Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-MS-2012-03

M.Sc. Thesis

A Novel Concurrent Validation Scheme for Hardware Transactional Memory

Anastasios Michos

Abstract

Transactional memory is a lock-free parallel programming model, which aims at replacing conventional lock-based threaded programming techniques, currently used by multi-core systems. These techniques are difficult to implement and impose unnecessary overheads caused by conservative programming practices. In this thesis, the scalability potential of a transactional memory system, called TMFab, was explored for different numbers of processors and it was concluded that for more than 4 processors the system presents reduced scalability, due to an increase in the validation overhead. In response to this observation, a novel validation scheme was proposed which reduces this overhead, first by allowing multiple transactions to perform their validations and commit operations concurrently, and second by removing the need for broadcasting messages between the active transactions. A distributed shared memory scheme was used to increase the validation and memory access throughput, as well as allow for transactions to commit concurrently on different memory partitions. The two architectures were compared by means of SystemC simulation, and a maximum of 2.5x validation speedup was observed for the modified design, together with a 2.7x reduction in memory access latency. In total, the modified design achieved a maximum execution speedup of 30% over the original, for the benchmarks that were used. Furthermore, the modified system guarantees sequential consistency even in corner case scenarios.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science Delft University of Technology

A Novel Concurrent Validation Scheme for Hardware Transactional Memory

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Anastasios Michos
born in Athens, Greece

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"A Novel Concurrent Validation Scheme for Hardware Transactional Memory"** by **Anastasios Michos** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: May 2012

Chairman: _____
prof.dr.ir. A-J van der Veen

Advisors: _____
prof.dr.ir. T.G.R.M van Leuken

_____
ir. Sumeet S. Kumar

Committee Members: _____
dr. ir. J.S.S.M. Wong

_____

Abstract

Transactional memory is a lock-free parallel programming model, which aims at replacing conventional lock-based threaded programming techniques, currently used by multi-core systems. These techniques are difficult to implement and impose unnecessary overheads caused by conservative programming practices. In this thesis, the scalability potential of a transactional memory system, called TMFab, was explored for different numbers of processors and it was concluded that for more than 4 processors the system presents reduced scalability, due to an increase in the validation overhead. In response to this observation, a novel validation scheme was proposed which reduces this overhead, first by allowing multiple transactions to perform their validations and commit operations concurrently, and second by removing the need for broadcasting messages between the active transactions. A distributed shared memory scheme was used to increase the validation and memory access throughput, as well as allow for transactions to commit concurrently on different memory partitions. The two architectures were compared by means of SystemC simulation, and a maximum of 2.5x validation speedup was observed for the modified design, together with a 2.7x reduction in memory access latency. In total, the modified design achieved a maximum execution speedup of 30% over the original, for the benchmarks that were used. Furthermore, the modified system guarantees sequential consistency even in corner case scenarios.

Acknowledgments

The great Greek poet Konstantinos Kavafis wrote in his poem Ithaca:

When you set out on your journey to Ithaca,
pray that the road is long,
full of adventures, full of knowledge.

My journey in Delft was indeed long, and one of the most defining ones I've had so far in my life. The experiences I gained in this journey both in a personal and an academic level will be a guiding light pointing towards the future.

For me, the most interesting part of this journey were the people I met and the friends that were close to me throughout my successes and failures. Therefore, I would like to express my gratitude to all those people that guided me throughout the way, helped me believe in my strengths and overcome my weaknesses.

First of all, I would like to thank my advisor Rene van Leuken, for entrusting me with this project, as well as helping me along the way with all the obstacles I encountered, while giving me the freedom to follow the direction I wanted to explore.

I would also like to thank my mentor Sumeet Kumar for his priceless support throughout the course of this thesis. Thank you for always being close to guide me, for motivating me, by simply making me love what I was doing as much as you did, and for always believing in me.

I would also like to thank Anupam for being a priceless companion in my never ending quest to find the bug...

A big fat thank you goes to my brother Giorgo, Aggeliki and Diana, my housemates, who always helped me see the positive side of life even when the world seemed to be falling apart.

Special thanks go to the SatCM group, Giorgo, Bjorn, Jeremy, Milan, Georgio and Max as well as Charlie the unicorn for bringing us all together and teaching us that there is an easy way to take the gloom away...Thank you guys for all the amazing times we had together and many more to come.

I also want to thank Gilbert for giving me good advice when I needed it the most, Zeki for boosting my spirit when I was down and, Norman and Wilko for all the mojito nights we shared together...or did we...I can hardly remember...

I also want to thank Nefeli and Isa for all the amazing times we had filled with dakos, compilers and failed portuguese lessons ...estoy bem?

I owe a very big thanks to Mark, for having a heart of pure gold...Kassandra for teaching me how to stand up for myself...and to Niv for reminding me that some things are worth fighting for...

At this point, I would like to thank all my friends back in Greece that were always supporting me throughout these years even though they couldn't be as close as I would want them to be.

Christo, my brother from another mother...there is always a place for you next to

me wherever I am... Ioanna, my beautiful p...I'm still waiting for you to come visit... Taso and Vasili thanks for filling up the lakos all these years...someone's got to do the hard work! Evaggelia thanks for promising to give me all your money...I will hold you to your promise! Nefeli thanks for always making me laugh... Sofia thanks for being the first one that believed in me and loved me for the dorky kid I was...I still am...I'm just hiding it better...

Just before the end I would like to thank Irini for just being awesome...and always standing by my side...listening to me whining and slapping me with her words whenever was needed...

And last but not least, I would like to thank my parents Argyri and Androulla as well as the rest of my family back in Greece, for always supporting me, believing in me and guiding me through good and bad times. Your love is like a big lighthouse that shows me where I come from and points to where I'm going.

Anastasios Michos
Delft, The Netherlands
May 2012

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In recent years, there has been an increased interest for the use of chip multi-processors (CMPs) in computer systems, as a means of overcoming the performance limitations imposed by single core systems. This change of direction has been strongly encouraged by the advances in integrated circuits (IC) lithographic techniques, which allow for tens of processors elements (PEs) to be integrated on the same chip. In order to exploit the increased number of resources, developers are requested to partition their programs in a way that there are multiple parallel tasks, known as threads, that can be executed concurrently on different PEs of the system. Conventional parallel programming models use locking to ensure correctness of execution and consistency of shared data used by multiple concurrently executing threads. When a thread reaches a part of the code, also known as critical section, in which there is access to a shared variable, it needs to acquire a lock protecting this variable in order to prevent any other thread from accessing it at the same time. Any other thread requesting for the same lock, is blocked until the task is finished and the lock is released, in order to be able to access its critical section in the same way. The drawback of locks, is that they can cause a number of issues, including the occurrence of deadlocks, convoying and priority inversion between threads of different importance. Furthermore, the complexity of programming with lock-based approaches is high given the need to explicitly manage accesses to shared data.

Transactional memory (TM), was originally proposed as an alternative to conventional lock-based parallel systems. Systems using this concept, replace the critical sections of a program with transactions. When a thread reaches a transaction, instead of acquiring a lock, it executes the transactional code speculatively, retrieving all the data it needs from the shared memory unobstructedly. The TM system keeps track of these accesses, and in case it realizes the presence of a conflict between two transactions, it forces one of the two to abort and discard the modifications it has performed in its local memory, in order to guarantee sequential consistency. In some cases, this execution scheme provides even better performance than the lock-based counterpart, while avoiding all the aforementioned issues tied with the latter[2].

A common issue when using locking techniques to ensure consistency, is that developers use the locks in a conservative manner. This means that the locks enclose a large part of a program's execution even though the shared variables are accessed in a small part of it. Consequently, the threads that are blocked need to wait much longer for a lock to be released in order to enter their own critical sections. On the other hand, in TM systems, even if the transactional part covers a large part of the total execution, the performance won't be affected considerably, since the transactions can still execute concurrently. The actual dependencies between these transactions are tracked

dynamically and only in the presence of a real conflict will the affected transactions be restarted and serialized. In this way, the task of tracking the dependencies between the threads is assigned to the TM system instead of the developer, thus simplifying programmability[3].

Most existing TM systems still make use of threads in order to achieve task parallelization, and only use transactions to replace originally locked critical sections. The TMFab system [1] on the other hand, considers that all the parallel tasks inside of a program are formed into transactional blocks which are assigned by a central hardware scheduler to different processing elements (PEs) to be executed concurrently. In this way, the overhead tied to context switching between different threads is avoided. Furthermore, TMFab provides a scalable processor independent TM framework intended for the use in CMP systems.

Nevertheless, the validation scheme proposed in TMFab doesn't scale well when the number of processors increases for the following reasons:

- Upon completion of a transactions execution, there is a set of messages broadcasted through the network, containing the addresses that have been speculatively modified. These messages need to reach all the other active transactions in the system. Consequently, when this number increases, the number of broadcasted messages increases proportionally, thus imposing a larger validation overhead.

- At any given time, only one transaction is allowed to validate its write-set. This effectively means, that any other transaction in the system needs to be stalled until the network is clear before it is given permission to validate. The computational time wasted this way increases considerably when the number of active transactions in the system increases.

- The increased number of validation packets causes for the interconnect to be congested, thus increasing the memory access latency for transactions that are still executing.

Furthermore, TMFab makes use of a compile-time prioritization scheme between the transactions, which causes for transactions to violate the rule of serializability in corner case scenarios, thus resulting to loss of sequential consistency.

The purpose of this thesis is to modify the original TMFab architecture in order to achieve better scalability of the system, while guaranteeing correctness of execution.

1.2 Thesis Goals

This thesis describes the optimizations performed on the original TMFab design in order to improve validation concurrency and scalability in general. The primary goals of this project are to:

- Improve the system's scalability by supporting validation and commit concurrency.

- Explore different conflict detection schemes in order to guarantee sequential consistency.

- Implement a banked memory system in order to better exploit the 3D stack architecture present in TMFab.

- Create a SystemC simulator for the original TMFab design in order to measure performance when the system scales up to more than 4 processors.

- Create a simulator of the modified design in order to compare performance with the original.

- Integrate additional units in the system, like a supervising processor and an instruction cache inside of every PE, in order to be able to run more complicated programs.

- Explore the system's performance using commonly used benchmarks for transactional memory systems.

1.3 Contribution

This work analyzed existing scalable hardware transactional memory (HTM) systems, in order to provide a suitable alternative validation scheme for TMFab, which is able to resolve the scalability and reliability issues of this TM system. Its major contributions are:

- Provides a novel validation scheme which allows for validation and commit concurrency between transactions and reduces the validation latency by a maximum of 2.5x .

- Incorporates a banked L2 cache memory in the system taking advantage of the stacked dice topology, reducing the memory access latency by 2.7x in the used benchmarks .

- Increases the system's scalability by reducing the number of validation messages that traverse the interconnect and by allowing multiple concurrent memory accesses either for data retrieval or for validation/commit operations

- Integrates a supervisor processor in the system which is responsible for executing the sequential code of a program, and for providing the necessary data initialization before the transactional blocks are executed by the remote processors.

- Provides a SystemC simulation framework that can be used for architectural exploration.

1.4 Thesis Organization

This thesis is organized into the following chapters:

Chapter 2 describes the fundamental concepts of parallel computing with respect to conventional parallel programming models and their weaknesses, which led to the use of transactional memory systems. Various TM proposals are then examined, with special focus being given to the ones that provide increased scalability. Furthermore, a brief overview of the original TMFab design is provided and the limitations that led to the modified architecture are outlined.

Chapter 3 provides an overview of the modified TMFab design. The different conflict detection scheme and contention management policy are explained and their contribution in increasing scalability is highlighted.

Chapter 4 gives a detailed description of the architecture of the modified TMFab design, as well as the practical decisions that were made during the development of the system and the reasoning behind them.

Chapter 5 evaluates the performance of the original and the modified architectures with respect to each other. Furthermore, the performance of the modified design is explored with respect to data distribution in different L2 data cache banks of the system.

Chapter 6 provides a summary of the work that was done and the achievements that were made, as well as recommendations for future work.

2 Background

This chapter presents the fundamentals of multi-processor systems and introduces the concept of transactional memory. Different Hardware Transactional Memory (HTM) architectures are summarized and their limitations with respect to scalability are explained. In response to these limitations, two HTM architectures, Scalable-TCC and KILO TM, are examined more extensively and their contribution in increasing scalability is highlighted. Further, the baseline design for this thesis, TMFab, is analyzed in detail, and both its scalability and reliability limitations, that led to the need for the current work, are listed.

2.1 Parallel Computing

Parallel computing is a programming method, in which larger tasks are divided into smaller ones, to be executed in parallel by multiple hardware resources. There are three main forms of parallelism :

1. ***Instruction Level Parallelism (ILP)***

   Every program is in essence a sequence of instructions. When two adjacent instructions are independent to each other, they can be executed in parallel by multiple functional units when they are present. This is referred to as Instruction Level Parallelism inside of a sequential code segment.

2. ***Data Level Parallelism (DLP)***

   Data Level Parallelism exists in computing loops, when a specific operation is performed on a large data structure. In this case, the loop can be unrolled, divided and distributed to multiple processors so that different parts of it are executed in parallel.

3. ***Task Level Parallelism (TLP)***

   Task level parallelism is a characteristic of parallel programs, where the different tasks that need to be performed are distributed into the available system resources and executed in parallel, without them having necessarily the same instruction code or data set.

ILP has been extensively used in the past in order to improve performance of uniprocessor systems, using instruction pipelining and superscalar execution among other techniques. However, there is a limitation to the parallelism that can be achieved in this manner. On the other hand,TLP and DLP, which is a special class of TLP, are very promising when used in multi-processor systems, even though the task of parallelization is shifting towards the programmer's side rather than the compiler's and hardware's in

ILP. This is the type of parallelism exploited by threaded programs. In those programs, processes that are able to run in parallel are referred to as *threads*.

There are two different types of multi-processor systems with respect to the communication between the different PEs:

1. ***Message Passing***

   In a message passing model, the different threads exchange data with each other through the use of messages. Every thread is then responsible for handling input messages and saving the enclosed data in their private memory.

2. ***Shared Memory***

   In shared memory systems, the communication between threads takes place simply through the use of shared variables, that reside in the system's shared memory.

The message passing model is more applicable in distributed memory systems, where every processor has its own private memory and when it needs data residing in another processor's local memory, it sends a message to fetch them, or waits for them to arrive according to the executing algorithm. The main drawback of this method is that it increases programming complexity by forcing the programmer to ensure correct communication between threads.

On the other hand, in shared memory systems there is no need for intra-processor communication in order to achieve data sharing. Every processor element (PE) updates its data directly to the shared memory, from where it can be accessed from any other PE in the system. The advantage in this case is that the programmer can achieve communication between different threads by the simple use of shared variables, thus reducing programming complexity. Chip multi-processor systems (CMPs), make use of this execution model, by having several processors integrated on the same chip sharing data through the use of an on-chip cache hierarchy. Usually, every processor has a private L1 data cache (L1D) in order to reduce data access latency and consequently improve execution performance, while all the processors share a larger L2D located in another block of the system. However, when using such a hierarchy, the system has to make sure that all the local caches are updated in such a way that all the processors have a coherent view of the shared memory.

2.2 Cache coherence

In case a specific cache line is requested by multiple PEs in a CMP, copies of it will eventually reside in all their local caches as well as the shared memory. Cache coherence protocols are needed, in order to guarantee that all of these copies contain the most recent version of data.In case that one of the PEs modifies the copy located in its own private cache, all the other copies need to be either updated or invalidated and retrieved again from the shared memory upon request.

In order for the caches to remain updated, they need to keep track of all the memory accesses performed by every PE in the system. In systems where the PEs communicate through a shared bus, *bus snooping* is the preferred method to perform this task. In

this case, every PE snoops on the common interconnect to see which cache lines are accessed by other PEs and updates its private cache accordingly. Even though this scheme is simple in its implementation, the use of a bus imposes limitations on the system's scalability. On the other hand, in systems where the communication is taking place through means of a scalable Network on Chip (NoC) architecture, *directory based coherence* is preferred. In this method, there is a central directory alongside the shared memory, which keeps track of each cache line's state inside of every PEs private cache. In this way, when a cache line is modified by one PE, this update is registered in the directory which in turn notifies all the other PEs to invalidate their copy.

2.3 Consistency

When multiple threads are running in different PEs of a system, it is possible that they are operating on the same set of data simultaneously.In the absence of a protocol that defines a way in which data are allowed to be accessed, every execution of the code would have unpredictable result, depending on the sequence in which the data are accessed from the threads.

In conventional lock-based systems, whenever a thread enters a section that is potentially conflicting with another thread's execution, also known as *critical section*, it needs to acquire a lock. If the lock is free, i.e no other thread has acquired it, then it is returned to the requesting transaction which is allowed to proceed further with its execution. Every consecutive request for the same lock by another thread will result in the requesting thread being stalled, until the thread that holds it reaches the end of the critical section and releases it. In this way, if the programmer handles the locks correctly, he can always predict the outcome of an execution sequence, thus guaranteeing sequential consistency.

The main problems resulting from this model are:

1. *Priority inversion* occurs when a process with lower priority holds a lock needed by a transaction with higher priority, thus delaying its execution.

2. *Convoying* occurs when a process holding a lock is, for some reason, descheduled without the lock being released, thus causing all the other processes in need of that lock to be stalled indefinitely.

3. *Deadlock* occurs when two processes try to lock the same set of objects in different orders and none of them manages to lock all of them so that it can enter the critical section.

4. *Reduced Scalability* appears when the same data set is shared by a large number of threads, which leads to threads being stalled for long periods while waiting to acquire the lock and access those data. The scalability is further reduced, in case of large critical sections, caused by conservative use of locks.

Transactional memory was proposed as a way to resolve all the above issues.

2.4 Transactional Memory

Transactional Memory was first proposed by M. Herlihy and J.E.B Moss in 1993 as a lock-free alternative to lock-based multiprocessor systems [2]. In this proposal, they replaced lock-based critical sections in programs, with transactions. In this model, conventional locks are replaced by transactional markers, signifying a transaction's boundaries.

When a a thread reaches a transaction during execution, it starts executing all the instructions included in it in a speculative manner. This means that, the thread can still retrieve data from the shared memory without acquiring a lock, but is allowed to modify them only in its own private memory. All the addresses that are read inside of this transaction constitute its *read-set*, while all the addresses that are speculatively modified from it, form its *write-set*.

The two main characteristic of a transaction stated in this proposal are:

1. *Serializability*: Transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another, even though the execution occurs concurrently.

2. *Atomicity*: Transactions perform speculative modifications only in their own private memory during execution. Upon completion of their task, they either commit and update the modified addresses to the shared memory, or abort and appear as they had never executed at all.

At the end of its execution, a transaction needs to compare the addresses it has speculatively accessed with the ones accessed by other transactions running on different PEs. This process is referred to as *validation*. When there is a conflict detected between two transactions, then one of them needs to be aborted according to a *contention management* scheme, discarding all its speculative modifications to private memory. In case the transaction manages to successfully complete the validation process, it is allowed to *commit* it's write-set, thus updating the modified data to the shared memory.

This programing model effectively guarantees serializability of execution, in the sense that concurrently executing transactions never appear to be interleaved with each other and the end result is the same as if they had been executed sequentially.

2.4.1 Transactional Memory Systems

Herlihy and Moss proposed a Hardware Transactional Memory (HTM) implementation, where the transactional and non-transactional part of the program are executing using different hardware resources. In other words, every processor has two local caches, one for transactional operations and one for the non-transactional. In their implementation, the non-transactional cache was a small fully-associative cache which accommodated all the speculative writes, before they became globally visible upon a successful validation/commit operation. However, since a cache line can't reside in both caches, this imposes a constraint on the number of cache lines that can be speculatively modified. This fact, limits the size of a transaction and imposes the additional task of respecting the hardware constraints on the programmer.

Transactional Memory Coherence and Consistency (TCC) [4], introduces an alternative to conventional cache line oriented coherence protocols, which incorporates both the reduced hardware complexity of message passing systems and the ease of programmability present in shared memory systems. In this proposal, the processors save all their speculative writes inside of their local cache memory and upon completion of the transaction's execution, they broadcast their write-set addresses to all the other active transactions in the system for validation. Upon successful validation, the transaction commits its write-set to the shared memory. This eliminates the need for a conventional snoopy cache coherence protocol, but increases the traffic in the interconnect due to the need of message broadcasting, thus impacting the system's scalability.

Hardware Support For Relaxed Concurrency [5] proposes a TM system, where the transactions are serialized upon commit according to a dynamic serialization protocol, which takes into consideration the dependencies between all executing transactions in the system. In this way, transactions abort only if is absolutely certain that sequential consistency will be violated otherwise. Using this scheme, the system achieved a maximum reduction in abort rate of 7.2, in comparison to other commonly used contention management schemes. This kind of system benefits the most in applications with long transactional sections and high contention, since in this case aborts result in more execution time being wasted. The drawback of this system is that conflicts are detected during a transaction's execution through means of broadcasting messages between all the PEs, which effectively congests the interconnect and reduces scalability.

2.4.2 Scalability

An important characteristic of CMPs is their ability to scale up their performance when the number of PEs in the system increases. Taking into consideration the current advances in integrated circuit technology, which make it feasible for hundreds of PEs to be packed together on a single chip, it becomes clear that any viability for transactional memory systems lies in improving their scalability.

One of the most important limiting factors for scalability in TM systems is the overhead imposed by the validation process. Even though threads are not forced to stall their execution due to the use of locks, there is now the additional overhead of transactions validating their read/write-sets in order to guarantee sequential consistency. When this takes place through a common bus interconnect [2], the transactions are actively tracking memory requests from other transactions during execution thus hiding the validation overhead inside of the execution time. However, these systems have considerable scalability limitations due to the fact that only one PE can access the bus at any given time. On the other hand, network based architectures force the transactions to broadcast messages to all other active transactions in the system in order to detect if there are conflicts between them[5].

This problem is solved by directory based schemes where a central directory is used to keep track of which private caches hold a copy of a specific cache line[4]. In this case, when a transaction is committing, the directory sends invalidating messages to the conflicting transactions thus causing them to abort. This way, there is no need for additional validation messages to be sent in the network. On the other hand, the size

of the directory increases proportionally to the number of PEs in the system, while the complexity of accessing all the tracking bits in every cache line increases accordingly.

In the following subsections, two HTM approaches that address the issue of scalability are going to be described with respect to both their achievements and their limitations.

2.4.2.1 Scalable - TCC

Scalable - TCC [6] is a variant of the TCC proposal, which aims at addressing problems concerning the original design's scalability. The main contribution of the new approach is that it allows for concurrent validation/commit operations to take place in different memory partitions.

In order for this to be achieved, the system is using a distributed shared memory architecture with multiple directories, one in each memory partition, to guarantee cache coherence. Transactions read from different partitions of the memory during execution, setting the tracking bits accordingly in the cache lines they retrieve. At the end of execution, every transaction requests for a Transaction ID (TID) which defines its priority in the commit sequence. Younger transactions, i.e with a higher TID number, commit always after an older transaction, except if it is absolutely guaranteed that there is no conflict between them, which is the case when they operate on different partitions of the memory. If the latter is true, then the transactions are allowed to validate and commit their write-set concurrently without violating sequential consistency, thus reducing the overall overhead related with using a TM model.

The main drawback of this system concerns the additional hardware resources needed to implement a directory when the number of PEs in the system increases. The system provided good scalability, by showing a maximum speedup of 57 for a 64 processor topology.

2.4.2.2 KILO TM

KILO TM [7] is a TM system proposal, intended for use in GPU architectures where there is a lot of inherent task parallelism and thousands of threads executing concurrently. The system combines aspects of value-based conflict detection, RingSTM [8] and Scalable - TCC [6] to achieve better scalability. Distributed shared memory is used the same way as in [6] in order to support parallelization of validation/commit operations. The difference in this case is that, instead of using a directory based cache coherence scheme which is not very scalable in terms of area consumption and complexity, a value-based conflict detection protocol is used.

At the end of execution every transaction sends its read/write log, containing all the address-value pairs of the memory locations speculatively accessed by it, to the corresponding memory partition in order for conflicts to be detected on a value-comparison basis. In order for this to be possible, the logs coming from all the transactions need to be saved in a temporary location in the memory partition, which effectively increases the storage requirements of the shared memory and imposes a limitation on the maximum number of transactions that are able to validate in parallel. On the other hand,

this architecture is intended for use in GPUs, where the data set of every thread is small, which is not the case in a GPP.

The system implements the validation scheme in a pipelined way. When a transaction first enters the pipeline, it needs to validate its read-set to the shared memory in order to detect if any addresses in its read-set where modified during execution, which would force the transaction to abort. If there are no conflicts detected, the transaction validates with the older transactions at later stages of the pipeline in order to detect possible conflicts, which are referred to as *hazards*. If the transaction, with which their is a hazard, manages to commit its write-set to memory, then the hazard will result in an actual conflict since the hazardous address will be modified. In order for this to be detected, the transactions are obliged to re-validate their hazardous addresses before they are allowed to commit. In case there is no hazard or conflict detected, the transaction is allowed to update the memory.

In order for this scheme to perform correctly, the transactions need to validate in the same order in all the memory partitions. Consequently, even if a younger transaction arrives earlier than an older in a specific partition, it needs to wait before it starts validating, possibly wasting validation time if there is no other transaction validating with the memory at that time. In addition to that, the value-based conflict resolution forces every transaction to also send the read-set data through the network to be validated with the shared memory, thus increasing the network traffic and decreasing the system's scalability.

2.4.3 TMFab

The baseline architecture for this work is called TMFab and was proposed by S. Kumar & R. van Leuken in 2010 [1]. TMFab is a processor independent TM architecture intended to be used for scalable shared memory CMP prototyping, ideally for technologies that support die stacking.

In this proposal, the use of threads is completely replaced by the use of transactions which is the basic unit of parallel work. In other words, the programmer instead of spawning threads to divide the different tasks in the program, he simply has to include the parallel workload inside of transactional markers, which reduces programming complexity. In addition to that, the overhead related with context switching in conventional threaded programs is avoided.

The system is based on a 3D stacked architecture where arrays of PEs can be stacked on top of each other on different dice in the same chip. In this way, the number of processors can be expanded in the vertical direction without increasing the chip's area footprint. The communication between the different PEs in the system is provided by a scalable 3D mesh NoC interconnect, illustrated in Figure 2.2 which uses Through Silicon Vias (TSVs) for the communication between different dice.

TMFab uses lazy *version management*, which means that every transaction's speculative writes are saved in an internal Speculative Write Buffer (SWB), instead of being updated directly in the L1D, in order to be updated to the L2 data cache upon successful validation, or easily discarded in case of a necessary abort. The use of such a buffer also facilitates the creation of validation packets during the validation stage, since the addresses to be validated lie in the SWB in consecutive locations.

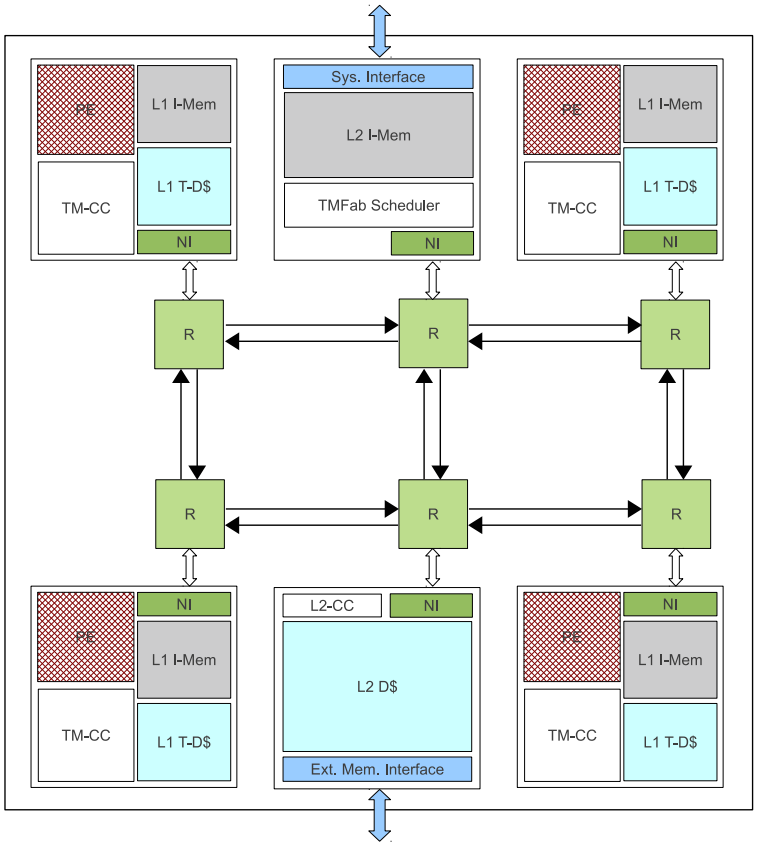

Figure 2.1: Original TMFab Design [1]

For the detection of conflicts between transactions executing on different PEs, a lazy *conflict detection* scheme has been used. In other words, the transactions validate their data-sets after they complete execution. This prevents unnecessary aborts of lower priority transactions caused by higher priority transactions that eventually don't manage to commit. Additionally, a word level granularity validation scheme has been deployed, which uses write masks to achieve cache line granularity overheads, resulting in smaller validation packets being broadcasted through the interconnect, thus reducing the congestion in the network and consequently the validation overhead in comparison to word granularity validation schemes.

Furthermore, there is no conventional cache coherence protocol used requiring either bus snooping or a directory to keep track of a cache line's state. Common cache lines between different transactions are invalidated according to the incoming validation packets sent by every validating transaction before commit, while the shared L2 data cache is updated by every transaction in the commit stage.

The contention management policy in this proposal is implemented through means of a priority scheme defined at compile time. Every transaction is given a phase and a sequence before it starts executing its transactional code. During validation, the transaction sends this information along with the validating addresses to the other active transactions in the system. These compare the incoming phase and sequence

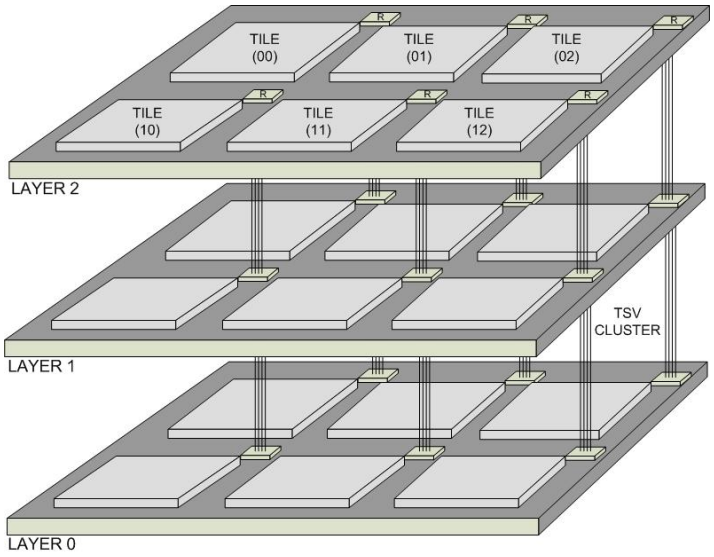

Figure 2.2: 3D Mesh NOC [1]

information with their own and according to whether they are younger or older, they abort or send an abort message to the validating transaction respectively. Even though the transactions have a predefined priority, they are allowed to commit out of order in case a younger transaction completes execution earlier and presents no conflicts with all the older transactions in the system. This prevents smaller transactions from being stalled by larger ones with higher priority, which would result in PEs being unnecessarily occupied by completed transactions for greater time periods before a new transaction would be scheduled on them.

The system provided good scalability for a topology of up to 4 PEs, showing a maximum speedup of 3.44x for 4 independent similar transactions with same workloads. However, the simulations that were part of this work showed that when the number of PEs increased in the system, the performance didn't increase accordingly. This happened for the following reasons:

1. At the end of execution, every transaction needs to broadcast validation packets to all the active transactions in the system. This effectively means that when the number of PEs increases, the number of validation packets that traverses the network increases exponentially, thus incurring a larger validation overhead.

2. The architecture as it is, allows for a single transaction to be able to validate and commit at any given time. In other words, even if a transaction has completed execution, it will be stalled until it has acquired an exclusive validation token in order to be able to start validating. When the number of PEs increases, the amount of time wasted in stalling increases accordingly, since there are more transactions in line to acquire the token.

3. The L2D is located only in the first die of the system, which means that when the system is scaled up in the vertical direction, it takes more time for requests in higher levels to reach the shared memory.

In addition to the scalability issues of the system, there are also problems that affect the correctness of execution. Those are:

1. The static compile time prioritization of transactions, in combination with the out of order commit scheme, allows for some Read-after-Write (RaW) conflicts to be disregarded in corner case scenarios. In case a younger transaction successfully validates with an older one before the former completes execution, all the conflicts that exist in the unexecuted part of the transaction are going to be disregarded, since none of the transaction will abort. This will result to violation of sequential consistency and loss of correctness.

2. The system doesn't take into consideration race conditions in which a transaction that has been aborted, accesses stale data in the shared memory, before they are actually updated by the committing transaction. This task is left to the inherent latency of the network communication. However, this latencies are not as predictable when the system scales up to the vertical direction and the distances between the shared memory and the PEs varies considerably.

Apart from the above problems that affect scalability and correctness of execution, the following practical issues were observed, which limit the system's functionality:

1. The lack of an operating system (OS), or a supervising processor (SP) make it difficult for the sequential part of the program to be executed. In the tested benchmarks that were small and fully transactionalized, this weakness didn't impose a considerable problem. However, when more complex programs are executed, the presence of either a SP or an OS is of outmost importance.

2. The lack of an instruction cache inside of the PEs makes it impossible to service instruction misses. For this reason, all the instruction code that could be used by a transaction had to be transfered and stored in the L1 Instruction memory of the PE before execution, thus increasing the communication overhead and the L1IMem requirements, as well as limiting the maximum size of a transaction.

2.5 Summary

This chapter presented an overview of parallel computing systems in general and explained the contribution of transactional memory in resolving problems related to conventional lock based parallel programming techniques. Further, some of the most influential transactional memory proposals were described with main focus being given on their scalability. One of these proposals, TMFab, was analyzed in more detail, and its issues concerning scalability were highlighted. Chapter 3 presents TMFv2, a new transactional memory proposal based on TMFab, which focuses on resolving all the aforementioned issues, by applying a novel concurrent validation scheme which reduces the validation overhead considerably.

3 TMFv2 Design Overview

Chapter 2 gave an overview of transactional memory, and described some influential HTM proposals focusing on their scalability potential. Further, the TMFab[1] design was described and its limitations were highlighted. This chapter describes a modified version of this system, TMFv2, which targets to overcome the original design's limitations with respect to scalability and correctness of execution.

3.1 TMFv2 Overview

The original TMFab design provides a processor independent transactional memory framework for CMP prototyping, which offers increased scalability, mainly in terms of the scalable NoC interconnect that is used. However, experimental results showed that the system's scalability decreased for topologies using more than 4 PEs. There are two main reasons behind this performance limitation:

- The number of validation messages every transaction sends increases proportionally to the number of PEs in the system, thus increasing the validation time per transaction.
- Only one transaction is able to validate its write-set at any given time, causing all other transactions to stall even if they have completed execution.

In addition to that, the system is not able to guarantee sequential consistency of execution.

TMFv2 is a modified version of TMFab, which uses a new, more scalable validation scheme which:

1. **Allows for validation and commit concurrency of transactions** Transactions are able to validate simultaneously on the shared L2D cache. A distributed banked L2D cache topology allows for increased validation throughput, by allowing multiple validation packets to be handled concurrently by different cache banks. Further, this topology also allows for different non-conflicting transactions to commit their write-sets concurrently thus reducing even more the validation overhead.
2. **Makes the number of validation packets independent from the number of PEs in the system** The number of validation packets is defined only by the size of a transaction's read-set, and by the number of potential conflicts between transactions. Thus, when the number of PEs is scaled up, the validation overhead does not increase which leads to faster execution, which is limited mainly by the available memory throughput.

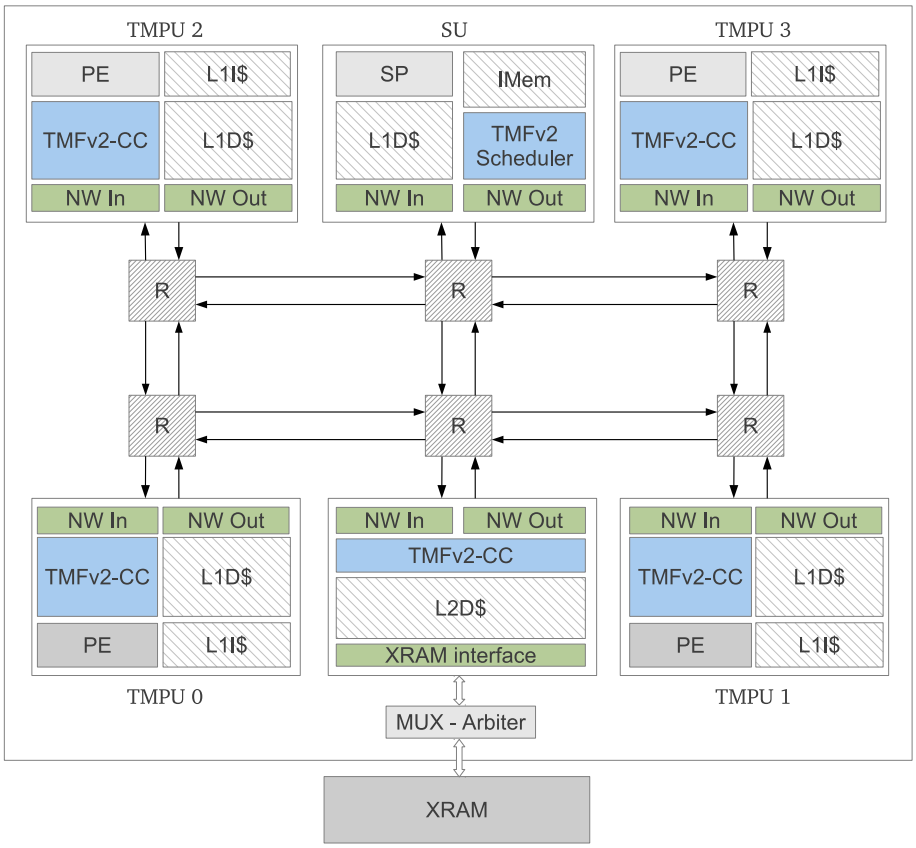

Figure 3.1: TMFv2 Design

3. **Guarantees sequential consistency of execution even in corner case scenarios** A dynamic prioritization scheme has been used which orders transaction upon completion of their execution. This fact, together with the conflict detection scheme that has been implemented, allows for all possible conflicts between transactions to be detected, thus providing correct execution in all cases.

TMFv2 consists of four main units appearing in Figure 3.1, the *Supervising Unit (SU)*, the *Transactional Memory Processing Element (TMPE)*, the *L2 Data Cache (L2D)* and the *Network-on-Chip (NoC)* interconnect.

At the beginning of execution, the SU reads and stores the instruction code of the transactional program in its internal *Instruction Memory(IMem)*. The *Supervising Processor(SP)* located inside of the SU starts executing the sequential part of the code until it encounters a transactional section, demarcated by a START_TXN transactional marker. At this point, the SP is stalled and the *TMFv2 Scheduler (TMS2)* takes control of execution by scheduling all the following transactions, which precede the next sequential section of the code, to the available TMPEs in the system to be executed in parallel.

When a transactional instruction block, sent from the SU, is received by the TMPE, it is stored inside of the *L1 Instruction Cache (L1I)*. Upon arrival of a start signal

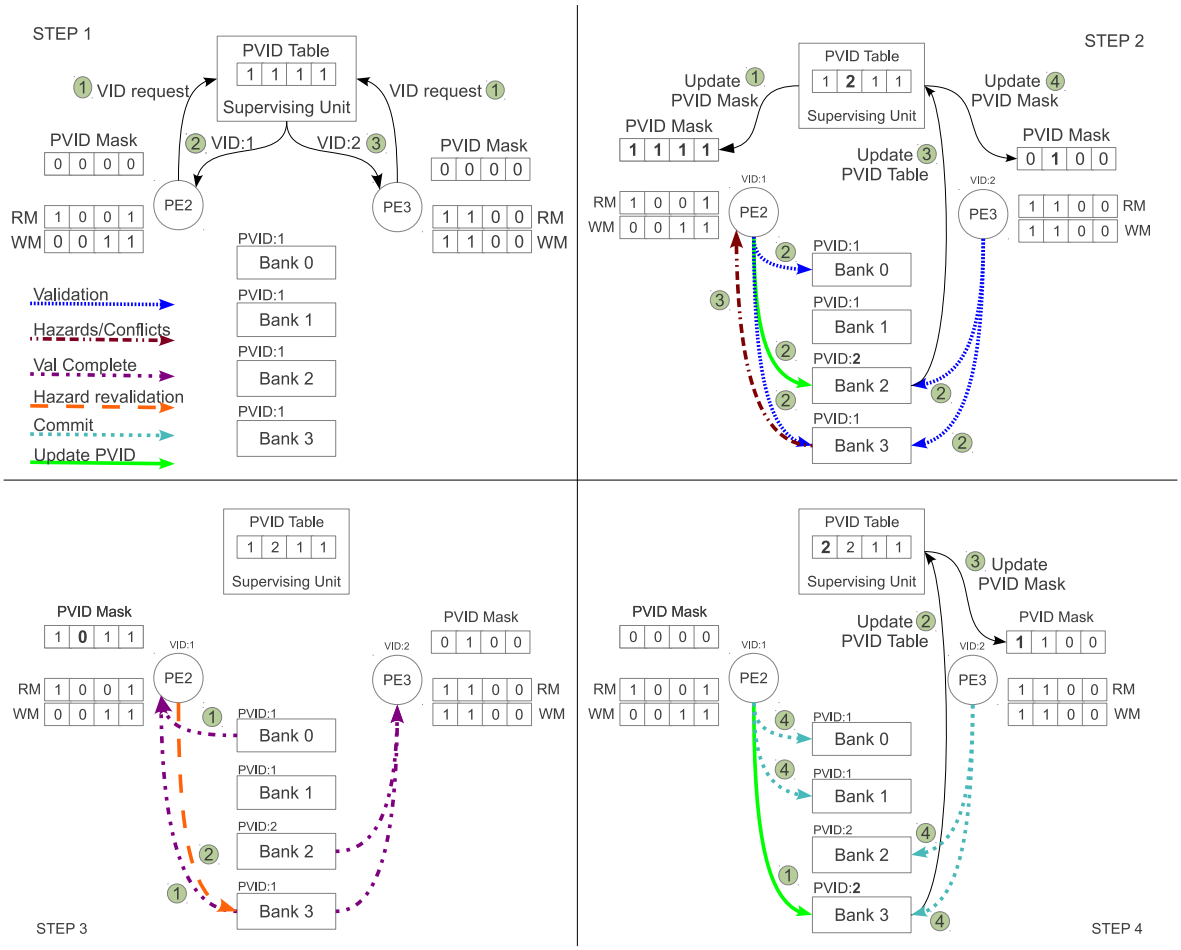

Figure 3.2: TMFv2 Concurrent Validation/Commit example

also originating from the SU, the *Processor Element (PE)* inside of the TMPE starts executing the transactional code speculatively, meaning that the transaction might be restarted at a later stage if it is conflicting with a higher priority transaction. If there is an instruction miss in the L1I during execution, caused by a call to a function located outside of the original transactional block, this is serviced by the IMem of the SU. In case of a cache miss in the local *L1 Data Cache (L1D)*, the data are retrieved from the shared L2D through the NoC. From now onwards the TMPE will be referred to as PE for simplicity reasons.

All the speculative modifications performed by a transaction during execution, are saved in a *Speculative Write Buffer (SWB)* inside of the PE. Upon completion of execution, the PE on which the transaction was executing, requests for a *Validation ID (VID)* from the SU (Step 1 of Figure 3.2), which defines its priority in comparison to other validating transactions in the system. The VID is provided in a first-come first-served basis, thus giving higher priority to smaller transactions that execute faster or transactions with similar workloads that have smaller communication latency to the SU.

When a transaction acquires a VID it starts sending the cache line addresses it has speculatively read , which form its read-set, to the L2D, to be validated with the cache lines there. If it is concluded that the cache line has been modified by another transaction in the system while the validating transaction was executing, the cache line data are returned to the validating PE in order to assert that there was an actual conflict. If the last is true, then the transaction is forced to abort, notify the TMS2 for this action, and restart. When it again reaches the end of execution, it needs to acquire a new VID before it starts validating anew.

Unlike the original TMFab design, TMFv2 allows for multiple transactions to validate their read-sets concurrently, as well as commit their write-sets concurrently if there are no conflicts detected. In order for this scheme to provide a performance improvement, there is need for the shared L2D to be partitioned into banks, so that it is possible for multiple validation packets originating from different transactions to be handled in parallel by multiple banks in the system. TMFv2 leverages the potential of the 3D stack topology, which is part of the original TMFab design, by placing every L2D bank on a different level in the chip, also known as *die*. The benefits derived from this placement method are:

- Transactions executing on different dice in the system, have similar latency while retrieving data from the shared L2D, provided that data are distributed evenly between the different banks in the stacked topology.

- Multiple validation packets can be handled in parallel by different banks in the system, thus reducing the validation overhead.

- In case that the read-set of a lower priority transaction and the write-set of a higher priority one are located in different banks, the transactions can potentially commit in parallel, thus freeing the PEs in which they are executing earlier and allowing for new transactions to be scheduled on them.

During execution, every PE keeps track of which banks it has speculatively read in a Read Mask (RM), and speculatively written in a Write Mask(WM). When execution completes and the PEs are assigned their VIDs, they start sending their read-sets to the banks they have accessed for validation. This is a task that can take place concurrently from multiple PEs irrelevant to their VID number. However, only one PE is allowed to commit to a specific bank at any given time. This is defined by every bank's *Priority VID (PVID)* number. When a PE gets priority over all the banks it has speculatively accessed, it is able to commit its write-set provided that no conflicts have occurred in the meantime.

When a PE has priority over a bank and has no more need for it, either because it is not in its read/write-set or because it has updated it already, it sends an update message to it. At that point, the bank increases the PVID number by one and informs the SU of this action. The SU updates its internal PVID table, and then locates the PE which has a VID matching the PVID of the updated bank, and sends a message to inform it that it has priority over this bank. Upon receival of this message, the PE updates its internal PVID Mask. This procedure is illustrated in Step 2 of Figure 3.2.

During validation, a PE receives messages from the L2D informing for potential conflicts with transactions that have already updated the cache, or with the ones that are still in the validation process. The last ones are called *hazards*. The first type of conflicts are resolved at the time they occur. However, the hazards are resolved only when a PE has priority over all the banks it has speculatively accessed. When this happens, the PE sends all the hazardous cache lines for re-validation. If no conflicts occur from this process, then the PE is allowed to commit its write-set to the L2D. However, before it commits, it sends messages to all the banks that are not in its WM to update their PVIDs because they are not needed. This allows for other PEs to get priority over those banks faster and complete their re-validation process earlier, as shown in Step 4 of Figure 3.2.

The communication between the different units in the system is serviced through a scalable 3D mesh NoC interconnect, similar to the one used in the original TMFab architecture. This kind of interconnect was chosen because NoCs show better scalability in comparison to a bus or a crossbar based architecture[9], while 3D meshes show lower cross network latency than 2D meshes of the same size [10].

The following section describes in more detail the policies used by the TMFv2 design.

3.2 Transactional Memory Policy

In this section, the different policies used by TMFv2 will be described and their differences with the original TMFab design will be highlighted.

3.2.1 Transactional Programming

Conventional TM programs follow the threaded programming approach, with the difference that they replace the locked critical sections with transactions, by including them in transactional markers. When the PE on which the thread is running reaches a transactional marker, it saves the state of the registers and starts executing the transactional part speculatively. In case the transaction aborts, the registers are loaded with their pre-transactional state and execution restarts. The drawback of this model is that the programmer is still responsible for identifying the critical section and enclose it in transactional markers.

Similarly to TMFab, TMFv2 uses a programming scheme in which the threads are completely replaced by transactions. In this case, instead of creating a thread to execute the parallel workload, the programmer needs to enclose it between a START_TXN and an END_TXN transactional marker. All the code inside of these markers is considered as one transaction and has to be sent to one remote PE to be executed. All the code outside of these markers is considered sequential and is executed by the SP of the SU. In this sense, threads turn into coarse grained transactions. The drawback of this scheme is that in case of an abort, there is more transactional work wasted. On the other hand, this model offers smaller programming complexity, since the programmer is not obliged to locate the conflicting sections of the code and transactionalize them.

When the program starts executing, the SP starts executing the initialization code which sets the original values in registers, like pointers to addresses in memory and other data necessary for the programs correct execution. In order to guarantee correctness of execution, those registers are sent to all the remote PEs that are going to execute one of the following transactions. This initialization process could also take place inside of the remote PEs, however this poses a considerable overhead since all of these instructions need to be sent to every single PE through the NoC.

During the SP's execution of the code, the requested cache lines are retrieved from the L2D through the NoC. These are temporarily stored inside of the SU's L1D. Every cache line that is modified inside of this cache needs to be updated in the L2D when the sequential segment reaches the end. At the end of this process, the L1D is flash cleared in order to guarantee that future sequential sections are not going to execute on stale data, since the modifications made by the PEs are not updated in the SU's local L1D.

When the SP arrives at a transactional section, its execution is stalled by the instruction interface and the scheduler starts assigning the transactions that follow to the idle PEs in the system. In order for the transactions to be executed concurrently, they need to be positioned adjacently, without intermediate code between them. Otherwise, the scheduler will consider that the transactional part is over, it will wait for the transactions that are scheduled to complete execution and then pass again control to the SP which will start executing the next sequential section. All the transactions that are placed between two sequential sections are considered to belong in the same *phase*, but have different *sequence* between them, while the transactions that are divided by a sequential section, are considered to belong in different phases.

3.2.2 Version Management

The version management policy refers to the way in which the speculative writes of every transaction get updated in their local L1 Data Cache. TMFv2 uses a *lazy version management* policy similar to the one used in the original TMFab. In this scheme, the speculative writes of a transaction are kept inside of a *Speculative Write Buffer(SWB)*, instead of being updated directly to the the L1D which is the case in an *eager version management* scheme.

The advantage of this policy is that the transaction's read-set is located already unaltered inside of the L1D, so in case of an abort, the SWB is flash cleared and the original data are directly accessed from the L1D, which speeds up the abort operation. Alternatively, in the eager scheme, the transaction would update directly the L1D and the original values would be held in an *undo log* from where they would be restored in the L1D in case of an abort. The advantage of this scheme is that it allows for faster commit operations since the modified data don't need to get updated in the L1D as well. However, in the TMFv2 there is no need for the data to be updated at the end of a transaction because the L1D will be flash cleared before a new transaction arrives as is going to be explained in subsection 3.2.5.

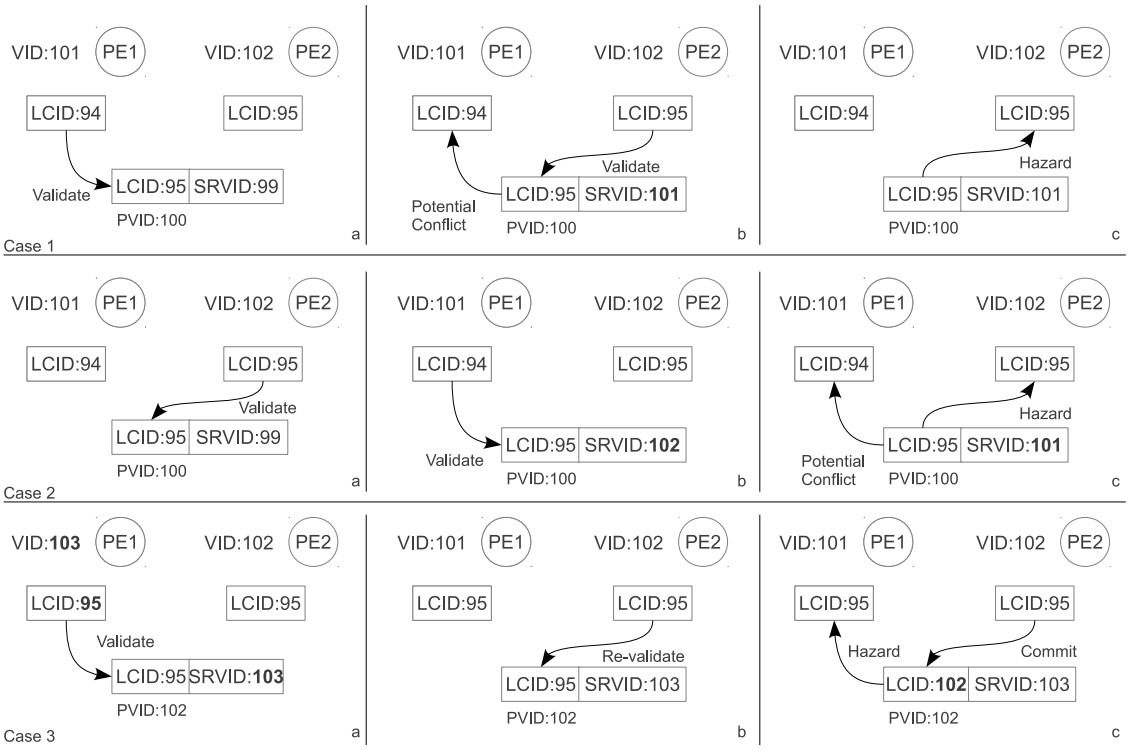

Figure 3.3: TMFv2 Conflict detection example

3.2.3 Conflict Detection

The conflict detection scheme used in TMFv2 differs considerably from the one used in the original TMFab design. Both proposals use *lazy* conflict detection, which means that the transactions validate at the end of their execution, instead of doing so whenever a memory access occurs, which is the case in an *eager* conflict detection scheme. However, unlike the original TMFab design, where every transaction validated its write-set with every other transaction in the system, in TMFv2 transactions validate their read-sets only with the L2D. In the first case, the number of broadcasted validation messages increases exponentially with the increase of the PEs in the system, which consequently reduces the system's scalability. On the other hand, in TMFv2 the number of validation packets is independent from the number of PEs in the system, and is defined only from the size of the read-set, as well as the number of potential conflicts that occur during validation.

When a transaction reaches the validation stage and acquires a VID from the SU, it starts sending the cache line addresses in its read-set to the L2D in order to check if they were modified during execution by another transaction. In order for this to be possible, every cache line in the L2D is enhanced with one more tag containing the VID of the last transaction that modified it, which is called *Last Commiter ID (LCID)*. When a cache line is sent to a PE upon request, the LCID tag is sent alongside the data and is stored in a similar tag inside of the L1D of the requesting PE. During validation,

this tag is sent back to the L2D in order to be checked with the most recent LCID of the cache line. If the two tags are different, it means that the cache line has been modified during this transaction's execution by another transaction, thus signifying a potential conflict (case 1a-1b,2b-2c of Figure 3.3). However, since the comparison takes place in a cache level granularity, it is not certain that the modified words are the same as the ones read by the validating transaction. For this reason, the cache line data are sent back to the validating transaction's PE to be compared in a word level granularity with the data there. If it is concluded that there was an actual conflict, then the transaction aborts and restarts, notifying the SU for this action.

Furthermore, since there are multiple transactions validating concurrently, which haven't updated their write-sets yet, there is additional need to validate with them too. In order for this to be possible, the L2D cache lines were augmented with one more field called *Speculative Reader VID (SRVID)*. When a transaction sends a validation packet to the L2D, it sends its VID alongside the cache line addresses. If the SRVID number is smaller that the bank's PVID number, implying that the transaction is no longer active, the SRVID fields of the validated cache lines are updated with the incoming VID (case 1b,2b of Figure 3.3). If at some point, another transaction updates this cache line during a commit operation, then the transaction whose VID is equal to the SRVID receives a message which indicates a *hazard* in that specific cache line address(case 3c of Figure 3.3). Before this transaction manages to commit, it needs to validate all the hazardous lines again, in order to detect any possible conflicts with transactions of higher priority, that hadn't committed yet at the time this transaction started validating. In case multiple transactions have read the same cache line during execution, then only the one with the highest priority will keep its VID registered in the SRVID field of that cache line. All the other transactions are going to receive hazard messages, even though the cache line hasn't been modified, in order to revalidate it before they are allowed commit(case 1c,2c of Figure 3.3).

In order for sequential consistency to be guaranteed, before a transaction is able to commit, it needs to have priority in all the banks that it has accessed. For this reason, every bank has a special entry, called *Priority VID (PVID)*, which contains the VID of the transaction that is allowed to commit on that specific bank. When the transaction gets priority over all the banks it has accessed, it re-validates all the hazardous lines and upon successful validation, commits its write-set to the L2D.

The advantage of this scheme over a directory based one, is that the former uses a predefined number of bits per cache line to save the LCID and SRVID fields, while in the latter the number of tracking bits per cache line increases according to the number of PEs in the system.

3.2.4 Contention Management

The contention management policy refers to the way two conflicting transactions decide which one is going to abort. Both TMFab and TMFv2 follow the *Oldest Contention Management (Oldest CM)* policy. In other words, when two transactions are conflicting, the youngest one is forced to abort while the oldest is allowed to commit its' write-set. The main difference between the two designs, is that in TMFab the transactions are assigned a predefined priority at compile time, while in TMFv2 the priority is defined

dynamically on a first-come first-served basis inside of the TMS2. In the original approach, the system presented two main drawbacks :

1. If a younger transaction finishes its execution earlier than an older conflicting one, it needs to abort and restart even though the older is not even at the validation stage yet. This causes for the younger transaction to re-execute all of its instruction code, while it could have committed and retired, releasing the PE for another transaction to be scheduled. In some cases, this transaction might abort multiple times before the older one has committed, resulting in an increase in the wasted work per PE and a decrease in the overall performance. This phenomenon is aggravated in case there is a chained dependency between multiple transactions, in which case the oldest transaction that aborts will force all the younger ones to also abort, thus actually serializing the code's execution.

2. In the case that, in the previous example, the younger transaction didn't detect any conflicts and committed its write-set, before the older one would arrive at the end of its execution, there is still a possibility that the older transaction would read on data that the younger updated. This already violates the principle of serializability because transactions appear to have executed in an interleaved way.

TMFv2 solves the above issues, by implementing a dynamic prioritization scheme for the transactions similar to the one proposed in [6]. In this case, transactions acquire a VID from the TMS2 upon completion of their execution. After that, they validate their read-set inside of the L2D together with other validating transactions. If there is a conflict between transactions that validate in parallel, the *older* transaction (lower VID) wins the conflict and the *younger* (higher VID) is forced to abort. However, in this case the transactions can be aborted only by another one in the validation stage and only if it is certain that the latter managed to commit, unlike the original TMFab design. In other words, a transaction will abort only once due to an older conflicting transaction, instead of multiple times in the previous TMFab design.

In addition to the reduction in the abort rate, TMFv2 is able to guarantee sequential consistency by detecting all possible conflicts. The reason is that in this case, all transactions participating in the validation scheme have completed execution and their read/write-sets are complete. In other words, it is not possible that a younger validating transaction will ignore a Read-after-Write conflict with an older executing one that has an uncompleted write-set, which was the case in TMFab.

3.2.5 Cache Coherence Protocol

The cache coherence protocol is one more point in which TMFv2 differs from the original TMFab proposal. In the latter, the L1D was kept coherent with the help of the broadcasted validation messages between the active transactions in the system. However, if a PE was idle and wasn't executing a transaction, it didn't receive any validation packets to guarantee its L1D coherence. Future execution of another transaction on the same PE could potentially result in stale data being used causing incorrect execution of the program.

In TMFv2 there are no validation messages being broadcasted between the different PEs which would invalidate the modified cache lines, since all the validation process is taking place inside of the L2D. When there is a potential conflict detected inside of the L2D, the conflicting cache line is sent back to the validating PE, both for the conflict to be resolved in a word level granularity, and for the modified data to be updated inside of the PE's local L1D. This way, even if the transaction is forced to abort, it has the updated data on its L1D and can execute on them during the second execution.

When a transaction commits and retires, the PE flash clears the L1D in order to be ready for the next transaction which will be scheduled on it. This action is necessary, because after a transaction retires, there is no mechanism to keep the L1D updated when the L2D is modified. Taking this into consideration, in the alternative case where the L1D wouldn't be erased, the following transaction would execute on stale data. At the time that this transaction would validate the data with the L2D it might be concluded that the data had been modified and the transaction would be forced to abort. By erasing the L1D the transaction will request for all the cache lines anew, thus reducing the possibility that it will access stale data.

If two consecutive transactions that run on the same PE are not correlated, which is usually the case in the current programming model, most of the second transaction's read operations would result in compulsory misses, which means that there is no benefit in keeping the L1D coherent between two different transaction that run on the same PE. However, if the two transactions have similar read-sets, then a directory based approach could be a better solution to guarantee cache coherence, with all the additional memory requirements this would imply.

3.3 Summary

This chapter presents TMFv2, a transactional memory system which applies a novel concurrent validation scheme on the original TMFab architecture. The new scheme, allows transactions to validate their read-sets and commit their write-sets concurrently, thus preventing transactions from being stalled for long periods of time before they are able to validate/commit. Furthermore, a banked L2D cache scheme has been used to increase the memory access throughput and consequently the validation throughput. Additionally, the number of packets needed for the validation to be performed, is independent from the number of PEs in the system, thus allowing for better scalability. All these factors contribute in reducing the validation/commit overhead, and improving overall the system's performance. Furthermore, the new system guarantees sequential consistency in all cases. The following chapter gives a detailed architectural description of the TMFv2 system.

4 Architecture

Chapter 3 describes the transactional memory policies implemented by TMFv2, highlighting the differences with the original TMFab system. In this chapter, the detailed architecture of TMFv2 is described with respect to the main four components : Supervising Unit, TM Processing Unit, L2 Data Cache and the NoC interconnect.

4.1 Supervising Unit

The Supervising Unit(SU) illustrated in Figure 4.1 is responsible for reading the input binary file which contains the transactional program's instruction code, execute the sequential part and distribute the transactional accordingly to the available PEs residing in the CMP. The SU consists of four basic units : the TMFv2 Scheduler(TMS2), the Supervising Processor(SP) the SU Input and the SU Output.

4.1.1 TMFv2 Scheduler

At the beginning of execution, the binary file containing the instruction code, is read and stored inside of the SU's instruction memory (IMem). This file contains both the sequential code that is meant to be executed by the SP and the transactional code that will be executed in parallel by the available PEs in the system.

The transactional code is separated by the sequential through means of two transactional markers, the START_TXN and END_TXN, demarcating the beginning and end of a single transaction respectively. The instruction code that is enclosed by these

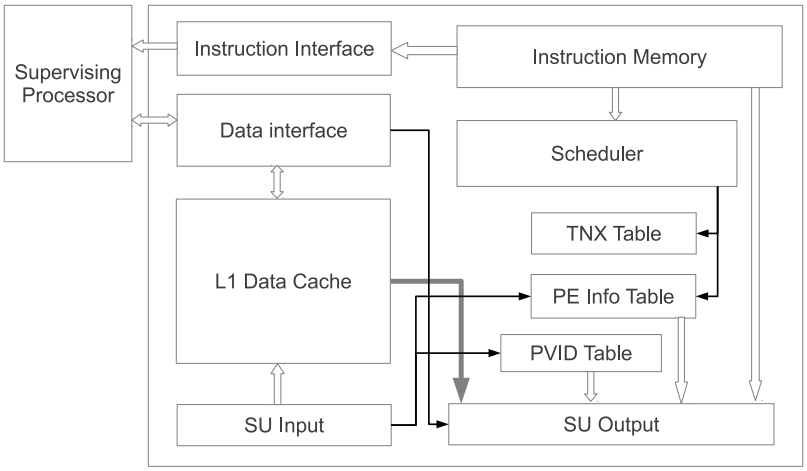

Figure 4.1: Supervising Unit

markers is meant to be executed by a single remote PE. Two consecutive transactions that have no intermediate instruction code between them are considered to belong in the same *Transactional Section(TS)* and are able to execute in parallel. If there is sequential code between two transactions, it means that they belong in different TS and the second needs to wait for, both the first one and the sequential section to complete execution, before it is able to execute.

Unlike the original TMFab design, TMFv2 integrates an additional processor inside of the SU, the SP, which executes the sequential sections of the code. At every clock cycle, the SP requests for the next instruction in line to be executed from the *Instruction Interface*. Before providing the requested instruction, this block checks whether this is actually a transactional marker. In case it is, the SP is stalled and the TMS2 takes over to start the execution of the TS.

In order for this task to be achieved, the TMS2 needs to be aware of the start and end addresses of every transaction. In the original TMFab design, the position of the transactional markers was defining those addresses. In that case, when the scheduler would detect a START_TXN marker it would start scanning and sending all the instructions to an idle PE until an END_TXN marker would be encountered. The drawback of this method is that all the instructions of a transaction are sent, even if some of them are not really used due to a conditional branch operation. In big transactions this could pose an unnecessary additional overhead. In addition to that, all transactions will be sent to the remote PEs one by one in their whole, regardless of their size. This effectively means, that every transaction will start executing potentially with a large phase difference, waiting for all the transactions that precede it to be transfered first.

For the aforementioned reasons, TMFv2 follows a slightly different approach. There are still two markers, one at the end and one at the beginning of every transaction, but their content is different. As illustrated in Table 4.1, the START_TXN marker comprises of 5 instructions. The first two instructions are NOPs, which are needed for practical reasons, in order to give time to the supervising processor to finish any pending store/load tasks. The third instruction is the actual start marker that signifies the beginning of a new transaction. When the scheduler comes across this instruction, it first disables the SP and then scans the following instruction which contains the end address of the transactional block. This way, the scheduler knows immediately both the start and end addresses of the block, and is able to send this information to the remote PE to start execution. At that point, the PE will register an instruction miss and will request the needed instructions from the IMem of the SU. The last instruction in this transactional marker is actually the first instruction that will be executed by the remote PE, and its purpose is to load the stack pointer address in the appropriate register of the PE[1].

The END_TXN marker depicted in Table 4.2 contains a NOP instruction and a branch label. The same label is used in the branch instruction of the start marker to signify the end of the transactional block. This label will be eventually translated by the assembler to an address, pointing at the next transactional or sequential block in

[1]In this system, the register r1 of the Microblaze processor is used for this purpose. This instruction is architecture specific and should be changed in case a different processor is used

START_TXN (sequence, phase, txn_stack_size)	NOP		
	NOP		
	0xAAF87 (20bit)	Sequence (8bit)	Phase (4bit)
	BRAI (16bit)	end_address (16bit)	
	ADDIK r1, r0 (16bit)	Stack pointer (address to dedicated stack space) (16bit)	

Table 4.1: Start transaction marker

END_TXN (sequence, phase)	NOP
	end_address(T_sequence_phase) (16bit)

Table 4.2: End transaction marker

the execution sequence.

After the SP is stalled and both the start and end addresses are extracted from the markers, TMS2 registers the new transaction in the transaction table shown in Table 4.3. Every transaction is defined by a *phase* and a *sequence* number. All the transactions that belong in the same phase have different sequences and are allowed to execute in any order. On the other hand, all the transactions that belong to an earlier phase (smaller phase number) need to appear as if they executed before a later phase (larger phase number). In the original TMFab design, the sequence was also defining the execution order for the transactions of a single phase. However, in TMFv2 there is dynamic ordering of transactions at validation time, and the sequence is only used for the identification of different transactions and to provide flexibility for future development of the project. The phase information is still necessary as a way of synchronizing transactions that belong in the same TS and is the equivalent of a *barrier* in a conventional pthreaded program. The priority field, is used from the SU Output to assign the transactions to the remote PEs in the same sequence they appear in the program.

After the transaction has been registered in the table, the TMS2 scans the next instructions to identify whether they are part of a new transactional marker, signifying the start of a new transaction, or if it is plain sequential code, which means that this is the end of a TS.

The TMS2 is also responsible for managing the PE info table, depicted in Table 4.4, which contains information about the current state of every remote PE in the system. The *Validation ID(VID)* is a number assigned by the TSM2 to every transaction that is ready to validate its read-set. Originally, when a transaction is scheduled, this number is set to 0 to reveal the lack of a valid VID. When a transaction reaches the validation

	Valid	Scheduled	Phase	Sequence	Start address	End address	Priority
Txn 0							
Txn 1							

Table 4.3: Transaction table

	VID	TTP	Assign Status	PE State	Req Tkn	Ack Tkn	Address Req
PE 0							
PE 1							

Table 4.4: PE info table

stage, it requests a VID from the TSM2, who responds to this request in a first-come first-served basis. This effectively means, that the *older* transaction that finished earlier receives a smaller VID number than a *younger* one. In other words, the transactions are assigned a priority upon completion of their execution, contrary to the original design which dictated a priority at compile time.

The *Transaction Table Pointer(TTP)* field contains a pointer to an entry in the transaction table, which holds the information related to the transaction scheduled on that PE. In the actual implementation, there are two pointers for this purpose, one for the current transaction executing and one for the transaction to be scheduled on the same PE. This is needed in order to be able to overlap the commit stage of a transaction, with the assignment of a new transaction to the same PE. Since a committing transaction is impossible to abort, and a committing PE is not performing any useful operation other than updating its write-set to the L2D, it is free for a new transaction to be transfered to it at this stage. For this reason, before the PE starts committing, it informs the TMS2 that it can send a new transaction if there is one.

The *Assign status* field keeps track of whether the transactional block is actually sent, as well as if the transaction has started execution. If not, the SU Output will send an appropriate message for the execution to start.

The PE state contains one of the three possible states of a PE : *PE_IDLE*, *PE_BUSY*, *PE_COMMITTING*.

When a transaction needs to validate and commit, it sends a message to the scheduler, who in his turn informs the PE info table by updating the *Request Token* field. The values in this field can be one of the following: *IDLE*, *VALIDATE*, *COMMIT*, *COMMITTED*. When the request is handled, the *Acknowledged Token* field is updated to avoid multiple responses to the same request.

The address request is needed in case of an instruction cache miss from a remote PE.

4.1.2 Supervising Processor

In order to perform the computation of the sequential part, the SU uses a *Supervising Processor(SP)*, which in this system has a MicroBlaze architecture [2]. This processor is responsible for the execution of all the code that is not inside of the transactional markers. When the processor is enabled, it requests for the next instruction to be executed by providing the instruction's address. At this point, the Instruction Interface examines the requested instruction which resides in the local IMem as shown in Figure 4.1 and decides whether it is a START_TXN marker or not. If yes, then it disables the

[2]The original MBLite instruction set simulator provided by another developer was modified, in order to support the instruction memory interface, as well as to support floating point operations and multiplication.

U	**Bank 0**	U	**Bank 1**	U	**Bank 2**	U	**Bank 3**
0	15	1	16	0	14	1	15

Table 4.5: PVID Table

SP by reseting the enable signal and then takes all the appropriate actions to start the execution of a TS. If not, it returns the read instruction to the SP to be executed in the next cycle.

When the SP starts executing, it requests for address 0 which is the correct address to be returned. However, every time the SP resumes execution, upon completion of a TS, it requests for the instruction address corresponding to the position of the encountered START_TXN marker because the PC was pointing in this address before the SP was disabled. In this case, instead of the marker, the scheduler returns a branch operation to the correct address in the execution sequence, which corresponds to the end of the transactional section.

Since the IMem can not be directly accessed by the SP, the executed instructions are requested in advance, one cycle before they are needed, in order for the Instruction Interface to be able to return the instruction. However, in case of a branch operation the returned instruction will be incorrect. In this case the SP recognizes the mistake, stalls execution and asks again for the new instruction.

4.1.3 SU Input

The SU Input block is responsible for receiving all the messages from the remote PEs in the system, as well as the L2D banks, and perform the appropriate actions. Table 4.6 shows the different messages arriving to the SU, as well as their origin.

Whenever an L2D bank updates its Priority VID (PVID), signifying which transaction is allowed to commit on it, a message is sent to the SU. At this point, the SU Input sets an *Update* bit in the PVID table (Table 4.5), informing the SU Output that there is an updated PVID number in an L2D bank. The latter will inform the transaction whose VID matches the PVID that it has priority over that specific bank. Furthermore, in case an L1D miss has occurred, the SU Input receives the appropriate cache line from the L2D and notifies the SP Data Interface to resume execution.

When the PE communicates with the SU, it's either to request for an instruction cache line in case of an L1I miss, request for a VID or inform it about its current state. In case there is a VID request or a message that the PE is committing, the SU Input sets the Request token in table 4.4 as VALIDATE or COMMIT respectively. In case of an abort, the Request token is set to IDLE and the VID number to 0, signifying that the active transaction does not have a valid VID anymore and will have to request for a new one upon completion of the new execution cycle. The same steps as in abort are followed also when a TXN_COMMITTED message arrives, with the addition that the counter keeping track of the active transactions is decreased by one. When all the transactions have finished execution, the counter will return to 0, signifying the end of a TS.

Source	Class	Comm ID	Scheduler Op
L2D	TM Communications	Update Scheduler	Update PVID Table
	Memory Operations	L2DRead	NA
PE	Instruction Block Transfer	Instruction Request	NA
	TM Communications	Scheduler Com	VID request
			Committing
			Abort/Restart
			Txn Committed

Table 4.6: SU Input Messages

4.1.4 SU Output

The SU Output is, as the name suggests, the output of the whole block, and the connection to the interconnect network. Its purpose is to send all the necessary messages to the PEs and the L2D banks. The possible messages are shown in Table 4.7.

Before a new TS starts executing in the remote PEs, the SU *restarts* all the L2D banks. This means that every bank's PVID number is reset to 0, as well as all the LCID and SRVID fields of every cache line that were described in Chapter 3. This is necessary for the correct execution of the conflict detection scheme. In addition to that, in cases where a transaction has aborted its execution, the SU takes its place in updating the PVID number in order for the following transactions to be able to commit.

Additionally, the SU contacts the L2D in case of an L1D miss during the execution of the sequential code, in order to retrieve the appropriate cache line. In case the miss is either a capacity or a conflict one, the SU Output evicts a cache line by updating it in the L2D in order to make space for the incoming cache line. At the end of the sequential section, all the modified L1D cache lines are updated in the L2D in order to be used from the transactions if they are needed.

When it comes to the PEs, the first thing that is sent to them is the instruction code of the transaction that has been assigned to them by the TMS2. Before this is done, the SU Output checks the assign status in Table 4.4 to see whether the transaction has been already sent. If not, then all the instruction cache lines that contain the transaction of interest are sent to the appropriate PE. This is not absolutely necessary in TMFv2, but as long as the transaction size is small, there is still gain in sending the transaction at the beginning, because it saves time from unnecessary instruction misses in the L1I of the PE. All the instructions needed during execution of the transaction, that are not included in the original instruction block, are sent upon request to the remote PEs.

At this point, in order to guarantee correctness of execution of the TS, the SU Output also sends out the content of all the registers in the SP. The reasoning behind that, is that the transaction is embedded inside of the sequential code, without the compiler being aware of that. In other words, the compiler will consider that the transactional code is a continuation of the sequential one, and will use some of the preloaded registers inside of the transaction. However, when the transaction starts

Destination	Class	Comm ID	Scheduler Op
L2D	TM Communications	Update Bank PVID	Restart Count
			No Sched
	Memory Operations	L2D write	NA
		L2D read	
PE	TM Communications	Scheduler Com	Restart Count
			Assign VID
			Update PVID mask
	Instruction Block Transfer	Instruction Request	Instruction Response
		Regfile Transfer	NA
		TXN state transfer	No Sched
		TXN state transfer	Start Transaction

Table 4.7: SU Output Messages

executing inside of a remote PE, the registers are all reset, which would eventually result in wrong execution of the program. For this reason, the register values of the SP are copied inside of the registers of the PEs, residing inside of the PEs, before a transaction's execution starts.

In case that the PE, where the transaction is assigned, is in the IDLE state, the SU Output sends at this point directly a START_TXN message to signal the beginning of the new transaction's execution. This message should also contain the start and end addresses of the transactional block. However, in case that the PE is committing, the scheduler waits for a TXN_COMMITTED message to arrive before it sends a starting message to the PE. This is necessary to avoid a case in which the new transaction overwrites the SWB before it is updated in the L2D or that stale data arrive from the L2D.

When a transaction reaches the validation stage and requests for a VID, the SU Output provides the first available VID that hasn't been assigned yet, starting from 1 and reaching up to 4095, which is the highest VID that can be depicted with 12 bits. If this threshold is exceeded, there are no more VIDs to be assigned. In this case, the SU waits for all the transactions that are active and have already acquired a VID to either commit or abort. Then, for all the banks to update their PVIDs and restart since they have reached the limit of available VIDs. When this process is over, the SU Output sends restarting messages to all the active transactions in the system, forcing them to abort and restart. During the new execution cycle, the VIDs can restart from 1 thus restarting the conflict detection process. This is actually an abort not dictated by the contention management scheme and is caused by limited system resources. Nevertheless, it is an abort that will rarely happen, if the available VIDs are much more than the number of transactions in a TS.

When a bank updates its PVID and informs the SU of this change, the SU is checking all the active transactions for one that has a VID equal to that PVID. When it finds the PE in which this transaction is active, it sends a message containing which bank was updated. This message indicates to the PE that it has priority over that bank, causing it to set the corresponding bit of the PVID mask.

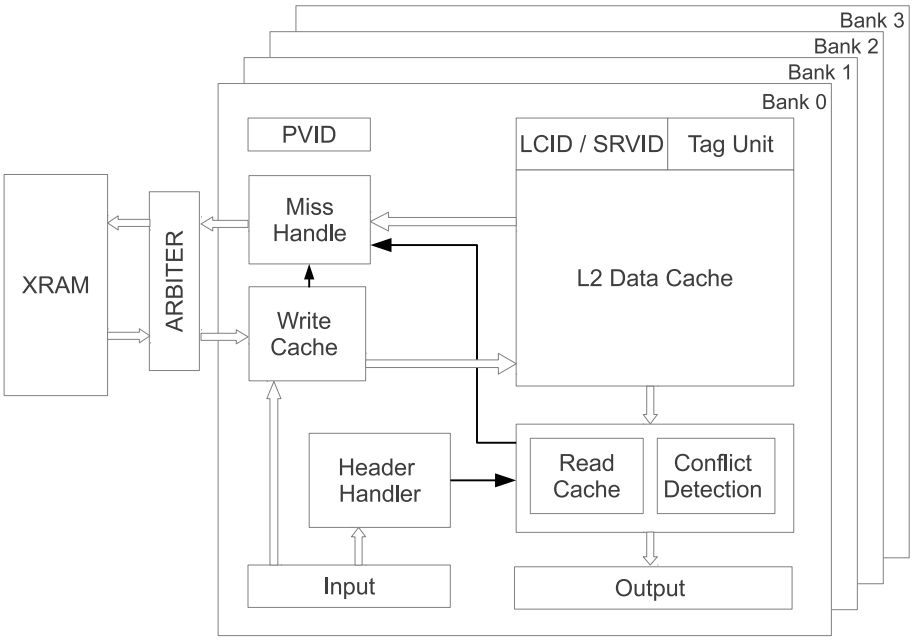

Figure 4.2: L2 Data Cache

In case that the incoming PVID doesn't have a matching VID in the PE table, this indicates that the transaction that had acquired that VID aborted, and its VID field was set either back to 0 or to another number in case the transaction requested for a new VID. Since the PEs are responsible for updating the PVIDs of all banks, this would cause for the PVIDs of all banks to be stuck in the aborted transaction's VID number, thus preventing any further commit operation to take place. In this case, the SU Output takes up the task of updating the bank's PVID, thus allowing the execution of the TS to continue.

4.2 L2 Data Cache

TMFv2 actively involves the L2D in the conflict detection scheme, as opposed to the original TMFab design where it was restricted to the L1D alone. Furthermore, considering the system's 3D stack topology, a multi dice L2D banking scheme was implemented in order to improve scalability and performance in general. For the main functionality of the cache, the architecture described in [11] was followed.

With respect to the simulation results provided in [1], an 8-way set associative cache was chosen, constituted from 8192 sets and 64 byte line size, resulting in a total size of 4MB. However, since a 4-level banked topology is currently in use, the cache was divided to 2048 sets(1MB) per bank.

The block diagram of the L2 Data Cache is illustrated in Figure 4.2. The following sections describe the functionality of every block.

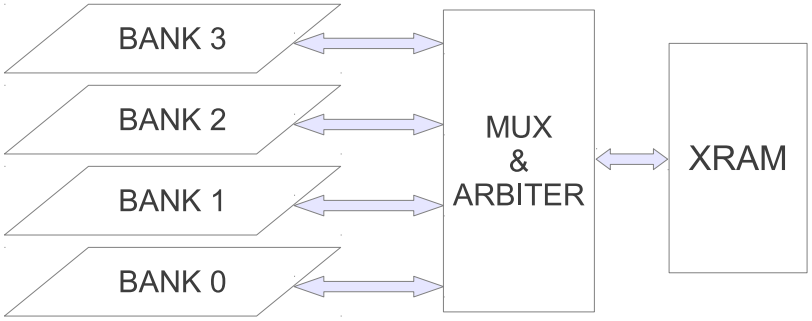

Figure 4.3: L2 Data Cache banks connection to the XRAM

4.2.1 Divide into banks

There are three main reasons for which a banked L2D approach was chosen over a single memory topology:

1. **Decrease memory access latency** In case that the L2D resides only in one layer of the stacked topology, all the PEs that are located closer to it will retrieve data faster than the ones that are located in different levels of the stack. Distributing the memory evenly on every layer results in similar access latencies, provided that there is even data distribution between the different banks.

2. **Allow for commit concurrency** According to the conflict detection scheme, in case two transactions have data sets located in completely different banks, they are allowed to commit simultaneously, taking into consideration that there are no conflicts with active transactions with higher priority.

3. **Increase communication bandwidth** By increasing the number of communication ports to the L2D (one in every level), it is possible for multiple transactions to access data in parallel, as well as validate their read-sets in parallel, thus decreasing the overall execution time.

4.2.2 Message Handler

When the messages arrive in a bank's Input block, they are inserted into a FIFO buffer from where the Message Handler unit starts servicing them. The possible messages arriving to the L2 can be seen in Table 4.8.

The SU communicates with the L2D for two reasons:

- To access data in case it is executing a sequential section. In this case, the procedure is very simple inside the L2D, since it reads and writes data without informing any other block of the unit.

- To inform the L2D to either restart the conflict detection scheme, in case that a new TS is starting, or to update the PVID number, in case the transaction responsible for this update has either aborted or committed and is not active anymore.

Source	Class	Comm ID	Scheduler Op
SU	Memory Operations	L2D Read	NA
		L2D Write	
		Program Ended	
	TM Communications	Update Bank PVID	No Sched
			Restart Count
PE	Memory Operations	L2D Read	NA
		L2D Write	
	TM Communications	Update Bank PVID	No Sched
		Validation	Read-set validation
			Validation packet sent

Table 4.8: L2 Data Cache Input Messages

In order for the pessages arriving from the PEs to be explained, first the role of the L2D in the conflict detection scheme needs to be described.

4.2.3 Conflict detection

As was already mentioned, TMFv2 actively includes the L2D in the conflict detection scheme. In order for that to be possible, the tag unit was enhanced with two more entries per cache line, the Last Commiter ID (LCID) number and the Speculative Reader Validation ID (SRVID) number. Both of these entries have a size of 12 bit, reflecting the size of the VID which is the same.

The LCID as the name implies contains the VID of the last transaction that modified this specific cache line. When a transaction is committing a cache line to the L2D, along with the address and the data, it also sends its VID embedded inside of the header flit. This way, when the cache line is updated with the incoming data, the corresponding LCID is also updated with the VID number of the committing transaction. When a transaction reads from that specific cache line at a future point, it will also access the LCID number. This number will be returned to the remote PE along with the data, and will be registered inside of the L1D. When this PE validates the cache line at a later stage, it will again transmit the formerly registered LCID along with the validating address.

Upon receival of the address/LCID information, the L2D will compare the incoming LCID to the most recently updated LCID residing in the L2D for that address. If the two LCIDs have the same value, it means that no transaction modified the cache line of interest from the time it was accessed since the time it was validated by that specific transaction. In the opposite case that the LCID is changed, it means that there was a transaction that committed in that specific cache line after the validating transaction accessed it. This fact doesn't necessarily mean that there is an actual conflict, because there is a chance that the two transactions are accessing different words in the same cache line.

At this point, it could be suggested a mask is used, marking the words that were actually modified by the last commiter, in order to be able to see if there is a conflict

at a word level granularity. However, there is no guarantee that there was only one transaction that modified that cache line, and no realistic way to keep track of all the transactions that modified it. For this reason, in order to decide whether there is an actual conflict, a value-based conflict detection mechanism is used, similar to the one used in KILO TM [7]. When there is an LCID mismatch signifying a possible conflict, the L2D sends a message back to the validating transaction, containing all the words of the cache line under examination. The PE where the transaction was executed receives the message and compares the data with the ones residing inside of its L1D, in order to decide whether there is an actual conflict, which would cause it to abort and restart the transaction.

Apart from the possible conflicts, the idea of hazard tracking was also followed [7]. This is where the SRVID field becomes useful. As was already described in Chapter 3, TMFv2 allows for multiple transactions to validate their read-sets in parallel independent of their VID number. However, they are still obliged to maintain the sequential order dictated by their VID. The SRVID field is needed in order to be able to keep track of these validations, and be able to detect possible conflicts (hazards). This number corresponds to the VID of the oldest speculative reader of a cache line.

When a transaction is validating a cache line, the L2D first checks the LCID number as was explained. If the LCID numbers are matching, i.e the cache line was not updated, the next step is to check the SRVID of that cache line.

If the SRVID is greater or equal to the PVID of the cache bank, it means that the transaction that speculatively read this cache line is still active, either validating, committing or has aborted without the L2D being aware of this fact. When this is the case, the L2D compares the incoming VID (of the validating transaction) with the SRVID. In case the first is smaller, it means that the incoming transaction has higher priority over the one that has already validated this cache line and the SRVID is updated with the incoming VID. When this happens, it is important for the transaction that already validated this cache line, to be informed that there is a hazard and it needs to revalidate the same cache line before it commits. This way, it is going to be certain that in case this cache line is modified at a later stage, the transactions that include it in their read-set will become aware of this fact when they re-validate.

On the other hand, if the incoming transaction has lower priority over the registered one (meaning a higher PVID than the registered SRVID) then the L2D is responding with a hazard message to this transaction instead. In case that the SRVID is smaller than the bank's PVID, there is no need for a hazard message, since the transaction registered in the SRVID field has either committed or aborted and is not active anymore, which implies that the SRVID can be simply replaced without further implications.

In order to be able to send hazard messages to the appropriate transactions, there is need to keep track of the VID of the transactions that executed in every processor. For this purpose, a *VID table* was used, with entry size equal to the number of PEs in the system. Every entry contains the VID of the transaction that is active in the corresponding PE as shown in table 4.9. Whenever there is a hazard detected, after identifying the VID of the transaction that has to be notified, the VID is looked up in the VID table. If the number is found there, then the L2D knows where to send the hazard message, while if it's not, it means that the transaction has already aborted and

	TXN VID
PE 0	100
PE 1	101
PE 2	104
PE 3	103

Table 4.9: VID Table

Valid	Modified	LRU tag	LCID	SRVID	Tag	Data

Table 4.10: L2 Data Cache line

has a new VID . This can be observed in the third entry of Table 4.9 where the VID number is 104 instead of 102 as would be expected. In the event that a transactions has aborted but hasn't acquired a new VID, the hazards from the previous VID will keep arriving, however they will not have any effect because the transaction is still in the EXECUTE stage.

In case there is a cache line eviction due to a capacity or a conflict miss, the LCID and SRVID information of the evicted cache line are going to be erased completely. If the transaction registered in the SRVID is still active, it will be informed for a hazard in this address. If there was a transaction that read this cache line and hasn't validated it yet, this will create a problem since during validation there will be no LCID to compare the original LCID value with. When the address of the evicted cache line will arrive for validation, the cache line will be retrieved from the external RAM (XRAM) and its LCID and SRVID fields will be set to 0. This implies that there is no information about which was the last transaction to update this cache line or which speculatively read from it. In this case, the cache line data will be returned to the validating transaction for their value to be compared, the same as if a potential conflict had occurred. Furthermore, the incoming VID would become the new SRVID of the cache line since this is the oldest transaction at this point that speculatively read from this cache line. The LCID will remain 0 until the next commit operation that will be performed on this cache line. The above process guarantees that in case of cache line eviction, only the performance will be affected and not the correctness of execution.

4.2.4 Cache access

In order for the L2 to be accessed, there are two blocks, one for writing to the cache and one that reads from it. The former is responsible for getting the data from the Input block, in case of a commit operation, and register them in the L2D, together with the VID of the commiter which is register in the LCID field. In case either the read block or the write block encounter a cache miss, they inform the miss handle block, which, in case of an eviction, updates the evicted cache line to the XRAM implementing thus a write back memory update policy and notifies the cache line of the requested memory address. After that, the XRAM returns the cache line containing this address back to the L2. Since there are multiple cache banks accessing the same XRAM through a single port, there is need for an arbiter who, according to a round-robin policy scheme,

Destination	Class	Comm ID	Scheduler Op
SU	Memory Operations	L2D Read	NA
	TM Communications	Update Scheduler	No Sched
PE	Memory Operations	L2D Read	NA
	TM Communications	Validation Response	Hazard Detected
			Conflict Detected
			Validation Done

Table 4.11: L2 Data Cache Output Messages

gives XRAM access privileges to one bank at a time.

Least Recently Used (LRU) was chosen as the cache line replacement policy, in case that a capacity or conflict miss occurs. In order for this policy to be implemented, there is an additional number tagged on each cache line ranging from 0 to N-1, where N is the cache's associativity [3]. For this number to be stored, the tag size has to be $\log_2(associativity)$ bits long, which means that in an 8-way associative cache there is need for 3 bits per cache line to keep track of which way is used less often in a set. In case this imposes a large storage overhead, or increases the hardware complexity, thus reducing the achieved clock frequency, a cache with lower associativity could be used or a pseudo LRU scheme scheme implemented, potentially without important performance deterioration.

The read cache/conflict detection block is responsible for sending the output messages to the PEs through the NoC as well as the SU. The possible messages are presented in Table 4.8, where the three main categories can be seen:

1. **Response to a cache line request** In this case, the requested cache line's data are returned to the either a PE or the SU. In case of the first,the LCID of the cache line is sent alongside the data, in accordance to the conflict detection scheme.

2. **Update message to the scheduler** In case the PVID of a bank is updated either from a committing transaction, or the scheduler itself, the updated bank sends a message to the TMFv2 scheduler to confirm this update and inform the PVID table located inside of the SU.

3. **Respond to a validation request** When a transaction is validating, there are three possible responses from the accessed banks. The one is to notify a hazard with the validated cache line, the other to notify a possible conflict and the last one to signify the end of the validation process. In case of a possible conflict, the data are returned as was already explained, in order for their value to be examined and to be concluded whether there is an actual conflict.

[3]When a cache line is accessed, its LRU tag is set to N-1, and all the other tags that originally had a higher value, are decreased by 1. The cache line whose tag is 0, is the one that is evicted in case there is a capacity or a conflict cache miss

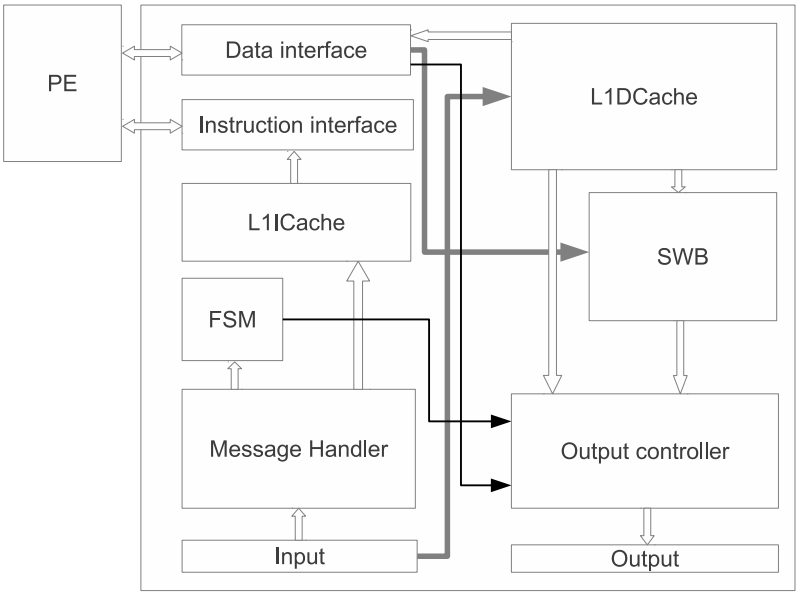

Figure 4.4: TM Processing Unit

4.3 TM Processing Element

At this point, the Transactional Memory Processing Element (TMPE) will be described. This is the unit that is responsible for executing the transactional code provided by the TMFv2 Scheduler. The TMPE comprises of four main parts: the Processor Element (PE), the TMPE State Machine , the internal memory and the IO units. Figure 4.4 illustrates the internal structure of the TMPE. From now onwards the TMPE will be just referred to as PE for simplicity reasons.

4.3.1 PE State Machine

In order for the functionality of the PE to be better described, one of the most important parts of the design, the FSM, will be analyzed. Figure 4.5 illustrates the eight possible states in which the PE can be found: *Idle*, *Execute*, *Wait VID*, *Read-set Validation*, *Wait Response*, *Wait PVID*, *Abort*, *Commit*.

1. **Idle State**

   When the state of the PE is Idle, this means that there is no active transaction executing, and the block is ready to service any incoming transaction provided by the SU. In order for the PE to leave this state, the SU has to send a state transfer message which signals the beginning of execution and provides the necessary initialization information. When this message is received, the state changes to Execute.

2. **Execution**

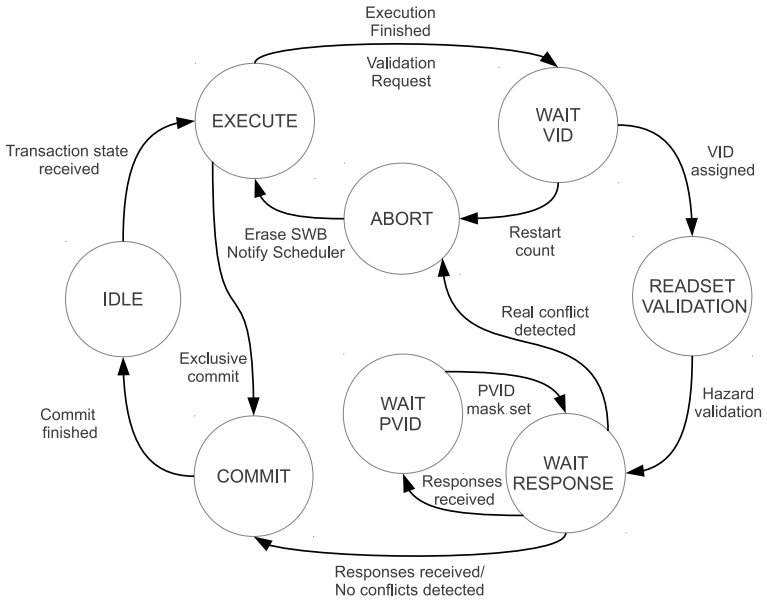

Figure 4.5: PE State Machine

During execution, the PE enables the PE to start executing the assigned transaction's instruction code. During execution, the PE keeps track of the instruction addresses requested from the PE and when it encounters the end address it disables the PE and signals the end of execution. After that, the FSM sends a validation request message to the SU in order to acquire a Validation ID (VID) and then goes to the Wait VID state to wait for it.

One of the advantages of TMFv2 over the original TMFab design, is that during the execution state there is no way for an abort to take place. The abort occurs only as a result of the validation process, and can not be forced by the validation of another transaction as was the case in the previous design, thus simplifying the design's complexity. On the other hand, because of this scheme, conflicting transactions are delayed in aborting which increases the wasted execution time. However, this also protects transactions from unnecessary aborts in case that older conflicting transactions don't manage to commit.

In case of cache overflow the execution needs to be split into two parts for sequential consistence to be guaranteed. There are two kinds of overflow that can occur during execution, the *L1D overflow* and the *SWB overflow*. In the first case, the L1D has to evict a cache line which has already been speculatively read and contains transactional information, while in the second case the SWB is full with speculative writes and there is no space for additional modified cache lines to be stored. In both cases, the execution needs to be stalled and data to be evicted to make space for new data. This will cause for transactional information to be lost, and consequently for the evicted cache lines to be excluded from future validation. For this reason, it needs to be certain that the overflowing transaction

is going to definitely commit in the end.

In order for this to be done, first the transaction is stalled and then the validation process described in the coming paragraphs is followed, until the transaction has exclusive commit rights in the system, i.e. no other transaction can modify the read-set of the transaction under consideration before it commits. The transaction can then resume execution and evict any cache line it needs, without the danger of any conflict being overseen. Upon completion of execution, the PE's state will go directly to Commit since there is no need for validation.

3. **Wait VID**

   As was already mentioned, when a transaction's execution phase is complete, the PE has to validate its read-set in the L2D. In order for this to be done, the PE requests and waits for a Validation ID (VID) to arrive from the SU, which will also define the transaction's priority in comparison to the other active transactions. If the SU has available VIDs to provide to the PE, a message will be returned containing the assigned VID and the PE will go over to the Read-set Validation state.

   In case there is a shortage of VIDs, because either there are too many transactions inside of a TS, or because too many aborts have occurred[4]. In the current design where the VID is 12bit long, the maximum number of VIDs that can be assigned is 4095[5] which means that it is quite unlikely that the VID limit will be reached. However, in case it does so, the SU can not assign any more VIDs and has to wait for all the validating and committing transactions to finish before it restarts the counter. This action will cause any transactions that have not yet acquired a VID to abort and restart, while their L1D cache lines are flushed, and all the LCID and SRVID fields of the L2D banks are reset. In this case, the next state will be Abort as depicted in Figure 4.5.

4. **Read-set Validation**

   In the most common case that the PE acquires a VID, it will start validating its read-set with the L2D. When all the validation packets have been sent, the PE goes to the Wait Response state.

5. **Wait Response**

   In this state, the PE waits for all the responses from the different banks that have been accessed. This happens twice during a transactional cycle, once after the read-set is sent for validation, and once after the hazards are sent. When all the responses have arrived, the Input Handler notifies the FSM about this event. In case there is a conflict detected, the next state will be Abort. If there is no conflict, in case of a read-set validation the next state will be Wait PVID, while in case of a hazard validation it will be the Commit state.

6. **Wait PVID**

[4] after every abort a new VID has to be assigned for the transaction that aborted in order to validate anew
[5] 4096 -1 because 0 depicts the lack of a VID

When the PE is in this state, it means that it has finished execution, has validated successfully its read-set and has received responses from all the L2D banks in the system. The only thing remaining to be done is to wait for the PVID mask to be set. This means that the PE will get priority over all the banks that it has accessed during execution. In other words, all the older transactions have either finished committing in those banks, or they don't have them in their read-set/write-set. After the PVID mask is set, all the hazardous lines are sent for re-validation to the L2D and the state changes once more to Wait Response.

7. **Abort**

   In case that there is an actual conflict between the transaction running on a PE and an older transaction that has already committed, the PE forces the transaction to abort and restart. Before this is done, it must first reset all the read and modified bits in the L1D as well as invalidate all the hazardous lines that have appeared so far in the validation process. This is not necessary to guarantee correctness, because the LCID will be checked anyway in the second execution cycle and any potential conflicts will be detected. However, the hazardous lines are more likely to actually by modified by an older transaction, in which case the aborting transaction will need the new data for the second execution cycle. If the hazardous cache lines are no invalidated, stale copies will be use leading to further aborts which could have been avoided.

8. **Commit**

   The last stage of a complete execution cycle is the Commit state. At this point the PE commits its write-set to the L2D, erases the whole L1D for the reasons described in Chapter 3, and resets the system to its initial values (e.g. Read/Write masks, PVID mask, SWB pointer). After these operations the system goes back to the Idle state from where it is able to start executing a new transaction.

4.3.2 Internal Memory

The internal memory of the PE is divided into two parts in order to implement the lazy version management policy. This means that the speculative writes of a transaction are kept in a Speculative Write Buffer (SWB) instead of being directly updated in the L1D. This way, in case of abort it is much easier to just erase the SWB and keep the L1D intact, except for the hazardous cache lines that are invalidated.

4.3.2.1 L1 Data Cache

For the L1D, a 4-way set associative topology was chosen as proposed in [1]. However, higher associativity could prove beneficial in reducing the overflow rate of the system. This is something that wasn't examined in the current thesis. Nevertheless, smaller associativity results in lower hardware complexity and better performance.

In order to implement the validation protocol, the standard cache architecture proposed by Hennessy and Patterson [11] was enhanced with some more fields shown in table 4.12. The R and M bits are needed to mark a cache line as Read and Modified

1bit	1bit	1bit	1bit	12bit	18bit	16words	16bits	9bits
V	R	M	RH	LCID	Tag	Data	Read Mask	Rel Addr

Table 4.12: L1 Data Cache line

respectively. When a cache line is marked as Read, it means that it has been speculatively read from the active transaction executing on the PE. Depending on which word of the cache line has been read, an additional Read Mask bit is set. This mask is comprised of 16 bits, one for each word of the cache line, and is needed during the validation process.

As we described earlier, when there is a possible conflict detected in a cache line of the L2D, the whole cache line is returned back to the validating transaction to be examined by value. The validating PE receives this cache line and starts comparing all the words, with the ones registered in its local copy of the same cache line. If there is a mismatch between two words, then the L1D is updated with the fresh incoming data. In case, the modified word has been speculatively read by the current transaction, i.e. the mask bit for that word is set, it means that there is an actual conflict and the transaction needs to abort and restart. Without this mask, conflicts wouldn't be able to be detected in a word level granularity, leading to unnecessary aborts.

When the PE needs to modify a word in a cache line, it first goes to the L1D and checks if the modified bit has been set. This would mean that the cache line of interest has been already speculatively modified and the latest data version is located inside of the SWB. If not, then the whole cache line is copied in the first available entry of the SWB, and its address is returned to the L1D to be registered in the Relocation Address (Rel Addr) field of the original cache line. Afterwards, the data coming from the PE are written on the appropriate word of the SWB cache line and the corresponding bit of the Write Mask field is set. Any consecutive accesses to the modified cache line will be diverted from the L1DCache to the SWB using the Relocation Address.

When a cache line has been either speculatively read or written, it contains transactional information and can not be evicted from the memory. In case there is a shortage in memory and an eviction is necessary to continue execution, then exclusive commit rights are needed in all the L2D banks of the system for the execution to resume.

In order for the proposed validation scheme to be implemented, there is need for two more fields per cache line, the Last Commiter ID(LCID) and the Read Hazard(RH) fields. The first one contains the VID of the last transaction that had modified that specific cache line before it was accessed by the executing transaction. The RH bit is necessary to keep track of the hazardous cache lines.

Unlike the L2D, the L1D does not have a cache line replacement policy, because the chance of eviction is very small, and all the cache lines are erased in any case after a transaction's execution. Flushing the cache should not affect the system's performance considerably, since in most cases different transactions will work on different data sets on the same PE.

4.3.2.2 Speculative Write Buffer (SWB)

The SWB is the place where all the speculative writes are stored. Similarly to [1], this memory contains 512 cache lines, having thus half the size of the L1D. This decision was made under the assumption that that the read-set and write-set of a transaction are of approximately the same size. However this depends greatly on the nature of the benchmarks, so it is possible that this ratio should be changed to serve a larger range of applications.

As was already mentioned, when a cache line is modified for the first time, the original data are copied from the L1D to the SWB in the first available entry. This entry is defined by the SWB pointer which always shows at the next available entry of the SWB. The advantages of using such a pointer are two:

1. It is easy to erase the SWB by just reseting this pointer to 0. This is necessary because actually erasing all the cache lines would take many cycles.

2. This way, the cache lines are written in consecutive spaces in memory, thus making it easier and faster to commit them to the L2D at the Commit stage.

It could be argued that for the above reason, the Valid (V) bit is not necessary, since whatever is located between the first location of the SWB array and the SWB pointer is considered valid. However, this is not the case when an overflow has occurred. If a modified cache line needs to be evicted from the L1D, the PE needs to update the modified data to the L2D after acquiring exclusive commit rights. In this case, the location in the SWB needs to be invalidated in order for the line not to be committed again at the Commit stage, since those data could be stale at that point.

The Tag and Index fields actually define the cache line's address, while the Write Mask reveals which words in a cache line have been modified. These are needed during the Commit operation, where the address of the cache line to be updated is sent to the L2D together with the Write Mask and the modified data.

At this point, it is important to explain a case in which the system could update the L2D with wrong data. Since the write mask refers to the words inside of a cache line and not the bytes, there is no way to define whether a single byte inside of the word was modified or the whole word. In case two transactions are modifying two adjacent bytes inside of the same word, they will both show this word in their write mask. However, if this word is not also in the read-set, there will be no conflict detected between the two transactions, because the Write-after-Write conflict is not affecting the execution of a transaction and thus is not considered as a conflict. The result will be that the two transactions will commit the same word with three stale bits and one updated, thus providing the wrong word to the memory, no matter in what order they commit.In other words, if the original word was 0xffff, T1 will write 0xfff1, T2 will write 0xff2f and the result will be one of the two, instead of 0xff21 as it should be. The best way to avoid this problem would be through the compiler, so that two different transactions can't access different bytes of the same word.

1bit	18bit	8bit	16bit	16Words
V	Tag	Index	Write Mask	Data

Table 4.13: SWB Cache line

4.3.3 TMPE Output

The TMPE Output unit is accessed by the different units of the system whenever they need to communicate with the SU or the L2D through the network interconnect. There are eight possible reasons for this unit to be accessed, which are presented in the following paragraphs.

1. **Read Instruction Memory** When the instruction interface encounters an instruction miss, it sends a message to the SU that contains the whole executable code in order to retrieve the appropriate instruction cache line.

2. **Read Data Memory** When there is a data miss registered by the data interface, there is a message sent to the L2D in order to retrieve the cache line containing the requested address.

3. **Write Data Memory** In general, during execution there is no need to write an individual cache line back to the L2D so this choice is not necessary in most cases. However, it is needed in case of an L1D overflow where there is need for only the evicted cache line to be updated in the shared L2D in case it has been previously modified.

4. **Validation Token Request** When a transaction reaches the end of execution it sends a VID request message to the SU in order to acquire a Validation ID

5. **Update Bank PVID** Every transaction that has a valid VID, will get priority in all of the L2D banks at some point. If the transaction is still active at that point, it needs to update the banks' PVIDs after it is done accessing them, either for hazard validation or commit operation. If a bank hasn't been accessed during execution it can directly be updated from the PE without any validation or commit operation first taking place. This way, transactions with datasets residing in different banks are not stalled and can directly commit in parallel, thus reducing the validation/commit overhead.

6. **Validate** When a transaction reaches the end of execution and acquires a VID, it has to start validating its read-set. This means that all the read cache line addresses will be sent to the L2D, together with the LCID information which was registered at the time that the cache line was first accessed.

   In the original TMFab design, every transaction would send its write-set to every other active transaction in the system, in order to be compared with the local speculative read-set and write-set and detect any potential conflicts. Since the write-set is located in consecutive entries in the SWB, it was easy for the validation packets to be formed and sent out.

	Index
Read line 1	
Read line 2	

Table 4.14: Read Table

On the other hand, in TMFv2 the read-set is sent to the L2D for validation instead of the write-set. Since the read cache lines are randomly distributed inside of the L1D, in order for the read-set addresses to be retrieved and gathered into packets, the whole L1D would need to be scanned. In order for this unnecessary overhead to be avoided, a Read Table depicted in figure 4.14 was used. Whenever a cache line is marked as read, its index is stored inside of this table and the read pointer is incremented similarly to the SWB pointer. This way, when the read-set needs to be validated, this buffer is scanned and one index is retrieved per cycle. This index is afterwards used in order for the actual address that was read to be retrieved from the L1D. However, since there is no information about which of the ways in a cache line have been read, the R bits in all of them need to be scanned in order to retrieve all the addresses that need to be validated.

The advantage of this methodology is that, one address can be sent for validation in every cycle, with very small area overhead for the implementation of the Read Table which needs to have entries equal to the number of sets in the L1DCache (which is 256 in our case).

One additional problem encountered in the new design is that since the data are located now in different banks, separate messages need to be sent to every bank according to where the validating addresses belong to. However, since data from different banks can be interleaved during execution, every few cache lines, the destination bank will be changing thus forcing the system to start a new validation message with the overhead of a new headerflit. Nevertheless, the smaller validation packets lead to better resource distribution since more processors can validate the L2D in parallel.

At the end of the validation, since there is no prior knowledge of which address is the last one in the read-set, an additional empty message is needed to notify the banks that the validation from this PE has finished. Upon receival of this message from the banks, they will notify the PE with a return message that all the addresses have been taken into consideration in the validation process.

When it comes to the hazard validation process, the steps are similar. The only difference is that in this case, a Hazard Table is used to keep track of where the hazardous cache lines are located inside of the L1D. The hazard table contains two fields instead of one, keeping also the information of the way in which the cache line is located. This increases the speed in which the cache line address can be found and does not pose a significant area overhead because the hazard table is considered generally small[6]. Nevertheless, the case in which the hazard

[6]Around 64 lines in the current system. However more simulations are needed to conclude what is an appropriate size for the hazard table

	Index	Way
Hazard 1		
Hazard 2		

Table 4.15: Hazard Table

Destination	Class	Comm ID	Scheduler Op
SU	Instr Blk Transfer	Instr Request	No Sched
	TM Communications	Scheduler Com	Validation Tkn Req
			Abort/Restart
L2 Data Cache	TM Communications	Update Bank PVID	No Sched
		Validation	Read-set Validation
	Memory Operations	L2D Read	No Sched
		L2D Write	

Table 4.16: PE Output Messages

table is not big enough to accommodate all the hazards needs to be taken into consideration. For this reason, when a hazard occurs, instead of just registering it in the Hazard Table, an additional Read Hazard(RH) bit is set in the cache line. In case that the table is overflown, all of the L1D is scanned for possible hazards for re-validation. In this case, it is possible that simple re-validation of the read-set would result in reduced validation overhead.

7. **Abort** In case of an abort, the only thing sent by the Output unit, is a message to the SU to notify it of this event and register it in the PE Table located there.

8. **Commit** The last thing that the PE Output unit is responsible of, is sending the commit messages upon successful validation of the read-set. When the transaction gets priority over the banks that it has speculatively accessed, and assuming that there are no conflicts detected, it starts sending all the cache lines registered in the SWB buffer to the L2D banks, together with its VID,in order to be registered in the LCID field of the cache lines that will be modified. This operation is straight forward since all the cache lines are saved in consecutive locations inside of the SWB. The only that needs to be taken into account is whether the cache lines are valid[7].

4.3.4 Input Handler

The Input Handler is responsible for receiving the messages coming from the L2D and service them accordingly. Before a transaction starts executing, the transactional block needs to be received and saved inside of the L1 Instruction Cache located inside of the PE. Even then, the transaction is not allowed to start executing before the start message is received, containing information about the phase and sequence of the transaction, as well as its start and end address. These informations are saved in special registers

[7]meaning that they weren't updated earlier to memory due to an overflow as explained in subsection 4.3.2.2

Source	Class	Comm ID	Scheduler Op
SU	TM Communications	Scheduler Com	Restart Count
			Assign VID
			Update PVID mask
	Instruction Block Transfer	Instruction Request	Instruction Response
		Regfile Transfer	NA
		TXN state transfer	No Sched
		TXN state transfer	Start Transaction
L2D Cache	Memory Operations	L2D Read	NA
	TM Communications	Validation Response	Hazard Detected
			Conflict Detected
			Validation Done

Table 4.17: PE Input Messages

which will be needed for the control of the transaction's execution. However, as was already mentioned in subsection 4.1.4, before execution starts, the register file of the SP needs to be transfered and copied inside of the PE of the PE in order to guarantee correctness of execution.

After the transaction's execution, the PE sends a request for a VID, which results in a return message with the assigned VID, or a restart signal in case there are no more VIDs to be assigned. When the VID returns, the Input Handler notifies the state machine to resume execution and start validating the read-set.

During validation, there are three possible messages returning from the L2D, one to signal a hazard, one to signal a possible conflict and one to mark the end of the validation process. In case of a possible conflict, the incoming cache line is compared with the local cache line to decide whether the conflict is actual, while the hazards are just registered inside of the Hazard Table for re-validation at a later stage. When the *done messages* from all the banks have arrived, the Input Handler gives a signal to the state machine to go to the Wait PVID state and wait until the PVID mask is set.

The PVID mask is considered to be set when the transaction gets priority over all the banks it has accessed. In order to be able to understand that, during execution the PE keeps track of the banks that are speculatively read by setting the bits of a Read Mask(RM) and the banks that are speculatively modified, using a Write Mask(WM) [8]. When the logic OR of these two masks coincide with the PVID mask, the transaction has priority over the banks that it has speculatively accessed and is able to re-validate the hazardous lines and then commit if there is no conflict detected.

In order for the PVID mask to be set, the SU sends a message to update it according to the changes in the its PVID table, as explained in subsection 4.1.4.

The rest of the messages concern incoming data or instruction cache lines coming from the L2D or the SU respectfully.

[8] every bit of this masks corresponds to a bank in the L2D

4.3.5 Processing Element (PE)

The processing element is responsible for the execution of the transactional code. In the current design a MicroBlaze processor architecture was used, similar to the one used as a Supervising Processor in the SU. The only difference in this case is that whenever the processor is restarted, either because of an abort operation or because of a new transaction starting, at the moment address 0 is requested, a branch instruction to the start address of the active transaction is provided instead.

Furthermore, one of the basic differences between the original TMFab design and TMFv2, is that the latter has a 2-way set associative L1I implemented inside of the PE, instead of a simple Instruction memory. This decision helps in three ways:

1. The presence of an instruction cache lifts any previous limitations on the size of the transaction's instruction code, since cache lines can be evicted and replaced in case of an instruction miss.

2. Furthermore, the instruction memory space is common for all the PEs in the system[9].

3. It is possible to fetch instructions from the IMem of the SU upon request and not preload them before the transaction starts, as was the case before. This means, that if two transactions that use the same functions run on the same PE, the instructions will be already there the second time, thus reducing the overhead caused by fetching the instruction code of the function.

4.4 Network on Chip

An important aspect of every multi-core system is the interconnect between the different functional units. Finding a suitable interconnect is a cumbersome task in every design, because all the possible options need to be weighted to find a network that best fits the system's needs with respect to throughput, scalability, complexity among other characteristics. In this design, the NoC architecture described in [12][1] was used without any modifications on it, since it was out of the current thesis' scope. Furthermore, the proposed NoC still fits the needs of the TMFv2 design. In this section the architecture of the NoC will be described, without emphasizing on the reasoning behind the choices that were made. For more detailed explanation, the reader is referred to the TMFab original design [1].

4.4.1 3D Mesh Topology

One of the most important attributes of a multi-core system is its scalability. Over the last years, the integration technologies have allowed for the number of transistors that fit in a single die to increase, as Moore's law predicted. Furthermore, the upcoming 3D stacking technologies allow the system to be expanded also in the vertical direction by

[9]There is no need for the memories used in the program to be translated into physical addresses used in the IMem

placing the functional units the one on top of the other in different dice inside of the same chip. These two facts contribute in having the ability to have hundreds of PEs on the same chip.

Since TMFv2 considers a 3D stacked topology, there is need for an interconnect suitable for this environment. A 3D mesh topology was preferred over a bus or a crossbar based interconnect for the following reasons:

1. A bus based interconnect has limited scalability, because when the number of processors increases, the arbitration time needed to decide which unit will take hold of the bus also increases. Furthermore, the use of a bus doesn't permit parallel network accesses from different blocks in the system, thus decreasing task parallelization, in case that the different units use different resources.

2. A crossbar based interconnect where all the units are connected with each other through a central switch, although it offers increased scalability, it is more difficult to be implemented efficiently for a 3D interconnect. Furthermore, the complexity of the network increases with the addition of extra components in the system.

3. The 3D Mesh topology is highly structured, which simplifies place and route, as well as timing closure and is easy to expand with simply reusing more of the network routers, without needed to modify any of the structure blocks.

4.4.2 Message Passing

The communication between the different functional units of the systems is done through the transmission of packets. Every packet consists of a header flit, a maximum of 16 body flits and a tail flit as shown in figure 4.6. Those packets are routed through the NoC using *wormhole routing*. This means that the flits are routed through the network whenever the resources are available, without waiting for the whole packet to be stored inside of a node before it is forwarded. The destination of the packet is located inside of the header flit which sets the path to be followed by the rest of the flits.

The maximum size of a packet is 18 flits long and is used in case of a cache line transfer: 1 header flit, 1 flit for the address and 16 flits that contain the words of a cache line[10].

The minimum packet size in 1 flit long. In case a single flit message is sent, both the MSB and the LSB of the flit have to be set to signify that this is both a header and a tail flit. In all other cases, the header flit has the MSB set and the tailflit has the LSB set as shown in figure 4.6.

The packets are routed through the network according the the value in the destination field of the header flit. The other fields are used to define the source of the packet and its nature. The Communication ID and the Scheduler Operation used for every different message are included in the Input/Output tables included in the previous sections.

[10]the cache line size both for the instruction and the data cache is 16 words

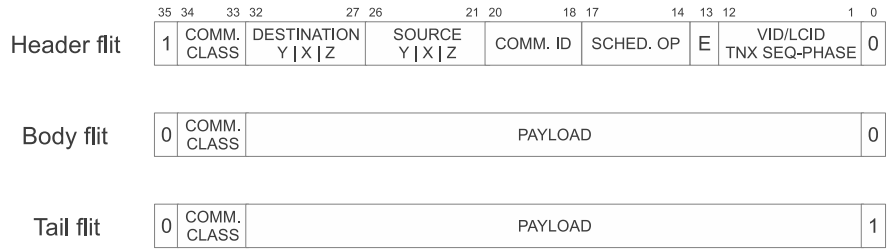

Figure 4.6: Packet Structure

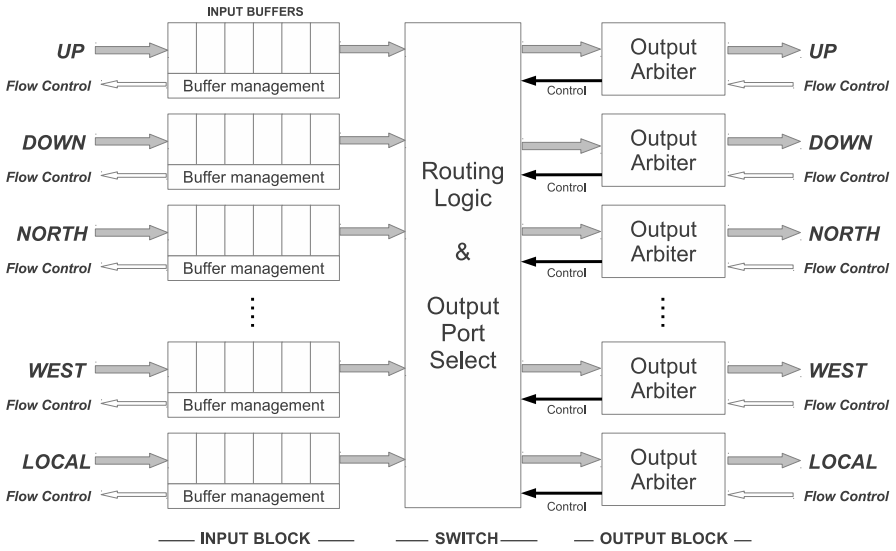

Figure 4.7: NoC router

The field originally used to transfer the sequence-phase of a transaction, now is used to transfer the VID of a validating/committing transaction, as well as the LCID of a cache line when the L2D returns it to a PE either because there was an L1D cache miss, or because there is need to resolve a potential conflict inside of the PE itself. In the future, it is possible that there will be need to transfer the phase - sequence of a transaction as well.

4.4.3 Router

Every node of the 3D Mesh is a 7-bidirectional port router. One port of every router is dedicated for the communication with the local functional unit (SU, L2D Bank or PE in the current system), while the others are needed for the communication with the adjacent nodes in the network. There are 6 directions apart from the local one, North, South, East, West, Up and Down, each of which have an input port and an output port. Furthermore, every router has a hardwired network address which is dependent on its position in the 3D Mesh.

When a flit comes through an input port, it is first saved inside of the corresponding

input buffer as shown in figure 4.7, before it is processed. If this flit is a header flit, the routing logic unit extracts the destination field from it and compares it to the routers hardwired network address. According to the difference between the two, the router decides what is the appropriate output port for this packet. The packets are routed always with priority to the z axis, then the y and then the x, which implies that there is only one path connecting two different units in the system. It is important to mention that a packet can't go out from the same direction they came in, thus preventing the occurrence of deadlock.

When the output port for a specific packet is decided, the packet waits until it gets priority over that port. This is decided from a dedicated arbiter located at every output which is responsible of providing fair access to the output port from all the input buffers except from the one that is located in the same direction. This is done following a Round Robin arbitration scheme.

In case that the input buffers get full, a backwards flow control signal is set in order to notify the previous nodes to stall the packet's forwarding, in order to avoid a case that flits get lost and packets corrupted and in need to be re-transmitted.

4.5 Summary

The current chapter gives a detailed description of the architecture of TMFv2, and explains all the choices that were made during the system's development. In Chapter 5, the results from the simulated benchmarks are presented, and the performance improvements of the modified design over the original are highlighted.

5 Results

This chapter, evaluates the performance of TMFv2 in comparison to the original TMFab design. First, a brief description of the simulation environment will be given, while afterwards the benchmarks that were chosen for the characterization of those systems are going to be explained. Further, the experimental results will be analyzed and the advantages and shortcomings of each design will be identified.

5.1 SystemC Simulators

In order for the performance, of both the original TMFab and the modified TMFv2 design, to be evaluated, two simulators were built using the SystemC programing language. This language is actually a superset of C++ enriched with classes and methods to simulate the actual hardware's behavior. One of the most important advantages of using such a language in comparison to VHDL is that it can be used in order to simulate hardware together with software. Furthermore, it offers flexibility as far as memory handling is concerned, communication between different modules and general architectural exploration.

At the beginning, the original TMFab design was implemented, and scaled up to 16 PEs instead of 4 that were used in [1]. Using the same benchmarks, it was observed that the speedup was flatlining at approximately 4x even after using more than 4 PEs. The reasons for that were the following three:

1. The validation overhead increased dramatically when the number of processors increased, because every active transaction had to validate with every other active transactions in the system, causing a dramatic increase in the number of validation packets.

2. Every transaction had to acquire exclusive validation and commit rights in the system, meaning than no other transaction could validate and commit its write set at the same time, causing for long stalling periods in every PE.

3. When the system was tested with multiple transactions being able to validate simultaneously, the NoC was congested from the increased network traffic resulting again in a performance flatlining.

In order for the above problems to be resolved, a new validation scheme was envisaged, which allowed for reduced validation overhead, improved scalability and better exploitation of the 3D architecture, leading to the creation of the TMFv2 design.

In both designs, the system was scaled up to 16 processors, in order for the simulation time to remain reasonable. However, this increase is indicative of the system's

performance and scalability and outlines a lot of the advantages and possible problems in both designs.

5.2 Choosing a benchmark

In order to compare the performance of the two designs with each other, as well as with other transactional memory systems, several transactional memory benchmarks were investigated. However, since both systems deviate from the conventional transactional memory practices, it was difficult to find an appropriate benchmark to suit the systems' needs.

One of the most common benchmarks used to characterize transactional memory systems' performance is the STAMP benchmark suite [13]. This package contains applications with different transactional length sizes, Read/Write set sizes, time spend executing transactions and contention magnitude. These applications are meant to cover a big range of realistic applications with different needs.

The problem with using these applications was that the transactions are integrated inside of the threaded code, in the places that normally conventional locks would be used. However, neither TMFab nor TMFv2 have support for threads, because the concept is that the transactions are spawn from the scheduler, and sent to the remote PEs to be executed individually, without being part of a thread in which they need to return control to.

In order for these benchmarks to be tailored for these systems, the threads had to be converted into transactions. This action lead to the transactional part becoming too big, which resulted in a lot of time being wasted in case of an abort operation for any benefit to be observed. This fact lead to the examination of benchmarks which use coarse grained transactions. In this case, a large part of the thread's execution was spent inside of the transaction, meaning that converting a thread into a transaction wouldn't affect the performance significantly. The STAMP benchmark suite includes some applications with this characteristics, however converting them wasn't that straight forward and presented many problems.

For this reason, another set of benchmarks, the RMS-TM Benchmark Suite [14], was also investigated. In this benchmark suite, there were two potential candidates that fitted the requirements. However, there were again problems in converting them for the two architectures under examination.

Even though converting these applications for the use in TMFv2 is an ongoing project which will show the benefits of this system in comparison to other transactional memory implementations, the used approach and as well as the simulator are still not mature enough to achieve this goal at the moment.

However, since the goal of this thesis was to improve the original TMFab design, another set of benchmarks was created, based on common algorithms found on the internet, in order for them to correspond to realistic applications.

5.3 Hash Table Benchmark

As was already mentioned, in order to explore the performance characteristics of the two systems, there is need for coarse grained benchmarks, i.e. benchmarks that have large critical sections. This requirement is satisfied by applications that perform operations on hash tables.

A hash table is a data structure, which is used to save information in a way that it can be easily located afterwards. Data are organized in the hash table according to their value, following a hash function which assigns them to the correct slot in an array. These data can be easily retrieved at a later point, by following the same hash function[1]. However, there is a big chance, that two values belong to the same slot in the array according to the hash function. When this happens, there are ways to resolve the issue such as using a list, or by just placing the data in the next available slot in the array.

However, using a hash table in a multi-core system is quite inefficient with current locking techniques, because a big part of the execution needs to be locked in order to prevent two threads from accessing the same position of the hash table simultaneously. If this happens for a read only operation there is no danger for the data consistence, however when the entries are modified the use of locks is necessary.

By using transactional memory, there is no longer need for locks to be used in order to protect the critical section, since the conflict detection takes place in the hardware level. This way, more transactions are able to operate on the table simultaneously, and since the chances that two keys are going to be hashed on the same table slot are small, the abort rate will also be relatively small.

The hash table benchmark that was used, is a transactionalized version of [15]. In the program there are 32 concurrent transactions, each of which is responsible for taking a keyword from a 32 entry table and installing it inside of the hash table. From the 32 keywords, there are 4 conflicting pairs and one conflicting triplet, which means that they need to be associated with the same entry in the hash table. When the keywords collide, then they are connected in a single linked list starting from the hash table entry.

The reason for which there were 32 transactions used, although there are only 16 PEs in the system, is to be able to show the system's performance when there are more transactions to be assigned than the number of PEs. Because of the nature of the system, the execution time of the second group of transactions, that is assigned on the same PEs, is hidden inside of the validation and commit time of the transactions that are still active from the first group. For this reason, if only 16 transactions would be used, the observed performance would be almost the same for both 8 and 16 PEs in the system.

The benchmark was executed for topologies using 1,2,4,8,12 and 16 PEs.

[1] e.g when we want to retrieve information about someone inside of a database and we search him by his name, the hash function will use his name and find his position in the hash table database where his name is located

5.3.1 Results

As can be observed in Figure 5.1, the modified design shows better performance when the number of PEs is above 4, while the original design shows slightly better results for a 4 PE system. The reason for that is that for a small number of processors, the validation overhead of TMFab is also small since there are not so many active transactions to validate with. On the other hand, in TMFv2, the transactions need to validate with all the banks that they read from, which imposes already a considerable overhead. However, when the number of processors increases, this overhead remains within the same order of magnitude, unlike the overhead of the inter-transactional validation that was used in the original TMFab system.

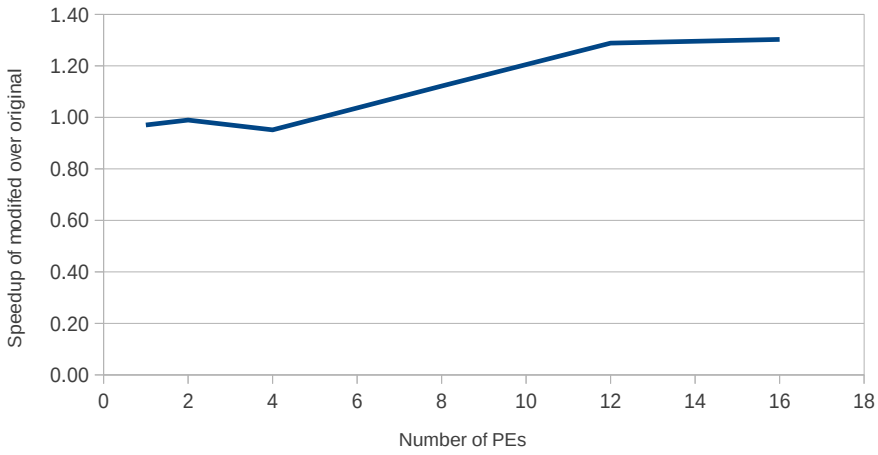

Figure 5.1: Overall speedup of TMFv2 over TMFab

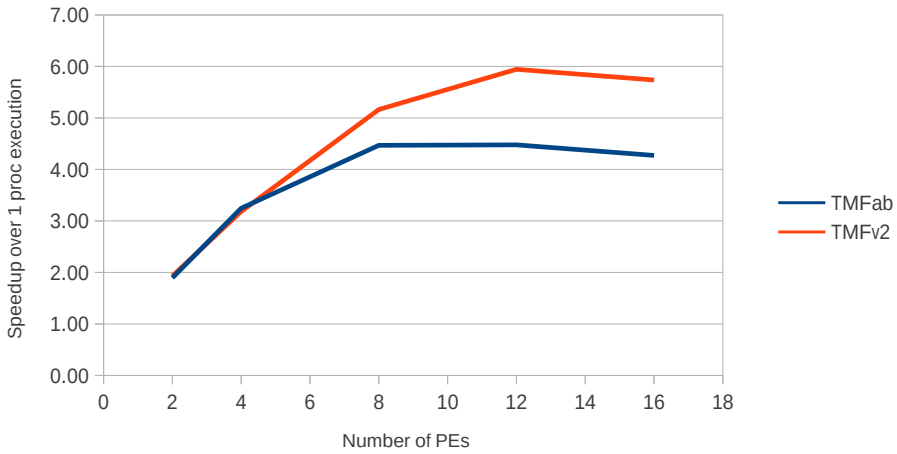

Figure 5.2: Scalability of both designs

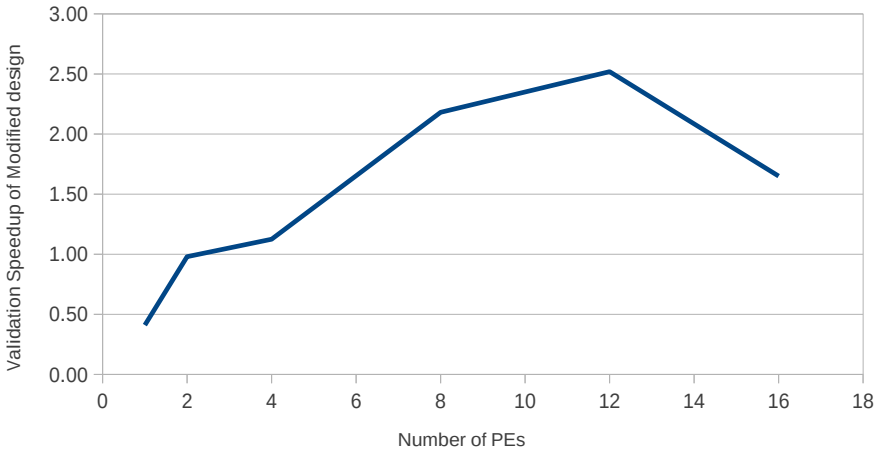

Figure 5.3: Validation speedup

This effect is also shown in Figure 5.2 where the scalability of the two designs can be observed, by calculating the speedup in comparison to the single PE performance for each design. The graph shows that TMFv2 is more scalable, however when the number of processors is increased above 12, the performance starts deteriorating. This is caused by an increase in validation overhead, and thus a reduction in validation speedup in comparison to the original TMFab design, which can be seen in Figure 5.3. The reason for that is, that by increasing the number of transactions that are able to execute in parallel, also the number of hazards increases, since more transactions are trying to validate the same cache lines at the same time. This causes for the transactions that are not able to register their VIDs in the SRVID field to receive hazard messages as shown in Figure 5.4. The effect this phenomenon has on the increase of the total validation packets sent to the L2D banks is presented in Figure 5.5. Furthermore, the additional level of PEs that is needed for a 16 PE topology causes the transactions to be more spread inside of the network interconnect, and thus the memory access latency to increase.

Even though the validation overhead of TMFv2 increases considerably when the design is scaled up, still it is smaller than that of the original TMFab design as can be seen in Figures 5.6 and 5.7. The reason is that in TMFab there is no validation concurrency, which implies that the average validation time also includes the time for which a transaction is stalled until it acquires the validation token. On the other hand, in TMFv2 shows a dramatic increase in the hazards, resulting in a consequent increase in address re-validation and of the validation overhead in total. Even though this has also to do with the current benchmark and the way that possible conflicts are distributed in the program, it already highlights on of the main disadvantages of TMFv2, which is that the number of potential conflicts and hazards greatly affects the system's performance.

As far as the execution time is concerned, the benefit of using a banked L2D is clear comparing the two designs, since the execution time drops considerably, because of the

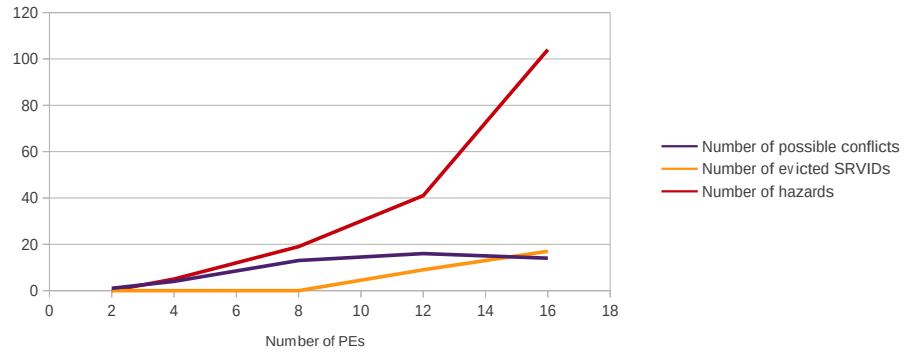

Figure 5.4: Number of possible conflicts, hazards and evicted SRVIDs for different number of PEs

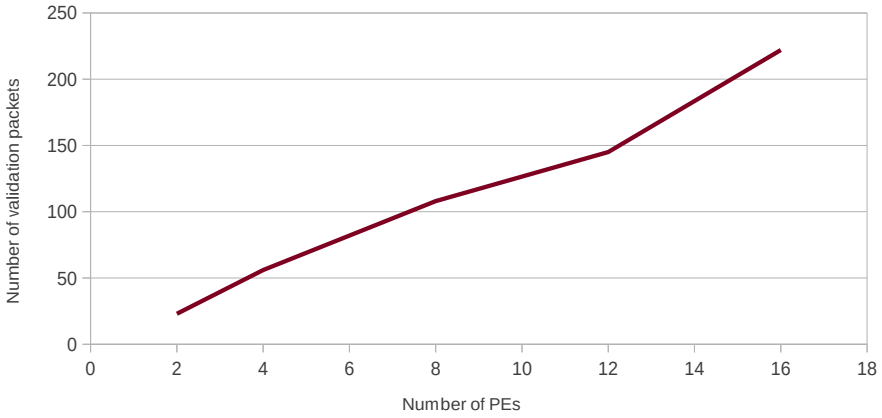

Figure 5.5: Number of validation packets received by all the banks for different number of PEs

decrease in the read latency as is shown in Figure 5.8

Finally, it was observed that there was a difference in the occurrence of real conflicts between the two designs. As can be seen in Figure 5.9 both designs wasted time because of aborts, however this happened in different topologies for each one. In the original design, fewer conflicts were observed in the presence of fewer PEs, because the conflicting transactions were not executing in parallel due to the limited PE resources. In TMFv2 the number of aborts, and consequently of wasted time was smaller when 16 PEs were active. This phenomenon is related to the dynamic ordering of transactions, which changes depending on the topology and on the way the transactions are distributed in the system. For the above reason, no direct conclusion can be derived about the abort rate of the two systems only by the results of the current benchmark.

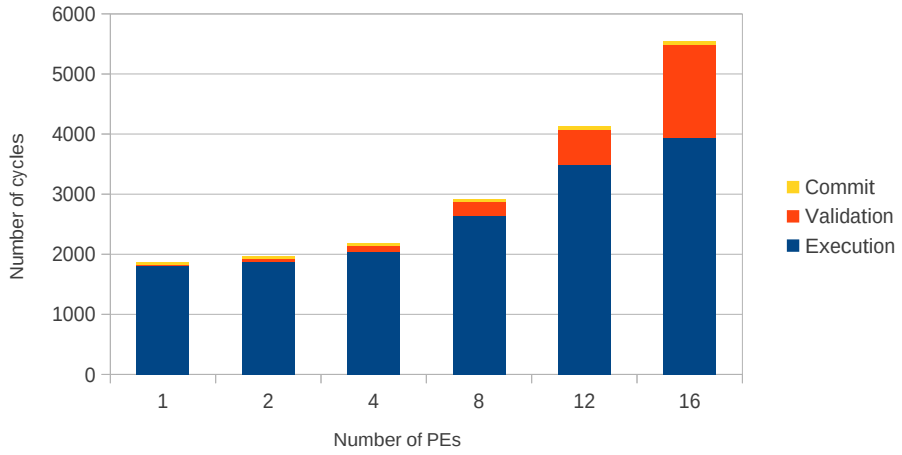

Figure 5.6: Distribution of execution time in transactional stages for TMFab

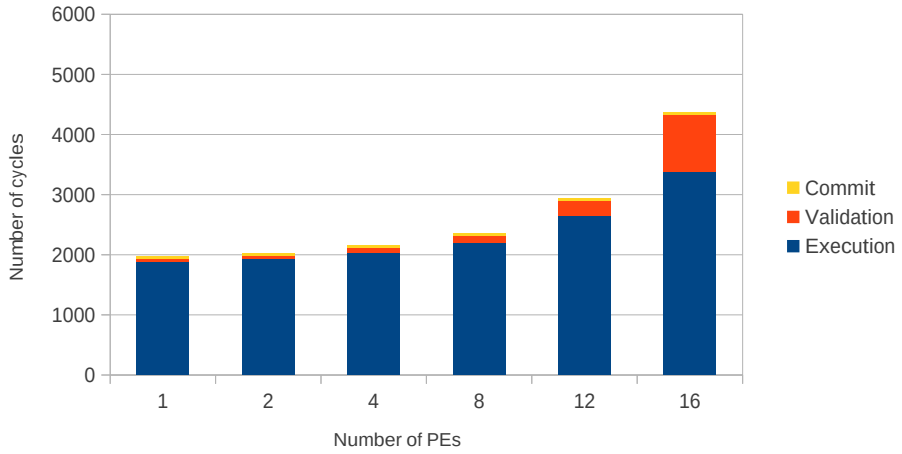

Figure 5.7: Distribution of execution time in transactional stages for TMFv2

5.4 Load-Store Benchmark

In the previous benchmark, the purpose was to characterize the systems' performance for realistic applications. In this section, a less realistic, worst case scenario application will be used to derive more conclusions about the performance of the two designs. The used benchmark is a variant of the load-store benchmark[2] used in [1] for a medium data size.

The benchmark comprises of 16 transactions with ranging number of dependencies between them as shown in Figure 5.10. The operations they have to perform are simple load-modify-store operations on the elements of an array. The dependencies, when

[2]coded in C instead of assembly

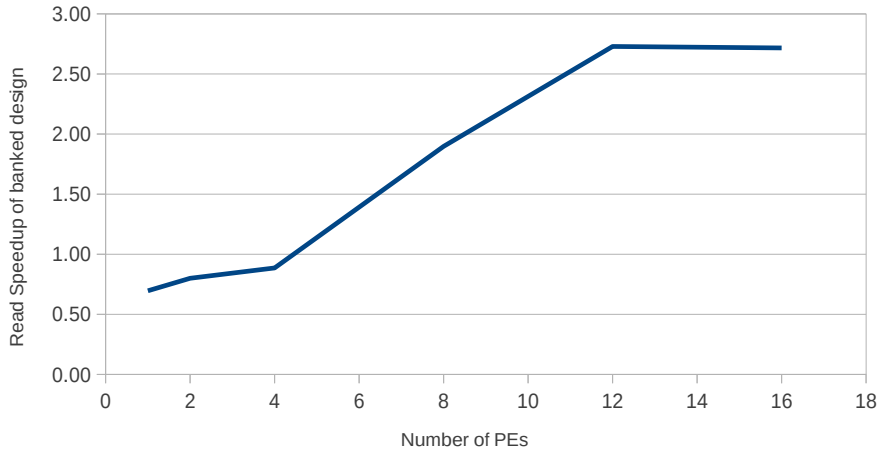

Figure 5.8: Speedup of the memory access

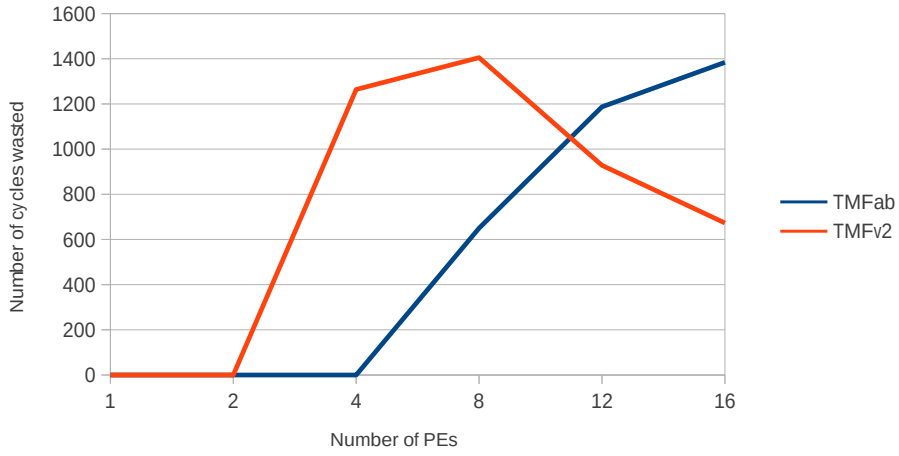

Figure 5.9: Execution time wasted

they exist, are located in the beginning and end of every individual array chunk, which reflects directly to their placement at the beginning and the end of every transactional block.

In the original TMFab, this placement of dependencies corresponded to worst case scenario execution because it forced all the transactions to abort and serialize their execution, causing a speedup of less than 1x in comparison to single core execution. However, this is not the case in TMFv2 as is going to be explained in the following paragraphs.

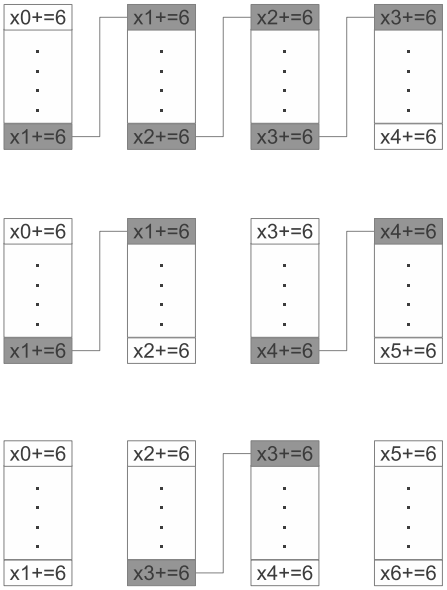

Figure 5.10: Varying number of conflicts distribution

5.4.1 Results

In Figures 5.11 and 5.12 the respective performance speedup of TMFab and TMFv2 can be seen, for different number of PEs and dependencies. The speedup was measured for 2,4,8 and 16 PE topologies in comparison to single PE execution. It must be taken into consideration that the range of the two plots is different in order for the results to be more legible, and that the TMFv2 design shows overall better performance than the original TMFab.

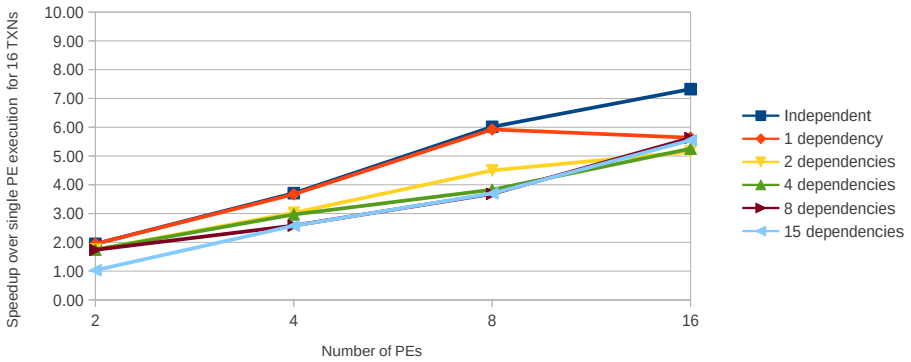

Figure 5.11: Speedup over single PE execution for original design

From the aforementioned plots, it can be observed that TMFv2 is more scalable than TMFab in all cases. As far as the independent execution is concerned, the reason for the speedup is the reduction of the validation overhead. As explained also in the

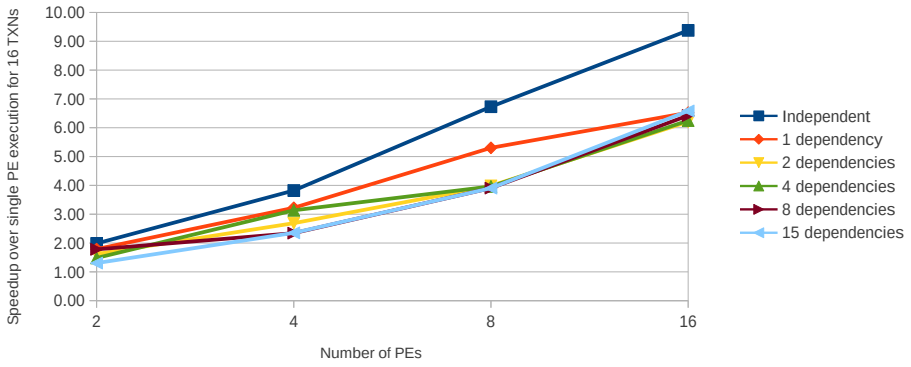

Figure 5.12: Speedup over single PE execution for modified design

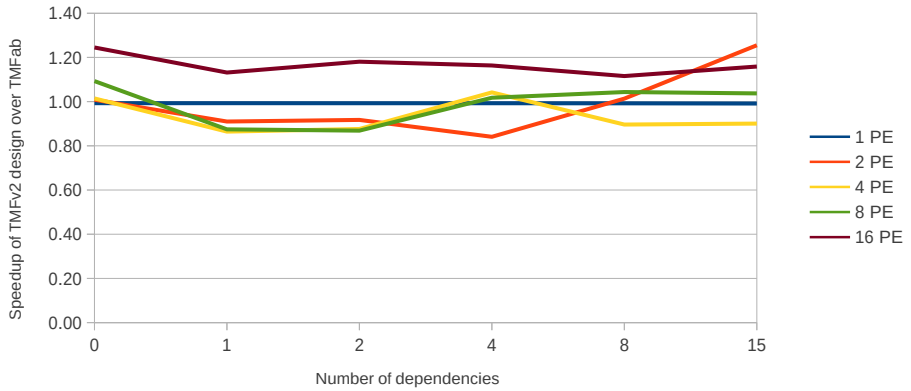

Figure 5.13: Speedup of Modified design over Original

hash table benchmark, the transactions need to validate only with the L2D instead of each other, which leads to reduction in network traffic because of the decrease in the number of validation packets that are injected in the network. In addition to that, the concurrent validation and commit scheme proves as expected more efficient than the alternative of exclusive validation/commit operations.

It is important to mention at this point, that since the original TMFab design validated the write-set, while TMFv2 validates the read-set, the difference in performance between the two systems greatly depends on the nature of the benchmarks that are used. In the current benchmark, the read-set and the write-set have exactly the same size so the improvements observed are independent of their sizes.

As far as the performance in the presence of dependencies is concerned, TMFv2 proves again more efficient. The main reason for that is that, as explained in the previous chapters, the original TMFab system assumed a predefined at compile time priority of transactions, while TMFv2 orders the transactions dynamically. In the first case, because of the chained dependency between the transactions, and the predefined priority between them, the transactions are going to be serialized because the older

transactions will force the younger ones continuously to abort. On the other hand, in TMFv2 the transactions acquire a VID dynamically at the end of execution and be prioritized in a first-come first-served basis. Following this scheme, when a transaction aborts, it will execute again and be re-scheduled at the end of all the transactions that are already validating, thus not affecting their validation process any longer. For example, in the first case of Figure 5.10, in TMFab the execution sequence would be T1, T2, T3, T4, while in TMFv2 the most probable execution pattern would be T1,T3,T2(previously aborted),T4(previously aborted).

Furthermore, the static priority scheme can potentially result in incorrect execution in TMFab, if for example a transaction with lower priority validates with a transaction of higher priority before the latter finishes execution. Even if there is no conflict detected, this doesn't guarantee that the conflict wouldn't have risen at the end of execution of the older transaction. The younger transaction will eventually commit and retire and not abort as it should.

A direct comparison between the two designs can be seen in Figure 5.13 where the speedup of TMFv2 over TMFab is presented, for different number of dependencies and PEs. As was also observed in the previous plots, the original design has better performance when fewer processors are used, while the modified shows its potential when the system uses more PEs.

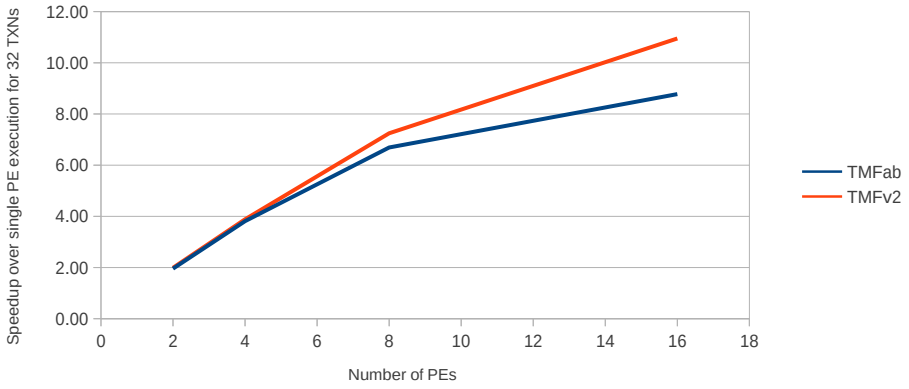

Figure 5.14: Speedup of both designs over single PE execution for 32 independent transactions

The performance of the two systems was also tested with 32 transactions inside of a single transactional section. As can be observed in Figure 5.14, in the absence of dependencies, the modified design shows better results even when the number of transactions increases. On the other hand, when the number of dependencies increases, the original TMFab shows better performance. The reason for that is, that the increased number of dependencies between transactions, causes for more hazard and potential conflict messages being sent through the network for their resolution.

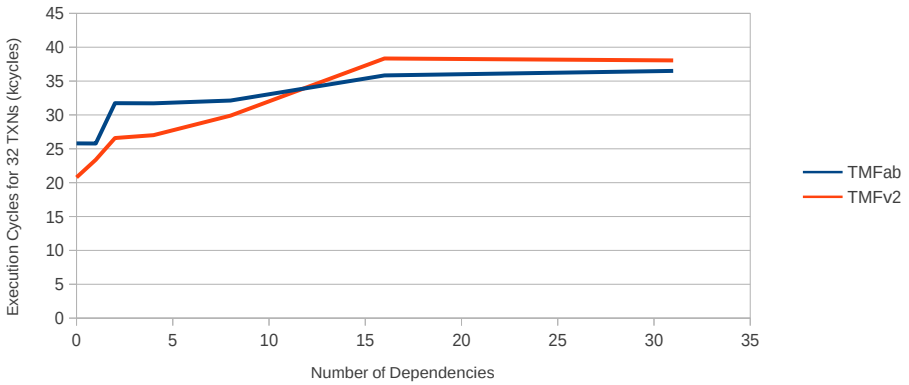

Figure 5.15: Total Execution Time in clock cycles for 32 transactions with 31 dependencies

5.5 Matrix multiplication Benchmark

The systems' performance was tested also with a transactionalized version of a pthreaded matrix multiplication program found on the internet. The characteristic of such a program is that the data workload of every transaction is partitioned very neatly between them and there is are no conflicts between transactions. Again the computation was divided between 32 transactions, which executed on the system's PE resources whenever they became available. The main difference of this benchmark in comparison to the Load-Store one is that in this case the computation time is much greater than the time needed for the validation/commit operations. The results that are going to be presented, show that both the system's performed well in terms of scalability, with the original TMFab design showing slightly lower performance, mainly because the lower memory access throughput.

5.5.1 Results

In Figure 5.16 it can be observed that TMFv2 shows a performance improvement of 12% in comparison to the original TMFab design. The main reason for this improvement in this case, is the memory access speedup appearing in Figure 5.17. Since the throughput increased 4x because of the banked memory topology, PEs are able to retrieve data faster, affecting this load intensive benchmark's execution considerably.

The validation speedup shown in Figure 5.18 is still important and is indicative of the system's scalability. s it can be observed, the speedup increases with the number of PEs in the system, confirming the original statement made in Chapter 3 that the validation overhead in TMFab increases according to the number of PEs, while in TMFv2 the validation time is independent of this parameter.

The overall speedup of both designs in comparison to single core execution, is depicted in Figure 5.19 and is compared to the ideal speedup of a perfectly scalable system. It is observed that the scalability of TMFv2 is better than that of TMFab. However, as was explained, this is mainly a result of the increased memory access throughput offered by the banked memory topology. In other words, if a banked mem-

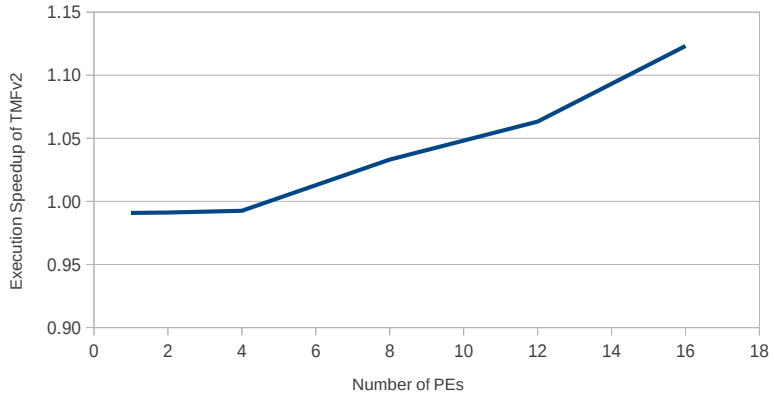

Figure 5.16: Execution speedup of TMFv2 over TMFab

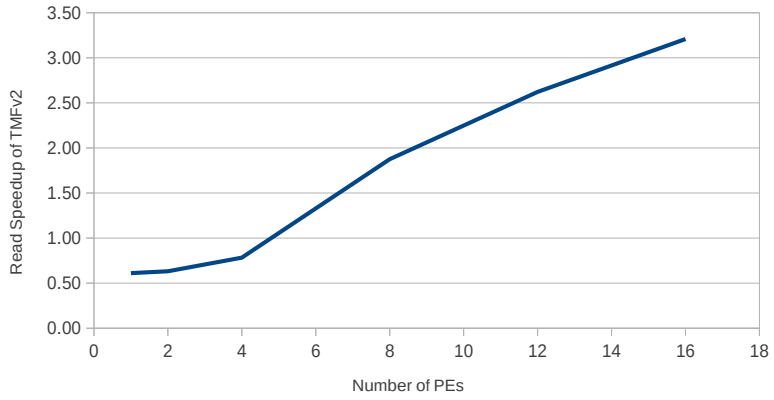

Figure 5.17: Memory access speedup of banked over non banked approach

ory system would be implemented in the original TMFab design, there would be an improvement in scalability as well.

In this benchmark, the performance improvement of TMFv2 is not as clear as in the previous designs, because the validation time in both cases is much smaller than the execution time. However, if this wouldn't be the case, or if there were few conflicts between the transactions, TMFv2 would probably show better performance than the original TMFab design.

5.6 Banked L2 Data Cache performance

One important difference between TMFab and TMFv2 is that the latter makes use of a stacked 3D banked L2D topology. There are two reasons for which a banked cache was used:

1. In order to decrease cache access latency by using the NoC more efficiently and

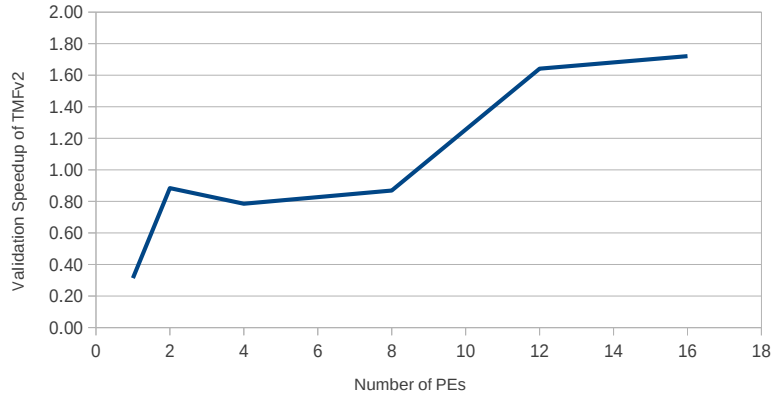

Figure 5.18: Validation speedup of TMFv2 over TMFab

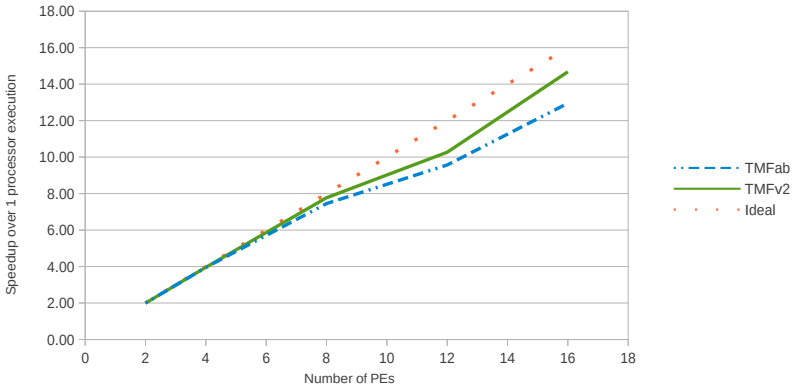

Figure 5.19: Scalability of TMFv2 and TMFab

being able to access multiple data in parallel.

2. In order to support validation and commit concurrency, by sending validation and commit packets to multiple transactions to different banks at the same time.

In this scope, a very important characteristic of the system that needs to be explored, is what is the effect of data distribution in the different banks, on the system's performance. The hash table benchmark was used once more for this purpose. The parameter that was changed in this case was the size of a memory block containing consecutive addresses, called *Contiguous Section(CS)*. The way these sections are distributed in the different banks is illustrated in Figure 5.20.

The following subsection discribes the effect of the different CS size on the data distribution, and consequently on the performance of the system.

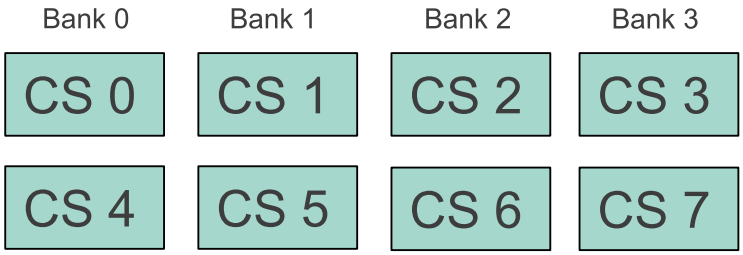

Figure 5.20: Contiguous Section distribution in the different banks

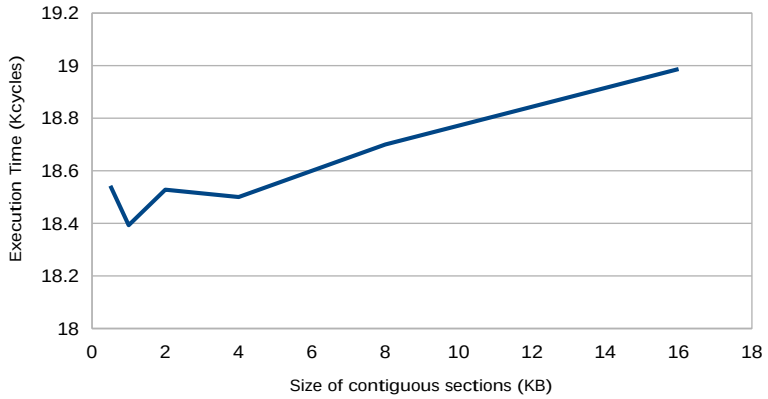

Figure 5.21: Total Execution Time in clock cycles

5.6.1 Results

In Figure 5.21 it can be observed that the total execution time increases proportionaly to the increase in the size of a CS. The reason is that when the size of a CS increases,

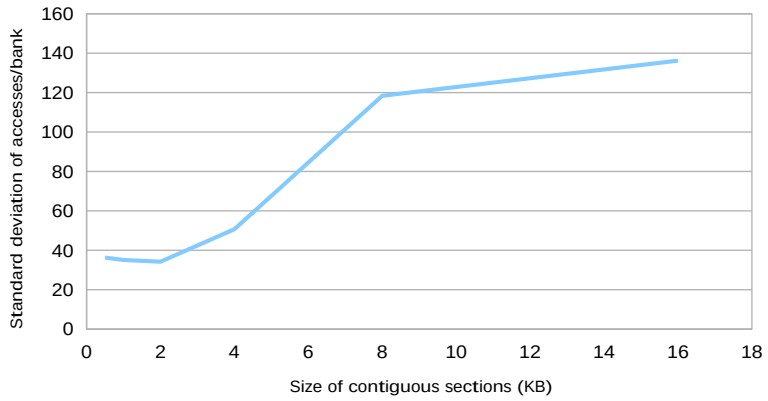

Figure 5.22: Standard deviation of memory accesses per bank

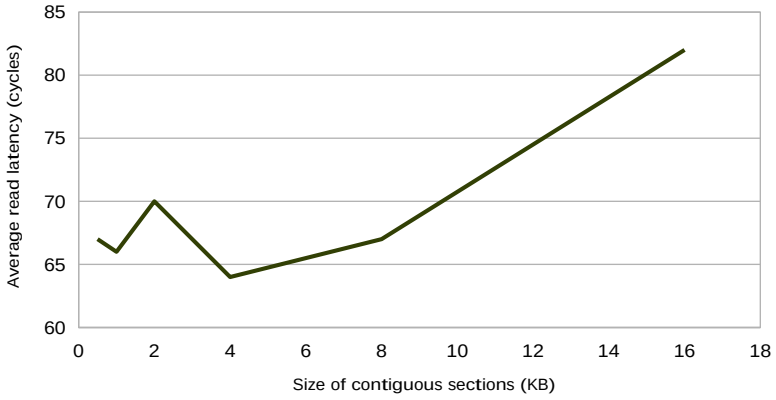

Figure 5.23: Read latency

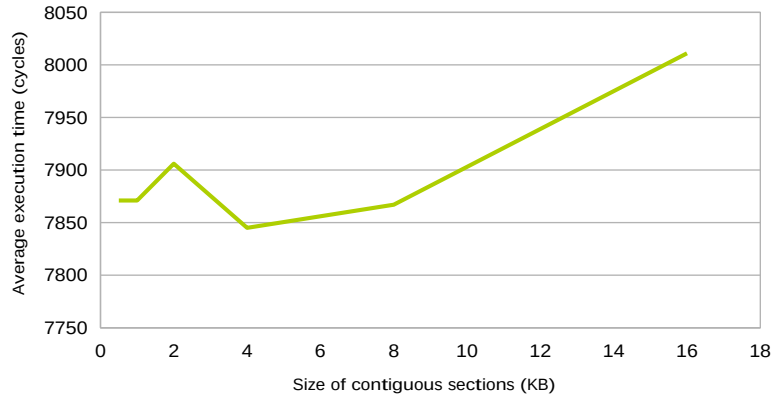

Figure 5.24: Average Execution Time in clock cycles

the data are concentrating in some of the banks, resulting in all of the transactions retrieving data from the same banks. This lead in a decrease in the memory access throughput, and thus in performance deterioration. The change in data distribution can be seen in Figure 5.22 which shows the standard daviation of the memory accesses per bank. For larger CS sizes, the deviation increases indicating that the data are distributed unevenly between the different banks.

This observation is asserted also by Figure 5.23, where it is shown that the read latency increase for larger CS, leading in an increase in the average execution time shown in Figure 5.24. Furthermore, the average validation time in Figure 5.25 and commit time in Figure 5.26 increase as a result of the decreased validation/commit throughput, since most of the validation/commit packets target few of the memory banks.

From the above results it is concluded that when the dataset distribution between the different banks is even, the system shows better performance. The reason for that is that the memory accesses are serviced in parallel by more resources, the validation

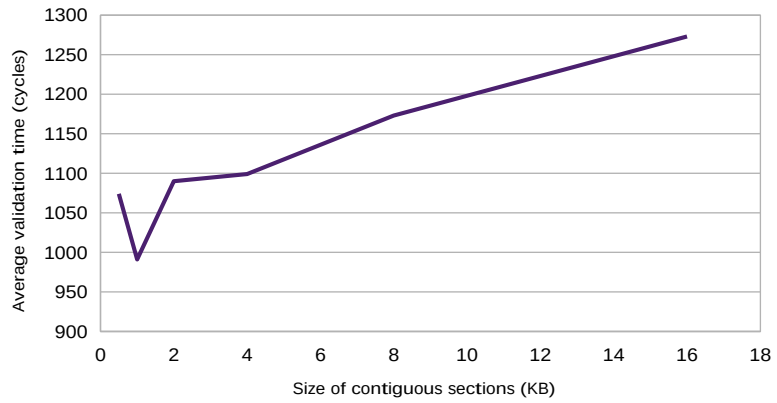

Figure 5.25: Average Validation Time in clock cycles

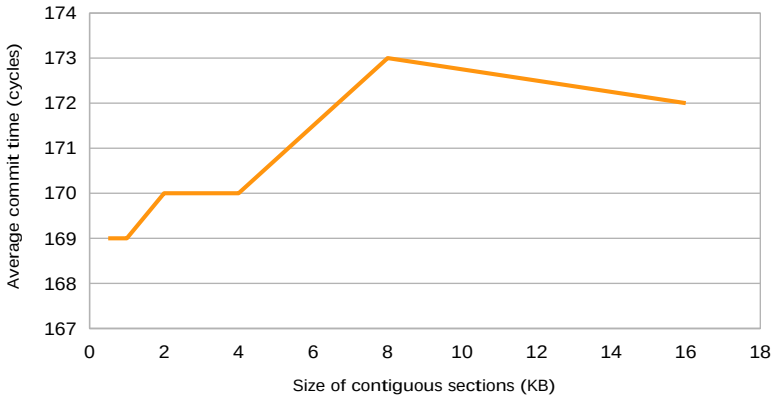

Figure 5.26: Average Commit Time in clock cycles

packets are divided more evenly into the different banks, which results in better use of both the NoC and the banks. Furthermore, when the dataset is well spread between the different banks the validation and commit concurrency improves.

In order to avoid performance deterioration because of uneven data distribution in the different banks, the size of the contiguous blocks should be set according to the needs of the most common applications. If the blocks are very small, then all the transactions need to validate with all the banks and the data are going to be interleaved between the banks for a single transaction. This will cause for transactions to need priority in all the banks in order to be able to commit, thus reducing the commit concurrency. On the other hand, if the blocks are very big, this could result in the dataset residing in the same bank thus excluding any real validation and commit parallelism. The validation packets are still going to be interleaved between the different validating transactions, but there will be only one reaching the L2D at any given point.

In other words, if the dataset that corresponds to every transaction is carefully divided by the compiler between different banks, this could result in higher validation

and commit concurrency which would lead to a more scalable system.

5.7 Limitations

After the results were analyzed, it was concluded that TMFv2 has three main limitations:

1. *The scalability of the system is limited by the available memory throughput* The number of memory accesses or validation packets that can be handled in parallel is related to the number of ports to the L2D. In the current architecture, since there were 4 banks in the system, the maximum throughput was 4x32 = 128 bits/cycle. The throughput increases with every additional die that is added in the system because one additional memory bank is in use. However, if the system is expanded in the lateral dimension, the only ways to increase the throughput is either to add memory banks at other points in the network, or replace the current banks with multiport banks which can service several requests simultaneously.

2. *The validation overhead is affected by the number of conflicts* When the number of potential conflicts increases, the data needed to by transfered through the interconnect network to resolve those conflicts, increase accordingly. Furthermore, when the number of hazards increases, the number of re-validations increases too. Both of those facts increase the network traffic and increase the validation overhead. This phenomenon is aggravated if the transactions are closely related, meaning that they have common read-sets even if they are not conflicting.

3. *Uneven data distribution can deteriorate performance* When the data are accumulated in few of the memory banks, the actual memory throughput decreases considerably. Consequently the memory access latency increases, and the validation/commit concurrency reduces, thus increasing the validation overhead.

6 Conclusion

This chapter summarizes the work done in the course of this MSc thesis project focusing on the achieved goals, as well as the conclusions derived from the experimental results. Furthermore, insight is given about future work needed to be done in order for the current system to be improved with respect to performance and functionality.

6.1 Summary

The primary goals of this thesis project was to improve the scalability of the TMFab transactional memory system, as well as guarantee sequential consistency even in corner cases where the original design couldn't. In order to do that, a SystemC simulator was built to test the performance of the original TMFab design. Based on the simulations it was concluded that when the system scaled up to more than 4 processors, the validation overhead increased considerably, due to the increased amount of time needed for every transaction to acquire validation privileges. When tha validation process was parallelized for the original validation scheme, the interconnect network was congested by all the messages sent from all the PEs in the system to all the other PEs. In order for the above issues to be tackled, the validation and commit process were altered using similar techniques as the ones proposed in [6] and [7].

TMFv2, the modified version of TMFab, parallelizes the validation and commit operations of transactions that have finished their execution. In order for this to be possible, the shared memory was divided into banks, each of which was placed on a different level inside of the NoC. This decision reduced the memory access overhead during execution, by allowing for multiple memory accesses to be serviced in parallel. These modifications resulted in a maximum speedup of 2.7x in comparison to the baseline design.

TMFv2 keeps the lazy version management and optimistic conflict detection policies used in the original TMFab design, however every transaction validates now only with the shared memory instead of each other. This choice reduces the network traffic since the transactions need to send validating packets to only one recipient, instead of all the active transactions. Additionally, the transactions are not able to abort before they finish execution which simplifies the design and protects from race conditions in which restarting transactions access stale data in the shared memory before they are updated by the commiting transaction.

Furthermore, TMFv2 uses dynamic ordering of transactions instead of compile-time ordering that was previously used, in order to guarantee sequential consistency and thus correctness of execution. The transactions request for a Validation ID from the scheduler, who provides this VID on a first-come first-served basis. All the transactions are allowed to validate their read-set if they acquire a VID, however they can not

commit their write-set until it is certain that there is no conflict with a higher-priority transaction. Nevertheless, they are allowed to commit in parallel if the write-set of the older transaction does not conflict with the read-set of the younger one.

Eventually, a SystemC simulator was built also for TMFv2, which allowed for the performance of the two designs to be compared. The derived experimental results show a validation speedup of up to 2.5x, of TMFv2 over TMFab, for a hash table benchmark. In total, a maximum of 30% speedup in comparison to the original design was observed.

Finally, the system's performance was evaluated for different data distribution patterns in the L2D banks, and it was concluded that the performace increases when the data are evenly distributed between the different banks as expected.

In spite of all the performance improvement, it was observed that when the number of dependencies increased considerably, performance scalability was seriously impacted. This indicates that there is need, for at least some of the dependencies, to be found and resolved in compile-time.

6.2 Future Work

The aforementioned problems, as well as other problems encountered throughout this project's development, motivate the need for future work. The exploration of the following are areas could lead to further system improvements:

1. In order to decrease the validation overhead in case of multiple conflicts, a scalable directory based scheme like the ones proposed in [16], [17] and [18] could be explored, which overcomes problems tied to conventional directory scalability.

2. Another possible solution to reduce the validation overhead is to leave transactional signatures in every cacheline in the shared memory during execution. This signature will contain the last transaction that accessed this cacheline, while upon eviction of a previous signature, the transaction that left it would be notified for a hazard. This method could potentially reduce the number of validation packets after execution since only the hazardous lines would need to be re-validated.

3. In the current design, there is no real need for a cache coherency scheme, because threads are replaced by transactions, and two consecutive transaction running on the same processinng unit are unlikely to have the same data-set. However, this reduces the advantage of having a local L1 data cache. For this reason, there might be a need in the future for thread support by the remote PEs. If this is the case, a cache coherence scheme needs to be implemented to support them.

4. In order to increase validation concurrency, there is need for even data distribution between the different units, which could be satisfied by an appropriate compiler.

5. The number of dependencies between transactions greatly affects the performance of the system. For this reason, a software toolchain is needed, to track the static and dynamic dependencies in compile-time. Using this information, a new scheduling scheme could be devised, which orders transactions in a way that

aborts are avoided, or stalls transactions before speculatively accessing conflicting addresses in the shared L2D.

6. The scheduler could provide the VIDs according to a scheme which takes into consideration which banks have been accessed by every transaction. This way, the transactions that have accessed fewer banks will acquire higher priority since they will finish their validation cycle faster and retire.

7. The support for different phases running simultaneously should be implemented, and the way in which transactions from different phases are allowed to validate, should be explored.

8. The system's performance should be evaluated for topologies using more than 16 processing units.

SystemC Simulators

A

During the making of this thesis, two cycle accurate simulators were created in the SystemC programming language (a superset of C++). One is describing the original TMFab design, while the second describes TMFv2, which is the newer version of TM-Fab. The simulators are similar in most aspects, apart from the validation scheme that was implemented in each case. TMFv2 simulator implements the design described in the current thesis, while TMFab simulator implements the one described in [1]. The current appendix is dedicated to the documentation of those designs.

A.1 Simulator Topology

Figure A.1 illustrates the block diagram of the the two designs.I the following paragraphs, a brief description of the content of every file will be given.

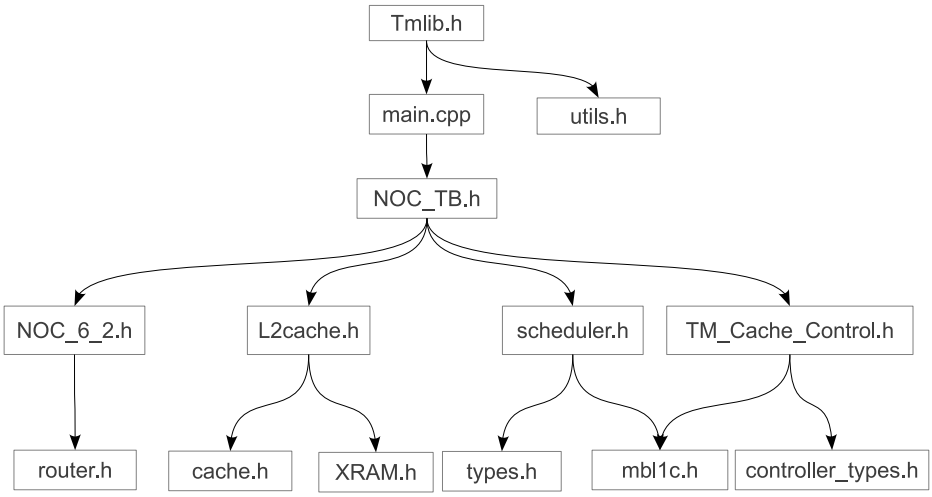

Figure A.1: Block diagram showing the relation between the files of the simulator

A.1.1 Description of files

- **TMlib.h**

  This file contains all the parameters pf both designs. The parameters that are meant for both TMFab and TMFv2 are shown in Table A.1, while Table A.2 contains additional parameters that are meant only for TMFv2.

- **main.cpp**(29 lines)

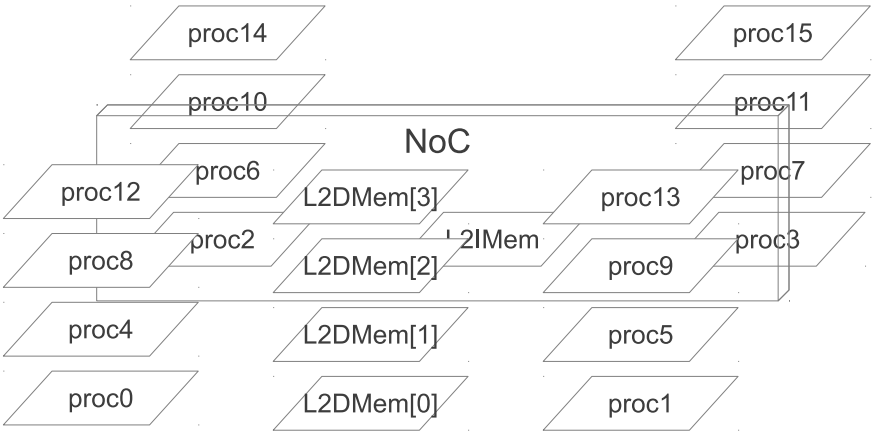

Figure A.2: 3D stacked architecture of TMFv2

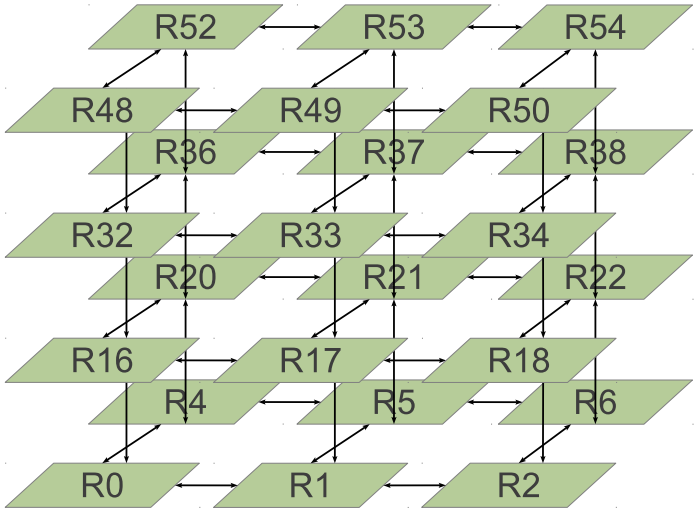

Figure A.3: NoC 3D mesh architecture

Is the file that is being compiled. In this file it is possible to set a predefined execution time. If not, the simulator executes until the program that is running on it completes execution. If for some reason the program doesn't complete, the simulator will keep running indefinitely.

- **utils.h**(74 code lines)

  Contains useful functions for compiler preprocessing and for data alignment needed for the communication with the PEs.

- **NOC_TB.h**(352 code lines)

  Contains the module that connects all the different tiles in the system with the NoC, as shown in Figure A.2.

- **NOC_3x2.h**(303 code lines)

Contains a module comprised of 6 routers which is located on a single die in the stacked topology. Four of these modules are connected in the NOC_TB.h file together with the functional units to create the whole architecture.

- **L2cache.h**(1045 code lines)

  Contains a single bank of the L2D in the TMFv2 design, while it contains the complete L2D in the original TMFab simulator. The methods used are:

  - network_input
  - header_handle
  - read_cache
  - write_cache
  - miss_handle
  - reset_methods

  Their architecture is analyzed in detail in Chapter 4.

- **types.h**(82 code lines)

  Header file which contains structures necessary for the Supervising Unit.

- **scheduler.h**(1184 code lines)

  Contains all the logic for the Supervising Unit.The methods used are:

  - read_program
  - MBSuper_connect
  - MBSuper_data_connect
  - Schedule
  - Network_in
  - Network_out

  Their architecture is analyzed in detail in Chapter 4.

- **controller_types.h**(52 code lines)

  Contains necessary structures for the TMPE.

- **TM_Cache_Control.h**(1834 code lines)

  Contains all the logic for the TMPE.The methods used are:

  - Reset
  - NW_input
  - NW_ouput
  - cache_access
  - input_handle

- state_handle
- MBLite_connect
- MBLite_instr_interface

Their architecture is analyzed in detail in Chapter 4.

- **router.h**(182 code lines)

  This is the basic unit of the NoC 3D Mesh architecture illustrated in Figure A.3. The number of the router defines its location inside of the network and is the reference point with which the routing is taking place. So the number has to be correct when every router is instantiated.

- **cache.h**(72 code lines)

  A header file containing necessary structures for the L2D cache.

- **XRAM.h**(230 code lines)

  This is the XRAM of the system. In the TMFv2 design, it has four ports and performs also the arbitration between the different banks in the L2D. In reality this arbiter would be located on the same chip as the banks and not inside of the XRAM.

- **mbl1c.h**(1294 code lines)

  An additional Floating Point Unit(FPU) has been added in the Microblaze simulator, together with the support for hardware multiplication and division, in order to allow for faster simulations. However, for the time being this instructions are executed in one cycle, which doesn't correspond with the real execution time in a Microblaze processor. If there is need for accuracy, either the FPU needs to be modified to allow for stalling, or the

  The two designs have a lot of similarities. However, since a new validation scheme had to be implemented, the TM_Cache_Control.h, the L2cache.h and the scheduler.h had to be modified in a great extend.

A.2 Output Files

The following output files contain information about execution. Some of them need to be specified in compile flags in order to be created. These are the ones that are used for debugging purposes. The ones that are used for statistics are always created.

A.2.1 scheduler.h

Overview file : **execution_files/execution_overview_(PE_NUMBER)procs.txt**

This file contains information about the time at which a transactional section starts being executed and its duration, the time when the transactions are spawned to the available PEs in the system, and the total parallel execution time. The messages appearing in the file are:

Param. Name	**Default Val.**	**Description**
BUF_DEPTH	100	Buffer depth of the L2D buffer
Param. Name	Default Value	Description
ADDR_SIZE	32	Address length
DATA_SIZE	32	Data word length
INSTR_LENGTH	32	Instruction length
NUMBER_OF_SETS	8192(/4 in TMFv2)	Number of sets per bank in TMFv2 - Number of total sets in TMFab
ASSOCIATIVITY	8	Associativity of L2D cache
WORDS_PER_LINE	16	Number of words per cacheline (CACHE-LINE_SIZE in some places)
FLITWIDTH	36	Size of single flit
FIFO_DEPTH	12	Buffer depth for NoC routers
PORTS	7	Number of NoC router ports
XRAM_SIZE_DEF	4194304	XRAM size in Bytes
IMEM_SIZE	8192	Instruction memory size in words (4 Bytes)
INS_L1_ASSOC	2	L1I cache associativity
INS_L1_LINE_ADDR	13	Number bits needed to address an L2D set (13 bits for cacheline addressing for 8192 sets)
DMEM_SIZE	256	Number of L1 Data cache sets
DAT_L1_ASSOC	4	L1 Data cache associativity
DAT_L1_LINE_ADDR	8	Number bits needed to address an L2D set (8 bits for cacheline addressing for 256 sets)
SWB_SIZE	512	Number of SWB cacheline entries
PROG_SIZE	131072	Max number of 32bit instructions in the program
SEQ_PHASE_LEN	12	Sequence - Phase bit length
INSTR_MASK	0xFFFF7000	Instruction mask to detect a START_TNX instruction
START_TNX	0xAAF87000	Instruction defining a new transaction (END_TNX is the same but has 0 as a parameter)
MAX_TNX	16	Maximum number of concurrent transactions
PE_NUMBER	16	Number of PEs in the system

Table A.1: Table of parameters for both designs

- @Timestamp: I start parallel part
- @Timestamp: I start sequential part
- @Timestamp: I sent transaction (transaction number)

Param. Name	Default Val.	Description
ID_SIZE	12	VID size
MAX_VID	4096	Maximum number of VIDs depicted with 12 bits
NUMBER_OF_BANKS	4	Number of L2D Banks
LSB_DEFINING_BANK	10	LSB in address for bank select
BITS_FOR_BANKS	2	Number of bank select bits
HAZARD_TBL_SIZE	128	Size of the Hazard table
READ_BUF_SIZE	256	Size of the Read table

Table A.2: Table of additional parameters for TMFv2

NAME	ACTUAL VALUE
SCHEDULER_NW_ADDR	"010100"
Communication Class options	
INSTR_BLOCK_TRANSFER	"00"
TM_COMMUNICATIONS	"01"
MEMORY_OPERATIONS	"10"
Communication ID options	
for INSTR_BLOCK_TRANSFER	
TNX_STATE_TRANSFER	"001"
REGFILE_TRANSFER	"010"
INSTRUCTION_REQUEST	"011"
for MEMORY_OPERATIONS	
from PE/SCHEDULER to L2DCache	
L2D_WRITE	"001"
from PE/SCHEDULER to L2DCache and back	
L2D_READ	"010"
from SCHEDULER to L2DCache	
PROGRAM_ENDED	"111"
for TM_COMMUNICATIONS	
SCHEDULER_COM	"000"
from PE/SCHEDULER to L2DCache	
UPDATE_BANK_PVID	"001"
from PE to L2DCache	
VALIDATION	"010"
from L2DCache to SCHEDULER	
UPDATE_SCHEDULER	"011"
from L2DCache to PE	
VALIDATION_RESPONSE	"100"
COMMIT_RESPONSE	"101"

Table A.3: Network Signals and their actual values (a)

NAME	ACTUAL VALUE
Scheduler Operation options	
for VALIDATION	
READSET_VALIDATION	"0001"
VALIDATION_PACKET_SENT	"0011"
for UPDATE_SCHEDULER	
from L2DCache to SCHEDULER	
UPDATE_PVID_TABLE	"0101"
for VALIDATION_RESPONSE	
from L2DCache to PE	
CONFLICT_DETECTED	"0001"
HAZARD_DETECTED	"0010"
VALIDATION_DONE	"0011"
COMMIT_DONE	"0100"
for SCHEDULER_COM	
from SCHEDULER to PE	
NO_SCHED	"0000"
START_TRANSACTION	"0001"
INSTRUCTION_RESPONSE	"0010"
ASSIGN_VID	"0100"
UPDATE_PVID_MASK	"0101"
RESTART_COUNT	"1000"
from PE to SCHEDULER	
VALIDATION_TKN_REQUEST	"0011"
COMMITING	"0110"
ABORT_RESTART	"0111"
TNX_COMMITED	"1001"

Table A.4: Network Signals and their actual values (b)

- Total parallel execution time: (parallel time)
- Time needed for parallel part: (parallel duration)

Memory file : **execution_files/scheduler_mem_access.txt**

This file contains the memory operations of the Supervising Processor (SP) and is meant for debugging purposes. In order for this file to be created, it is necessary to add the -DSAVE_SU_MEM during compilation of the simulator. The messages appearing in this file are:

- @Timestamp: I read from XRAM[(address)] = (data)
- @Timestamp: I wrote to XRAM[(address)] = (data)

A.2.2 L2cache.h

Memory operations files : **mem_operations/write_sequence_bank_(Bank_id).txt**

This file contains the commit operations in the L2D cache banks and is meant for debugging purposes. In order for this file to be created, it is necessary to add the -DSAVE_L2D_COMMIT during compilation of the simulator. The messages appearing in this file are:

- Scheduler is committing a cacheline
- Proc (PE_id) with VID (input_VID) is commiting a cacheline
- @Timestamp: XRAM[(address)] = (data)

Bank statistics files : **cache_stats/Cache_stats_(Bank_id).txt**
This file usage statistics for every L2D cache bank. The statistics shown at the end of execution are:

- Number of reads
- Number of write packets
- Number of PVID updates
- Number of invoked restarts
- Number of validation packets
- Number of transactions validating
- Number of read misses
- Number of read misses while validating
- Number of possible conflicts
- Number of evicted SRVIDs
- Number of hazards
- Number of evictions

A.2.3 TM_Cache_Control.h

Log file : **logfiles/PE(proc_id)_log_file.txt**
This is a log file of what is happening inside of every TMPU. In order for it to appear, the -DSAVE_LOGFILE flag has to be included during compilation. The possible messages are:

- @Timestamp: NW_READ_IMEM :(proc_id) (OUTPUT FLIT)
- @Timestamp: NW_READ_DMEM :(proc_id) (OUTPUT FLIT)
- @Timestamp: NW_WRITE_DMEM :(proc_id) (OUTPUT FLIT)

- @Timestamp: NW_VAL_TKN_RQST :(proc_id) (OUTPUT FLIT)
- @Timestamp: NW_UPDATE_BANK :(proc_id) (OUTPUT FLIT)
- @Timestamp: NW_VALIDATE_HAZARD :(proc_id) (OUTPUT FLIT)
- @Timestamp: NW_VALIDATE :(proc_id) (OUTPUT FLIT)
- @Timestamp: NW_ABORT :(proc_id) (OUTPUT FLIT)
- @Timestamp: NW_COMMIT :(proc_id) (OUTPUT FLIT)
- There are also other messages that are meant mainly for debugging purposes and if the user wants to see the execution steps.

Statistics file: **exec_stats/PE(proc_id)_stats_file.txt**
This file contains information about TMPU statistics. The information is written only at the end of a successful execution that completed and contain:

- The execution time is:
- The validation time was:
- The commit time is:
- The average read latency is:

A.2.4 mbl1c.h

Instruction files : **execution_files/mblite_instructions(proc_id)run(restart_times).txt**
This file is also for debuging purposes and contains the instruction file of every PE in the system. In order for it to be saved, the -DSHOW_INSTRUCTIONS flag must be set. The only message contained is the one below:

- @Timestamp: (Intsruction)

This file becomes very big and is difficult to open, and it also slows down execution considerably. So it should be used only if absolutely necessary.

A.2.5 XRAM.h

Memory file : **mem_operations/memory_file.txt**
At the end of execution the XRAM prints the content of all the modified addresses in the file above, in order for the outcome of execution to be obvious. In order for this file to be saved, the -DSAVE_XRAM_MODIF must be set during compilation.
The only message appearing is:

- XRAM[(address)] = (data)

A.3 Makefile

A.3.1 Simulator

The source file of the simulator is the main.cpp, while the executable is called main_sim. The SystemC libraries need to be installed in a folder in order for the compilation to work. This is the folder that is currently set as the library directory : SYSTEMC_HOME = /usr/local/systemc-2.2

The possible flags that have to be set during compilation are:

- -DSHOW_INSTRUCTIONS
- -DSHOW_MEM_ACCESS
- -DSAVE_SU_MEM
- -DSAVE_L2D_COMMIT
- -DSAVE_LOGFILE
- -DSAVE_XRAM_MODIF

A.3.2 Executable

In order to compile a program for the current system, the Makefile in the sw_stuff folder needs to be set correctly. The output of the compilation will be a binary file named imem.bin. This file needs to be in the same directory as the simulator when the simulator is executed. By default, if the directories are not changed, the imem.bin will be automatically copied in that directory from the sw_stuff directory.

In order for the compilation to work, there is need for the mb-gcc compiler provided by Xilinx. The path to this compiler has to be set in the Makefile in the MBPATH. The name of the source file has to be set in the SRCS variable.

There are some more parameters and flags that need to be set inside of the Makefile.

Xilinx flags:

- -mxl-float-sqrt : allows for floating point squared root(faster simulations)
- -mhard-float : allows for hardware floating point operations(faster simulations)
- -mxl-soft-div : hardware division(faster simulations)
- -mno-xl-soft-mul : hardware multiplication(faster simulations)
- -mxl-soft-div :software division
- -msoft-float :software floating point operations
- -mxl-soft-mul : software multiplication

Other parameters:

- _STACK_SIZE : max stack size needed for sequential section

- _HEAP_SIZE : heap size
- _STACK_TXN_SIZE : max stack size needed for transactional section

Abreviations

B

BRAI :	Branch to Immediate
CM :	Contention Management
CMP :	Chip multi-processor
CS:	Contiguous Section
DLP :	Data Level Parallelism
GPP :	General Purpose Processor
GPU :	Graphic Processor Unit
HTM :	Hardware Transactional Memory
IC :	Integrated Circuit
ILP :	Intstruction Level Parallelism
IMem :	Instruction Memory
L1D :	L1 Data Cache
L1I :	L1 Instruction Cache
L2D :	L2 Data Cache
LCID :	Last Committer ID
NoC :	Network-on-Chip
NOP :	No Operation
OS :	Operating System
PE :	Processing Element
PVID :	Priority VID
Raw :	Read-after-Write
RM :	Read Mask
SP :	Supervising Processor
SRVID :	Speculative Reader VID
SU :	Supervising Unit
SWB :	Speculative Write Buffer
TID :	Transaction ID
TLP :	Task Level Parallelism
TM :	Transactional Memory
TMPE :	Transactional Memory Processing Element
TMS2 :	TMFv2 Scheduler
TS :	Transactional Section
TTP :	Transaction Table Pointer
TXN :	Transaction
VID :	Validation ID
WM :	Write Mask
XRAM :	External RAM

Bibliography

[1] S. S. Kumar, *TMFab: A Transactional Memory Fabric for Chip Multiprocessors*, November 2010.

[2] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 289–300.

[3] R. Rajwar and J. 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*, ser. ASPLOS-X. New York, NY, USA: ACM, 2002, pp. 5–17. [Online]. Available: http://doi.acm.org/10.1145/605397.605399

[4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st annual international symposium on Computer architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 102–.

[5] U. Aydonat and T. S. Abdelrahman, "Hardware support for relaxed concurrency control in transactional memory," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 15–26. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.25

[6] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 97–108. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2007.346189

[7] W. W. L. Fung, I. Singh, A. Brownsword, and T. Aamodt, "Hardware transactional memory for gpu architectures," *IEEE Micro*, vol. 99, no. PrePrints, 2012.

[8] M. F. Spear, M. M. Michael, and C. von Praun, "Ringstm: scalable transactions with a single atomic instruction," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '08. New York, NY, USA: ACM, 2008, pp. 275–284. [Online]. Available: http://doi.acm.org/10.1145/1378533.1378583

[9] F. Angiolini, P. Meloni, S. Carta, L. Benini, and L. Raffo, "Contrasting a noc and a traditional interconnect fabric with layout awareness," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, ser. DATE '06. 3001 Leuven, Belgium, Belgium: European

Design and Automation Association, 2006, pp. 124–129. [Online]. Available: http://dl.acm.org/citation.cfm?id=1131481.1131520

[10] V. F. Pavlidis and E. G. Friedman, "3-d topologies for networks-on-chip," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 10, pp. 1081–1090, Oct. 2007. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2007.893649

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[12] S. Kumar and T. van Leuken, "A 3D network-on-chip for stacked-die transactional chip multiprocessors using through silicon vias," in *2011 6th Int. Conf. on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*. Athens, Greece: IEEE, April 2011, pp. 1–6.

[13] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[14] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero, "Rms-tm: a comprehensive benchmark suite for transactional memory systems," in *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ser. ICPE '11. New York, NY, USA: ACM, 2011, pp. 335–346. [Online]. Available: http://doi.acm.org/10.1145/1958746.1958795

[15] S. Datta, "A hashtable in c." [Online]. Available: http://www.sourcecodesworld.com/source/show.asp?ScriptID=1188

[16] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "Waypoint: scaling coherence to thousand-core architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854291

[17] H. Zhao, A. Shriraman, and S. Dwarkadas, "Space: sharing pattern-based directory coherence for multicore scalability," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854294

[18] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," *High-Performance Computer Architecture, International Symposium on*, vol. 0, pp. 169–180, 2011.