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

CAS-2010-MS-11

M.Sc. Thesis

TMFab: A Transactional Memory Fabric for Chip Multiprocessors

Sumeet S. Kumar

Abstract

With the performance of single-core processors approaching its limits, an increased amount of research effort is focused on chip multiprocessors (CMP). However, existing lock-based synchronization methods that are critical to performing parallel computation possess limited scalability and are inherently complex to use while programming. This thesis uses the concept of transactional memory implemented within a synthesizable fabric named TMFab, containing all the requisite hardware components needed to prototype a scalable chip-multiprocessor. Its processor independent nature enables the instantiation and use of any suitable soft-processor core inside the fabric without significant modifications to the fabric hardware. Additionally, the fabric offers scalability on account of its 3D interconnect architecture that supports die-stacking to add additional processor cores to the CMP without increasing its area footprint. The hardware transactional memory system of the fabric reduces performance overheads of transactional operations, allowing transactions to complete execution faster. TMFab is shown to provide speed up as high as 3.44× for correctly partitioned independent transactions and can be used to analyze the points of contention for conflicting transactions. The fabric was synthesized for both Field Programmable Gate Array (FPGA) as well as 90nm semi-custom targets.

TMFab: A Transactional Memory Fabric for Chip Multiprocessors

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Microelectronics

by

Sumeet S. Kumar
born in Kuwait City, Kuwait

This work was performed in:

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

Delft University of Technology

Delft University of Technology
Department of
Microelectronics

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 **"TMFab: A Transactional Memory Fabric for Chip Multiprocessors"** by **Sumeet S. Kumar** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 26/11/2010

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

Advisor: _____
dr.ir. T.G.R.M. van Leuken

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

_____

Abstract

With the performance of single-core processors approaching its limits, an increased amount of research effort is focused on chip multiprocessors (CMP). However, existing lock-based synchronization methods that are critical to performing parallel computation possess limited scalability and are inherently complex to use while programming. This thesis uses the concept of transactional memory implemented within a synthesizable fabric named TMFab, containing all the requisite hardware components needed to prototype a scalable chip-multiprocessor. Its processor independent nature enables the instantiation and use of any suitable soft-processor core inside the fabric without significant modifications to the fabric hardware. Additionally, the fabric offers scalability on account of its 3D interconnect architecture that supports die-stacking to add additional processor cores to the CMP without increasing its area footprint. The hardware transactional memory system of the fabric reduces performance overheads of transactional operations, allowing transactions to complete execution faster. TMFab is shown to provide speed up as high as 3.44× for correctly partitioned independent transactions and can be used to analyze the points of contention for conflicting transactions. The fabric was synthesized for both Field Programmable Gate Array (FPGA) as well as 90nm semi-custom targets.

Acknowledgments

Two years ago, I arrived in the Netherlands from India with the enthusiasm of a child. I was in a new country, with new people and was about to set out on an incredible experience. In these two years I've come across situations that have taken me out of my comfort zone, met strangers I felt I knew for ages and had an opportunity to understand that its only when you give people a chance do you know how much they can impress you. The last two years have been about the people, who knowingly and unknowingly supported me and enriched my journey a little. It has been about experiences, of knowing that an ego comes in the way of true learning. And it has been of appreciation, for all the little things people do for you that always go unnoticed. Therefore, at the conclusion of my Masters, I would like to express my appreciation for all those people, for the experiences they gave me, and the little things they did that went a long way in sustaining me in these two years.

First, Rene. You took me on as a fresh Masters student two years ago, and played host to my strange questions and whimsical ideas throughout that time. It is with you that I had some wonderful discussions which made think like an academic engineer. I always appreciated the freedom you gave me with design decisions, and how we could proceed from the many brick walls we ran into. You kept my feet on the ground, and taught me to be realistic when it was needed the most. For this, and so much more, thank you.

Alexander and Huib, through my messing around with the MB-Lite, playing with cache simulators and fiddling with the ASIC design flow you remained patient. Without your help, a lot of issues which I encountered wouldn't have been solved very easily. Antoon - missing license file, crashing design flows and noisy fans - you dealt with them all even on gloomy Monday mornings. Thank you.

Laura, Judith and Minaksie, your presence on the 17th floor is always reassuring to every member of the group. You are the glue that holds the group together, managing every administrative task so students can focus on what they are meant to be doing.

To all my friends, you were always around to share my happiness, and never seemed to leave when I was sad. Without your friendship and your support, life wouldn't be as colourful.

Mum, Dad, Nisha, Nicky and Sharmishta - Your unfaltering faith in me always kept me going when I thought that hope was fading. Without your love, your understanding and your support, none of this would have been possible. You all have been my brightest stars, in my darkest nights.

Sumeet S. Kumar
Delft, The Netherlands
26/11/2010

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The quest for higher levels of performance from microprocessors has resulted in the limitations of single core processors being uncovered. Chip multiprocessors are seen as the future in an environment requiring high performance processor architectures capable of performing several tasks in parallel. However, such an increase in performance is not easily obtainable and is accompanied by its own set of challenges, most significant of which is the programming complexity associated with developing parallel applications. Conventional CMPs view parallel tasks as threads, requiring those that operate on shared data to be synchronized and perform their writes in a mutually exclusive manner. These requirements necessitate the use of *lock* and *barrier* primitives that increase both effort and complexity of programming applications for CMPs. Furthermore, locks impose a serious limitation on the scalability of the system as contention for shared data increases [1]. *Transactional memory (TM)* was first suggested as an alternative for locks in critical sections of code [2], but has evolved into a general parallel programming paradigm for chip-multiprocessors. TM uses transactions as the basic unit of work, executing them speculatively on cores and performing all synchronization with the shared memory only upon completion without requiring programmer specified locks and synchronization primitives. Consistency of shared data and correctness of execution are inherently guaranteed by transactional memory due to its strict checking for dependencies between transactions, and the fixed order in which their operations appear to complete. Most importantly, transactional applications have been shown to execute faster than their lock-based counterparts [2][1].

Despite its promising performance and relatively simple programming interface, there exist only two hardware implementations of TM - Atlas [3] and the discontinued Sun Rock [4]. While the Atlas CMP provides an excellent platform for research on TM, it uses hard PowerPC processor cores and offers no options for using alternative cores in its place. Similarly, the Rock's TM implementation uses a customized instruction set for its processor cores. In addition, the interconnect in both these CMPs offers limited scalability. Therefore, there exists no consolidated transactional memory infrastructure to prototype and implement TM based CMPs using other processor architectures, in hardware.

TMFab aims to address this by providing a transactional chip-multiprocessing fabric containing requisite TM hardware and a scalable interconnect to enable the rapid prototyping and implementation of CMPs composed of synthesizable soft-processor cores. Additionally, it reduces programming complexity by simplifying the transactional primitives available to the programmer.

1.2 Thesis Goals

This thesis describes the development of a hardware fabric which implements transactional memory to enable the parallel execution of transactional applications on an array of processor cores that share memory. The primary objectives of this work are to:

- Design a light-weight scheduler to manage execution of transactions on processor cores
- Design a hardware transactional cache memory system that is independent of processor architecture
- Design a low-overhead scalable baseline interconnect architecture
- Define a system policy that governs the operation of the TM system.
- Reduce performance overhead of TM operations
- Establish the speed up achieved in applications with the proposed architecture, along with the penalty incurred due to conflicts between transactions
- Implement the designed scalable hardware TM fabric for an FPGA target and in 90nm UMC technology

1.3 Contributions

This work analyzed existing state of the art transactional memory proposals to determine an appropriate TM system architecture for the envisaged fabric. This architecture incorporated additional features to improve performance and enable the realization of a hardware fabric for chip multiprocessor prototyping. Its significant contributions include:

- Proposes the first such hardware transactional memory fabric for shared memory chip-multiprocessor prototyping
- Processor architecture independent TM implementation
- Reduced performance overhead for scheduling and transactional validations
- Scalable 3D interconnect with full-custom Through Silicon Vias (TSV)
- Evaluation of the limitations of stacking, and area penalty incurred by the use of TSVs with varying keep-out areas

1.4 Thesis Organisation

This thesis is organized into the following chapters:
Chapter 2 introduces the concepts of parallelism, and describes the reasons that necessitated chip multiprocessing and subsequently, transactional memory. Various TM proposals and their contribution to the evolution of the methodology itself are examined, along with the only two hardware implementations of TM. Further, the concepts of die stacking and 3D networks-on-chip are examined with a survey of recent work in the field.
Chapter 3 provides an overview of TMFab, explaining its organisation and key components. Using this as a foundation, a system level transactional memory policy that defines protocols for version management, conflict detection, contention management and cache coherence is described.
Chapter 4 examines the architecture of TMFab in terms of its four primary components - scheduler, cache controller, L2 data cache and interconnect. Each is described separately along with information on design decisions, as well as their implications.
Chapter 5 evaluates the performance of TMFab in two parts. The interconnect is first tested in single and multi-layer configurations to determine performance, and stacking limitations. Thereafter, the complete system is tested after integration of a light soft-processor core. The performance overhead incurred for transactional operations is analyzed along with the penalty due to conflicts between transactions.
Chapter 6 details the hardware implementation of the fabric on FPGA and 90nm UMC technology targets. Additionally, it describes the full-custom design of a TSV cell in its three configurations, and reports the area overhead associated with the use of each.
Chapter 7 includes concluding remarks, and recommendations for future work.

2 Background

This chapter provides an overview of the basics of chip multiprocessing, and the various issues surrounding it. The concept of transactional memory is introduced, along with the different classifications of its implementation. One such class of TM systems - Hardware Transactional Memory (HTM), is explained, and the most influential HTM proposals from the last decade summarized. With this foundation, three transactional chip-multiprocessors are reviewed, and their limitations highlighted. Further, the concepts of 3D Networks on Chip using die-stacking and Through Silicon Vias are explained in this chapter.

2.1 Parallelism

Processors exploit different types of parallelism in order to speed up execution of application processes. There exist three primary types of parallelism: Instruction Level Parallelism, Data Level Parallelism and Task Level Parallelism.

Instruction Level Parallelism (ILP) During execution, the processor fetches instructions from the waiting stream of instructions and executes each on an appropriate function unit. Subsequent instructions in the stream that require a different function unit may be executed in parallel, provided no dependencies exist between the two. This is known as Instruction Level Parallelism and refers to the inherent parallelism in sequential code.

Data Level Parallelism (DLP) Loops that perform a certain operation on large data structures like arrays may be unrolled and executed in parallel by more than one processor. This is referred to as Data Level Parallelism.

Task Level Parallelism (TLP) A process may contain several groups of instructions, with each performing a particular task. Task Level Parallelism is exploited by the parallel execution of each one of these task on one or more processors. Note that DLP may be considered as a form of TLP.

The exploitation of ILP coupled with the consequences of Moore's Law have enabled the rapid increase in uniprocessor performance with every successive shift in technology node [5][6]. However, with realistic limitations on the amount of such parallelism that can be extracted from application code, other techniques are required to conserve the rate of performance increase. For this reason, chip-multiprocessors (CMP) are used to exploit additional TLP that may exist in the application code alongside any ILP.

CMPs consist of several processor cores (also called processing elements) with private caches, connected over a common interconnect. Those in which the cores share a common memory apart from having a local memory are referred to as Shared Memory chip multiprocessors.

Each processor core in the CMP executes a process task, also called a *Thread*. Since threads are executed in parallel with each modifying data in the core's private cache, it is imperative that appropriate measures are taken to ensure the consistency of shared data in all private caches in the system and the shared memory. This means that every private cache must hold a coherent view of shared data, ensuring that it caches the most recent version.

2.2 Cache Coherence

CMPs implement a cache coherence protocol to ensure that private caches remain coherent with each other as well as with the shared memory. This is achieved by maintaining a special set of tags for every cache line present in the memory. These track the status of cache lines and serve to indicate if a line is valid, is shared by other caches, or has been updated by another processor in the CMP. To maintain these tags, each cache actively listens for cache line address traffic on the interconnect to determine if the data it is caching has been modified by another core, and subsequently invalidates its copy of the modified line. This is referred to as *snooping*. However in CMPs using a scalable network based interconnect where communication between a core and shared memory is not globally visible to other cores, *directory based coherence* is used. This method uses a central directory structure alongside the shared memory which tracks shared cache lines present in the private cache of each core. When shared lines are modified, the directory notifies other cores caching that line to take appropriate action - invalidate the line, or update the line with the new value. The number of tracking bits present with each cache line is determined by the coherence protocol in use. Commonly used protocols include MSI, MESI, MOSI and MOESI [5].

While cache coherence mechanisms ensure that caches always hold the most recent versions of data, their implementation results in increased traffic in the interconnect. This is especially true for CMPs with a large number of cores and large amounts of shared data.

2.3 Consistency

Writes to shared data by threads executing in parallel must be serialized and performed in the correct order, i.e. writes to shared data must be *mutually exclusive*. Locks provide this mutual exclusion by permitting only the thread that owns them to modify shared data. Threads that try to modify shared data while the lock is held by another thread are forced to wait until the lock is released. Several important issues arise as a result of this behavior of locks:

- Deadlock: Threads waiting to acquire a lock make no progress in their execution until the lock-holding thread completes its write. In the event that the thread fails

or enters an infinite loop, the lock is never released causing all waiting threads to deadlock.

- Priority Inversion: Higher-priority threads are forced to wait due to a low-priority thread holding the lock.
- Convoying: Threads waiting on a lock experience an increased waiting time if the lock-holding thread is descheduled.
- Scalability: Protecting data shared by a large number of cores with locks increases contention for writes, and places limits on the application's scalability.
- Complexity: Managing locks for multiple shared data over multiple threads increases complexity of the application code.

2.4 Transactional Memory

Transactional Memory was proposed as a replacement for performance degrading lock-based critical sections in shared memory multiprocessors by Herlihy and Moss in their 1993 paper [2]. They defined transactions as code segments that read or modify shared data in the system, representing the part of the thread executed immediately after acquisition of the lock. Transactions are executed in parallel based on the assumption that they are atomic and do not share data with any other transaction. This assumption is tested at the end of the transaction's execution. If the assumption is found valid, the transaction is allowed to commit its writes to the shared memory. However, if the assumption is found invalid, the transaction is restarted, rolling back all its previous writes.

The proposal defined two important properties of transactions:

- Serializability: The operations of every transaction are viewed independently from those of other transactions. Consequently, transactions appear to execute in a sequential order.
- Atomicity: Operations within a transaction either complete fully, or appear as if they were never performed. The latter case results from a violation of the serializability property by a transaction.

Data read by a transaction constitutes its *read-set* while the data written by it constitutes the *write-set*. Collectively, these are referred to as the transaction's *data set*. Loads and stores during execution of a transaction occur according to a strict protocol, replacing the need for locks during shared data accesses.

1. After performing a load from a shared location, a *validation* is performed to determine if the transaction is valid, and if the read data is consistent. If the validation fails, the transaction restarts execution
2. After completing all stores to a shared location, the transaction attempts to commit its writes to the shared memory. Other transactions executing in parallel may

thwart this attempt to commit if the transaction's writes violate the sequential write order. Consequently, the transaction aborts, and restarts execution. If the transaction succeeds, its writes are made global and appear in the order in which they occurred

Herlihy and Moss used custom hardware in each core to implement such a protocol. An additional transactional cache was added to store the transaction's speculative writes before commit. As writes are performed, their addresses are put onto the system bus where they are snooped by other cores to determine if the write causes a violation. Upon detection of a violation, the transaction is forced to clear its transactional cache and restart. However, if no violations are detected, the transaction completes its writes by committing them to the non-transactional private cache. The system was found to provide considerable performance gains while executing transactional applications when compared to their lock-based counterparts.

2.4.1 Types of Transactional Memory Systems

Herlihy and Moss's TM implementation represents a *Hardware Transactional Memory (HTM)* system, utilizing specialized hardware to enable transactional operations in private caches. The primary disadvantages of HTMs lie in this additional hardware itself. Firstly, since implementation of a HTM system requires modifications to private caches in cores, design complexity as well as hardware cost is increased. Secondly, the fixed nature of these additional hardware resources imposes limitations on the size of transactions that may execute on cores. Consequently, HTMs require programmers to understand how transactions execute on cores in the underlying system, thus increasing program complexity.

Transactional Coherence and Consistency (TCC) [7] was an influential proposal for HTMs, advocating the replacement of cache coherence protocols in shared memory transactional CMPs. Transactions are executed speculatively on different cores in parallel, with their caches maintained in the coherent state by TCC, with the assumption that the transactions are independent of each other. Similar to the first TM proposal, TCC isolates the data sets of executing transactions until commit time, when transactions arbitrate for commit permission and make their write-sets globally visible. An executing transaction's writes are buffered by TCC hardware present in each core. At commit time, each of these writes is made global, and committed to the shared memory. Transactions are allowed to commit one at a time, according to the program's sequential execution order. The TCC system was shown to guarantee correctness of execution, while maintaining coherence between caches and consistency of shared data. In addition, the system was shown to scale well with moderate overheads for validation and commit operations in simulations.

However, most of TCCs operations were based on the shared system bus. For instance, each core maintains an active snapshot of the progress of other transactions in the system by snooping the shared bus for transaction updates. Maintaining an active snapshot of all executing transactions will result in increased broadcast traffic on the bus, increasing contention. Further, the bus arbitration and transfer latency is assumed to be 3-cycles, a rather low figure for a CMP. These issues are further

examined in Section 2.4.2.

Other proposals include *Virtual Transactional Memory (VTM)* [8] which provides transactions access to a virtual memory space, enabling them to continue execution inspite of eviction of transactional data-set elements and overflow of transactional hardware resources. *Large Transactional Memory (LTM)*[9] offers another alternative to transactions that have exceeded their transactional hardware resources by storing their speculative writes into an in-memory hash table. However, the conflict detection mechanism for such an implementation is complex since the conflict detector must scan both the hash table as well as the core's private cache in order to detect conflicts.

Software Transactional Memory (STM) systems were proposed to overcome the costs and limitations of available HTMs. STMs implement all transactional operations including validation and commit in software itself. Shavit and Touitou in 1995 proposed the first software transactional memory system [Shavit and Touitou], comparing it with Herlihy and Moss's HTM. Their STM implementation reduced the penalty incurred on restarts, which in HTMs depends on the size of the transaction's data set. This penalty was observed constant for STMs allowing them to scale to large processor counts without losing performance. Several other proposals followed Shavit and Touitou over the years claiming easy programmability and increased scalability [10], [11], [12]. However, the 2008 paper by Cascaval et al. weighed such claims against the overheads incurred in supporting them [13]. Analyzing the performance and overheads of various STMs, the paper concluded that the overheads incurred in supporting transactional memory operations simply exceeded the obtained speed up. This was attributed to the software implementation of transactional primitives which results in tens of extra instructions being executed after every load and store in order to provide conflict detection.

In a best of both worlds approach, Hybrid Transactional Memory (HyTM) was proposed by Damron et al. as a combination of STM and HTM [14]. The proposal primarily uses an STM system, with transactions managed in software. However, HyTM allows transactions within resource limits to execute on an underlying HTM system, if available. This solves two issues characteristic to HTMs and STMs:

1. Performance: The overheads of STM can be mitigated by executing suitable transactions on the HTM system

2. Resource Limitations: The limitations imposed by the fixed size of transactional hardware are overcome by providing STM execution for transactions that exceed the HTM's resources

Although designed as a hybrid, HyTM is primarily an STM system with support for HTM when available. Baugh et al. in [15] present a similar hybrid architecture, however, extend HyTM to include a best-effort HTM system similar to Herlihy and Moss's 1993 proposal. Other proposals such as [16], [17], [18] suggest similar architectures with appropriate features to address specific issues from the earlier HyTM proposals. The general consensus from these proposals is that while transactional execution in HyTMs does not explicitly depend on hardware, the presence of a low commit-overhead HTM may significantly improve performance.

2.4.2 Transactional CMPs

Although over ten different TM systems have been proposed since the first in 1993, only two actual chip-multiprocessors have resulted from the research. These are: Sun Rock and Atlas.

2.4.2.1 Rock

The first and only commercial implementation of transactional memory was Sun's Rock processor. The multithreaded Rock consisted of 16 cores based on the SPARC V9 instruction set architecture, with the ability to run upto two threads on each. The specifications of Rock are listed in Table 2.1.

Table 2.1: Rock specifications

Cores	16 SPARC V9 cores at 2.1GHz 8×32KB 4-way L1-D cache 4×32KB 4-way L1-I cache 4×512KB 8-way L2-D cache 16MB (off chip)
Gate count	5.5 Million

Cores are arranged in clusters of four in order to effectively share on-chip memory resources. Each cluster contains two L1 data caches shared by two cores. Further, clusters are interconnected over a crossbar, and also to four banks of shared L2 data caches. The memory management unit and the system interface are connected over the same crossbar.

Rock enables hardware transactional memory operations by implementing a checkpoint based architecture. During execution, an architectural checkpoint is captured at the start of a transaction to begin speculative execution, and at the end, if the speculation is found to be valid, the speculative state is converted into the current architectural state. To enable transactional programming, Rock includes two custom instructions: *checkpoint* and *commit*. The checkpoint instruction is used to indicate the start of a transaction and also specify a *fail_pc* address. The speculative write-set of transactions is buffered in a local store buffer similar to TCC. Since transactions must validate their write-set before committing, write-set addresses are forwarded to Rock's directory which performs conflict detection. In the event that validation succeeds, the contents of the local store buffer are drained into the L2 data cache. However, should the validation fail, execution continues from the address indicated in the *fail-pc* attribute of the checkpoint instruction.

Dice et al. experimented with early versions of the HTM implementation in Rock, and published an account in [19]. It was observed that even though some benchmarks showed speed up with transactional execution, others showed a slow down, i.e. increased run time when compared to uniprocessor execution. In addition, reported scalability for applications was lower than expected. Dice's paper provided much needed insight

into the functioning of HTMs in an actual hardware implementation. However, rather than address the issues raised by Dice, Sun cancelled the Rock project.

Although Rock's HTM system was the first hardware implementation of TM, it possessed certain limitations along with the processor itself.

1. The local store buffer was small at only 32 cache lines, placing serious limitations on the data-set size of transactions

2. The *best-effort* HTM system implemented did not take into account the interrupt and IO capabilities of Rock. As a consequence, transactions were failed on interrupts, exceptions, and any *difficult* situations occurring in the system.

3. The implemented crossbar is not the most scalable of interconnect architectures. With increasing processor count, wire length to the central switch increases, thus increasing latency. Further, with additional cores the area of the crossbar itself increases, along with that of the overall die.

4. The HTM implementation was designed for the Rock alone, and is not portable to other processor architectures

2.4.2.2 Atlas

Atlas forms a full system prototype for the TCC HTM architecture with eight embedded PowerPC cores in a shared memory transactional CMP. The design is implemented on a multi-FPGA (Field Programmable Gate Array) BEE2 board consisting of a single control FPGA and four user FPGAs [20]. Fig. 2.1 shows an overview of the Atlas CMP.

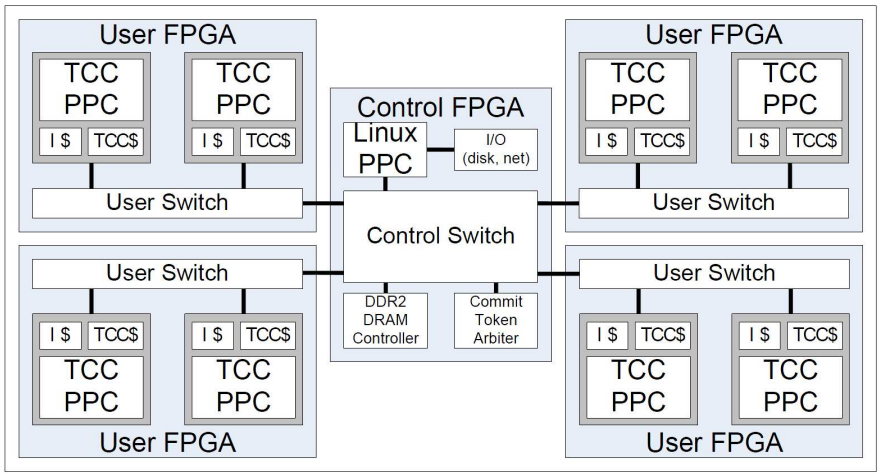


Figure 2.1: Atlas CMP

Each user FPGA in Atlas contains two hard PowerPC 405 cores with a custom synthesized transactional cache. The control FPGA contains a single PowerPC core that handles the OS and input/output operations. The five FPGAs are connected in a

star topology to a central switch which handles all control and data transfers between cores.

The specifications of the Atlas CMP are listed in Table 2.2.

Table 2.2: ATLAS Specifications

Cores	9 IBM PowerPC 405 cores at 100MHz 16KB 2-way I-cache 32KB 4-way Transactional D-cache 16KB 2-way D-cache (Control CPU)
Main Memory	512MB DDR2 at 200MHz
User FPGA	Xilinx XC2VP70 17641 LUTs (26%) 212KB BRAMs (32%)
Control FPGA	Xilinx XC2VP70 16284 LUTs (24%) 66KB BRAMs (10%)
OS	Montavista Linux 3.1

Applications for the Atlas are partitioned into threads with each containing one or more transactions. It is subsequently compiled and linked with a library containing optimized assembly implementations of transaction control primitives. Additionally, Atlas also includes performance tuning features to log the number and causes of violations and overflows, allowing applications to be optimized for execution on the CMP.

While the Atlas does an excellent job at prototyping the TCC system, a few shortcomings are observed in its design approach.

1. The star topology interconnect shows limited scalability since increasing the number of processors in the system causes increased contention at the control switch. As a result of the increased network latency, validation and commit communications would incur a larger performance overhead.

2. Operations such as commit request, commit writes, and conflict-driven data-set invalidation are all performed using the Atlas library in software. Making such operations visible to the programmer exposes the underlying TM system and increases programming complexity.

3. Since the library is implemented as an optimized set of assembly instructions for each transaction control primitive, switching to a lighter embedded processor would entail rewriting the entire library. As a result, retargeting the Atlas for a different processor architecture is not simple.

4. The Atlas system itself only consists of the required transactional memory hardware for each CMP core. Since it uses a hard processor core, an ASIC implementation of the CMP would require the addition of processor core IPs, a suitable interconnect, and appropriate memory blocks. While this is considered as a limitation in the scope of this thesis, it must be noted that the Atlas project itself focuses on developing a platform for transactional memory research, and not one for CMP research as this work endeavours to.

2.5 Interconnect

A common trait to both the Rock and the Atlas was the limited scalability offered by their interconnects. Addition of cores onto such interconnects would increase network latency and serve as a performance bottleneck. Furthermore, ignoring their effect on the Rock's crossbar switch, these additional cores would significant increase the already high area of its die. While scalable networks-on-chip (NoC) [21][22] provide a solution to the interconnect issues faced in the Rock and the Atlas, a different approach is necessary to address the issue of scalability limits on account of area constraints.

As conventional networks-on-chip increase in size, the number of hops between nodes at opposite edges of the network increases, i.e. network latency increases, with network size increasing accordingly. In order to maintain the area of the network while still increasing the number of nodes it contains, *die-stacking* is used. This technology utilizes several dice stacked one above the other, interconnected with a vertical wire, known as the *Through Silicon Via (TSV)*. Fig. 2.2 illustrates this concept.

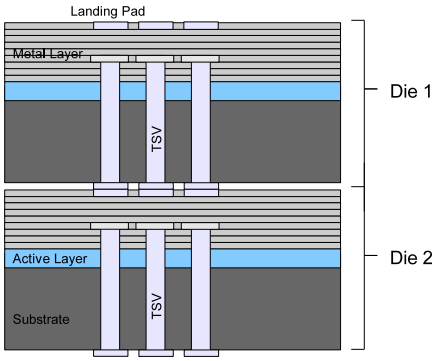


Figure 2.2: Die Stacking with Through Silicon Vias

In addition to decreasing the area footprint of the overall design, die-stacking reduces wire length in the vertical dimension. This is a consequence of the TSVs shorter height when compared to that of ordinary lateral wires [23][24][25][26]. In addition, TSVs also exhibit superior parasitic performance and thus reduce signal propagation delay [27].

Pavlidis and Friedman in [28] provided the earliest evidence of the packet latency and performance benefits of 3D NoCs. They evaluated 3D interconnects in terms of hops, and the zero-load network latency for different network configurations. They found that with scaling in the vertical dimension, average network latency followed the same increasing trend that is observed in scaling 2D networks. However, they noted that the increase in number of hops per transmission with scaling in the vertical dimension was considerably lower than that for 2D network. Additionally, with reduced channel length to nodes in adjacent layers, the average packet latency was reduced. These findings highlight the superiority of scaling in the vertical dimension over scaling in a single plane.

Further to the work by Pavlidis, Weldezion et al. in [29] explored the scalability of

3D buffer-less networks-on-chip, analyzing the performance benefits over 2D meshes. 3D mesh and bus based networks were compared against a 2D mesh to determine the increase in latency associated with adding network nodes. In general, their work served to highlight the benefits of scaling in the vertical dimension. However, it did not explore the penalties incurred in achieving such scaling.

Patti in [30] discussed the processes involved in fabricating TSVs in a real stacked system-on-chip. It demonstrated the ease of integrating TSVs into standard digital designs, and demonstrated the potential for partitioning systems in the vertical dimension, allowing for dice from different processes to be integrated within a single system. However, a methodology for use of TSVs and details on their integration were only provided by Loi et al. in [31] who describe a design flow for 3D NoCs using TSVs. They present an electrical model for TSVs based on extracted parasitics, and describe an implementation of a TSV based 3D NoC. ASIC processes use a fixed supply voltage at each technology node. Loi et al. do not specify a process technology making it difficult to compare the TSV size with interconnect dimensions and area metrics for the implemented logic. Additionally, process technologies specify a minimum wire pitch to mitigate the effects of capacitive coupling between adjacent wires, and this is implemented as a keep-out area in TSVs to prevent signal nets from being routed in close proximity. The authors again do not specify such a keep-out area for their TSVs, and only mention a fixed TSV pitch. The actual area overhead incurred by the use of TSVs can be accurately determined only by factoring in the keep-out area during floorplanning.

Apart from the 3D NoC itself, router architectures utilizing TSVs were investigated in MIRA [32] and Picoserver [33]. Park et al. presented a 3D router architecture decomposed into blocks and placed on separate layers in a stacked configuration. MIRA represented the first such architecture spanning several layers in a stack. However, inspite of its design complexity, MIRA performed only marginally better than a baseline 3D network in terms of hop-count and latency. Additionally, MIRA assumed that processor cores themselves are decomposed into multiple layers. Such processor cores do not find their way into the mainstream very often due to the relative infancy of the design flow for such decomposition over multiple layers. This effectively limits the potential application of MIRA as a 3D interconnect architecture for CMPs.

Picoserver on the other hand was designed for CMPs in a stacked configuration. It used a shared bus architecture composed primarily of TSVs to enable communication between cores and shared memory in the CMP. However, its primary limitation lies in its scalability since it uses a bus. The authors note that system performance would saturate at 16 cores with their bus architecture.

3 TMFab System Overview

Chapter 2 studied the principles of transactional memory, and examined the different approaches proposed for its implementation, highlighting the limitations of existing transactional chip multiprocessors - Atlas and Sun's ROCK. This chapter presents a system-level overview of TMFab, and describes the transactional memory policy implemented in the fabric.

3.1 Overview of TMFab

The transactional memory proposals have traditionally focused on developing TM systems and policies closely tied to a single processor architecture such as the x86 and PowerPC. This was observed in both the Atlas and Rock. Furthermore, most proposed TM implementations proposed features which are trivial in bus based systems, but incur large communication overheads in scalable interconnects.

TMFab was conceived with the intent of providing a means to prototype transactional chip multiprocessors using any suitable processor cores. The primary goal of such a fabric was the reduction in design effort required to develop scalable CMPs. From the background study performed in Chapter 2, it was observed that transactional memory proposals have traditionally focussed on developing TM systems and policies closely tied to a single processor architecture, a fact evidenced by both the Atlas and the Rock. Furthermore, the proposed HTM implementations utilized features that are relatively trivial to perform in bus based systems, however, incur large commuication overheads in scalable interconects. It must also be reiterated that inspite of an abundance of TM proposals over the last decade, the lack of hardware implementations have resulted in a dearth of realistic performance estimates and system characterization.

From these observations, we arrive at a few key requirements for TMFab. These include

- Processor architecture independence
- Scalability
- Synthesizability

Deploying soft-processor cores in such a fabric would require minimal modifications to the underlying transactional memory system, and reduce the design effort involved in prototyping transactional chip multiprocessors. With these requirements in mind, a novel stackable hardware transactional memory fabric was designed, as illustrated in Fig.3.1.

The TMFab transactional memory fabric is intended to work under a non-transaction supervising processor (SP). Traditional non-transactional applications and

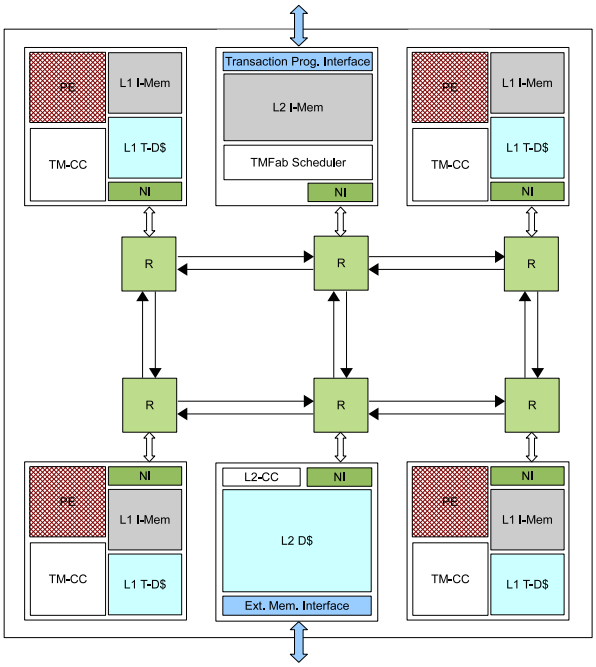


Figure 3.1: TMFab Transactional Memory Fabric

the operating system execute on the SP while transactional applications execute on PEs[1] across TMFab.

A network-on-chip based interconnect architecture was chosen on account of its excellent scalability when compared to bus and crossbar based architectures [34]. NoCs are composed of routers interconnected with fixed-length links to form a highly structured network. Fixed-length links regularize wire delays between routers, allowing the interconnect to operate at higher frequencies, and reducing the effort required in achieving timing closure. Function units such as processing elements and memory blocks are placed inside tiles, and connected to the network by means of a *Network Interface(NI)*. The TMFab network is organized as a mesh for simplicity.

Furthermore, the TMFab network-on-chip was designed as a three dimensional interconnect, i.e. it interconnects tiles not only in the X- and Y-planes, but also in the Z-plane through vertical ports at each router employing advanced Through Silicon Vias (TSV). These vertical links are established when dies are stacked one above the other, with each die containing a mesh of routers, as illustrated in Fig.3.2. This effectively results in a three dimensional network spanning several dies in the stack, with an overall area footprint of a single die. Data is transferred across the network as packets and is delivered to a tile based on the destination address in the packet header. The TMFab network architecture is examined in detail in Chapter 4.

The TMFab network contains three types of tiles: Scheduler, PE and L2 Data

[1]Processor cores deployed in TMFab are referred to as Processing Elements (PE)

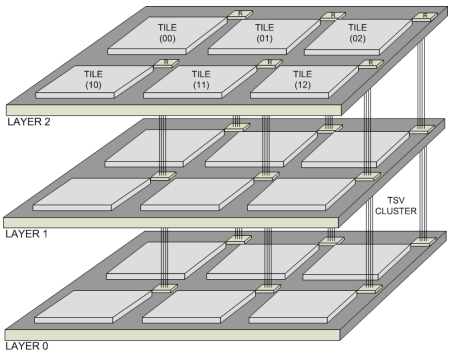


Figure 3.2: TSV based vertical links with die-stacking

Cache. When a transactional application is encountered by the SP, it is transferred to the *TMFab Scheduler (TMS)* tile through a dedicated *Transaction Programming Interface (TPI)*. These transactions are stored in the L2 Instruction Memory inside the Scheduler tile. Transactions are demarcated by transactional markers indicating the start and end of transactional code segments, as well as the transaction's logical position in the sequential execution order of the program. TMS assumes all transactions to be atomic and independent of each other, and schedules them for speculative execution on its PEs. Scheduled transactions are transferred to their assigned PE over the network, where they begin execution.

Loads during a transaction's execution are serviced over the three-level memory hierarchy of TMFab, starting with the L1 Transactional Data Cache (L1-D). This cache is augmented with transactional fields to track speculative reads and writes to cached data. TMFab's Transactional Cache Controllers buffer all of the transaction's speculative writes in a local buffer until the transaction completes. In the event of a write, the L1-D registers the local buffer address where the speculative write is temporarily stored in order for subsequent reads to access the data.

If the data requested by the transaction is not found in the L1-D or the local buffer, the request is forwarded to the larger, shared L2 Data Cache. This cache is non-transactional, and only tracks if the data it is caching is valid, or has been modified. In the event that the requested data is not found in the L2-D, an L2 miss is registered, and the data is fetched from the slower external memory over the L2 data cache controller's external memory interface. Retrieved data is forwarded back to the requesting PE. TMS monitors the progress of all transactions executing on PEs within TMFab, and determines which transaction should be allowed to commit its write-set to the L2-D. It implements a three-stage commit sequence - Initiate, Validate and Commit, wherein, upon completing execution, a transaction first initiates a validation request to the TMS. Validation is managed using a *validation token*, which serves as validate permission. Once permitted to validate, the transaction transmits its read and write-set addresses to other active transactions in the system, and waits for their response. If the transaction's reads and writes are legal, and causes no violations, the contents of its local buffer are merged with the L1-D, and all its writes are committed to the L2. In the event of a conflict arising due to the intersection of a committing transaction's

write-set with that of an executing transaction, one of the two is forced to abort and restart execution according to the implemented *Contention Management (CM)* policy. TMFab implements the *Oldest* CM policy [7] with *Aggressive retry* which forces the younger transaction to abort in the event of a conflict, and restart immediately. The combination of Oldest CM and aggressive retry may result in wasted work should repeated conflicts occur, however, this issue is examined more closely in Section 3.2.4.

The next section examines the system-level transactional memory policy that defines version management, contention management and other policies for the TMFab system.

3.2 System-level Transactional Memory Policy

The system-level TM policy is described as a collection of sub-policies for the TMFab transactional memory system. It defines the system protocol for transaction versioning and version management, contention management, conflict detection and cache coherence within TMFab. Each policy is examined separately.

3.2.1 Transaction Programming

Transactional programming for the TMFab differs from the conventional methodology in a number of ways. In the conventional approach, application code is divided into transactional threads that execute in parallel on different PEs. Critical sections are demarcated by atomic transactions that must validate their write-set before it becomes globally visible to other transactions. However, this means that non-transactional program code is executed before the critical section is entered, necessitating the use of register checkpointing during transaction restart. The TMFab approach considers transactional threads of the conventional method themselves as atomic transactions. Therefore, on transaction restart, execution simply begins from the first instruction of the transactional thread, implying that each transaction assumes execution to start with an empty register set. While eliminating the need for register checkpointing, this approach adds the overhead of each transaction needing its own initialization code at the start of execution. Since the entire thread is considered as an atomic transaction, this initialization sequence is also considered as part of the transaction. This marks a significant trade-off in the design process, where support for transactional code outside the transaction's atomic section was dropped in favor of a simpler programming interface, thereby decreasing the complexity of partitioning code into transactions.

Application code for TMFab is therefore divided into coarse grain transactions, each demarcated by the transactional markers *START_TXN* and *END_TXN*, with no program code outside of the transaction markers. To ensure correct execution, shared memory operations are ordered as they occur during sequential execution. This determines the commit order [35]. Therefore, the position of a transaction in the program's sequential execution order is indicated during programming by means of a 12-bit *sequence* and *phase* field. Although TMFab assumes all transactions to be independent, it forces them to validate their write-set before committing it to the shared L2-D. However, if a transaction is guaranteed by the programmer to be independent, with no dependencies with other transactions, it can be allowed to commit to the shared

L2-D without validating its write-set against other transactions. The transaction will nonetheless have to request for the commit token from the scheduler, which will allow the transaction to commit only if no other commit is in progress.

In order to do this, the transaction is marked with a different 4-bit phase code than the default of *0000*. Each different phase may contain only one guaranteed atomic transaction sequence for it to commit without validation. Multiple sequences within each phase result in the transaction requiring validation against other active transactions in the phase. It must be re-iterated here that transactions marked with different phase codes must be guaranteed to be independent, with no dependencies with transactions in other phases, as this would result in conflicts passing undetected causing loss of correctness in program execution.

Transactions within a phase, including the default, are marked with an 8-bit sequence code that describes their position in the global commit order. The sequence and phase codes are included in the *START_TXN* marker which marks the beginning of the transaction. The *END_TXN* marker contains no transaction information, and simply marks the end of the transaction.

TMFab offers a best effort correctness guarantee to applications. This means that TMFab guarantees the correctness of a transactional application's results if its transactions do not differ significantly in length, and in the sizes of their read/write sets. Applications that deviate from this criterion may require additional tuning to ensure correct results.

3.2.2 Version Management

TMFab implements *lazy version management* [36], which means that a transaction's speculative writes are held in a local write-buffer until commit time, at which point they are copied into the L1-D and committed to the L2-D. With *eager version management* the transaction's writes are performed directly to the L1-D itself, thus allowing for fast commits. However, it incurs a higher per-write latency than lazy version management due to its need to save the old value of data in an *undo log* before writing the new value in place. Lazy version management on the other hand results in fast aborts since the old values are retained in the L1-D, while the local write-buffer can be flash cleared on abort. It however results in slow commits since the contents of the write-buffer need to be merged with the L1-D. TMFab overlaps the merging operation with the commit to L2-D in order to reduce the overhead incurred during commits.

3.2.3 Conflict Detection

Transactions that have completed execution validate their write-set against other active transactions in the system. Executing transactions that detect no conflicts with the validating transaction's write-set permit it to commit by responding with an *acknowledgment (ACK)*. If a conflict is detected and the validating transaction is determined to be the loser in the contention according to the CM policy, the executing transaction responds with a *negative-acknowledgement (NACK)*. This method of detecting conflicts at commit time refers to *optimistic conflict detection*, as opposed to the *pessimistic* method of detecting conflicts at each memory access.

The primary drawback of delayed conflict detection with the optimistic method is that contentious data that would have otherwise registered a conflict in the early stages of execution with the pessimistic method now remains unchecked until the transaction begins validation, at which point a conflict is detected forcing the transaction to abort and restart. The pessimistic method, however, increases traffic in the interconnect by broadcasting all memory operations inside a transaction to other active transactions, increasing the average communication latency and thus impacting the overall performance of the system. Although these broadcasts occur in the optimistic method as well, they do not occur for every memory operation, but instead in a burst during the validation phase of the transaction. Additionally, the communication overhead involved in performing validations is considerably lower than per-access broadcasts, as examined in Chapter 4. Conflicts are detected at word granularity, preventing false conflicts from being detected when two transactions modify different data words in the same cache line.

For these reasons, the Optimistic method with word granularity was implemented as TMFab's conflict detection policy.

3.2.4 Contention Management

Contention Management (CM) refers to the policy used in determining which transaction aborts in the event of a conflict. TMFab implements the Oldest CM policy which causes the transaction with the larger sequence code, i.e. the younger transaction, to abort and restart upon detection of a conflict. Since conflicts are detected lazily during validation of the transaction's write-set, the CM policy effectively determines whether it is the validating transaction, or the executing transaction that is forced to abort and restart when a conflict is detected.

The CM policy must be specified along with a retry policy that defines how long an aborted transaction should wait before it can restart. In *back-off* based policies, the aborted transaction waits for a period of time before it restarts, where the period of time involved may be fixed, random, linear or exponentially increasing. The *aggressive* policy on the other hand, restarts the aborted transaction immediately. However, if the transaction is small, it may abort once again if restarted immediately, resulting in wasted work. The effects of this drawback can be reduced by ensuring that transactions are similarly sized, causing the aborted transaction to restart fewer times. TMFab uses a modified version of the aggressive policy, with two different restart strategies for transactions. The first causes transactions that are forced to abort and restart due to a conflict during validation of their write-set to be restarted immediately. This strategy was used since the TMFab interconnect requires tens of cycles to transfer the validation and validation response packets between PEs. Additionally, the executing transaction waits till the entire validation packet has been received before responding with an ACK or a NACK to the transaction committing. These latencies effectively induce a delay between the detection of a conflict at a remote executing transaction, and the actual restart of the validating transaction, providing an inherent back-off with the aggressive policy itself. The second strategy causes executing transactions that are forced to abort and restart due to a conflict with a validating transaction to wait until the entire incoming validation packet is received, before aborting and restarting.

However, the ACK for the validation is sent as soon as the local abort causing conflict is detected. The conflicting data is invalidated upon detection of the conflict, and therefore, when the transaction restarts, the invalidated data will be fetched from the L2-D. The delay between detection of the conflict and the local abort is used to delay restart in order to allow the conflicting data to be updated in the L2-D by the committing transaction. Further, delaying restart until completion of validation allows the local L1-D to invalidate all cache-line addresses specified in the validation packet thus ensuring that the cache remains coherent.

3.2.5 Validate and Commit Contention

TMFab does not implement an explicit contention management policy for issue of validate and commit tokens for three reasons. First, since all transactions are assumed to be independent, delaying the issue of the validate token for transactions that have completed early results in serialization of execution, with completed transactions forming a convoy waiting to acquire the token. This effectively converts the "*All transactions are independent*" assumption into "*No transaction is independent*".

Second, since the interconnect inherently serializes all validation token requests, the scheduler can simply issue the token to the first request it receives, and buffer the remaining till the first completes. On the other hand, with an explicitly ordered sequence for validation token grants, transactions ready to commit are forced to wait for older transactions to complete execution and commit. Without any such ordering, transactions ready to commit are forced to wait only if another transaction is currently validating or committing its write-set, and not for the execution of older transactions to complete.

And third, by granting the validate token to the first transaction that requests it, the scheduler is freed from the conflict detection operation, allowing truly independent transactions that have completed execution to commit their writes to the L2-D without having to wait on other executing transactions. While this results in independent transactions being retired faster from the system, it may result in wasted work if the transaction tries to commit before an older transaction it shares a dependency with. This penalty may be avoided by reducing the number of dependencies that exist between transactions executing in parallel.

Commit contention does not occur since a transaction is only allowed to commit upon successful validation of its write-set. Since the validation-token is issued to the first requesting transaction, it is implicit that should the transaction complete validation successfully, it will commit to the L2-D atomically.

3.2.6 Cache Coherence Protocol

Updates to the L2-D by executing transactions are performed only when the transaction commits to memory. The nature of transactional memory operations thus ensures coherence between caches in TMFab's PEs.

The validation phase before commit causes write-set addresses of the committing transaction to become visible to other active transactions in the system. The validation packet thus serves as a coherence invalidation message between PEs. Based on

the outcome of validation, four possibilities exist for the L1-D cache of the executing transaction's PE. These possibilities illustrate the method by which L1-D caches are kept coherent with each other and the L2-D cache.

1. If the incoming validation causes the executing transaction to abort, the transaction's data set are invalidated. Additionally, all cache-lines in the local L1-D cache with addresses matching those in the validating transaction's write-set are invalidated as well. This is done because the executing transaction's dependence on the validating transaction has been detected, and therefore, its local L1-D cache must be refilled with data updated in the L2-D cache by the validating transaction during commit.

2. If the incoming validation causes a conflict and the CM policy determines the validating transaction to be the loser in the contention, no invalidations are performed in the executing transaction's L1-D cache.

3. If the incoming validation causes no conflicts, no dependencies exist between the validating and executing transactions. However, in the event that the validating transaction is older than the executing transaction, its write-set addresses are used to invalidate all matching cache lines in the executing transaction's L1-D cache to maintain coherence.

4. On the other hand, if the validating transaction is younger than the executing transaction, no such invalidations are performed. Instead, cache-lines in the executing transaction's L1-D cache are marked for a delayed invalidation at commit. Should the older transaction perform a read to an address that was modified by the younger transaction, it will see the old value and not the new updated value. This ensures that the younger transaction's updates do not affect the older transaction's execution, while still allowing the local L1-D cache to maintain coherence by tracking which updated cache-lines to fetch from the L2-D cache after commit.

It is important to mention here that possibilities 3 and 4 could result in undetected conflicts if the two involved conflicts differ significantly in their transaction length and read/write-set size. Incorrectly ordered writes therefore may be attributed to improperly load-balanced transactions, requiring the application code to be modified for correct execution.

4 Architecture

Based on the System-level Transactional Memory Policy, this chapter examines the architecture of TMFab in terms of its four key components: Scheduler, TM Cache Controller, L2 Data Cache and the scalable interconnect. Each of these components is described in terms of their architecture, and their function in the overall operation of the fabric.

4.1 TMFab Scheduler (TMS)

The performance of a TMFab based CMP depends on the design of the applications that are run on it. Most threaded applications utilize performance degrading lock-based synchronisation methods to ensure correctness and consistency in shared data. Transactional memory, however, replaces lock-based synchronisation with atomic code sections, allowing for the same correctness and consistency of shared data, without the associated cost in performance. The relative infancy of the transactional programming methodology, however, has meant lower acceptance and consequently fewer available real world transactional applications.

Therefore, the transactional fabric works under a non-transactional supervising processor (SP) which runs the operating system and user applications. When a transactional application is executed from within the operating system, the SP transfers the transactional code block to the TMFab Scheduler (TMS) which subsequently spawns the required number of transactions.

4.1.1 Scheduler Architecture

The TMFab Scheduler (TMS) architecture is shown in Fig.4.1

TMS is responsible for mapping transactions received from the Supervisor Processor onto available Processing Elements (PE) as well as managing validation and commit tokens within the system. Each of these functions is placed inside one of TMS's three primary modules: the Scheduler Core, Transaction Programming Interface (TPI) and the L2 Instruction Memory.

4.1.1.1 Transaction Programming Interface (TPI)

Transfers of transactional code blocks from the SP are performed through the Transaction Programming Interface. The signal descriptions of the TPI are listed in Table 4.1.

The SP initiates the transfer of a transactional code block by asserting the VALID_I signal of the TPI. Subsequently, 32-bit instructions are transferred through the DAT_I port to be written at ADDR_I of the L2 Instruction Memory at the rising edge of every

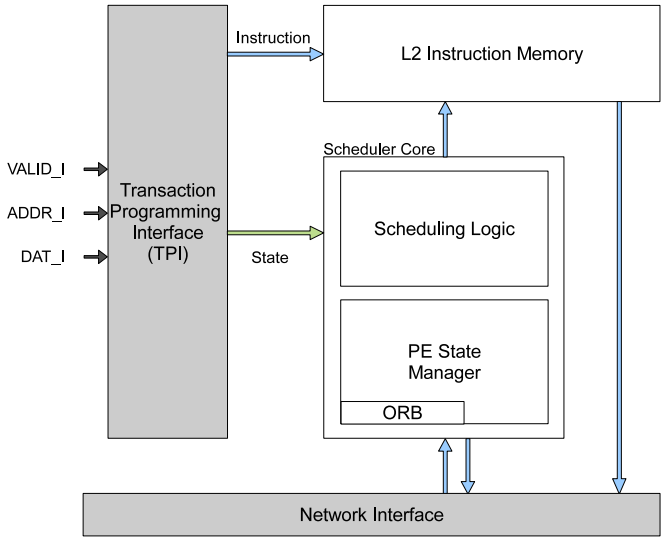


Figure 4.1: TMFab Scheduler Architecture

Table 4.1: TPI SIGNAL DESCRIPTIONS

Signal	Description	Type	Direction
VALID_I	Asserts validity of transfer	In	std_logic
ADDR_I [15:0]	Memory address for current instruction	In	std_logic
DAT_I [31:0]	32-bit instruction	In	std_logic_vector

clock cycle. Transactions in this instruction stream are demarcated by special transaction markers *START_TXN* and *END_TXN*, indicating the start and end instructions of the atomic transactional code segment, as illustrated in Fig.4.2. These markers are not stored in the instruction memory, and therefore, bear no relation to the instruction set architecture of PEs in the system. The format of these markers is listed in Fig.4.3.

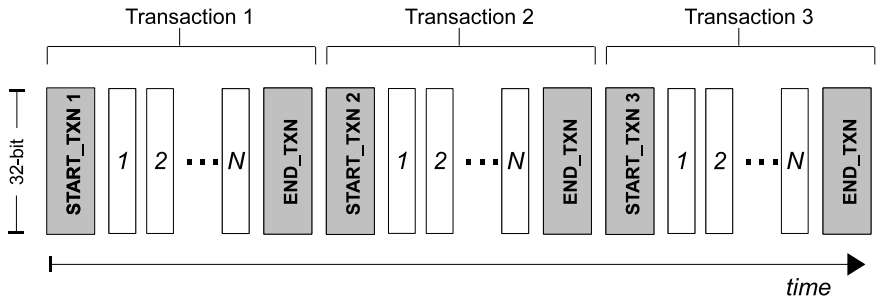


Figure 4.2: a. Demarcated transactional instruction stream, b.

The *SEQ* and *PHASE* fields of the START_TXN marker indicate the position of a transactional code segment in the overall sequential execution order of the program. The Scheduler uses these fields to schedule transactions on PEs as explained in Trans-

actional Programming in Section 3.2.1.

Upon receiving transactional application code through the TPI, the SEQ and PHASE fields of each transaction are registered, along with the L2 instruction memory addresses of its first and last instructions. Collectively, these form the transaction's State. Upon receiving the complete transaction code segment, the TPI forwards the transaction's recorded state to the Scheduler Core. The transaction may now be scheduled for execution on an available PE.

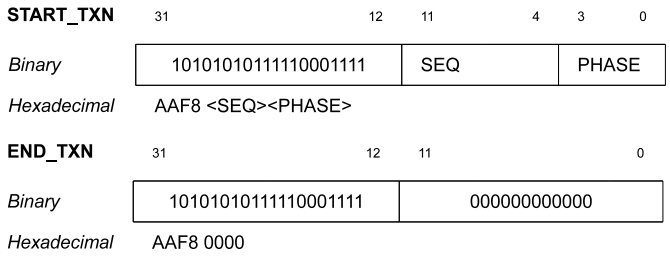


Figure 4.3: Format of transactional markers

4.1.1.2 Scheduler Core

The *Scheduler Core (SC)* is responsible for mapping new transactions onto available processing elements, and monitoring their status until they commit. Additionally, it manages the validation and commit tokens, and arbitrates between transactions contending to commit their write sets to memory. The SC is thus comprised of two parts: the Scheduling Logic and the PE State Management.

Scheduling Logic Transaction states received from the TPI are entered into a *transaction status table* containing state information for all transactions currently executing on the system as well as those awaiting scheduling. The received transaction is assigned a 10-bit *Transaction ID (TXN ID)*, and entered in the first available slot in the table.

A transaction may be scheduled for execution only when an idle PE is available. Upon being assigned a PE, the Scheduling Logic retrieves the transactional code segment from the L2 instruction memory using the addresses stored in the transaction state. Instructions from the code segment are transferred to the idle PE's private instruction memory over the fabric interconnect. The transaction state is subsequently transferred to the PE in order to commence execution.

An ideal interconnect offers single cycle communication latency between nodes. Since the fabric network interconnects several PEs and functional units, it offers a non-ideal multi-cycle communication latency. Therefore, even a 100-instruction code segment may require a few hundred clock cycles before it is delivered to the appropriate PE, which remains idle till the entire segment is received and stored in its L1 instruction memory. In effect, the overhead incurred in transferring code segments between the TMS and the PEs would degrade the system's performance.

Recalling the three-stage commit sequence from Section 3.1, we observe that after completing the validation phase without aborting, a transaction with a commit token will complete successfully and commit its write set to the L2. The PE for that transaction, therefore, will remain idle till its L1 instruction memory is loaded with the next transaction. Leveraging this, the TMS performs the code segment transfer for the next waiting transaction as soon as the transaction on a PE completes its validation phase successfully, and begins to commit its write set to the L2. This significantly reduces the incurred code segment transfer overhead, and its performance degrading effects. However, in doing this, the TMS must ensure that the committing transaction completes its commit operation before the next transaction can begin execution on the PE. This is achieved by halting the transfer of transaction's state, until the committing transaction notifies the TMS of completion. Since the PE cannot commence execution of a new transaction without receiving the transaction state from the TMS, completion of committing transactions is guaranteed.

Processing Element State Manager (PESM) The *PE Status Manager* tracks the status of individual processing elements, as well as the collective system state. Since the system-level TM policy only allows one active transaction in the system to validate and commit at a time, the PESM must monitor the status of all PEs within the fabric in order to effectively track the progress of transactions. The PESM uses validate and commit tokens to preserve proper transactional commit order. Processing elements may assume one of three states of operation: Idle, Busy and Committing. An idle PE transitions to the Busy state when a newly scheduled transaction is transferred to it for execution. Upon completion, this transaction must enter a validation phase to validate its write-sets against other active transactions in the system. In order to enter this phase, the PE forwards a validation token request to the PESM, which grants the token if no other transaction in the system is currently in the validation or commit phase.

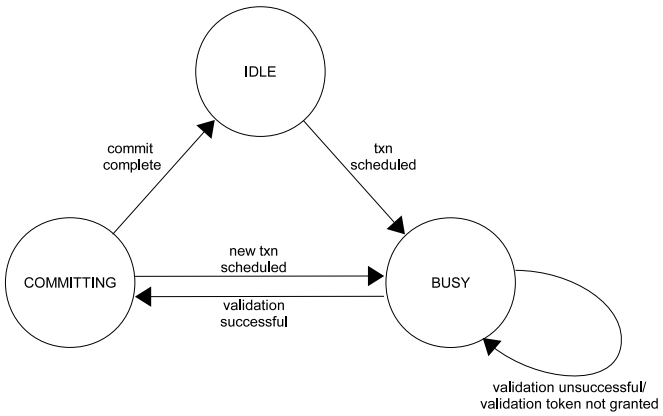


Figure 4.4: State transitions in the PESM

The PESM then switches to a waiting state, buffering incoming validation token requests from other active transactions in the *Outstanding Request Buffer (ORB)* until

the validating transaction completes its validation. In the event that the validation succeeds, the PESM is notified and the PEs status subsequently updated to Committing. PESM returns to the normal running state once the committing transaction has completed its commit. However, should the validation fail causing a local abort, the transaction restarts by returning the validation token to the PESM, which grants it to the next buffered token request. These state transitions are illustrated in the state diagram in Fig. 4.4. A summary of the transfers that occur between the PE and PESM is given in Table 4.2

Table 4.2: SUMMARY OF PE-PESM TRANSFERS

PE State	Type	Direction	Transaction Status
Idle	Transaction code segment transfer	PESM-PE	Scheduled
	Transaction state transfer	PESM-PE	Start Execution
Busy	Validation Token Request	PE-PESM	Execution complete
	Validation Token	PESM-PE	Validating
	Validation Successful	PE-PESM	Committing
	Validation Unsuccessful	PE-PESM	Restarting
Committing	Commit Complete	PE-PESM	Idle
	Transaction code segment transfer	PESM-PE	Scheduled

L2 Instruction Memory (L2-I) The L2 Instruction Memory stores the transactional code segments received through the TPI. Since the memory is written to by the TPI, and read by the Scheduler Core, it is implemented as a dual port memory, with separate read and write ports. From the circuit perspective, no additional logic is required to ensure that the read and write ports don't access the same memory locations. This is because instructions from a transactional code segment are read out by the Scheduler Core only once the entire transaction has been received, and its state recorded.

Transactional code segments contain all the program code the transaction requires for execution, and therefore instruction misses do not occur in PEs. Consequently, the L2 Instruction Memory is read from only when a transaction is scheduled for execution, and its size holds no significance to the performance of the application. In the TMFab design, it was sized in order to accommodate transactional applications with instruction counts of the order 10^4. Thus, it has a storage capacity of 256KB, organised as 8192 lines with 16 instructions of 32-bits per line ($8192 \times 16 \times 32$). The fabric interconnect which is examined later in this chapter, transfers transactional code segments in blocks of 16 instructions from the TMS to PEs, therefore, the line size of 16-instructions serves to reduce the latency in retrieving instruction blocks from the L2-I.

4.1.1.3 System Communications

The TMS engages in two primary types of communication with PEs in the fabric Transactional code segment transfer, and Validation-commit message transfers. These communications occur at specific phases in the transaction's lifetime, starting with code

segment transfer when the transactional application code is transferred into the L2-I, upto the final commit operation.

The communication type for each transfer is indicated in the packet header at the start of the transfer. However, since the interconnect itself supports only three major types of traffic, these communications must be augmented with special identifiers in order to allow the receiving network interface to determine the context of the communication. These identifiers are listed in Table 4.3. While all requests carry a communication identifier, communications to the scheduler are denoted by an additional scheduler operation identifier indicating the actual request or notification.

Table 4.3: COMMUNICATION IDENTIFIERS

Communication	**Classification**	**Comm. Identifier**	**Scheduler Operation**
Code segment transfer	Instruction Block Transfer	-	-
Transaction state transfer	Instruction Block Transfer	001	-
Validation token request	TM Communications	010	0001
Validation token granted	TM Communications	-	-
Committing	TM Communications	010	0100
Commit Complete	TM Communications	010	0011
Abort and Restart	TM Communications	010	0101
Partial validation token request	TM Communications	010	0010

Transactional Code Segment Transfer The start and end addresses of the transactional code segment specified by the stored transaction state information are used by the TMS to retrieve instructions from the L2-I. Immediately after a transaction is scheduled, its instructions are retrieved from the L2-I in blocks of 16 and transferred to the assigned PE. Transactional markers are not stored in the memory by the TPI, and therefore, they are not transferred to PEs.

This transfer belongs to the *Instruction Block Transfer* communication class of the fabric network and is not denoted by any special communication identifier. Such transfers occur only when waiting transactions are assigned to a PE in the fabric, and their duration depends on the instruction count of transactional code segments.

Transactional State Transfer Transactional code segment transfers are succeeded by a state transfer operation once the assigned PE assumes the Idle state. This transfer signals the start of operation, and consists of the scheduled transaction's ID, Sequence and Phase, along with an offset derived from the start and end memory address of the transaction indicating the position of the last instruction of the code segment. It is classified as part of the instruction block transfer communication class of the network, and is denoted by the communication identifier *001*.

Validation Token Request/Validation Token Granted Upon completion of execution at a PE, a request for the validation token is sent to the TMS. This type of transfer

is classified under the *TM Communications* class since it involves a transactional system request/notification, and carries the scheduler operation identifier *0001* indicating a validation token request. The TMS immediately responds to this request with a *Validation Token Granted* notification to the requesting PE if the PESM asserts that no other transaction is currently validating or committing. If the PESM indicates an on-going validation or commit operation, the token request is buffered, and the token grant notification is delayed till the operation completes.

The TMS does not perform any transfers to a PE after the initial code and transaction state transfers, and therefore the TM communications class notification from the scheduler in itself serves as the validation token grant, removing the need for any communications identifiers. Therefore the token grant notification does not bear a communications identifier.

Alongside serving as a notification to begin validation, the token grant also serves to update the requesting PE with a snapshot of the status of other PEs in the fabric. This is done by means of the core status field that indicates which other PEs are active, i.e. in the Busy state.

Committing/Restarting The PE notifies the TMS with the *Committing* notification if the validation operation is successful. If an unscheduled waiting transaction exists in the TMS, it is scheduled for execution on the PE when this notification arrives. However, if the validation is unsuccessful, the TMS is notified with a *Restarting* notification, and the transaction restarts. This notification is used only at the end of a validation, and not on conflict-driven restarts during execution.

Commit Complete Transactions that have completed committing their write-sets to the L2-D are no longer active in the system. Thus, the PE notifies the TMS of commit completion in order for the transaction to be retired and for the next scheduled transaction's state to be transferred to it.

Partial Validation Token Request Previously mentioned requests and notifications are performed atleast once in the lifetime of every transaction. However, an additional request is built into the TMS to allow the system to cope with transactions that overflow their PEs local write buffer. The *Partial Validation Token Request* allows PEs with overflowing transactions to request for a validation token before execution completes. Effectively, the transaction partially commits its write-set to the L2-D before resuming execution, in a manner similar to TCC [tcc]. Since the transaction already holds the partial validation token, once it completes, it immediately validates the remainder of its write-set, before committing it. During an overflow, the TMS withholds the issue of validation token grants to other transactions until the overflowing transaction fully commits, and is retired. This represents a corner case in applications, and leads to serialization of transactions, and possibly the loss of correctness.

4.2 TM Cache Controller

The architecture of the TM Cache Controller (TM-CC) is shown in Fig. 4.5. The PE block is shaded red to indicate that TMFab in itself does not provide processing elements, which must be instantiated separately in the fabric.

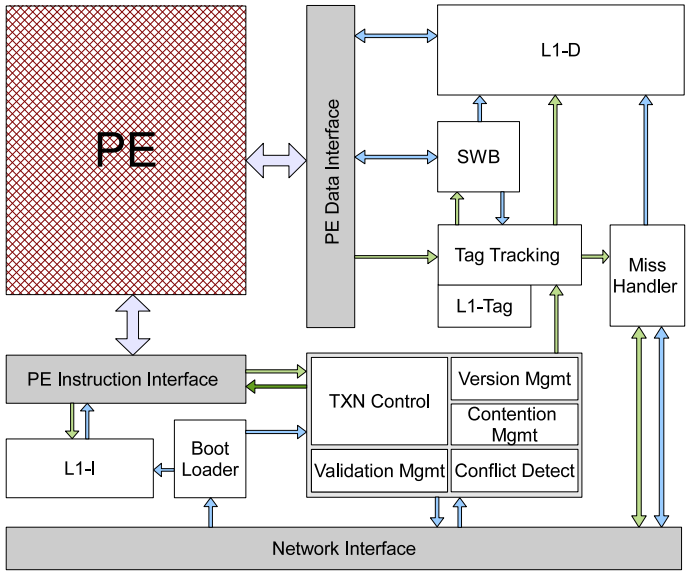


Figure 4.5: TM Cache Controller architecture

This section examines the architecture of the TM-CC in terms of its key components:

1. Bootloader
2. L1 Instruction Memory (L1-I)
3. L1 Data Cache (L1-D)
4. PE Instruction and Data Interfaces
5. Tag Unit
6. Speculative Write Buffer (SWB)
7. Transaction Control

- Conflict Detection and Contention Management
- Validation Management
- Version Management

4.2.1 Bootloader

Transactional code segments transferred from the TMS are stored in the local instruction memory for retrieval by the PE. Execution can begin only once the state is received, containing the transaction's ID, sequence and phase, and instruction memory addresses of the transaction's boundaries. These operations are managed by TM-CC's integrated bootloader.

The code segment is received in blocks of 16 instructions from the scheduler and stored sequentially in the L1-Instruction Memory. The bootloader manages memory addressing and records the local address of the first and the last instructions of the code segment as the operation proceeds. When the code segment transfer completes, the bootloader is switched into a waiting state, awaiting notification from the TMS.

The transaction is ready for execution when its code segment is loaded into the L1-I. Execution is initiated when transaction state data is received from the TMS and transferred to Transaction Control, marking the end of the code segment transfer phase, and transitioning the system into the execution phase.

4.2.2 L1 Instruction Memory (L1-I)

Similar to the L2 Instruction Memory, the size of the L1-I holds no significance to the performance of a transaction executing on the PE. Four executing transactions of equal length occupying the entire 256KB L2-I implies that each of the four PEs in the system executes a transaction with a 64KB code segment. The L1-I is thus structured as a single port memory of size 16384×32b, or 64KB.

4.2.3 L1 Data Cache (L1-D)

In a chip multiprocessor, the L2-D cache is generally shared by all processing elements in the system, and is accessible over the system interconnect. The shared nature of the interconnect implies a non-ideal communication latency resulting in accesses between the L1 Data cache and L2-D taking few tens of processor cycles before the requested data is made available to the PE. During this time, the PE is stalled, performing no useful work. Effectively, a performance penalty is incurred on every miss in the L1-D cache, and subsequent misses in the L2-D incur a larger penalty as the requested data must be fetched from the off-chip memory.

Unlike the L1-I, the L1-D cache influences performance of a transaction executing on the PE. The performance of a cache depends on the memory access characteristics of applications. Applications that read data randomly from the memory incur higher performance penalties on account of higher miss rates. On the other hand, applications that read data such as arrays incur a lower performance penalty due to a reduced miss rate. This is because data in arrays is stored at consecutive addresses in the memory, and since cache refills on misses in the L1-D consist of the entire cache line, misses occur only when the requested address targets a word in an uncached line. Therefore, to determine the optimal cache size for the system, a set of transactional applications from the Stanford Transactional Applications for Multi-Processing (STAMP) [37] benchmark suite was employed. The suite consists of eight transactional applications, of which four

were used on account of their varied memory access patterns. These include bayes, labyrinth, vacation-low and kmeans. With a memory trace driven cache simulator, different cache organizations were analyzed for this set of applications.

Since TMFab uses two levels of on-chip data cache, the optimal size for both the L1-D and the L2-D must be determined. The miss rate of the L1-D is not influenced by that of the L2-D, which was therefore fixed at 1MB during L1-D cache sizing. Cache simulations were performed for each application with a range of cache sizes, associativity and line sizes as listed in Table 4.4. Subsequently, the L1-D miss rate for each combination was computed.

Table 4.4: L1-D SIMULATION PARAMETERS

Cache Size	512B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB
Associativity	1-way (Direct), 2-way, 4-way, 8-way, Fully Associative
Line Size	16B(4 words), 32B(8 words), 64B(16 words)

High runtimes were encountered for simulations using 64B line size, especially in the case of the bayes and labyrinth applications. Thus, the simulation process was split into two parts. In the first part, line sizes of 16B and 32B were used with combinations of different cache size and associativity. The miss rates for the four applications are shown in Fig. 4.6.

As associativity increases, a decrease in miss rate is observed, stemming from the decrease in conflict misses in the cache. Since each set may hold more than one cache line, contention for sets is decreased thus reducing the number of evictions. As cache size increases, a greater decrease in miss rate is observed, although this decrease is due to the reduced occurrence of capacity misses on account of the larger number of sets available to store cache lines. Compulsory misses, however, remain unaffected by cache size, line size and associativity since they represent accesses to data uncached since the start of the transaction's execution. The effects of these types of misses are observed in the graphs in Fig. 4.6.

From the results in Fig.4.6, the cache size range was determined as between 16KB and 64KB, and associativity range as between 2-way and 8-way. The second part of the simulation used this determined range of cache sizes with a 64B line size. The miss rates for these simulations are shown in Fig. 4.7.

The number of memory references by each application is listed in Table 4.5.

The magnitude of these memory references highlights the importance of reducing miss rates in the L1-D cache. Fig.4.6 and 4.7 indicate a general decreasing trend in

Table 4.5: DATA REFERENCES PER APPLICATION

Application	**Data references**
bayes	1.172×10^9
labyrinth	2.6×10^7
vacation-low	9.3×10^7
kmeans	6.7×10^5

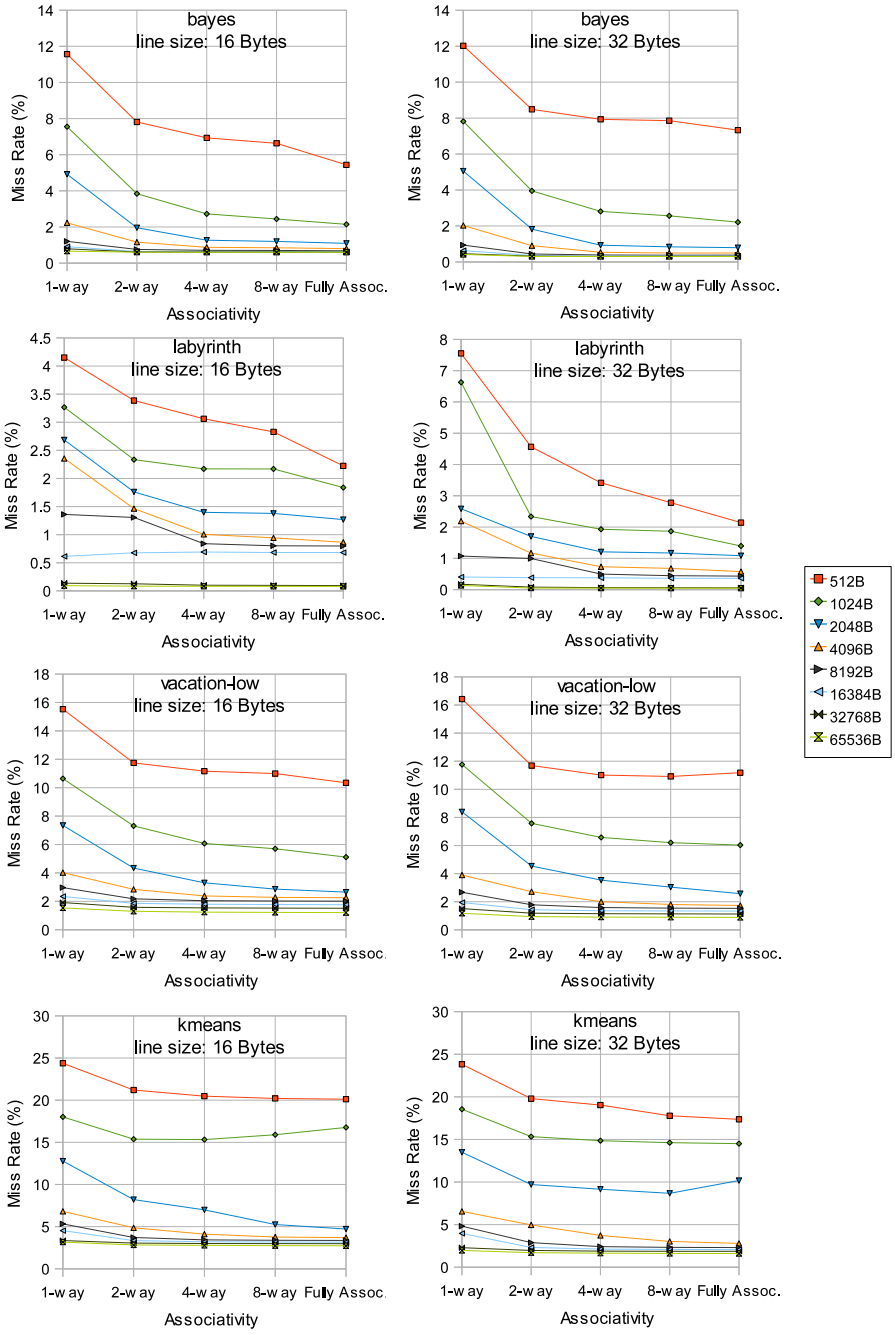


Figure 4.6: Miss rates for varying cache size and associativity (Line Size: 16 Bytes and 32 Bytes)

miss rates with increasing cache size and associativity. Even though the 8-way and fully associative caches are observed to be the best performers in terms of miss rate, they bear a high hardware cost. Set associative caches are divided into sets, each containing N cache lines, N being the associativity of the cache. During the tag lookup

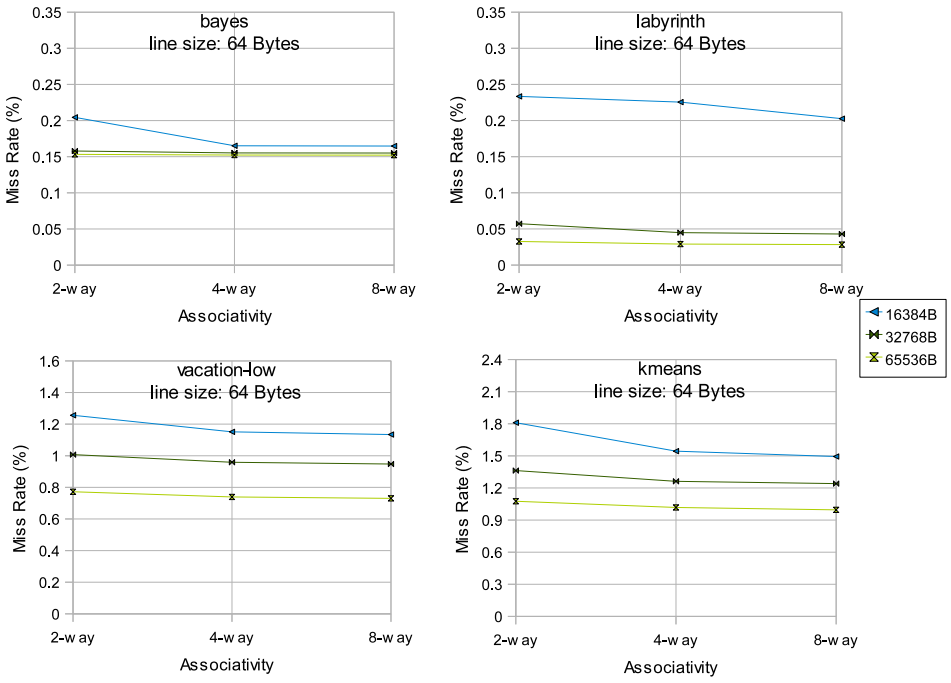


Figure 4.7: Miss rates for varying cache size and associativity (Line Size: 64 Bytes)

operation to locate the requested data, all N cache lines in the addressed set must be searched in parallel, thus requiring N comparators. Direct mapped caches have the lowest hardware cost as each cache line is mapped to a single address, eliminating the need for tag lookup. However, they have the highest miss rates in all combinations for all tested applications.

The combination of 64KB cache size and 64 byte line size resulted in the lowest miss rates in all simulations. While the miss rate for the same cache with 4-way associativity was low, no significant decrease in miss rates was observed upon increasing associativity to 8-way, indicating that the miss rate in this region is dominated by capacity misses that occur due to the limited size of the cache itself and not due to conflict misses arising from eviction of cache lines being accessed by the PE.

Consequently, the L1-D was organized as a 64KB, 4-way set-associate cache with a 64 byte line size. It is thus structured as a 256 set memory, with each set containing four cache lines of 64 bytes.

4.2.4 Tag Unit

A separate memory structure called the *Tag memory* is required to track locations of the L1-D that are actively caching cache lines. It stores this information in the form of tag data that describes the complete memory address of cache lines, their location in the L1-D, their validity, and whether they have been speculatively read or written by an executing transaction. When the PE performs a memory access, the requested

memory address is used by the tag memory to determine the location of the requested data in the L1-D. When the data is found, its address is forwarded to the L1-D which responds to the PE with the requested data. The structure of a tag memory set is shown in Fig 4.8.

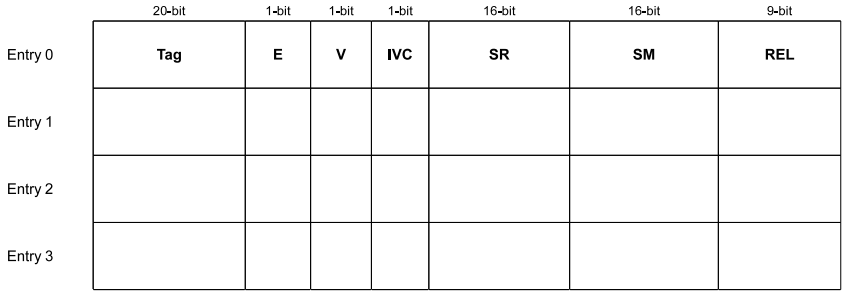


Figure 4.8: Structure of L1-Tag set

The tag memory is also a 256 set, 4-way set associative memory with each set containing four cache line entries. Each entry in the set contains 64 bits of tracking data for a cache line stored in the L1-D equivalent location. The tag memory thus has a total capacity of 8KB, and is implemented as a two port memory, with one port for read and write operations, and a second port for incoming validations and invalidating cache lines for coherence.

Every cache line stored in the L1-D is tracked in the Tag memory with a 20-bit identifier that corresponds to the most significant 20-bits of its memory address. This identifier is called the Tag and is used in determining if a requested cache line is in the cache, or must be fetched from the L2-D.

The Valid bit marks the validity of cache lines stored in the L1-D. When an updated version of the line exists in the L2-D, or when the writes to the cache line are invalidated, the valid bit for the line is reset.

The 16-bit SR field tracks speculative reads from data words in cache lines during transaction execution. However, in the event that the data has been speculatively modified, subsequent reads from that location result in the transaction using locally generated data, and thus the SR field is unmodified. Similarly, the SM field tracks speculative writes to data words in cache lines during transaction execution. Writes to memory during a transaction's execution are held in a local write buffer until commit, and therefore, cache lines containing speculatively written data, also contain a *Relocation Address (REL)* that indicates the location of the written data in the write buffer. The relocation address is provided by the write buffer during every write operation on a previously unmodified cache line. Because a cache line's relocation address must be maintained till the transaction commits, entries in a tag memory set with marked SM bits cannot be evicted from the cache. If the relocation address for a cache line is reset during execution, the tag memory no longer contains a record of the PE's speculative writes to the cache line. As a result, conflicts may pass undetected, and the PE's speculative writes may never reflect in the L2-D.

The IVC bit is used to mark cache lines for delayed invalidation. This is particularly useful when a cache line unused by the executing transaction needs to be invalidated by a younger committing transaction. The IVC bit marks cache lines for invalidation after the executing transaction has committed, ensuring that the cache remains coherent, and that the read-set of older transactions is protected from modification by newer transactions.

On memory read and write requests from the PE, the tag unit looks up the requested cache line in the addressed set of the tag memory by matching the 20-bit cache line tags in each entry of the set with the requested cache line tag. A miss is registered if no tags in the set match the requested cache line tag. On the other hand, a hit or miss may be registered when the requested cache line tag is found in the set. Each of these possibilites is described below.

Requested line is cached If a matching tag is found in the set, the requested cache line is present in the cache. However, the cache line's validity must be checked with the valid bit. The cache may contain stale copies of cache lines that have been updated by committing transactions. Validating transactions invalidate copies of modified cache lines present in other caches by means of a validation packet.

Accesses to an invalidated cache line result in the L1-D cache requesting the L2-D for a refill with an updated copy of the cache line. In effect, the L1-D registers a miss and the PE is stalled till the updated cache line is received from the L2-D.

Requested cache line is uncached If no tags are found matching the requested cache line, a miss is registered in the L1-D. The requested cache line must be fetched from the L2-D, and placed in one of the set's four entries. New cache lines are stored in empty entries of the set. However, if no empty entries exist, an unused cache line marked with the *Empty* bit (E) is evicted from the cache and replaced with the requested cache line, similar to the pseudo Least Recently Used (LRU) replacement policy [1]. Speculatively read and written lines are not candidates for eviction since their removal from the cache causes the loss of read and write-set tracking information, thus preventing conflicts from being detected. A special case may result from this limitation, where no empty entries are found in the set, and no candidates for replacement exist, i.e. all cache lines in the set have been speculatively read or written by the executing transaction. This condition is referred to as Overflow, and since the pending memory access cannot be completed till the requested cache line is available in the cache, execution stops and the PE is stalled.

During overflow, the TM-CC's transaction control obtains a *Partial Validation Token*, causing the entire write-set upto the point of validation to be validated, and if successful, committed to the L2-D. Conflicting cache lines in sets may now be replaced with the requested cache line. However, until this transaction completes execution and completely commits its write-set, no other transactions are allowed to commit, ensuring isolation of the overflowed transaction's writes. Although this scheme manages overflows in transactions, it may result in loss of correctness guarantees in the system in the

[1] Explained in Section 4.3.2

event that a conflict is detected after the partial commit, since at this stage, there is no way to roll back the overflowed transaction's writes to the L2-D. Therefore, overflows must be prevented in the software design itself by ensuring that data sets don't exceed the PEs transactional memory resources. Transactions operating on moderately sized sets of data are preferred over transactions that use very large data sets causing overflows. Additionally, if the overflowing transaction is coded free of dependencies with other transactions, correctness guarantees will remain in place since the transaction's writes to the L2-D may occur in any order with respect to other transactions.

4.2.5 Speculative Write Buffer (SWB)

During execution of a transaction, data writes are speculative since dependencies with read and write sets of other concurrently executing transactions remain unknown until validation. Therefore, speculative writes to data are buffered until commit, in accordance with the lazy version management policy. This is achieved using a *Speculative Write Buffer (SWB)*. The format of a single SWB entry is shown in Fig. 4.9.

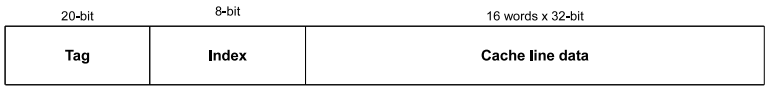


Figure 4.9: Cache line entry in the Speculative Write Buffer

The size of the SWB determines the maximum number of writes a transaction can perform to sequential memory addresses before overflow occurs. For the 64KB 4-way set associative L1 cache, assuming accesses occur sequentially starting from address *0x00000000*, contention for set entries occurs after it contains 1024 cache lines, after which the cache overflows since all cache lines have been speculatively read or written to. Assuming that memory accesses contain an equal number of reads and writes, write-set overflow would occur after 512 cache lines have been written. Thus the SWB was implemented as a 512 cache line deep buffer, i.e. 32KB. While a small buffer hampers performance with frequent overflows in applications with write-sets exceeding its capacity, a large buffer imposes a significant overhead in terms of area.

The SWB serves two purposes, necessitated by a lazy versioning transactional memory system. First, it isolates a transaction's speculative writes from the L1-D, and second, it serves as a log of all data speculatively modified during execution. The latter is used to create the validation packet when the transaction completes execution and obtains the validation token.

On the first write to a cache line, the SWB acts as a FIFO by placing the cache line address in the first empty slot of the buffer pointed at by the SWB pointer. Data is written into the buffer at word granularity, and therefore, only words modified by an executing transaction exist in cache line entries within the SWB. Upon adding this data to the buffer, the 9-bit relocation address (REL) is passed back to the tag memory indicating where the cache line is located in the buffer. For subsequent writes within

the same cache line, the SWB entry is selected using the Tag Unit supplied REL address instead of the SWB pointer. In order to access data that was previously written by the transaction, the Tag unit addresses the SWB entry with a REL address, and the block offset of the word requested by the transaction. Thus the SWB behaves like a random access memory during reads from buffered cache lines.

4.2.6 Transaction Control

The progress of an executing transaction is tracked at instruction granularity by the *Transaction Control*. Every instruction fetch address is compared with the address of the last instruction of the transactional code segment to determine if the transaction has completed execution. The TM-CC is accordingly maintained in one of the eight states of operation. These states are illustrated in Fig. 4.10

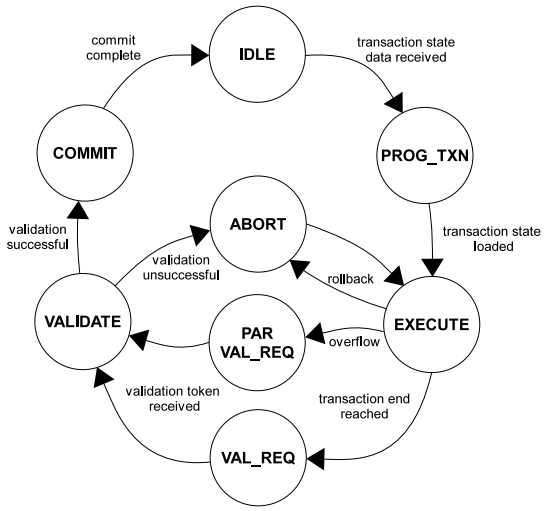


Figure 4.10: States of operation of the TM-CC

Fig. 4.10 illustrates the various states of operation, of which transaction execution forms one state. Execution begins only once the transaction state data is loaded in the *Transaction Control Registers*. As execution progresses, instruction fetch addresses are compared every cycle with the stored transactions state data to determine if the last instruction of the transaction has been reached.

Upon reaching the end of the transaction, the TM-CC transitions to the *VAL_REQ* state where the TMS is requested for a validation token, with this state being maintained until the token is received. Subsequently in the *VALIDATE* state, validation begins under the control of the *Validation Management (VaM)*.

During validation, VaM cycles through the SWB one entry at a time to determine the contents of the transaction's write-set. The SM field for the buffered cache lines is copied into a validation packet containing the cache line tag, and is sent to the network interface for transfer. The VM uses core status information received from the TMS in the validation token to determine which other PEs to send the validation packet to, thus directing validation packets only towards those with active transactions.

Should the validating transaction be the only active one in the fabric, the validation immediately succeeds and proceeds to commit. However, if other active transactions exist in the fabric, the validation packet is replicated and sent to each, while the TM-CC holds the *VALIDATE* state until validation responses are received from them. A received *ACK* implies that the validating transaction has been cleared to commit by an active transaction. The validating transaction requires an ACK from every active transaction the validation packet was sent to in order to advance to the *COMMIT* state. A *NACK* on the other hand implies a conflict requiring the validating transaction to restart execution after invalidating its read and write sets. This is done in the *ABORT* state. Two possible outcomes exist for the validation.

Validation Successful In the COMMIT state, the SWB is cycled through by the V*ersion Management (VeM)*, and buffered cache lines retrieved. Based on the SM bits set in the tag memory, data words are selected from the retrieved cache line and merged with the old cache line in the L1-D. This refers to the *Merge* operation that commits the transaction's write set to memory.

In order to make these writes visible to other transactions, updated cache lines are written back to the L2-D in a single stream of writes. It is important that these writes to the L2-D remain uninterrupted till completion in order to isolate them from intermediate reads by other transactions which must see these writes appear in the L2-D simultaneously. Therefore, during a commit write operation, L1-D misses are buffered in the L2-D until the committing transaction completes its *write-back*.

The merge and write-back operations are performed concurrently, in order to hide the latency involved in retrieving cache lines from the SWB, merging them with the L1-D and subsequently writing back the merged lines to the L2-D

Validation Unsuccessful NACK responses to validating transactions cause the TM-CC to transition into the ABORT state. In this state, speculative writes buffered in the SWB are cleared by resetting the SWB pointer to zero. Further, the SR and SM fields of the tag memory are cleared, and speculatively read and written cache lines invalidated. The PE is reset in order to clear the contents of all registers, and execution is restarted by placing the address of the first instruction of the transaction in the *Program Counter (PC)*. The transaction is now restarted, and commences execution with the TM-CC's transition to the *EXECUTE* state.
As a consequence of transmitting the validation packet only to PEs indicated as active in the core status information, the performance overhead incurred during validations successively decreases as transactions complete and are retired. In addition, by transmitting the SM field instead of individual word addresses, the benefits of word-granularity conflict-detection are obtained without the associated communication overhead. This technique reduces overhead by over 80% when compared to validations using per-word addresses.

During the *EXECUTE* and *VAL_REQ/PAR_VAL_REQ* states, the TM-CC may encounter incoming validation packets from another validating transaction, thus triggering the conflict detection manager. Cache line addresses from the validation packet are used in a tag lookup operation to determine if cache lines from the validation write-

set are cached in the local L1-D. If the particular cache line is present, its SR and SM fields are compared with the SM bits carried in the validation packet in order to detect intersections between the validation write-set and the local data set. The Oldest contention management policy is invoked upon detection of an intersection.

The contention manager uses the age of the validating and executing transactions along with conflict information in deciding which transaction is aborted. Once again, two possibilities exist.

Validating transaction is older In the event of a read conflict, where the SR field of the executing transaction's read-set intersects with the SM field of the validation packet, the contention manager restarts the younger executing transaction, according to the Oldest CM policy. In addition, the restart is actually performed only after the entire incoming validation completes, thus allowing for other conflicts to be detected, and locally cached versions of the validation write-set to be marked for invalidation.

Intersections between the SM fields of the execution transaction's write-set and the SM field of the validation packet do not cause conflicts, as these writes occur according to the program order. Therefore, neither transaction is aborted.

Validating transaction is younger The read-set of older transactions is protected against invalidations by younger committing transactions by means of the IVC tag bit for delayed invalidation. Therefore, intersections between the SR field of the older executing transaction's read-set and the SM field of the validation packet do not cause conflicts. However, intersections between the SM fields of the older executing transaction's write-set and the validation packet cause a write conflict. In accordance with the Oldest policy, the older transaction's writes are considered valid and thus the younger validating transaction is restarted.

Conflict checking is performed through the second port of the tag memory and therefore, does not block accesses to the L1-D by the executing transaction, preventing the validation operation from increasing execution time.

Special Case: Overflow In the event of an overflow caused by a full SWB, or contention for entries in L1-D cache sets, the TM-CC is switched to the PAR_VAL_REQ state. The process that follows is identical to the validation and commit phases during regular operation. Updated L1-D cache lines are written back, and the SWB subsequently cleared. Since hardware resources are once again available in the TM-CC, execution resumes.

The primary difference from regular operation lies in the handling of the validation token. While at the end of a regular validation and commit the token is returned to the TMS, in the case of a partial validation and commit, it is retained by the overflowed transaction. At the end of execution, the TM-CC is switched to the VAL_REQ state to request for the validation token, however, since it was retained after the overflow occurred, validation is initiated immediately.

4.2.7 PE Interfaces

The processor independent architecture of the fabric thus requires customizable PE interfaces. The TM-CC implements two separate PE interfaces, a data interface and an instruction interface.

The PE performs data memory accesses through the data interface which connects it to the L1-D, SWB and Tag Unit. During reads, the appropriate local data source is selected by the interface depending on whether the read data was stored in the L1-D, or in the SWB.

Instruction fetches are performed through the PE instruction interface, which links the PE with the L1-I. Fetch addresses are monitored by Transaction Control to determine if the last instruction of the transaction has been reached. During restarts, the PE is reset, clearing all registers and setting the PC to zero. The instruction interface also contains the enable signal used to stall execution on cache misses, and also after the end of the transaction has been reached.

4.3 L2 Data Cache

PEs within TMFab cannot address data in the L1 data caches of other PEs directly. However, since transactional multiprocessing requires sharing of data between PEs, the fabric uses a shared L2 data cache. The L2-D tile also includes a 32-bit *external memory interface* to fetch data on compulsory misses from a larger off-chip memory, or a lower level cache.

The architecture of the L2-D and its 32-bit external memory interface is shown in Fig. 4.11. This architecture is not examined in detail as it is based on the standard cache architecture from Hennessey and Patterson [henessey and Patterson]. It is thus described only briefly.

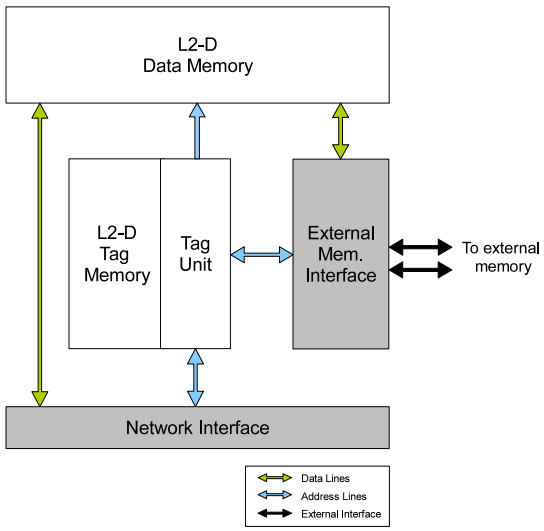


Figure 4.11: Architecture of L2-D tile

The L2-D primarily consists of three blocks: Data Memory, Tag Unit and the External Memory Interface.

4.3.1 Data Memory

When a miss is registered in the L1-D cache, requested data is fetched from lower levels of the memory hierarchy. Accesses to the external memory are slow, forcing PEs to be stalled for long periods of time until data is retrieved. Additionally, since the contents of the L1-D cache are not globally visible, accesses to the same data by different PEs results in multiple external memory accesses for the same cache line. This degradation of performance is addressed by the use of a shared second-level cache. The L2-D stores data cached by the different L1-D caches in the fabric, thus reducing their miss penalty. Consequently, on the first L1-D miss, the requested cache line is looked up in the L2-D. Upon registering a miss, it is fetched from the external memory and cached in the L2-D. Subsequent misses from other L1-D caches to the same cache line result in the data simply being fetched from the L2-D, eliminating the need for multiple memory accesses. For this reason, it is sized larger and has higher associativity than the L1-D caches.

In order to determine the appropriate size, cache simulations were performed with the same four applications used in the sizing of the L1-D. These applications are: bayes, labyrinth, vacation-low and kmeans. A range of cache sizes and associativities were used in simulation runs to determine miss rate. The L2-D's miss rate may be specified in two ways - global miss rate and local miss rate. The global miss rate refers to the ratio of L2-D misses to the total data memory accesses performed in the system, indicating the overall performance of the cache hierarchy. The local miss rate on the other hand refers to the ratio of L2-D misses to the total number of L2-D references. Table 4.6 lists the cache sizes and associativities used in cache sizing simulations. The L1-D was set at 64KB, 4-way with 64 byte line size. The same line size was used for the L2-D to reduce design complexity of its tag unit.

Table 4.6: L2-D SIMULATION PARAMETERS

Cache Size	512KB, 1MB, 2MB, 4MB, 8MB
Associativity	4-way, 8-way, 16-way, 32-way

Table 4.7 lists the logged reference metrics for each application while the local and global miss rates for each application over a range of cache sizes and associativities are presented in Fig. 4.12. These miss rates do not vary significantly between cache sizes and associativites, a characteristic of compulsory cache misses. In bayes, large amounts of data are read from the memory during execution. Since only a small amount of locally generated data is used by transactions, miss rates are not significantly affected by increasing cache size or associativity. Similar observations are made for labyrinth and kmeans.

Increasing cache sizes across all applications reduced the L2-D miss rate although by a rather small percentage. However, this increase diminished between cache sizes of 4MB and 8MB. Increasing associativity causes a very small reduction in miss rates.

Table 4.7: L2-D REFERENCES PER APPLICATION

Application	Total data references	Total L2-D references
bayes	1.172×10^{9}	1.78×10^{6}
labyrinth	2.6×10^{7}	7.7×10^{3}
vacation-low	9.3×10^{7}	6.89×10^{5}
kmeans	6.7×10^{5}	6.8×10^{3}

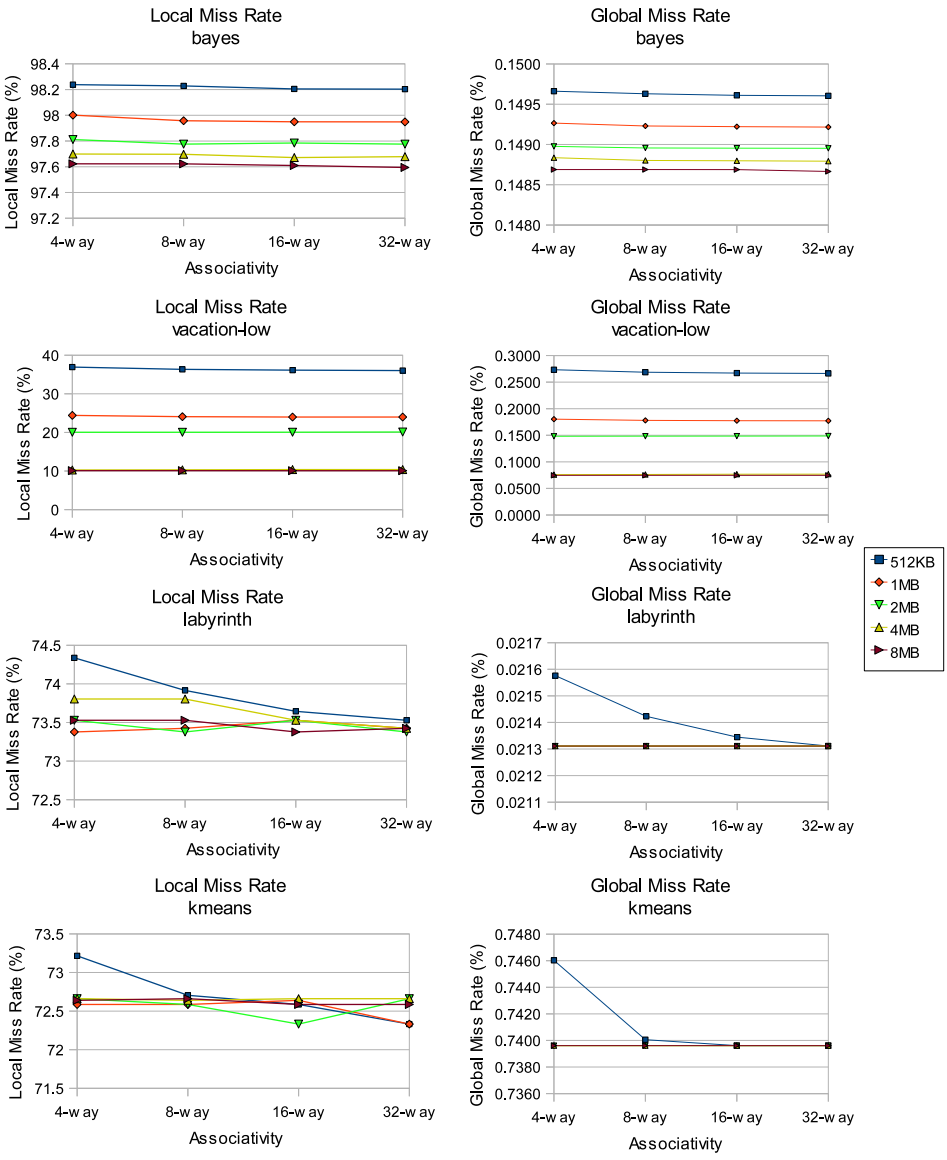


Figure 4.12: Miss rates in the L2-D for varying cache sizes and associativity

However, taking into account the number of L2 references for each application, small miss rate reductions translate to hundreds of fewer memory accesses, such as in the

case of bayes. Increasing associativity beyond 8-way however yields no improvement in miss rate. Therefore, the L2-D is organized as a 4MB, 8-way set associative cache with 64 byte line size, i.e. 8192 sets with each containing 8 cache lines of 16 words.

4.3.2 Tag Unit

The L2-D Tag Unit is a simplified version of that in the L1-D. It performs basic tag lookup operations when a read or write request is received to determine if the addressed cache line is in the cache. If a miss is registered, the requested cache line is fetched from the external memory through the external memory interface. In addition to tag lookup, the tag unit also maintains tag entries for cache lines stored in the L2-D Data memory, and implements a replacement policy to handle contention for cache sets.

Least Recently Used (LRU) is considered as the best performer amongst cache replacement algorithms [38]. LRU tracks the least recently used entry in every set, and replaces it when contention for the set occurs. It is implemented by adding tracking bits to every entry in the set, and updating all tracking bits on every access to the set. Implementing LRU for high-associativity caches (more than 4-way) is therefore complex, and costly in terms of hardware. In order to obtain LRU-class performance without the associated hardware cost and complexity, a pseudo-LRU algorithm is implemented in the L2-D tag unit.

Most Recently Used (MRU) is one such pseudo-LRU algorithm, tracking the cache set entry that was most recently accessed, and thus estimating the least recently used entry of the set. It incurs a hardware cost of only one extra bit per set entry, and negligible logic overhead owing to its simple implementation. Eviction decisions are made based on the status of MRU bits in each entry of the set. An entry with an MRU bit set to 0 is considered as a candidate for eviction.

On accesses to an entry in a set, its MRU bit is set to 1, while the MRU bits of other entries are preserved. In the event that all but one MRU bits are set to 1, a subsequent access to the unused entry will cause an inversion of all MRU bits in the set. This ensures that there is always atleast one candidate for eviction in each set. If multiple candidates exist, one is randomly selected for eviction. The performance of the MRU application was found to be superior to the Random and FIFO cache replacement algorithms in most cases, and comparable to the performance of LRU according to Al-Zoubi et al. in [39].

Apart from the MRU bit, each set entry contains the Empty (E), Valid (V) and Updated (U) tags. The U bit of each set entry is used to determine whether a write-back is required on eviction. The bit is set when the entry is updated by writes from a committing transaction. If a candidate for eviction is marked with the U bit, its contents must be written back to memory before it can be evicted from the L2-D. The L2-D is thus a write-back cache that updates memory only on cache evictions.

4.3.3 External Memory Interface

The external memory interface connects the L2-D to external memory or a lower level cache outside the fabric. This interface only implements the requisite address and data buses, and not any additional control logic required for memory accesses. A 32MB

external DDR400 memory module is assumed for determination of timing information. A static memory access latency of 16 cycles is considered for burst reads and writes, with burst length of 16 words (one cache line) [40][41].

4.4 Interconnect

Different application domains may require additional processing power from transactional CMPs, necessitating the addition of PEs to the system. Scalability is therefore an important requirement and must be taken into account in the design of the interconnect. The TMFab interconnect links PEs, the Scheduler and the L2-D cache to form a transactional multiprocessing system capable of speculatively executing transactions in parallel. The interconnect must support addition of PEs and other required components to the system without significant degradation in performance. An increase in the PE count must be supported by an expansion of the fabric interconnect. However, in silicon implementations additional PEs increase the physical area of the die, an undesirable consequence in area constrained designs.

Die stacking technologies solve this issue by stacking individual dice one above the other, interconnected with advanced Through Silicon Vias (TSV). The area footprint of the stack however, is equivalent to that of a single die. Additional PEs may therefore be added to the system with die stacking, without increasing its area footprint.

Interconnect architectures were evaluated based on their scalability, and suitability for implementation with die stacking. Bus-based interconnects offer limited scalability due to the increased bus arbitration time that results from adding additional components to the system. This directly translates to an increased waiting time for components wishing to perform a transfer over the bus. In the context of a transactional multiprocessing system, miss penalties and validation/commit latencies are increased due to the higher communication latency.

Crossbar based interconnects on the other hand offer increased scalability by interconnecting all system components over a large central switch. However, implementing a crossbar for communication in a 3D interconnect requires careful placement and routing to achieve proper timing closure. Additionally, it may lead to increased design complexity, and reduces reusability since adding layers to the stack would require the crossbar switch to be modified to support components in the added die.

A mesh topology based network-on-chip was found to be a suitable interconnect architecture for the stacked-die transactional CMP. The highly structured mesh topology network simplifies place and route, and enables the faster achievement of timing closure. Additionally, TSVs allow the expansion of the mesh topology from 2D to 3D. Most importantly, the reusability of network-on-chip routers allows for the design to be scaled up without requiring modifications to the routers themselves.

Hence, TMFab uses a packet switched network-on-chip interconnect with TSV based vertical links to route packets between tiles on different layers of the stack. This section examines the architecture of the network, its constituting routers and network interfaces.

Note that since this work only considers TMFab with one to four PEs, only a single layer 2D interconnect is considered for complete system simulations. Nevertheless, subsequent sections describe the 3D interconnect since scaling up beyond four PEs while maintaining a low area footprint will require the use of such an architecture, of which the 2D interconnect is a subset.

4.4.1 Network Architecture

The TMFab network is organized as a packet-switched stack of 3x2 meshes with 36-bit input and output links interconnecting nodes. While the network here is described in the context of a 3D interconnect in the stacked-die configuration, it can also be deployed in a single layer mesh as a 2D network.

Wormhole routing is used to route packets across the network. In this method, packets are transferred as flits, with the destination address indicated only in the header flit, which is routed through by routers as soon as resources are available, without waiting for the remaining flits which simply follow the header. As a consequence, buffering requirements are reduced, and the packet is routed through faster, since routers do not need to wait for the entire packet to be received before forwarding it to the next router [42].

Network packets may contain upto 64 bytes of payload data, and are transferred as 36-bit flits, with each flit containing 4 bytes of data. The header flit of the packet contains addressing and control data essential for routing the packet across the network, and additional fields to aid transactions' communications with the scheduler and L2-D cache. The packet format is illustrated in Fig. 4.13.

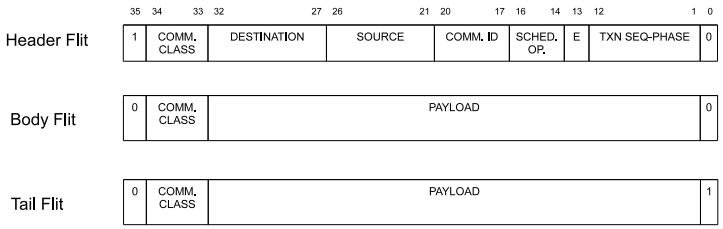


Figure 4.13: Packet Format

The network offers Best Effort Service to traffic within three communication classes: Instruction Block Transfer, TM Communications and Memory Operations. These classes simply categorize network traffic in order to clearly structure network services for the PE, Scheduler and L2-D tile.

Packets are routed across the network based on the *Destination Network Address* they carry in their header flit. Each router and its local tile are assigned a unique *Local Network Address* in the network. When a packet is received at a router with a destination network address that matches the local network address, it is forwarded to the local tile. The header flit also contains a 4 bit communication identifier, which indicates the purpose of the packet within a communication class. The complete list of identifiers is in Table 4.8.

In addition to the communication identifier, the scheduler tile uses the *Scheduler Operation identifier* to determine the requested scheduler operation. However, this identifier is only used during scheduler communications.

During validations by transactions, the packet header also carries the transaction sequence and phase information in order to indicate the transaction's age during conflict

Table 4.8: COMMUNICATION IDENTIFIERS

Comm. Identifier	Comm. Class	Purpose
001	Instruction Block Transfer	Transaction state transfer
010	TM Communications	Scheduler communication
011	TM Communications	Validation Response
100	TM Communications	Validation
101	Memory Operations	L2-D Write
110	Memory Operations	L2-D Read

detection and contention management operations. Since executing transactions wait for the entire validation write-set to be received before responding with an ACK/NACK, the E bit of the header is used to demarcate the last packet of the validation write-set. In addition, during the commit operation, the L2-D buffers all read requests from other transactions until it receives a packet with a set E bit. Therefore, the E bit is also used to indicate the end of a commit operation.

4.4.2 Router Architecture

The fabric router facilitates a three dimensional interconnect utilizing advanced TSVs and die stacking. This input buffered wormhole router contains five lateral ports and two vertical ports, Up and Down for routing between layers in the stack, as illustrated in Fig. 4.14.

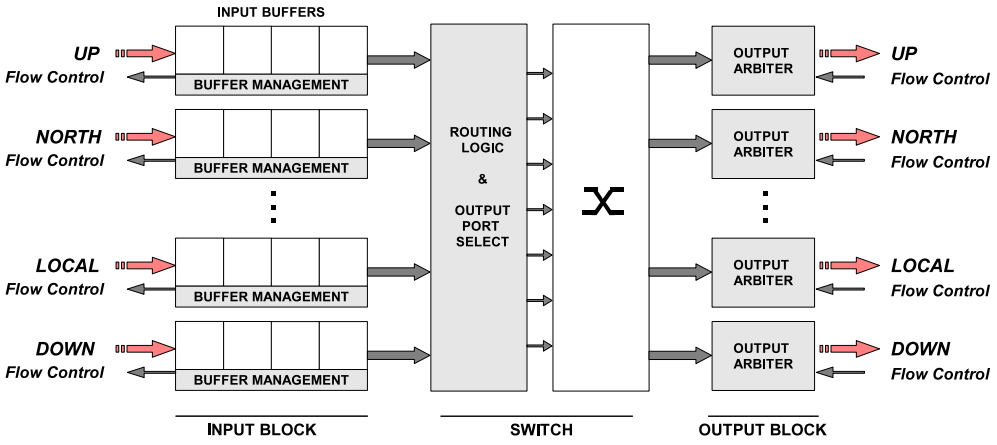


Figure 4.14: Fabric router architecture

The router implements a *dimension ordered static Z-X-Y* routing algorithm that routes packets along the shortest path from source to destination. Packets are first routed in the Z-dimension, and consequently in the X and Y-dimensions. This ensures that packets in transit to other layers in the stack are routed through the TSVs onto their destination layer immediately upon injection into the network, reducing congestion in the layer meshes. Once packets arrive in their destination layer, they are routed

first in the X-dimension, followed by the Y-dimension. The nature of the algorithm ensures that incoming packets are never routed through the same port that they were received at, nor misrouted. Consequently, conditions such as *Deadlock* and *Livelock* are prevented from occurring in the network [43].

The router uses *On-Off* flowcontrol between nodes in the network due to its simple and low overhead implementation. This is achieved by means of a flow control line between upstream and downstream nodes, which switches to a high state when the occupancy of the input buffer at the downstream router reaches a certain threshold. The pipelined nature of the router induces a two-cycle latency for transmission stalls in the event of congestion at downstream routers. The Buffer Management module is designed to take this latency into account, and consequently, the flowcontrol threshold is *N-2*, *N* being the input buffer depth at each port. This flowcontrol mechanism ensures that flits are never dropped, thereby eliminating the need for retransmissions.

The optimal input buffer depth was determined from simulations by examining the variation in average end-to-end packet latency and throughput with increasing buffer depth in a single layer 3×2 mesh. Fig. 4.15 shows the variation in average end-to-end packet latency and the corresponding raw aggregate throughput for different input buffer sizes.

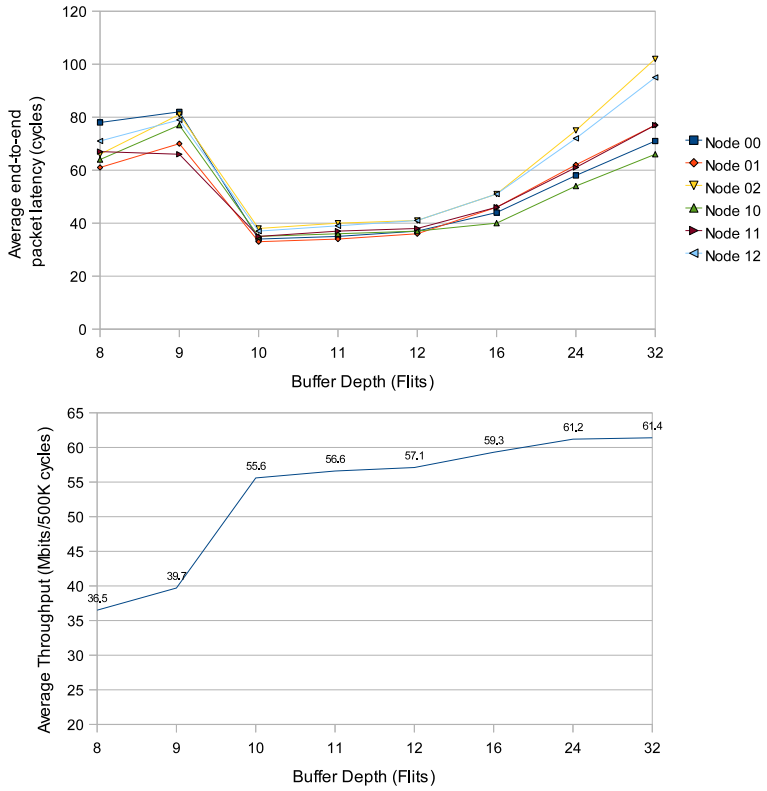


Figure 4.15: Average end-to-end packet latency and raw aggregate throughput with varying input buffer depth

The lowest average latency is observed at an input buffer depth of 10-flits. However,

at a buffer depth of 12-flits, a slightly higher latency is observed although which a significantly higher throughput. Therefore, the router uses 12-flit deep input buffers, with a flowcontrol threshold of 10-flits.

The fabric router uses dedicated arbiters for each output port, implementing a *Round Robin* arbitration scheme in order to ensure fair, *Best Effort* service to all input ports. Due to the routing restriction that prevents packets from being routed through the same port they arrived at, each output arbiter only polls six out of the seven input ports. When a header flit arrives at an input port, the *Routing Logic & Output Port Select* block raises a request to the appropriate output arbiter. If the requested output port is idle and the downstream router has free input buffer slots, the arbiter grants the request by asserting the *Drain* signal for the input buffer. Round robin arbitration is deactivated, and remains in that state until the tail flit of the packet has advanced through the output port. This ensures the integrity of routed packets, and prevents flits from contending packets merging in the output stream.

Contending input ports now wait for access to the output port, keeping their request lines asserted, and input buffers in the stalled state. In the event of input buffers becoming empty while flits of a packet are advancing through the router, the output arbiter sets the link as idle while maintaining the round robin arbitration in the deactivated state. This state of the output port holds until the tail flit has advanced and the complete packet has been routed. Upon completion, the arbiter asserts the *Next* signal and resumes round robin arbitration to service the next waiting input stream. The maximum packet size of 17 flits, i.e. a 64 byte payload, imposed by the network architecture ensures fair and timely arbitration to all input ports, and places a limit on the time the arbiter spends servicing any particular input. The three-stage router thus has a minimum fall through latency of four cycles. The performance of the fabric router is analyzed in Chapter 5, while the design of Through Silicon Vias is described in Chapter 6.

4.4.3 Network Interface Architecture

Specialized network interfaces are used to connect the tile to the fabric network. The *Network Interface (NI)* architecture is illustrated in Fig. 4.16.

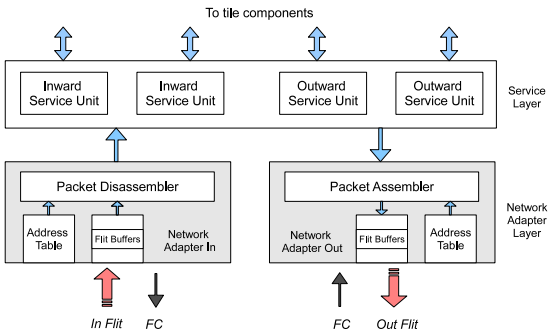


Figure 4.16: Network Interface Architecture

The NI is designed as a modular and customizable interface, split in two layers - Service Layer and Network Adapter layer. The Service Layer consists of several Service Units, each customizable according to the function of the tile. These convert communication signals from the tile into fragments that are assembled by the network adapter, and relayed to the intended recipient over the network, in compliance with the network architecture. Service units are small, and can be easily increased in number to support additional tile functionality.

Inward and Outward service units may also be linked with one another for fast request-response communications. Such a scheme is used in the PE network interfaces for responses to incoming validation write-set packets. During the start of validation, the inward service unit passes the source address of the validation to the appropriate outward service unit responsible for sending the validation response. As a result, the network address and ID of the PE that the validating transaction is mapped to remains abstracted from the other TM-CCs which are only aware of the validating transaction's sequence and phase.

The Network Adapter layer contains inward and outward flit buffers with the same depth as that of input buffers in the fabric routers. *Network Adapter In* performs packet disassembly for network packets. Data that has been received from a tile is stripped of its header flit, and other control information before it is forwarded to the appropriate service unit of the Service layer. Similarly, the *Network Adapter Out* performs packet assembly utilizing control information supplied from the appropriate outward service unit regarding the intended recipient of the packet, its purpose and any additionally required control data for the header. Network adapters contain an address table with the network addresses of all tiles in the system, and their function. In the network interface for the PE tile for instance, individual service units exist in the service layer for each transfer type between the tile and the Scheduler tile. Requests to the scheduler cause the scheduler's network address to be looked up by the network adapter out during packet assembly automatically. The address table is specified as a generic to the NI during instantiation of tiles in the design prior to synthesis.

5 Performance Evaluation

This chapter evaluates the performance of TMFab in two parts. First, the interconnect is characterized with synthetic traffic to establish its performance over a range of injection rates, and in a stacked configuration. Second, the performance of the complete TMFab system with four MB-Lite processor cores is evaluated using test applications, in terms of obtained speed-up over single core execution and execution characteristics of transactions. The results from these simulations determine the overhead incurred from the use of transactional memory, and thus indicate the scalability of TMFab.

5.1 Interconnect

The interconnect is independently characterized from TMFab to determine its performance over a range of traffic injection rates, using tunable traffic to estimate its performance for different traffic conditions. This section examines the testing methodology employed in the evaluation of single layer and stacked meshes.

5.1.1 Methodology

Since the interconnect is characterized independently from the rest of the system, functional units within tiles are replaced with synthetic traffic injectors and evaluators. The fabric router was thus instantiated in a single layer 3×2 mesh, with such *traffic injectors* and *evaluators* connected to the local port of each. In addition, the router's Up and Down ports were disabled since only a single layer mesh was used. Transactional traffic was injected at each port with a variable injection rate. This variation is achieved by inducing a delay between packets offered by the traffic generator at each router. Traffic injectors determined packet destinations based on the order of operations in a transaction, i.e. transactional code segment transfer, L1-D misses, validation and commit arbitration, and commit to the L2-D. Fixed packet size of 64 bytes was used for the code segment transfers and the commit, while other traffic used a packet size of 32 bytes. The destination distribution for PE-emulating injectors is shown in Fig. 5.1 (illustrates the distribution for PE 0). The traffic generators emulating the scheduler and L2-D used a uniform distribution for traffic to PEs.

5.1.2 Interconnect performance

End-to-end packet latency is defined as the time difference between the injection of the head flit into the network at the injector node, and the delivery of the tail flit to the evaluator node from the network. Since each evaluator receives traffic from injectors situated at different distances, the end-to-end packet latency is averaged over 500K cycles, and considered as the average latency to a particular node in the network. The

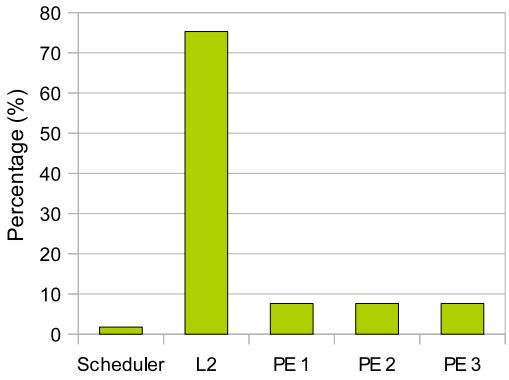


Figure 5.1: Destination distribution for PE-emulating injectors

raw aggregate throughput of the mesh was computed using the total data received at all traffic evaluators in the mesh over 500K cycles at 200MHz. The obtained average packet latency over injection rates is shown in Fig. 5.2.

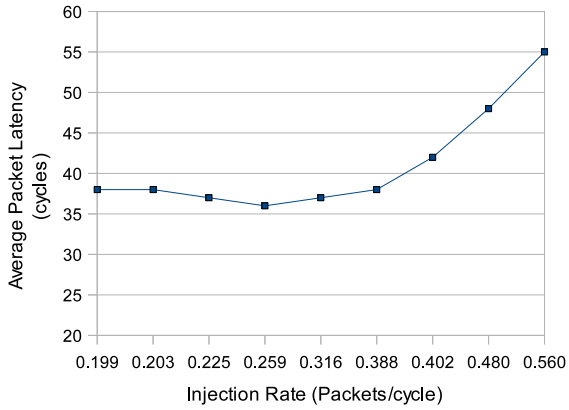


Figure 5.2: Average packet latency for single layer 3×2 mesh

The average packet latency is considered as the average latency to a particular node from any node in the mesh, i.e the cross network latency. It is observed to be flat at lower injection rates due to the fixed maximum packet size imposed by the network architecture that prevents the domination of any particular input stream at the output arbiter of routers. At higher injection rates however packets are injected into the network faster than they can be routed by the output arbiters, consequently increasing the waiting time for input streams and thus the latency. A maximum raw aggregate throughput of 2.85GBps was obtained for the single layer mesh.

The same test setup was extended to study the effects of increasing stack-height while using a 3D network-on-chip. Therefore, the Up and Down router ports were enabled in the previously mentioned mesh, and stacked in three-layers. This increases the round robin arbitration cycle since the each arbiter now polls two extra ports. Fig. 5.3 shows the average packet latency for the three-layer stack.

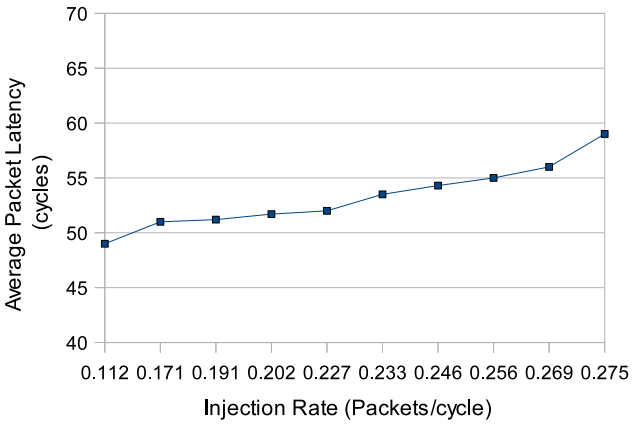


Figure 5.3: Average packet latency for stacked 3×3×2 mesh

The average latency shows a gradual increase with increasing injection rates. This is due in part to the Z-X-Y routing algorithm that routes packets over the TSVs to their destination layer immediately on injection into the network, preserving X-Y routing resources on each layer for packets in their destination layer. The Z-first nature of the routing algorithm also reduces contention for the TSVs by utilizing them primarily at injection. A maximum raw aggregate throughput of 8.3GBps was obtained for the three-layer stack of meshes.

In order to highlight the effects of scaling, the increase in maximum throughput obtained by adding each layer to the stack is weighed against the increase in minimum network latency. The results of this comparison in Fig. 5.4 indicate that as the number of layers in the stack increase, the average latency increases as expected. However, towards higher layers in the stack, this latency increases steeply on account of increased contention on the vertical links, translating to an increased waiting time for inter-layer packets on the TSVs. On the other hand, while raw aggregate throughput increases steeply over the first few layers of the stack, the higher average latency due to the increased network size begins to outweigh the increase in throughput at large stack heights, evidenced by the flattening of the throughput curve past five layers. However, this flattening can be postponed by prioritizing TSV traffic, and by possibly exploiting the superior parasitic performance of TSVs [27].

5.2 TMFab System

5.2.1 Methodology

Since TMFab is only a fabric, and does not include PEs, a suitable soft-processor core must first be instantiated in the design, resulting in a CMP capable of executing transactional application code. Since the fabric is processor independent, no restrictions are placed on the choice of soft-processor core integrated into the design. However, testing the fabric requires integration of a suitable soft-processor core. The MB-Lite

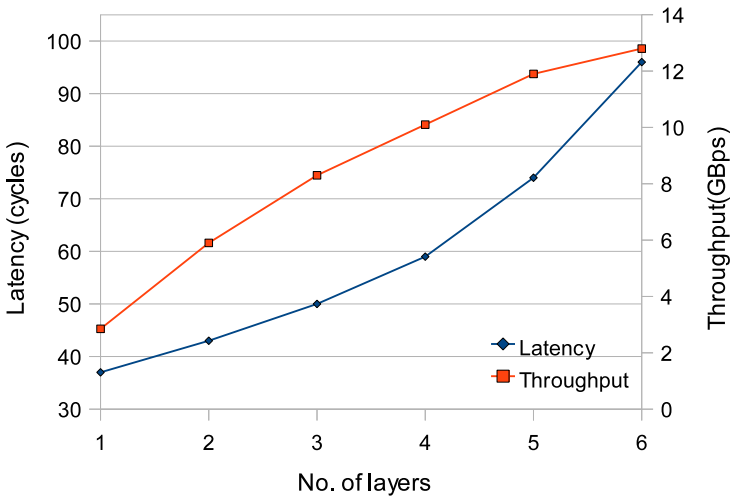


Figure 5.4: Variation in Throughput and minimum network latency with stacking

[44] is an open source soft-processor core based on the Xilinx Microblaze architecture [45]. It was found suitable for the performance evaluation of the fabric for the following reasons:

1. Open source
2. Good documentation

Since each processor architecture is accompanied by its own set of signaling conventions, quality documentation significantly reduces the design effort in integrating the core into the fabric.

In order to evaluate the performance of the fabric, transactional applications must be run on PEs to determine the amount of speedup the system provides over a single core processor. Conventional processors use a set of standard benchmarks composed of several applications designed to stress a particular part of the system. While some benchmarks operate on large sets of data, stressing the underlying memory system, others are computationally intensive, highlighting the performance of the processor core itself. Evaluating a transactional memory system however, requires the use of transactional benchmark applications. The Stanford Transactional Applications for Multi-Processing (STAMP) is the broadest benchmark suite available for transactional memory performance evaluation, consisting of eight applications with varying transactional characteristics. However, the suite was not usable on TMFab since:

1. STAMP applications use the POSIX thread (pthread) library to spawn transactional threads. However, pthreads is not supported by the MB-Lite toolchain.
2. Not all applications are completely transactional, and contain non-transactional code.

Removing pthread references from the code was not a trivial task, and resulted in incorrect results over several iterations in porting the benchmark to TMFab. Manually

parallelizing the application after eliminating pthreads was not feasible. In addition, converting non-transactional code into coarse grain transactions also impacted the correctness of the applications. Therefore, STAMP could not be ported to the fabric within the context of this work, however it holds a high priority in future work with TMFab.

A custom set of test programs was coded in order to test the system. The programs were coded in C, with coarse grain transactional sections demarcated with transactional markers. The code was compiled with the -S switch in order to obtain its assembly code. However, compiling the partitioned code with transactional markers in place yielded incorrectly executing transactions. This highlighted the need for a toolchain augmented with capabilities to maintain the integrity of code between transactional markers, and to manage initialization code for each transaction.

Therefore, the four test programs - *MAT-SMALL*, *MAT-MED*, *MAT-LARGE* and *MAT-LARGE-HIGH*, were hand-coded in assembly, with initialization code for each of their four transactions included within their transactional boundaries. The programs were not designed to be computationally intensive, but instead to generate L2-D accesses, create conditions for false sharing, and illustrate the effects of conflicts on obtained speed-up.

MAT-SMALL : A 4×4 matrix is read in by four transactions, with each transaction assigned one 2×2 section. Transactions increment values in the matrix, and write them back to the L2-D as they commit. Since different transactions modify words in the same cache line, conventional cache line granularity conflict detection mechanisms would flag a conflict here.

***MAT-MED*:** A 32×16 matrix is read in by four transactions, with each transaction assigned one 16×8 section. Transactions increment values, similar to the *MAT-SMALL* program and write them back to the L2-D. Each transaction works on individual cache lines, and therefore, no conflicts are detected.

***MAT-MED-HIGH*:** is identical to *MAT-MED* except in that it contains dependencies between transactions. *MAT-MED-HIGH* uses four transactions with a configurable number of dependencies between them. *MAT-MED-HIGH* thus defines dependencies as the sharing of data between the boundaries of a pair of transactions. Dependencies are defined for pairs of transactions as illustrated in Fig. 5.5.

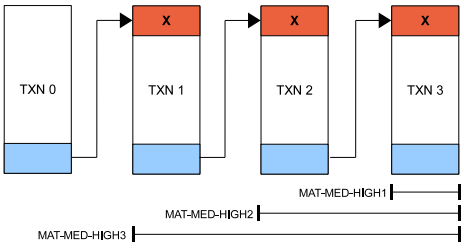


Figure 5.5: Dependencies between *MAT-MED-HIGH* transactions

Thus transactions 0 through 2 modify one data word in another transaction's write-set during execution. Transaction 3, being the last transaction to execute, has no such characteristic.

MAT-LARGE : A 64×64 matrix is read in by four transactions, with each assigned one 32×32 section. Transactions increment values in the matrix, and write them back to the L2-D. This program stresses the L2-D and the interconnect since it generates atleast 64 L1-D misses per transaction.

Each of these applications was assembled, and loaded onto an instruction ROM in the testbench, performing the function of the supervisor processor. The run timer was started as soon as the transfer of to the L2-Instruction Memory through the Scheduler's TPI began, and stopped when the commit complete notification from the last executing transaction was received by the PESM. Each simulation was run in three iterations, with the number of PEs ranging from one to four, in order to show speed up. Upon completion of all transactions, L2-D contents were manually evaluated to check for correctness. The normalized speed-up for the *MAT-SMALL*, *MAT-MED* and *MAT-LARGE* applications is shown in Fig. 5.6.

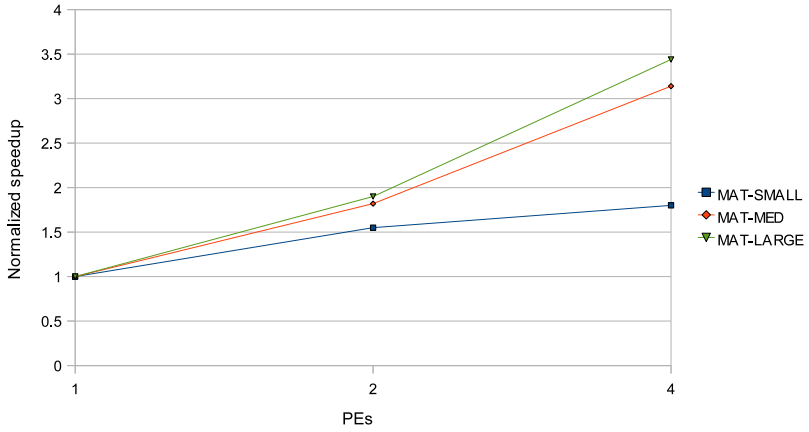
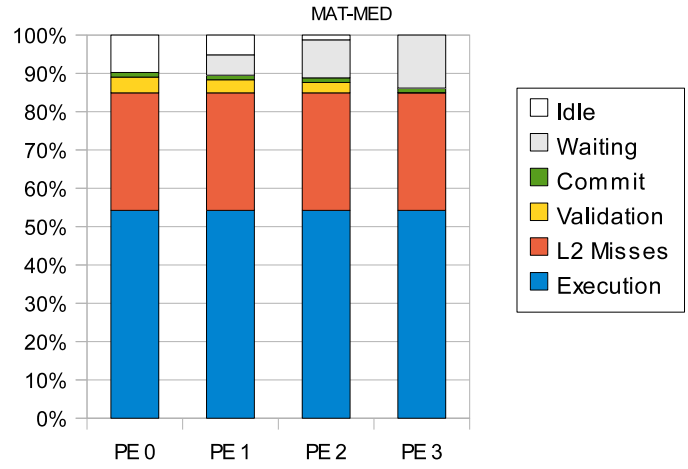


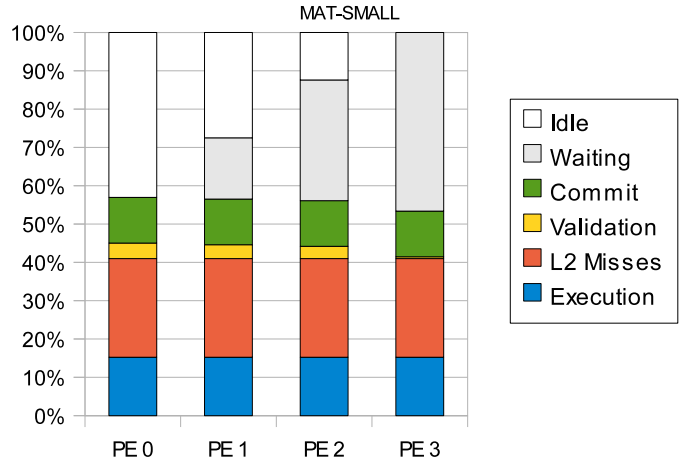
Figure 5.6: Normalized speed up for *MAT-SMALL*, *MAT-MED* and *MAT-LARGE*

Amongst the three test applications, *MAT-LARGE* shows the highest speed up at 3.44× with four PEs in the fabric while *MAT-MED* performs closely at 3.14×. *MAT-SMALL* in comparison exhibits the lowest speed up of 1.8× over execution on a single PE system. However, the significant difference between the three lies in the manner in which their performance scales with an increasing number of PEs. The breakdown of execution time in Fig. 5.7 contrasts the characteristics of execution of the *MAT-SMALL* application on a four-PE system against those for *MAT-MED*, indicating why application performance scales in the manner observed in Fig. 5.6.

The percentage bar graphs in Fig. 5.7 break the execution of transactions on individual PEs down into constituent transactional operations. Sections shaded blue correspond to the time spent by the PE in actively fetching and executing instructions.



(a)



(b)

Figure 5.7: Breakdown of execution time (a) *MAT-MED* (Runtime = 18430ns), (b) *MAT-SMALL* (Runtime = 2098ns)

Each of the four transactions in *MAT-MED* operate on eight cache lines, while *MAT-SMALL*'s transactions, on the other hand, operate on four distinct sets of words in a single cache line. The Red areas on each bar correspond to the total duration for which the PE remained stalled under cache misses registered on the first reference to a cache line by a transaction.

Before a transaction can commit, its write-set is validated against the data sets of other active transactions. Reiterating from Chapter 4, validation packets only contain a cache line address and its corresponding SM field, thus acting as a coherence invalidation packet. As execution progresses and transactions complete their commits, fewer active transactions remain in the system. Consequently, the number of validation packets

sent by each remaining transaction decreases, effectively shortening the length of its validation phase. This is illustrated by the gradual decrease in the duration of the validation phase for each PE. The commit phase, however, is of constant duration regardless of the number of active transactions in the system, since it depends on the write-set size alone.

The validation and commit phases collectively account for 6% of each transaction's execution time in the case of *MAT-MED*, while they form 39% of the execution time for *MAT-SMALL*. The overhead of validation and the commit operation is observed in Fig. Fig. 5.7B to be of comparable duration to that of active execution of the transaction itself. Further, the decrease in the duration of the validation phase in the case of *MAT-SMALL* is negligible on account of its small write-set, unlike *MAT-MED* where such a decrease causes commits to commence sooner. Fig. 5.8 shows the breakdown of execution for the *MAT-LARGE*, clearly illustrating the performance advantage of using longer duration transactions.

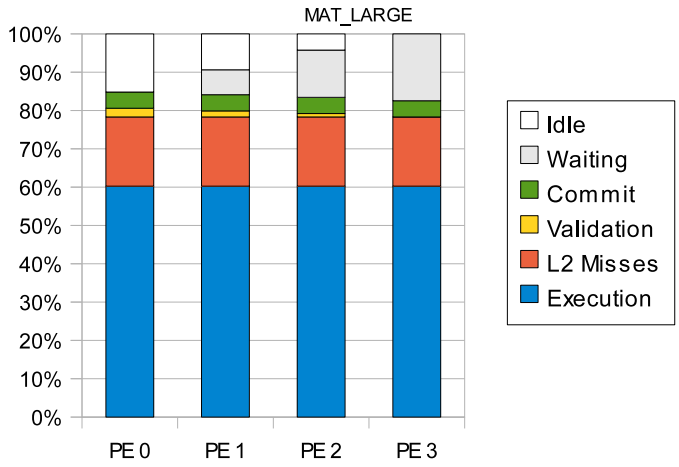


Figure 5.8: Breakdown of execution time for *MAT-LARGE*(Runtime = 144725ns)

The validation and commit phases in *MAT-LARGE* account for a maximum of 8% of each transaction's execution time. With progress in execution, this overhead is reduced to 5% for the last active transaction. Fig. 5.9 shows the magnitude of the overhead for different numbers of active transactions in the system.

Therefore, for applications with short duration transactions, the overhead imposed by the validation and commit operations severely limits the speed-up obtainable from their concurrent execution on TMFab. This overhead is offset in long duration transactions by the length of the execute phase itself. Consequently, such transactions achieve a high speed up from concurrent speculative execution on the fabric. The execution characteristics and speed up of *MAT-SMALL*, *MAT-MED* and *MAT-LARGE* illustrate the performance of TMFab with atomic transactions free of dependencies. In order to examine the effects of non-atomic transactions with dependencies, the *MAT-MED-HIGH* application is used, with dependencies between transactions leading to contention for shared data. Fig.5.10 illustrates the effects of dependencies between transactions on the speed up obtained from execution on TMFab.

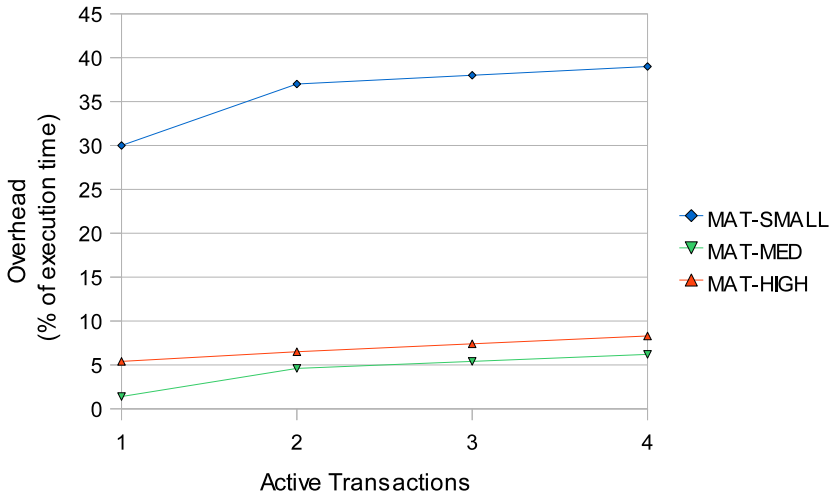


Figure 5.9: Magnitude of overhead for varying number of active transactions

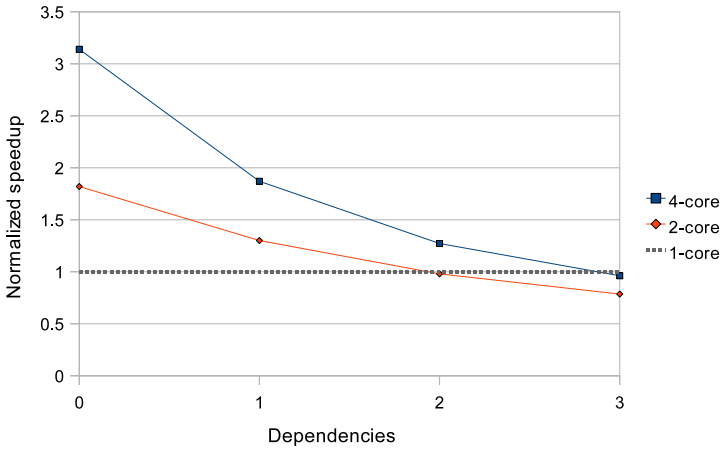


Figure 5.10: Effect dependencies on speed up

Dependencies effectively serialize execution, with contention-losing transactions forced to abort and restart. As a consequence of this serialization, overall application run time is increased, drastically decreasing speed up. The decrease in speed up becomes more significant with the amount of concurrency exploited in the system. Thus, the gradient in speed-up is observed to be much steeper for the four-PE system than for the two-PE system. Execution on the one-PE system is inherently sequential and remains unaffected by dependencies. However, It is interesting to note that transactions with dependencies cause the performance of the four-PE system to drop below the levels of a one-PE system. This underlines the importance of ensuring the independence of transactions.

In the VHDL simulation model of TMFab, contentious data is identified by noting its read/write-set address, the sequence and phase of the two contending transactions

and the type of conflict that was flagged. Using this data, transactional boundaries may shifted, or the operations on the shared data restructured in order to eliminate contention.

The L2-D contents were evaluated at completion for each application and found to reflect expected values, indicating correctness of the result, and overall execution of the application. This observation was also made in the case of MAT-MED-HIGH, affirming that transactional operations in the fabric and the enforced system policy maintain coherence of the L1-D caches.

6 Implementation

In this chapter, the results from synthesis of each TMFab tile for a Field Programmable Gate Array (FPGA) target as well as a semi-custom ASIC flow are examined. Additionally, results from the place and route (PAR) of the fabric router are examined, along with the full-custom design of the Through Silicon Via (TSV) cell in different configurations.

6.1 TMFab

The primary objective of this synthesis is to determine the area utilization of components in order to establish a base line for future optimizations. This design was not taken through the place and route flow for reasons of time.

6.1.1 FPGA

Each tile was decomposed into its constituting components - network interface and the TMFab functional unit, and each of these components was synthesized separately on a Xilinx Virtex6 LX75T [46] target in order to determine its resource utilization. However, memories were not included in this synthesis on account of their size. Therefore, the stated resource utilization only considers logic components. The resource utilization and maximum clock frequency automatically determined during synthesis for each TMFab component is listed in Table 6.1.

Table 6.1: RESOURCE UTILIZATION AND CLOCK FREQ. - TMFAB

Component	Resources (LUTs)	Max. Clock Frequency (MHz)
TM-CC	3089	262
Scheduler	880	417
L2-D Control Logic	5800	397
Scheduler NI	2886	321
PE NI	2285	250
L2-D NI	1265	334

In comparison, the MB-Lite utilizes 1221 LUTs on the same device.

6.1.2 ASIC

Synthesis was also performed using the Faraday FSD0A_A standard cell library [47] built on the 90nm Standard Performance logic process from UMC (L90SP) using worst

case operating conditions (125°C temperature and 0.9V core supply) and a 200MHz clock frequency.

To highlight the area occupied by the different caches and memories in the fabric, synchronous SRAM blocks were generated according to the required size and organization. These memory blocks are built on the same 90nm process, and are part of the FSD0A_A library as well. Synthesis results are presented separately for each tile. Table 6.2 lists the area utilization for each TMFab component in comparison with the area of a synthesized MB-Lite core in Table 6.3. Additionally, Table 6.4 summarizes the sizes of various memory blocks within the fabric.

Table 6.2: POST-SYNTHESIS AREA UTILIZATION - TMFAB

Scheduler Tile		PE Tile		L2-D Tile	
Component	**Area** (mm^2)	**Component**	**Area** (mm^2)	**Component**	**Area** (mm^2)
Scheduler	0.045	TM-CC	0.063	Control Logic	0.23
L2-I	3.068	L1-D	0.96	L2-D	49.08
NI	0.076	L1-Tag	0.24	L2-Tag	30.84
		SWB	0.636	NI	0.09
		L1-I	0.767		
		NI	0.1		
Total Area	3.15		2.76		80.24

Table 6.3: POST-SYNTHESIS AREA UTILIZATION - MB-LITE

Component	**Area** (mm^2)
MB-Lite	0.034
Register File	0.023
Total Area	0.057

Table 6.4: SUMMARY OF MEMORY SIZES

Type	**Capacity (KB)**
L1-I	64
L1-D	64
L1-Tag	8
SWB	32
L2-I	256
L2-D	4096
L2-Tag	152

It is prudent to note here that these represent estimates of area, and not the actual area itself. During place and route, nets are routed between standard cells, and buffers

inserted to meet timing requirements, thus altering the design area. However, these numbers still serve as reasonable first estimates of the area of the design.

The L2-D occupies the most area on chip in comparison with other components. This large area penalty can be mitigated by splitting this large memory into smaller blocks, and placing them in a stacked-die configuration, connected to the L2-D control logic by means of TSVs. This is analogous to the work by [48].

6.2 Fabric router

The 3D 7-port fabric router was designed to enable a 3D interconnect for the TMFab system, facilitating communication between PEs in a stacked die configuration. However, since only a single-layer mesh with four PEs was considered for system simulation, the vertical ports of the router were disabled, resulting in a 2D 5-port router. Both the 5- and the 7-port router were taken through the synthesis-place and route flow to determine area utilization.

6.2.1 FPGA

The two router variants were synthesized for the same Virtex 6 target mentioned earlier. It is important to note that only the 5-port router can be implemented on an FPGA, while the 7-port router can be implemented only to realize 2D NoCs. Its resource utilization is therefore presented only for comparison with the 5-port router.

The resource utilization along with the determined maximum clock frequency for the two routers is listed in Table 6.5.

Table 6.5: RESOURCE UTILIZATION AND CLOCK FREQ. - FABRIC ROUTERS

Router Variant	Resources (LUTs)	Max. Clock Frequency (MHz)
5-port	4141	260
7-port	5942	220

6.2.2 ASIC

The Through Silicon Via (TSV) forms a critical part of the 3D router, since it acts as the vertical link between routers in a stack. Their large height, spanning from the fourth metal layer through the first metal layer may induce signal integrity issues in neighbouring nets, and thus a buffer zone known as the keep out area is maintained around the via. This area contains routing and placement blockages, preventing nets from being routed close to the via. The keep out area also determines the spacing between TSVs, i.e. the pitch. Thus, to investigate the area penalty associated with the use of TSVs, three full-custom cells were designed in 90nm UMC with varying keep out areas. Fig. 6.1 illustrates the general structure of this cell.

The three TSV configurations are listed in Table 6.6.

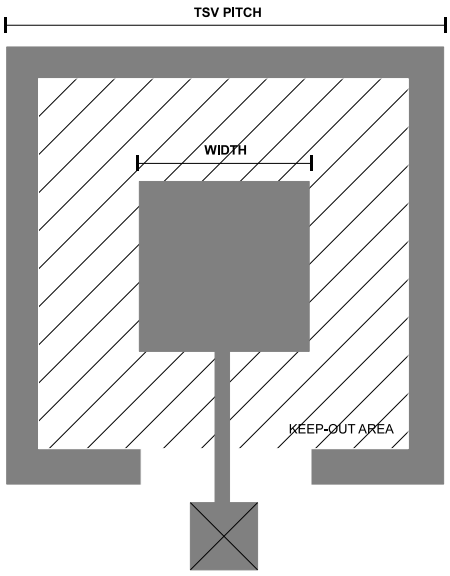


Figure 6.1: Structure of custom TSV cell

Table 6.6: TSV CONFIGURATIONS

TSV Width (μm)	5.6		
TSV Pitch (μm)	11.2	25.2	50.4
Aggregate Cell Area (μm^2)	282.2	953.5	3144.96

The 2D 5-port router was first synthesized and subsequently taken through a series of floorplanning, placement and routing steps. At a clock frequency of 200MHz, the 2D router occupied an aggregate area of 0.158mm^2, inclusive of the 0.0177mm^2 of area occupied by the 12-flit deep input buffers at each port. Fig. 6.2 provides a view of the placed and routed designs, in which the five ports of the router are easily observed. Note that the router's local port is located at the upper left corner of the die area shown.

The 3D 7-port router was subsequently synthesized after instantiating 74 TSVs for each of its two vertical ports, thus yielding a total of 148 TSVs. The design was floorplanned, placed and routed iteratively, with a different TSV configuration in each run. The obtained area estimates with each configuration is listed in Table 6.7.

Table 6.7: AREA ESTIMATES FOR 7-PORT ROUTER

TSV Pitch (μm)	11.2	25.2	50.4
Router area without TSV (mm^2)	0.158		
Router area with TSV (mm^2)	0.199	0.298	0.62

Fig. 6.3 shows the placed and routed 7-port router with 11.2μm pitch TSVs surrounding the router core. The TSVs are clustered in the corners as a precaution against vertical ports inducing signal integrity issues in lateral nets in close proximity with

them.

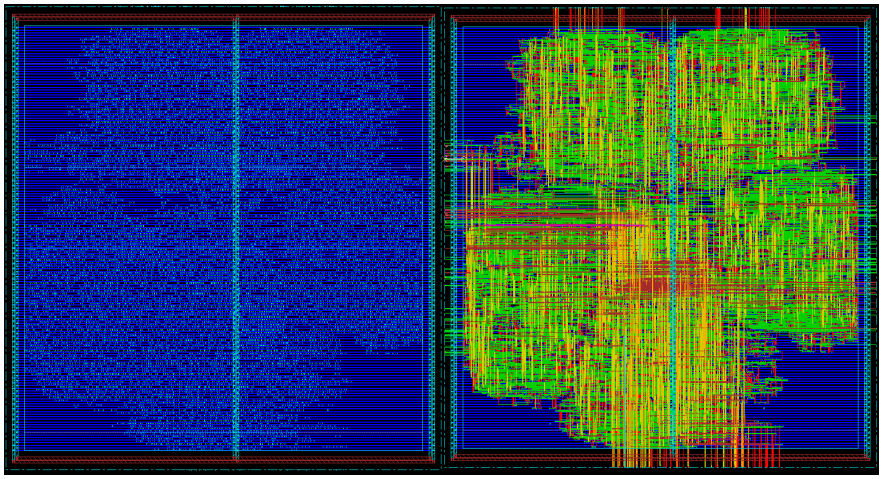


Figure 6.2: Placed (L) and Routed (R) 5-port router

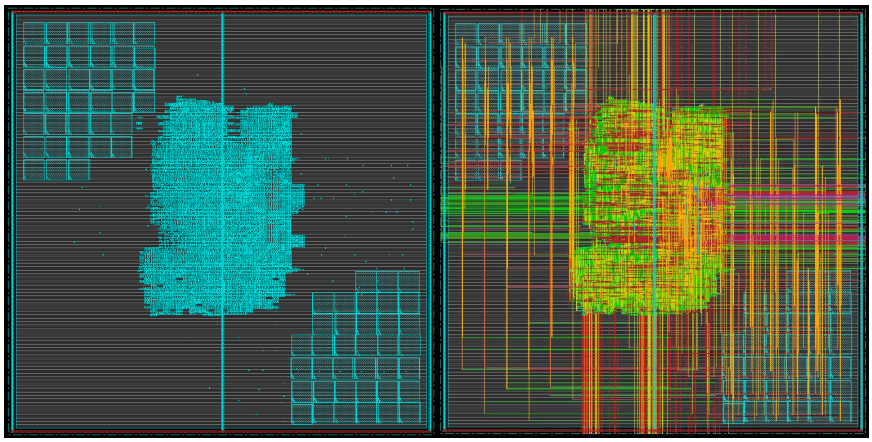


Figure 6.3: Placed (L) and Routed (R) 7-port router with 11.2μm TSVs

7 Conclusion

This chapter presents concluding remarks on the TMFab design alongside summarizing the work that was carried out during the course of the project, and emphasizing the goals that were achieved. Additionally, areas requiring further exploration are identified, and the scope for future work highlighted.

7.1 Summary

It was determined from initial background research that existing hardware transactional memory implementations did not offer enough flexibility to enable scalable chip multiprocessors with different processor cores than the ones they were designed with. The TMFab project was started with the intent to develop of a transactional memory fabric containing requisite hardware infrastructure to prototype and deploy scalable chip multiprocessors. With this objective, a fabric architecture was developed after an extensive survey of existing transactional memory proposals and their implementations. The developed architecture contained a best effort hardware transactional memory system along with a scalable network-on-chip based interconnect architecture. To support such an architecture, a system-level transactional memory policy was defined, establishing the protocol for transactional operations, and defining policies to reduce incurred performance overheads, thus providing a roadmap for the development of the required hardware. TMFab implements a best-effort transactional memory system, with lazy data versioning and optimistic conflict detection. Application code coarsely partitioned into transactions using simple transactional primitives, is executed on the fabric's PEs through a light-weight transaction scheduler using a Fixed-Priority First-Come-First-Served scheduling policy. Transactional caches and associated hardware were designed as processor independent modules, performing all conflict detection, version management, contention management, and validation operations outside the PE. This approach eliminated the need for register checkpointing in PEs, and allowed for their easy integration without any modifications to the core itself. Transactional operations were designed keeping a scalable network in mind, and a conscious effort was made to decrease the amount of traffic that each PE injected into the network. This resulted in the reduction in validation overheads generally incurred at the end of transactions. By using scheduler tracked core-status information, validation packets were only transmitted to active transactions in the system, reducing the total number of packets injected into the interconnect at the end of a transaction's execution. This provided an additional benefit in the form of speeding up the application as transactions committed. This was attributed to the reduced validation requirements for every subsequent transaction on account of the fewer number of active transactions in the fabric. Additionally, by using a 16-bit field indicating the speculatively modified words within a cache line,

validation overhead was decreased by over 80% when compared to per-word address transmission based validations. And lastly, a baseline 3D interconnect architecture utilizing advanced Through Silicon Vias was built into the fabric to allow scalability in the vertical plane using die-stacking. Experiments with the interconnect were used to highlight the area penalty resulting from the use of TSVs of different pitches, and to determine the maximum feasible height of a stacked-die configuration for the baseline 3D architecture. The performance of TMFab was evaluated by instantiating four lightweight MB-Lite processor cores inside the fabric, and executing a set of test applications that stressed the conflict detection, contention management and version management policies of the fabric. Over three application configurations, the TMFab based CMP showed a worst case speed up of 1.8× and a best case speed of 3.44× over single core execution. With dependencies, the worst case speed up was 0.96× highlighting the importance of reducing dependencies between transactions. The overall performance overhead was determined at under 10% for medium and long running transactions, while it settled at a minimum value which translated to 38% overhead for the short duration transaction The fabric was decomposed into blocks and synthesized to determine its resource utilization and post-synthesis area utilization in a semi-custom design flow, along with the automatically determined maximum clock frequency which was found to be atleast 220MHz for all components.

7.2 Future Work

In the course of the design of TMFab, several ideas were conceived that couldn't be realistically implemented given the magnitude of the project. These are listed as possible avenues for future work:

1. The highest priority task involves the development of an augmented toolchain for TMFab. Test applications were manually coded in assembly during benchmarking however for very large applications with several transactions, such a method will not suffice. A tool to automatically partition compiler generated assembly code from a high level user program is envisaged. Initialization code for every transaction would have to be included by the tool within the transactional boundaries. Such a tool, needless to say, would have to remain processor independent, or possibly offer support for a broad range of processors. This would certainly pose a significant challenge, however the benefits are plenty.

2. Although a hardware transactional memory system has been implemented within the fabric, it is by no means optimal. Optimizations to further reduce performance overheads, and scaling up of the fabric beyond the test four cores would improve the design past its current state.

3. In addition, the possibilities of decomposing the scheduler into multiple dice on separate layers may be looked into to increase system scalability

4. Differential clocking of TSVs and bufferless vertical links must be explored in order to adequately exploit the electrical characteristics of the TSVs, and overcome the performance loss observed during experiments with stacking.

7.3 Publications

These contributions of TMFab are described in two papers:

- Sumeet S. Kumar, T.G.R.M. van Leuken; **A 3D Network-on-Chip for Stacked-Die Transactional Chip Multiprocessors using Through Silicon Vias**, submitted to the 6th International conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS'11), Athens, Greece.
- Sumeet S. Kumar, T.G.R.M. van Leuken; **TMFab: A Transactional Memory Fabric for Chip Multiprocessors**, currently being compiled

Bibliography

[1] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *In Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 5–17, ACM Press, 2002.

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

[3] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "Atlas: A chip-multiprocessor with transactional memory support," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pp. 1 –6, april 2007.

[4] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance sparc cmt processor," *Micro, IEEE*, vol. 29, pp. 6 –16, march-april 2009.

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

[6] P. Stravers and J. Hoogerbrugge, "Homogeneous multiprocessing and the future of silicon design paradigms," in *VLSI Technology, Systems, and Applications, 2001. Proceedings of Technical Papers. 2001 International Symposium on*, 2001.

[7] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pp. 102 – 113, june 2004.

[8] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pp. 494 – 505, june 2005.

[9] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie, "Unbounded transactional memory," *Micro, IEEE*, vol. 26, pp. 59 –69, feb. 2006.

[10] O. S. D. Dice and N. Shavit, "Transactional locking ii," in *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pp. 194–208, 2006.

[11] M. Herlihy, V. Luchangco, and M. Moir, "Software transactional memory for dynamic-sized data structures," pp. 92–101, ACM Press, 2003.

[12] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi- core

runtime," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, (New York, NY, USA), pp. 187–197, ACM, 2006.

[13] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?," *Queue*, vol. 6, pp. 46–58, September 2008.

[14] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory.," in *ASPLOS'06*, pp. 336–346, 2006.

[15] L. Baugh, N. Neelakantam, and C. Zilles, "Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory," *SIGARCH Comput. Archit. News*, vol. 36, pp. 115–126, June 2008.

[16] E. Vallejo, T. Harris, A. Cristal, O. S. Unsal, and M. Valero, "Hybrid transactional memory to accelerate safe lock-based transactions," in *WORKSHOP ON TRANSACTIONAL COMPUTING (TRANSACT 2008)*, 2008.

[17] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "Logtm: log-based transactional memory," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, feb 2006.

[18] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," 2006.

[19] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," *SIGPLAN Not.*, vol. 44, pp. 157–168, March 2009.

[20] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "Ramp: Research accelerator for multiple processors," *Micro, IEEE*, vol. 27, pp. 46 –57, march-april 2007.

[21] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, 2001.

[22] L. Benini, "Networks on chip: a new paradigm for systems on chip design," in *In Proceedings of Conference on Design, Automation and Test in Europe*, pp. 418–419, IEEE Computer Society, 2002.

[23] S. W. Yoon, D. W. Yang, J. H. Koo, M. Padmanathan, and F. Carson, "3d tsv processes and its assembly/packaging technology," in *3D System Integration, 2009. 3DIC 2009. IEEE International Conference on*, pp. 1 –5, 2009.

[24] P. Franzon, W. Davis, and T. Thorolffson, "Creating 3d specific systems: Architecture, design and cad," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 1684 –1688, 2010.

[25] E. Beyne and B. Swinnen, "3d system integration technologies," in *Integrated Circuit Design and Technology, 2007. ICICDT '07. IEEE International Conference on*, 30 2007.

[26] K. Puttaswamy and G. Loh, "3d-integrated sram components for high-performance microprocessors," *Computers, IEEE Transactions on*, vol. 58, no. 10, pp. 1369 – 1381, 2009.

[27] G. Loi, B. Agrawal, N. Srivastava, S.-C. Lin, T. Sherwood, and K. Banerjee, "A thermally-aware performance analysis of vertically integrated (3-d) processor-memory hierarchy," in *Design Automation Conference, 2006 43rd ACM/IEEE*, 0 2006.

[28] V. F. Pavlidis and E. G. Friedman, "3-d topologies for networks-on-chip," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, pp. 1081–1090, October 2007.

[29] A. Y. Weldezion, M. Grange, D. Pamunuwa, Z. Lu, A. Jantsch, R. Weerasekera, and H. Tenhunen, "Scalability of network-on-chip communication architecture for 3-d meshes," in *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '09, (Washington, DC, USA), pp. 114–123, IEEE Computer Society, 2009.

[30] R. Patti, "Three-dimensional integrated circuits and the future of system-on-chip designs," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1214 –1224, 2006.

[31] I. Loi, F. Angiolini, and L. Benini, "Supporting vertical links for 3d networks-on-chip: toward an automated design and analysis flow," in *Proceedings of the 2nd international conference on Nano-Networks*, Nano-Net '07, (ICST, Brussels, Belgium, Belgium), pp. 15:1–15:5, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[32] D. Park, S. Eachempati, R. Das, A. Mishra, Y. Xie, N. Vijaykrishnan, and C. Das, "Mira: A multi-layered on-chip interconnect router architecture," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pp. 251 –261, 2008.

[33] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner, "Picoserver: using 3d stacking technology to enable a compact energy efficient chip multiprocessor," *SIGPLAN Not.*, vol. 41, pp. 117–128, October 2006.

[34] F. Angiolini, P. Meloni, S. Carta, L. Benini, and L. Raffo, "Contrasting a noc and a traditional interconnect fabric with layout awareness," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, pp. 1 –6, 2006.

[35] T. F. Knight, "An architecture for mostly functional languages," in *Proceedings of ACM Lisp and Functional Programming Conference*, pp. 500–519, Aug 1986.

[36] M. Lupon, G. Magklis, and A. González, "Version management alternatives for hardware transactional memory," in *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, MEDEA '08, (New York, NY, USA), pp. 69–76, ACM, 2008.

[37] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 35 –46, sept 2008.

[38] A. Dan and D. Towsley, "An approximate analysis of the lru and fifo buffer replacement schemes," *SIGMETRICS Perform. Eval. Rev.*, vol. 18, pp. 143–152, April 1990.

[39] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, (New York, NY, USA), pp. 267–272, ACM, 2004.

[40] I. S. S. Inc, *IS43R32800B 256Mb DDR Synchronous DRAM, Rev. 00D.*

[41] M. T. Inc, *Micron MT46V128M4 Core DDR Rev. B 2/09 EN.*

[42] C. Scheideler and B. Vöcking, "Universal continuous routing strategies," in *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, (New York, NY, USA), pp. 142–151, ACM, 1996.

[43] W. Dally and C. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *Computers, IEEE Transactions on*, vol. C-36, pp. 547 –553, May 1987.

[44] T. Kranenburg and R. van Leuken, "Mb-lite: A robust, light-weight soft-core implementation of the microblaze architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 997 –1000, 2010.

[45] X. Inc, *Microblaze Processor Reference Guide.*

[46] X. Inc, *DS150 Virtex-6 Family Overview.*

[47] F. T. Corporation, *FSD0A_A Standard Cell Library.*

[48] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *In International Symposium on Computer Architecture.*