ABSTRACT

Title of dissertation:	HARDWARE DESIGN, PROTOTYPING AND STUDIES OF THE EXPLICIT MULTI-THREADING (XMT) PARADIGM
	Xingzhi Wen Doctor of Philosophy, 2008
Dissertation directed by:	Professor Uzi Vishkin Department of Electrical and Computer Engineering

With the end of exponential performance improvements in sequential computers, parallel computers, dubbed “chip multiprocessor”, “multicore”, or “manycore”, has been introduced. Unfortunately, programming current parallel computers tends to be far more difficult than programming sequential computers. The Parallel Random Access Model (PRAM) is known to be an easy-to-program parallel computer model and has been widely used by theorists to develop parallel algorithms because it abstracts away architecture details and allows algorithm designers to focus on critical issues. The eXplicit Multi-Threading (XMT) PRAM-On-Chip project seeks to build an easy-to-program on-chip parallel processor by supporting a PRAM-like programming (performance) model. This dissertation focuses on the design, study of the micro-architecture of the XMT processor as well as performance optimization.

The main contributions are:(1) Presented a scalable micro-architecture of the XMT based on high level description of the architecture. (2) Designed a synthesizable Verilog HDL (hardware design language) description of XMT, which lead to the first commitment to the silicon of the XMT processor, a 75 MHz XMT FPGA computer. With the same design, we expect to see the first XMT ASIC processor using IBM 90nm technology. (3) Proposed and implemented some architecture upgrades to the XMT: (i)value broadcasting, (ii)hardware/software co-managed prefetch buffers and (iii) hardware/software co-managed read-only buffers. (4) Quantitatively studied the performance of XMT using non-trivial application kernels with the 75 MHz XMT FPGA computer, in addition, the performance of a 800MHz XMT processor is projected. (5) The choice of not having local private caches in the XMT architecture is studied by comparing current architecture with an alternative one that includes conventional coherent private caches.

HARDWARE DESIGN, PROTOTYPING AND STUDIES OF THE EXPLICIT MULTI-THREADING (XMT) PARADIGM

by

Xingzhi Wen

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Uzi Vishkin, Chair/Advisor
Professor Joseph F. JaJa
Associate Professor Manoj Franklin
Associate Professor Clyde Kruskal
Assistant Professor Tali Moreshet

© Copyright by
Xingzhi Wen
2008

Dedication

To my wife, Lihua, my daughter, Annie and my parents.

Acknowledgements

I would like to express my gratitude to my advisor, Professor Uzi Vishkin, for his guidance and support throughout my doctoral studies. I greatly appreciate the research opportunity he provided to me. Dr. Jacob and Dr. Franklin provided many good suggestions for the research. Dr. Yeung and Dr. Qu also gave me very helpful comments about my work and Dr. Moreshet provided many valuable comments on an earlier version of this dissertation.

Michael Horak graciously found the time to help with English editing in spite of being extremely busy with finalizing his own M.S. thesis. I also want to thank Aydin Balkan, George Caragea, Fuat Keceli, Mary Kiemb, and Alexandros Tzannes as well as other members of the XMT team.

I would like to thank my wife, Lihua and my parents for their patience and encouraging words. Without their support, I would not have finished this long journey.

Table of Contents

List of Figures

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BFS	Breath First Search
BST	Binary Search Tree
CMP	Chip MultiProcessor
CRCW	Concurrent Read Concurent Write
DAG	Directed Acyclic Graph
FPGA	Field Programmable Gate Array
GR	Global Register
GRF	Global Register File
HDL	Hardware Description Language
ILP	Instruction level parallelism
IOS	Independence of Order Semantics
LSRTM	Length of Sequence of Round Trips to Memory
MTCU	Master Thread Control Unit
MC	Memory Controller
MSI	Modified Shared Invalid
OS	Operating System
PRAM	Parallel Random Access Machine(Model)
PS	Prefix Sum (to register)
PSM	Prefix Sum to Memory
QRQW	Queue-Read Queue-Write
SIMD	Single Instruction Multiple Data
SMP	Symetric multi-processor
SMT	Simultaneous multi-threading
SPMD	Single Program Multiple Data
TCU	Thread Control Unit
TLP	Thread level parallelism
XMT	eXplicit Multi-Threading
XMT P	XMT with coherent Private caches
XMT S	XMT Shared caches only (without private caches)

Chapter 1

Introduction

Micro-processor performance has been improved continuously over the past decades, primarily because an increasing number of faster transistors are available on-chip, thanks to the development of the semiconductor industry. Processors can operate at a higher clock rate with faster transistors and more transistors enabled designers to implement various optimizations in hardware, namely targeting extraction of instruction level parallelism (ILP) in hardware, parallelizing execution of serial code.

However, the extensive power consumption in high density chips became a new challenge for further improvement of processor performance with the traditional techniques. For example, clock rates increase with shrinking transistor sizes, but since faster clocks are the major contributor to the increasing power consumption, they are unlikely to increase as they used to.

On top of that, ILP has reached its diminishing return stage. As the finest-grain parallelism, ILP techniques attempt to exploit parallelism among instructions in serial programs. To increase the amount of parallelism, researchers have studied a variety of techniques such as: dynamic scheduling, multi-instruction issue, branch prediction, and register renaming. However, the parallelism available in a serial program is inherently limited [49, 39].

The new trend in processor industry is turning to multi-core processors, as the name indicates, consisting of multiple processor cores in a single chip. These processors are also referred to as CMPs (chip multi-processor). This is a natural choice since it is no longer cost effective to improve the performance of a single core due to design complexity, increased power consumption and diminishing performance increase. In a typical multi-core processor, cores are quite independent of each other since their sharing are limited to lower levels of a memory hierarchy. These multi-core processors are used in the symmetric multi-processor (SMP) systems and each core is scheduled by an operating system (OS) independently.

The XMT (explicit multi-threading) platform attempts to take advantage of the fast-growing number of transistors available on a chip in a different way from simply replicating advanced serial processor cores as in the CMP. The XMT processor is designed to support thread level parallelism using thread-aware hardware, targeting shortened single task completion time. More importantly, XMT is a parallel algorithmic architecture that efficiently executes PRAM (parallel random access machine/model) algorithms.

1.1 Parallel Computing

Although On-chip parallel computing is relatively new, parallel computing based on multiple processors(chips) have been around for many decades [2, 3]. Shared memory and message passing are the two main parallel programming models. In a shared memory parallel computer architecture, multiple processors share the same memory address space and the communication between processors is done through the shared memory implicitly. Programmers do not need to know the physical location of the data. Message passing is an explicit communication method, since processors have to exchange messages to communicate with each other. Generally, it is believed that programming in shared memory is easier than in message passing. With hardware and software support, the combination of shared memory and message passing can be implemented[37, 14].

Communication bandwidth and latency are among the most important measurements of any parallel computer system. Until recently, all parallel computers were built from multiple chips and communications between processors had to cross chip boundary and rely on an off-chip interconnection network. The communication overhead in such a system is very expensive due to the long latencies and low bandwidth. Caches are used in multiprocessors to reduce the latencies, but it has also introduced the coherence problem, resulting from possible multiple copies of a single memory location. Bus snooping and directory based protocols are used to address the cache coherence problem. Bus snooping is simple but it is limited to a small scale of the parallel computer due to the limited bus bandwidth. The directory base cache coherence protocol is scalable, but it needs extra storage space for directories and its hardware implementation is very complicated. It is well known that cache coherence protocols are inefficient for some data access patterns, typically those using extremely fine-grained parallelism. As a result, caches in parallel computers are not as effective as in uniprocessors in providing low-latency, high-bandwidth memory accesses for certain types of programs.

Programming in parallel computing is quite challenging [5]. First, it is difficult to develop a correct parallel program, because of asynchronous events and potential deadlocks from many possible execution paths. It is more difficult for programmers to reason about a multithreaded program than a single-thread program. Second, it is difficult to develop a high performance parallel program, since programmers need to know the details of the targeted parallel computer. The communication overhead in parallel computing is quite difficult to estimate. Indeed, the challenges of parallel programming has been preventing it from broad application for many decades.

1.2 What is PRAM?

The abstract model for serial computing is RAM (Random Access Machine/-Model) and PRAM (Parallel Random Access Machine/Model) is its counterpart in parallel computing. The main assumption of the PRAM is that the latency for an arbitrary number of memory accesses is the same as for one access. PRAM was extensively used in 1980s and 1990s for the theorists to develop and study parallel algorithms for various applications. With PRAM, algorithm designers can concentrate on the problem itself and they are freed from dealing with the details of a specific architecture. However, because of the abstraction, PRAM underestimates the communication overhead that is not negligible in real multi-chip parallel computers. In 1993, Culler et al. published a paper [13] that pointed out that PRAM is an oversimplified model and proposed a new parallel machine model. From the mid-1990s, PRAM was deemed useless and research about PRAM faltered because most researchers abandoned it.

Researchers have tried to approximate the theoretical performance of the PRAM using multi-chip parallel computing, like NYU-Ultracomputer [20] in the 1980s and the SB-PRAM [6, 16, 38] in the 1990s. However, the paper [13] and the later book [14] explain why the high latency, and even more importantly the limited bandwidth among the processors, make it hard to accomplish.

The XMT PRAM-On-Chip project at the University of Maryland is based on the observation that the fast growing number of transistors will make it possible to build a PRAM on a single chip. When multiple cores reside on the same chip, they can be connected with a very high-bandwidth, low-latency network, and the communication overhead among them can be significantly reduced compared to the multi-chip parallel computers. Before we jump into the the details of the XMT, representative CMPs will be reviewed, since both CMP [24, 23] and XMT are single-chip multi-threaded architecture.

1.3 Chip Multi-Processor(CMP))

The first example of CMP, a dual-core processor, was introduced by IBM [28] and now both Intel and AMD deliver quad-core processors. As the name indicates, two or four processors are placed on one chip. The cores in these chips are rather independent and powerful, while they share some chip-wide resources such as a lower-level cache. OS considers these cores as independent processors when assigning a thread or process to them. A CMP system can improve overall throughput of a chip and execute parallel programs like in SMP, but it also has similar programming challenges as SMP.

Recent research[32, 4] on CMP shows that a heterogeneous architecture performs better than a homogeneous one under the same power and area constraints. The advanced serial processors typically apply many microarchitecture optimization techniques to achieve a better single-thread performance, e.g. out-of-order execution, branch prediction, and pipelining. These techniques generally are not cost effective in terms of performance-per-transistor and performance-per-watt. The hardware support of fine-grained parallelism is a desirable feature for future multi-core processors [30].

The CELL[26, 52] processor designed by Sony, Toshiba and IBM is an architecture that is radically different from conventional multi-core processors. The CELL uses a high performance PowerPC core that controls eight relatively simple SIMD cores, called Synergistic Processing Elements (SPE). The processor is designed for the Sony PS3 game console, but it is also considered to be a good building block of high performance scientific computing solutions[52]. Each SPE in CELL has its own local storage and the data transfers from local storage to the global storage need to be taken care of by the programmer explicitly, which makes programming for CELL quite challenging.

The Niagara processor[17, 29] from SUN is a multi-core multi-threaded processor, which is targeted for server applications. The latest generation of Niagara 2 has 8 cores and each core can execute up to 8 threads. The threads are switched to utilize processors better by hiding latencies from cache miss, branch misprediction and exceptions. The power consumption per thread is extremely low, slightly more than 2 Watts per thread. The processor is designed to achieve better throughput with less power consumption, but it is not designed for shortening single task completion time as XMT.

Tile-based architectures, such as MITs Raw [43], Stanfords Smart Memories [33] and UT-Austins TRIPS [41], also expect to scale to high levels of parallelism. The XMT, unlike Raw, Smart Memories and TRIPS, provides hardware support for efficient load balancing and better support for a shared-memory model, both of which are critical for many irregular applications.

Amdahl's law states that the overall speedup a parallel program can provide is limited by the serial portion of the program. Therefore it is reasonable to use a more powerful processor for the serial portion while using many simple in-order RISC processors for parallel execution, which are efficient both in power and area. In the XMT system, we propose to have one powerful MTCU(Master Thread Control Unit) for the serial part, which applies various optimizations for single-thread execution, and have as many as 1024 relatively simple, in-order execution TCUs (Thread Control Units) for the parallel part to achieve better overall area and power utilization. It is also possible for an XMT processor to have multiple MTCUs to support multiple OS threads, where the parallel processors are dynamically assigned to the MTCUs.

Although both CMP and XMT are trying to take advantage of an increasing number of transistors and high on-chip communication bandwidth available, the XMT architecture addresses the long standing challenge: programmability. It does so by natively supporting thread level parallelism in hardware, specifically supporting fine-grained parallelism from PRAM algorithms.

1.4 eXplicit Multi-Threading (XMT)

The XMT project attempts to design an architecture that executes PRAM-style programs efficiently by taking advantage of the low-latency and high-bandwidth communication network available on a single chip. Load balancing that needs to be carefully handled in other parallel computers is naturally achieved in the XMT by a dynamic task assigning scheme. The programming model of the XMT is simple, and there are plenty of PRAM algorithms available which are easy to implement on the XMT. The multi-operand prefix-sum operation, a special instruction introduced in XMT, further lowers the latency of the synchronization among multiple threads and reduces the overhead of generating new threads.

1.4.1 Background review

Early papers on the XMT presented the fine-grained programming model, architectural features and some initial performance results [48, 36, 15, 11, 34, 35, 46, 45, 44].

XMT Programming Model

The programming model underlying the XMT framework is the SPMD (Single Program Multiple Data) programming model that has two executing modes: serial

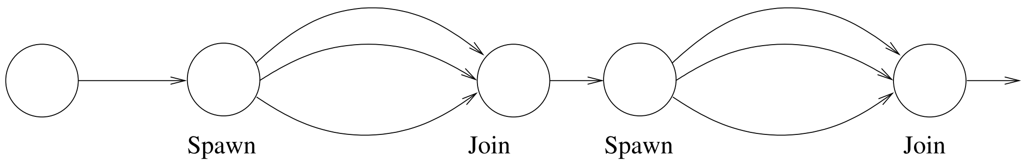


Figure 1.1: Serial and parallel execution modes

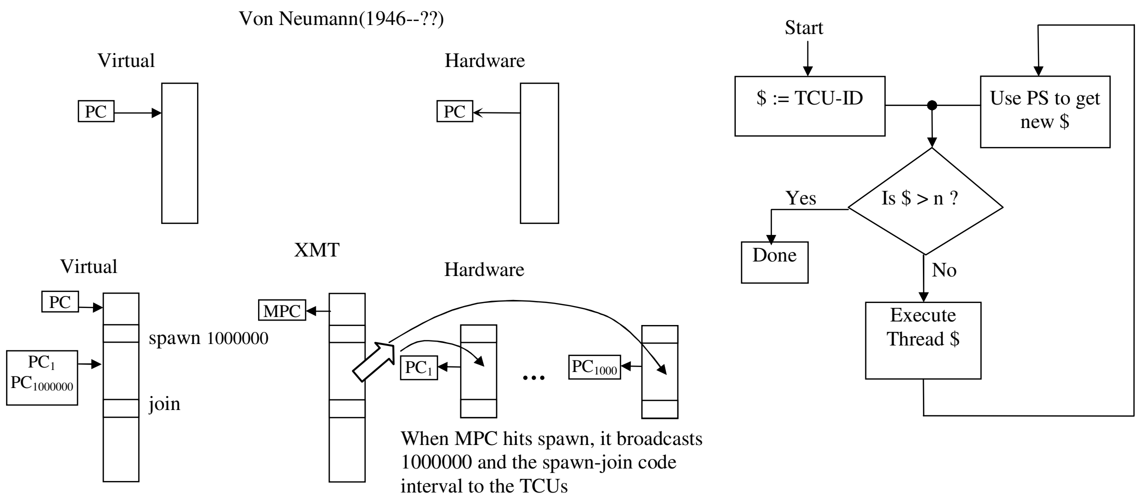
and parallel. The *spawn* and *join*, instructions specify the beginning and end of a parallel section (executed in parallel), respectively. See figure 1.1.

An arbitrary number of virtual threads, initiated by a *spawn* and terminated by a *join*, share the same code. The memory access model of the XMT processor is a hybrid of so-called arbitrary CRCW (Concurrent Read Concurrent Write) and QRQW (Queue Read Queue Write) [18]. The arbitrary CRCW aspect dictates that concurrent writes to the same memory location result in one arbitrary write committing. No assumption needs to be made beforehand about which will succeed. An algorithm designed with this property in mind permits each thread to progress at its own speed from its initiating *spawn* to its terminating *join*, without ever having to wait for other threads; that is, no thread busy-waits for another thread. We call this "independence of order semantics" (IOS).

An advantage of using this easier to implement SPMD model is that it is also an extension of the classical PRAM model, for which a vast body of parallel algorithms is available in the literature [25, 27]. The programming model also incorporates the prefix-sum statement. The prefix-sum operates on a base variable, B, and an increment variable, R. The result of a prefix-sum (similar to an atomic fetch-and-increment [20]) is that B gets the value B + R, while the return value is the initial value of B. The primitive is especially useful when several threads simultaneously perform a prefix-sum against a common base, because multiple prefix-sum operations can be combined by the hardware to form a very fast multi-operand prefix-sum operation. Because each prefix-sum is atomic, each thread will receive a different return value. This way, the parallel prefix-sum command can be used for implementing efficient and scalable inter-thread synchronization, by arbitrating an ordering between the threads.

The XMT Architecture

Perhaps the most important distinguishing characteristics of an XMT architecture are low-overhead mechanisms for the management of parallelism. New elements not present in standard microprocessor design are introduced for the purpose of supporting the parallel programming model. The XMT programming model allows programmers to specify an arbitrary degree of parallelism in their code. Clearly, real hardware has finite execution resources, so in general all threads cannot execute simultaneously. In an XMT machine, a thread control unit (TCU) executes an individual virtual thread. Upon termination of a virtual thread, the TCU performs a prefix-sum operation in order to receive a new (virtual) thread ID. The TCU will then execute the thread with that new ID. All TCUs repeat the process until all the virtual threads have been completed. A Master Thread Control Unit (MTCU) orchestrates the TCUs. Figure 1.2 illustrates this: (i) through a comparison with the von Neumann stored program and program counter apparatus (1.2 (a)), and (ii) through a snippet of the program of a TCU (1.2 (b)).



(a) Program counter + stored program (b) TCU program snippet

Figure 1.2: XMT execution model

We begin with Figure 1.2 (a). Its upper part, entitled "von Neumann (1946–??)," illustrates the program counter apparatus in serial machines, which has dominated general-purpose computing since 1946; it is not yet clear whether and when its reign will end. The right hand side (of the upper part of Figure 1.2 (a)) depicts the hardware apparatus, where one command at a time is brought to the program counter. The left hand side (of the upper part of Figure 1.2 (a)) demonstrates how the programmer is often educated to think about this apparatus–"the virtual outlook". Here the program counter is the one to move; it moves from one location of the memory to another, perhaps like a "book analogy", where the finger of a reader advances from one line of the book to another. The fact that this von Neumann apparatus has survived orders of magnitude improvements in speed since the 1940s makes it a remarkable "Darwinistic success story". For this reason we sought to upgrade, rather than replace in a disruptive manner, this successful apparatus.

The lower part of Figure 1.2 (a), entitled “XMT”, illustrates the new apparatus. The left hand side (of the lower part of Figure 1.2 (a)) depicts the virtual description. There is still one computer program as in the von Neumann apparatus. In the above book analogy, one finger (marked as PC, for program counter) moves from one line of the book to another, until it reaches a special command called *spawn*. The *spawn* command specifies a number of “threads” which can be performed in parallel. Since we discuss now the virtual side, any number of threads can be specified. Figure 1.2 (a) mentions 1,000,000 threads. The virtual threads, initiated by a *spawn* and terminated by a *join*, share the same code. At run-time, different threads may have different lengths, based on individual control flow decisions. The programmer’s understanding will be that each of the threads can progress (guided by one finger per thread) from the *spawn* command to a subsequent *join* command at its own speed. At the *join*, the thread expires. Once all the virtual threads expire, the finger marked PC continues. The main difference in the hardware description on the right-hand-side (of the lower part of Figure 1.2 (a)), is that the number of program counters is fixed (the figure mentions 1,000), and does not change as a function of the *spawn* command at hand. The program counter of the MTCU (denoted MPC) executes the serial code, prior to the *spawn* command. The program counter of the MTCU executes a *spawn* command and then broadcasts the following instructions until a *join* instruction to the other program counters. The program counters start by executing the first 1,000 among the 1,000,000 threads, one thread each. When a program counter completes its thread, it starts executing one of the yet-to-be-executed threads. This is done until all of the 1,000,000 threads finish.

Figure 1.2 (b) illustrates the program of a TCU. Suppose that n = 1,000,000 threads are to be executed as a result of a *spawn* command. The figure assumes that n and the SPMD code were broadcast to all TCUs. TCU i starts by executing the respective virtual thread i, but only if i is not larger than n. Upon finishing the execution of a virtual thread, the TCU uses a prefix-sum computation to obtain the ID of the next virtual thread it should execute, and proceeds to execute it if that ID is not larger than n. Note that the only communication among TCUs is through the prefix-sum computation. An extension of the architecture that allows some nesting of *spawn* commands (using an *sspawn* command, noted later) is not reviewed here.

Expected Performance of XMT

Previous papers on XMT have presented simulation results of the performance of the XMT. The work reported in [48] is the speedup of several parallel algorithms based on hand-coded assembly programs (see table 1.1). The DAGs problem aims to find the longest path leading to each vertex in the directed acyclic graphs. Integer sorting uses bin-sort iterations to sort an array. List ranking finds for each element in the list its total distance from the end of the list, given that each element has a pointer pointing to its successor. Detailed information about these problems and algorithms can be found in [48].

The XMTC compiler made it possible to evaluate the performance of the XMT

Table 1.1: Empirical performance result

		Number of TCUs					
Problem	Input Size	20	50	100	200	500	1000
DAGs	graph4	4.17	7.59	10.71	13.78	16.45	17.72
	graph6	5.79	14.02	26.69	43.97	100.46	159.69
Integer sort	$5 \cdot 10^3$	3.61	7.71	13.21	21.84	40.15	60.88
List ranking	10^3	2.68	5.29	7.83	12.81	23.47	32.49
	$5 \cdot 10^3$	3.09	7.62	14.89	28.53	63.30	106.6

on more complex applications. Figure 1.3 shows the speedup results reported in [34].

The XMT showed good performance on applications which have either small input size or exhibit irregular behavior, for which traditional parallel computing cannot employ a simple static scheme because of potential high load-imbalance or communication overhead. From previous work[48, 36, 34, 35], the speedup of the XMT is about one magnitude order lower than the hardware cost, which is much better than multiprocessor parallel computing[48].

XMT speedups exceeding 100-fold for a standard VHDL gate-level simulation benchmark suite relative to serial were reported in [22].

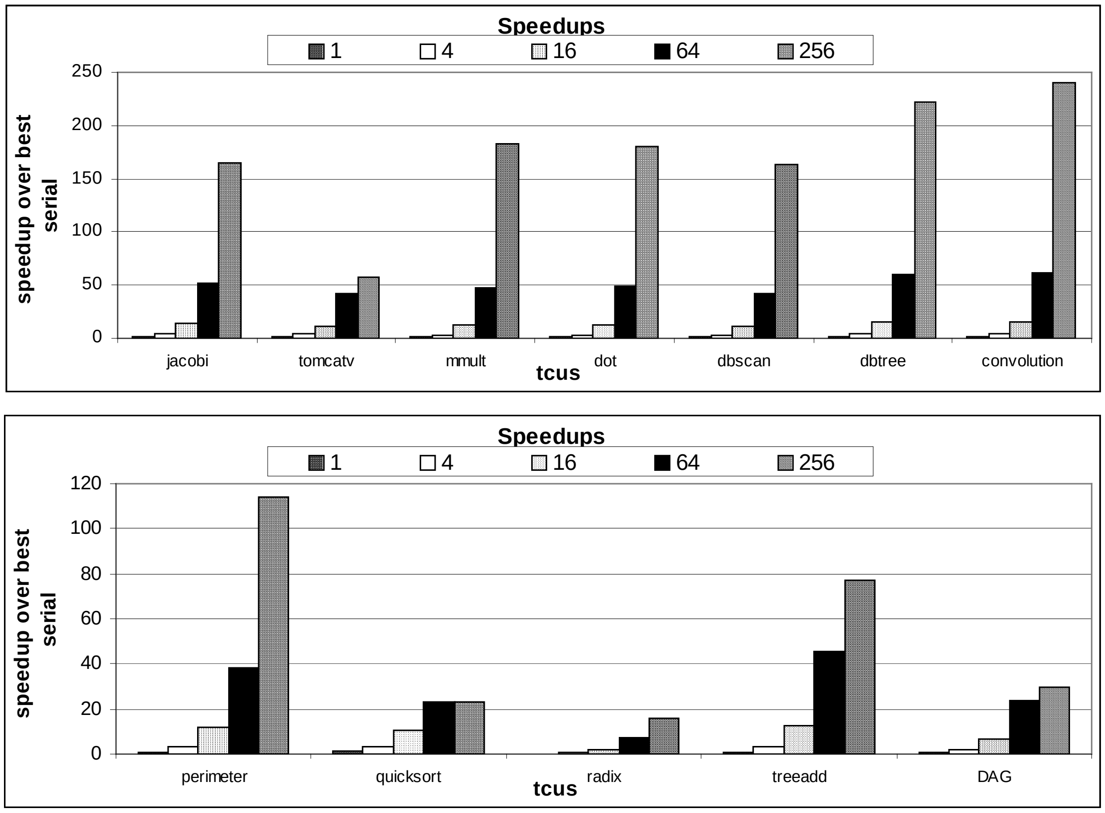


Figure 1.3: Speedups on XMT simulator

1.4.2 Overview of Thesis

This dissertation presents the first hardware implementation of the XMT architecture, which is extended from the high level description and software simulator of the XMT architecture presented in early papers. The performance of the XMT architecture is also studied with an FPGA-based prototype [51, 50].

Following this introduction, the XMT architecture is presented. The components of the XMT processor, transitions between parallel and serial modes, instruction and value broadcasting, software/hardware co-managed prefetch buffer, and the read-only buffer are explained in detail. The overview of the XMT framework is also discussed in this chapter for better understanding of a broad concept of XMT.

In chapter 3, the performance of the XMT processor is studied from different aspects with the XMT FPGA prototype. The execution time of a parallel implementation over the best serial program in the prototype is reported. The absolute execution time (wall-clock time) is compared with a commodity 2.6GHz AMD Opteron processor. The performance of an envisioned XMT ASIC 800MHz processor is projected using the XMT FPGA prototype as a cycle-accurate emulator by slowing down the DRAM.

In chapter 4, the choice of not having coherent private caches is studied. An imaginary system that includes conventional coherent private caches is simulated and its performance is compared to the preferred default configuration: XMT without coherent private caches. The two systems are compared in execution time, traffic in the interconnection network, and average latency for read operations. The simulation results suggest that coherent private caches do not have an advantage in the context of the XMT system, which is designed to support fine-grained, PRAM-based algorithms efficiently.

Chapter 5 concludes this thesis by summarizing the results from this study and briefly discusses the future work towards a complete XMT processor.

The description of the Verilog model of the two XMT prototypes: ASIC and FPGA versions are presented in the appendices. Following an overview of the XMT Verilog model, each chapter of the appendices describes a component of the XMT in great detail. Since the XMT processor is still under development, the description of modules is only valid for the current snapshot. The XMT ASIC prototype is discussed as the default, and the differences between ASIC and FPGA versions are presented in the last chapter of the appendices.

Chapter 2

Architecture of the XMT Processor

In this chapter, the micro-architecture of the XMT will be presented in great detail.

2.1 Micro-architecture of the XMT Processor

2.1.1 Overview

The XMT processor includes a MTCU (Master Thread Control Unit), clusters comprising of TCUs (Thread Control Units) and functional units, an interconnection network, shared on-chip cache modules, memory controllers (MC), a global register file (GRF) and a prefix-sum unit.

Figure 2.1 depicts the block diagram of the XMT processor. The MTCU executes the serial portion of the program and clusters of TCUs execute the parallel sections. MTCU and clusters of TCUs access shared memory space through an interconnection network. The memory space is partitioned into multiple memory modules, which can be independently accessed. The prefix-sum unit can execute common base prefix-sum operations from multiple TCUs in a unit time, which will provide low overhead for extremely fine-grained parallelism.

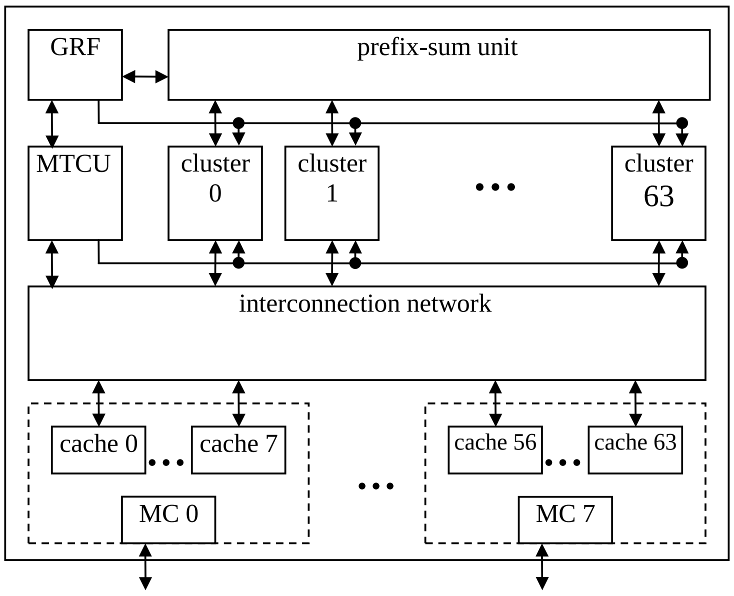


Figure 2.1: Block diagram of the XMT processor

2.1.2 Master TCU

The master TCU is the only TCU active in serial mode. Similar to advanced serial microprocessors, the master TCU can incorporate standard techniques related to ILP, such as branch prediction, out-of-order execution, and register renaming. The main difference between the MTCU and a serial microprocessor is its support of special XMT instructions such as *spawn* and *join*. When the XMT processor switches to parallel mode, the MTCU broadcasts the instructions in the parallel section to all clusters where they are copied to a local instruction buffer and later fetched by TCUs inside clusters. After all instructions in the parallel section have been broadcast, the MTCU waits for TCUs to finish the execution of the parallel section.

The Master TCU has its own cache, L0, that is only active during serial mode and applies write-through. Whenever the local cache in MTCU is written, the shared cache is also updated with the new value. When the XMT processor enters parallel mode, the Master TCU discards its local cache. The overhead of the flushing L0 cache is trivial since write-through mechanism is chosen. The MTCU will have compulsory misses when XMT switches back to serial mode, but it can be solved by advanced hardware/software prefetch. The other option is to apply a hardware cache coherence protocol between shared cache modules and MTCU local cache. If a memory location cached in MTCU local cache is updated by a TCU, it can either update or invalidate the cache line in the MTCU local cache. When XMT operates in serial mode, L0 cache is the first level cache of the MTCU and multiple shared memory modules provide the lower level of the memory hierarchy (similar to a serial processor). With this memory hierarchy, the MTCU mainly relies on L0 in serial mode, instead of shared caches.

2.1.3 TCUs and Clusters

A TCU can execute a thread in parallel mode. TCUs have their own local registers and they incorporate simple in-order pipelines including fetch, decode, execute/memory access and write-back stages. The TCUs have a very simple structure and do not aggressively pursue maximum performance. Given the limited chip area, the overall performance of the XMT is likely better when it has more simple TCUs rather than fewer, but more advanced, TCUs because of the well-known diminishing

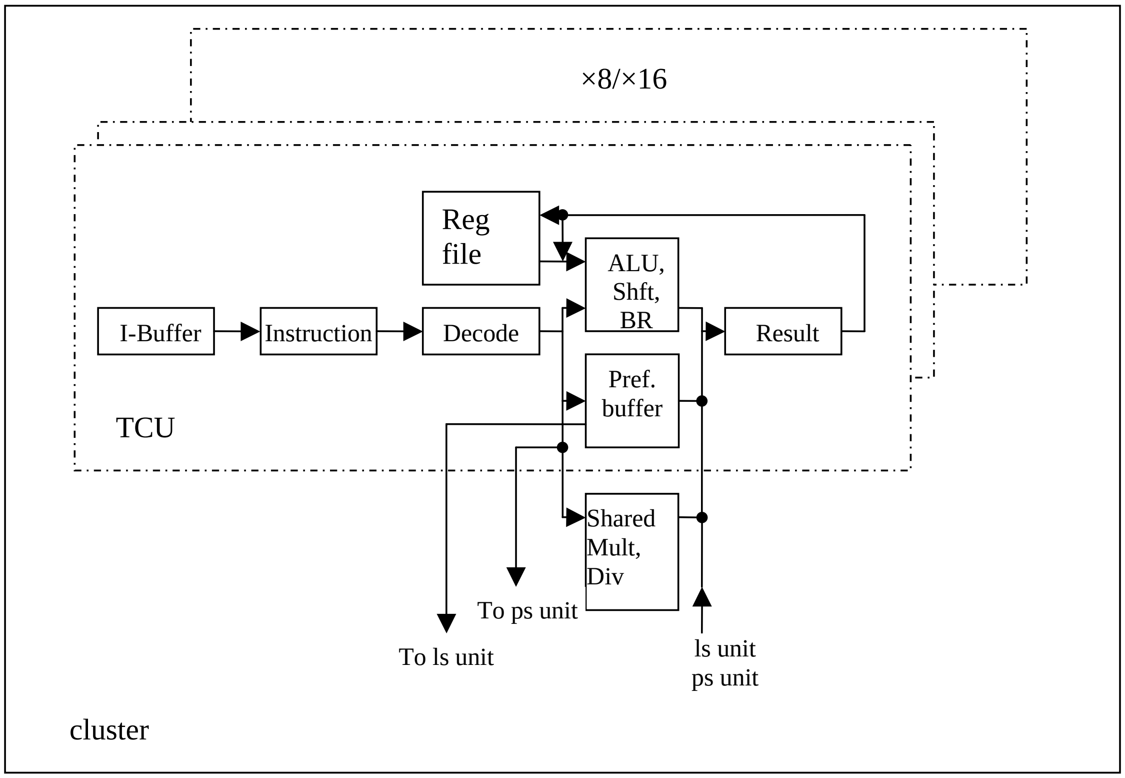


Figure 2.2: Cluster and TCUs

return of many ILP techniques. However, XMT does not prevent TCUs from introducing any advanced techniques, since the TLP (Thread Level Parallelism), which XMT is taking advantage of, is orthogonal to that. A TCU is an in-order processor that only issues a new instruction when the previous instruction has been executed, so there is only one instruction from a TCU being executed by functional units at any moment. A cluster is a group of 16 TCUs and accompanying functional units. TCUs share some functional units with several other TCUs in the same cluster; this is similar to an SMT (Simultaneous Multi-Threading) processor. For complicated and time-consuming operations, such as multiplication and division, multiple TCUs are assigned to one functional unit. Some simple functional units are not shared, because the overhead of sharing is comparable with that of replicating them and it may be intensively used by TCUs. If several TCUs assigned to a functional unit try to access it, proper arbitration is used to queue all requests. Each cluster has one load/store port to the interconnection network, which is shared by all TCUs inside the cluster. Prefix-sum requests from TCUs are combined in the cluster and sent to the prefix-sum unit. Figure 2.2 shows the block diagram of the cluster.

2.1.4 Prefix Sum Unit and Global Register File

ps R, B, an individual prefix-sum, indicates the following atomic operations.

$R <= B$, R will get the old value of B.

$B <= B + R$, B will get the sum of B and R.

B is called base register

A series of individual prefix-sum instructions, *ps* R_0, B, *ps* R_1, B $\cdots$, will result in

$B <= B + R_0 + R_1 + R_2 + \cdots$

$R_0 <= B$

$R_1 <= B + R_0$

$R_2 <= B + R_0 + R_1$

$\vdots$

In XMT, the common base prefix-sum instructions from different threads can be executed by the prefix-sum unit simultaneously. In this case, the result of the execution of these prefix-sum instructions are the same as executed in serial mode at an arbitrary order, but the execution will take a constant time on XMT, regardless of how many threads participate. Due to the hardware implementation challenges, the incremental values are limited to 0 or 1. There is no limit on the base register and the result of a ps operations is 32-bit value. This hardware implemented multi-operands prefix-sum operation can be used in efficient inter-thread synchronization by arbitrating an ordering between threads.

TCUs send 1 bit input for a ps operation request and get the prefix-sum results in local registers. The prefix-sum unit is based on binary tree implementation. After the prefix-sum unit finishes calculation, each TCU gets its own value (prefix-sum of inputs with initial value of 0 in base register) and the base value broadcast from the prefix sum unit; then, the final results are the sum of these two numbers which is calculated locally by TCUs. The registers in GRF are used for base registers of the prefix-sum operations. The master TCU can access these registers directly, but regular TCUs can only access them through a prefix-sum operation. There is only one prefix-sum unit, but the global register file has multiple registers, so a proper sharing mechanism is needed. At any cycle, the inputs of the prefix-sum unit will handle only one base register. Requests to other base registers have to wait their turn. The base register number for a particular clock cycle is determined by the requests from all TCUs. Each TCU has multi-bit output, where each bit represents a base register, and assert 1 at a proper bit when there is a request for a ps operation. In other words, there is a decoder for the ps operation. The bit-wise logic OR of all these outputs of the decoders from TCUs are used to determine the next base register. The base registers are chosen in a round-robin fashion among those who actually have a request. This may introduce some delays in the beginning of a prefix-sum operation for a new base register, but not for the following operations.

2.1.5 Memory Modules and Hashing

To provide better bandwidth, XMT uses multiple memory modules, which operate independently. Clusters and memory modules are connected by an interconnection network. The memory address space is evenly divided among these memory modules. The shared caches are used for instructions only by the MTCU, not for TCUs, since the instructions for TCUs are broadcast by the MTCU and buffered in the instruction buffer. Within each memory module, the order of operations to the same location is preserved and a store operation can be acknowledged as soon as the cache module accepts the request, regardless if it results in a cache hit or miss. Each memory module also has an adder to support a fetch-and-add operation for any value of increment, unlike the ps instruction that can only support a binary input, 0 or 1. Resembling the NYU-Ultracomputer fetch-and-add, the prefix-sum-to-memory (psm) instruction is executed on cache modules and they are executed one after the other (in contrast to the ps instruction that processes requests concurrently).

The memory space is divided evenly among memory modules. Each memory request will be dispatched from a cluster to one of those memory modules. In our

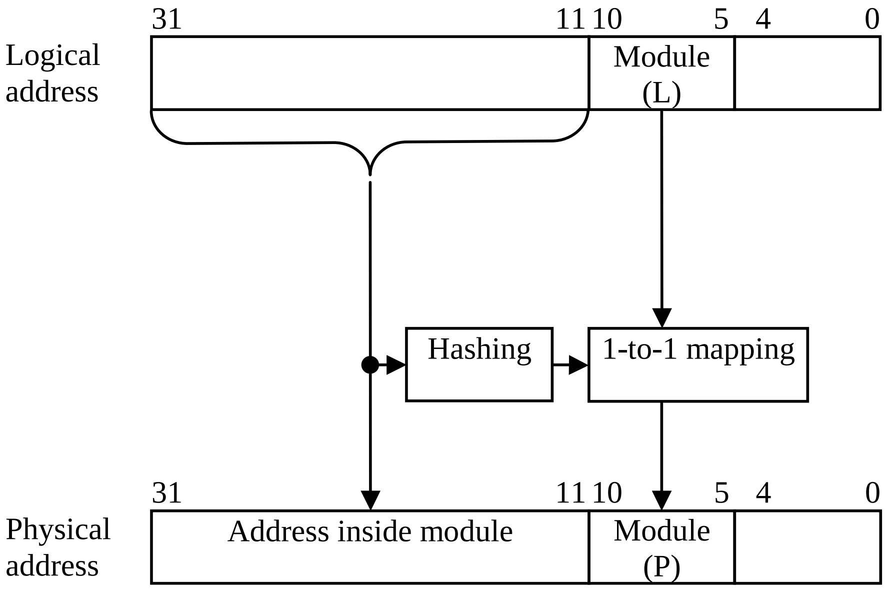


Figure 2.3: Physical address calculation with hashing

implementation, we assumed that the address is 32 bits and cache line is 32 bytes, which is the base element of mapping. One possible implementation of the allocation of memory addresses to memory modules is outlined below. Some bit fields of the address can be used to determine the cache module, such as the Module(L) (L means logical address) field consisting of bits 5 through 10 (4:0 is used as offset inside a cache line), as shown in figure 2.3. Bits 5 through 10 are chosen to have the smallest granuality, a cache line, in distributing to different cache modules. But then code exhibiting certain regularities could result in an imbalance among the number of accesses to different memory modules. The basic idea that has been used in the literature for coping with such imbalance is to employ the concept of hashing. This thesis does not claim that the problem of how best to use hashing for reducing imbalance, as well as for achieving the best overall performance, has been resolved. We only present a straw man solution for this fundamental problem, as we plan to thoroughly investigate it in the future. For a group of addresses that share the same upper field with different Module(L), a 1-to-1 mapping is used to map the natural module index (Module(L)) to the module index (Module(P), P means physical address), as in figure 2.3, so that they are always distributed to 64 memory modules. The hashed value is used to choose different 1-to-1 maps for different groups. Note that there are 64! 1-to-1 mappings of the set $0,1,\cdots,63$ onto itself.

2.1.6 Interconnection Network

Since an efficient interconnection network is so important to the XMT processor, we decided to custom design it and study its performance carefully. The behavioral description was incorporated into the current verilog HDL description to allow simulation of applications. The network has uniform delays for any source-destination pairs and it is designed to have zero interference between traffic for different destinations unless the network is heavily loaded or extremely unbalanced. More detailed description of the interconnection network appears in [8, 9, 7, 10].

2.2 Program Execution Flow

2.2.1 SPMD Programming Model

The XMT uses fine-grained SPMD (Single Program Multiple Data) programming model. The actual performance model is a hybrid of so-called arbitrary CRCW

(Concurrent Read, Concurrent Write) and QRQW (Queue-Read, Queue-Write) [18]. XMT program starts with serial mode and it changes to parallel mode when a *spawn* instruction is executed. The spawn instruction carries two parameters: *low* and *high*, meaning the parallel thread id ranges between *low* and *high* inclusively. After all parallel threads finish, the MTCU executes *join* instruction and the processor changes back to serial mode. See figure 1.1 on page 9. When the *spawn* instruction is executed, the *low* and *high* are copied to global registers, GR-LOW and GR-HIGH respectively. GR-LOW is used as the base of prefix-sum operation, that assigns new thread IDs to the new available TCUs. GR-HIGH is used to store the upper bound of the valid thread ID and TCUs can generate a new thread during parallel mode by increasing GR-HIGH through *sspawn*, a prefix-sum operation to GR-HIGH with incremental value of one.

2.2.2 Parallel and Serial Mode Switch

The one important difference between the XMT and CMP is who manages the parallel threads. In CMP, normally the the OS (Operating System) is aware of the existence of the parallel threads and OS is responsible for the scheduling of the threads on the available processors. In the XMT, the parallel threads are not OS threads. All parallel threads are managed by hardware, this includes TCU assignment, termination of threads and completion of all parallel threads.

The change from serial to parallel mode is simpler than the reverse. When the MTCU encounters a *spawn* instruction, it initializes both GR-LOW and GR-

HIGH and the XMT processor enters parallel mode. The MTCU also starts the broadcasting of instructions in the parallel section through the instruction bus at the beginning of parallel mode.

The transition from parallel to serial mode is more involved because of the *sspawn* instruction that can increase the GR-HIGH dynamically (see figure 2.4. The *sspawn* instruction is a prefix-sum operation with GR-HIGH as the base register and incremental value of one (1). The differences between *spawn* and *sspawn* instructions are listed in table 2.1.

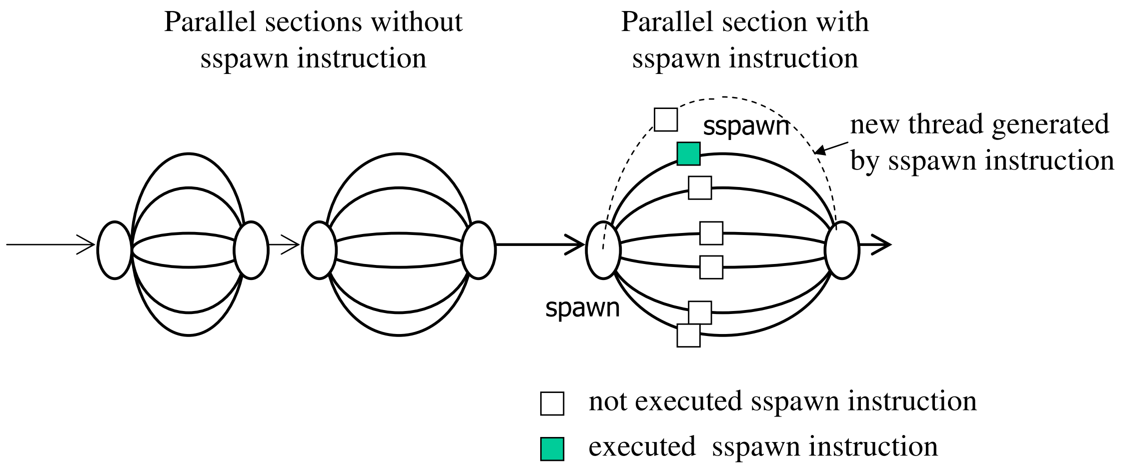

Parallel sections without sspawn instruction
Parallel section with sspawn instruction
sspawn
new thread generated by sspawn instruction
spawn
□ not executed sspawn instruction
🟩 executed sspawn instruction

Figure 2.4: Single Spawn (sspawn). A parallel thread can generate a new thread by increasing GR-HIGH. The figure shows that not all threads executed the sspawn instruction in a parallel section because of different execution paths

When the XMT processor enters parallel mode, GR-LOW and GR-HIGH, two designated global registers, are loaded with the low and high bounds of thread IDs and the value of GR-HIGH is broadcast to all TCUs. The spawn instruction prompts the master TCU to broadcast the instructions between the spawn and join

Table 2.1: Comparison between spawn and sspawn instruction

	spawn	sspawn
executed by	MTCU	TCU
executed in	serial mode	parallel mode
number of generated threads	any	1

instructions and then wait for completion of the parallel execution. Clusters, groups of TCUs, are activated by broadcasting the instructions (link ① in Figure 2.5). The broadcasted instructions are saved to local instruction buffers at clusters. Such broadcasting is more efficient than having the TCUs fetch instructions from main memory and load the memory system with intensive read requests. However, the number of instructions that can be stored in the instruction buffers is limited and TCUs have to fetch instructions from shared caches through the interconnection network for larger parallel sections.

A TCU executes one virtual thread at a time. TCU i starts by executing virtual thread i (its natural ID). When the number of virtual threads exceed the number of TCUs in the XMT processor, a TCU may serially execute multiple threads. A TCU obtains a new virtual thread whenever it finishes executing the previous one. Unlike the assignment of the virtual thread IDs to the TCUs for the first virtual thread they execute, more effort is needed for later ID assignments. This is done using dedicated prefix-sum hardware with GR-LOW as the base. Upon reaching a join

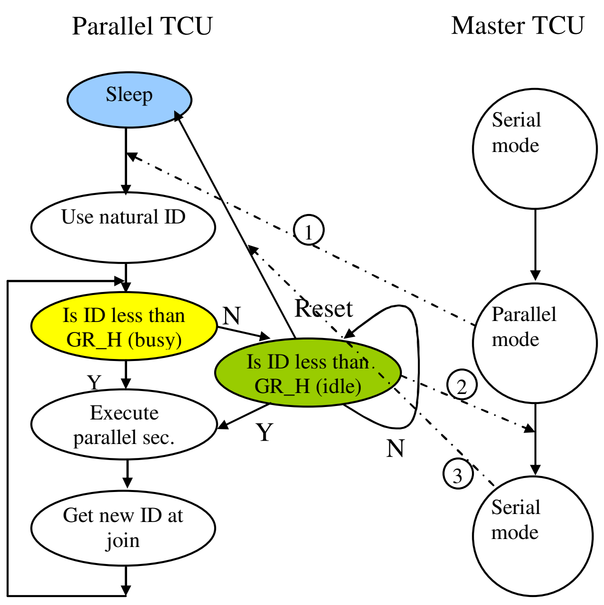


Figure 2.5: Transitions between serial and parallel modes with sspawn

instruction, a TCU gets a new virtual thread ID from the ps operation and jumps to the beginning of the parallel section. The TCU either enters an idle state when its allocated ID exceeds GR-HIGH (namely, it is beyond the scope of the current spawn command), or executes the parallel section with the valid thread ID.

It is possible that an allocated thread ID that exceeds GR-HIGH at a certain point of time becomes valid later, after GR-HIGH increases. To handle this case, TCUs in idle state will keep comparing its ID with GR-HIGH. If the ID becomes valid due to an *sspawn* in another TCU, the TCU will execute the virtual thread immediately. The MTCU is capable of detecting when all TCUs are in their idle state by checking the logic AND of all idle flags as this will signal that the execution of the current parallel section has been completed. Then the MTCU would change execution mode back to serial mode and continue (link ② in Figure 2.5). The Master

TCU resets all clusters after it changes back to serial mode (link ③ in Figure 2.5).

2.2.3 Observation on Synchronization Needs of Nested Spawn (sspawn) Implementation

In the current XMT processor model, nesting of parallel sections is allowed through the use of a *sspawn* instruction. The basic mechanism of increasing the number of virtual threads that need to be executed by a prefix-sum to register GR-HIGH was noted before. But, should the programmer use nested spawn commands, and have the compiler translate them to *sspawn* commands, or use *sspawn* commands directly? This question is beyond the scope of the thesis. However, regardless of how this question is resolved, proper management of the *sspawn* instruction is necessary to prevent potential explosion in the number of new virtual threads (and possibly the memory that will be needed for storing initialization data). Quite a few studies, including Leiserson's MIT Cilk project [12], have considered ways for coping with the memory explosion possibility, and we do not have anything new to add to it. Below, we chose to highlight one new subtle observation with respect to the initialization data. The observation appears to be fundamental and could be of wider interest beyond just the XMT platform.

An attractive feature of XMT is its so-called independence of order semantics (IOS). While applicable in different ways to different levels of abstraction, IOS means that a parallel thread can advance to the end of a parallel section (i.e., until it reaches a *join* command) without ever needing to wait for any other thread. In general, IOS greatly reduces (though it does not always completely eliminate, as the new observation below shows) busy-wait (or spin-wait) between parallel threads. (As an example for IOS, consider use of a ps command to GR-HIGH by a thread executing a *sspawn* command. The relevant point is that this parent thread needs only wait for the feedback from the ps functional units but not on any other thread.) However, the observation presented below will lead to the conclusion that the introduction of *sspawn* instruction makes some synchronization between the parent thread and child thread inevitable.

The busy-wait inevitability observation: Suppose that the parent thread is responsible for: (i) initializing input data for the child thread, and (ii) declaring it (through an *sspawn* command). Then the child thread must busy-wait at least once on its parent thread.

To understand the observation, note that the parent thread first gets an ID for the child thread only when it gets its result from the ps functional unit that updates GR-HIGH. Only after the parent thread has the ID it can store initialization data (or even just a pointer to such data) for the SPMD-type program of the child thread. Recall that SPMD programs can distinguish between threads only based on their IDs. However, without imposing synchronization on the child thread, nothing can prevent the child thread from getting started before parent thread has finished initializing data. After understanding the problem, the solution is not difficult. The child thread simply needs to wait for a signal from the parent thread that it finished storing initialization data.

2.3 Features of the XMT Processor Memory Hierarchy

2.3.1 Overview

A cache system is very important for any processor design including the XMT processor. As a part of multi-level memory hierarchy, the cache plays a very important role in mitigating the speed mismatch between the processor and off-chip memory. Most importantly, the cache system can take advantage of: (1) spacial locality and (2) temporal locality. Although there are some limitations on the cache [26], the cache system is still an essential part for better performance of processors. The XMT processor uses caches as part of the memory hierarchy, but also adopted other types of storage components, which will be explained in this section.

As described in chapter 1, each core in most multi-core processors has a private L1 cache, but a certain lower-level (L2 or L3) cache is shared by all cores in the chip. As a result, a cache coherence protocol is needed for such a system. The two popular cache coherence protocols are bus-snooping and directory-based protocol. It is widely believed that bus-snooping will not scale well beyond 4 or 8 cores. Directory-based cache coherent protocols scale up much better, but the overhead increases with the number of processors and it becomes very expensive for certain cases, like false cache line sharing. When there is false cache line sharing, which occurs when two or more processors write to different words in the same cache line, the cache line needs to be transferred back and forth between those processors.

The XMT processor is designed for fine-grained parallelism and it is supposed to have many more TCUs than that of current multi-core processors. After careful evaluation and comparison (see chapter 4), we concluded that the XMT will not have local caches. Instead, it will only rely on shared caches. In section 2.1, it is stated that the TCUs and shared caches are connected by a high-performance interconnection network. Because there is no local cache for TCUs, each and every memory access needs to make a round trip to a cache module through the interconnection network, if no special optimization is applied. The concept of the length of sequence of the round trip to memory (LSRTM) [47] accounts for the effects of the long memory access latency in the performance of an algorithm. To alleviate long memory access latency, the following features are incorporated in the XMT architecture: (1) value broadcasting, (2) release consistency, (3) hardware/software co-managed prefetch buffers per TCU, and (4) hardware/software co-managed read-only buffer per cluster. These four improvements will be discussed in this section. These features, together with compiler optimizations, enable the XMT to perform at least as good as an alternative XMT architecture that incorporates cluster-level private caches with a directory-based cache coherence protocol, as shown in chapter 4.

2.3.2 Value Broadcasting

Consider the following implementation problem in a parallel algorithm. Suppose that all, or nearly all, threads of a parallel section of an XMT program use a certain variable. Without giving special attention to this case, each thread needs to read the variable through the interconnection network. Furthermore, the read requests will be queued at the same memory module and handled one at a time, significantly increasing the implementation overhead of the parallel algorithm. Suppose further that the variable can be determined at run time and it does not change during this particular parallel section. The broadcasting mechanism reduces the execution time of such concurrent reads. To broadcast a parameter to all parallel threads, the compiler moves the *lw* instruction to the serial section and two broadcasting instructions, *broadh* and *broadl* are inserted to the original place of the *lw* instruction in parallel section. (See ① in figure 2.6.)

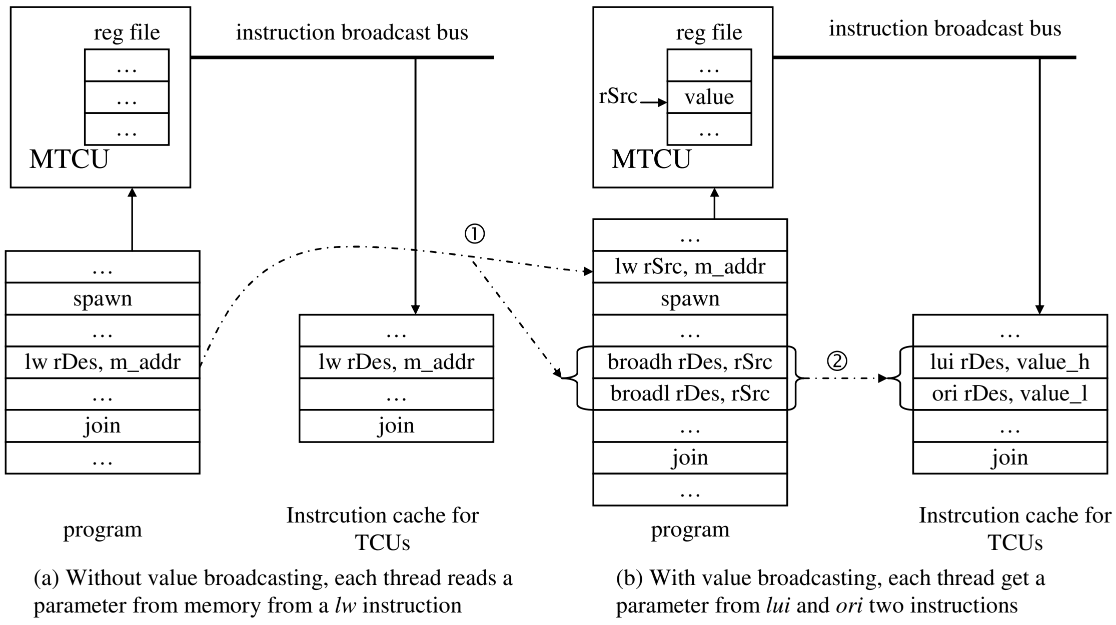
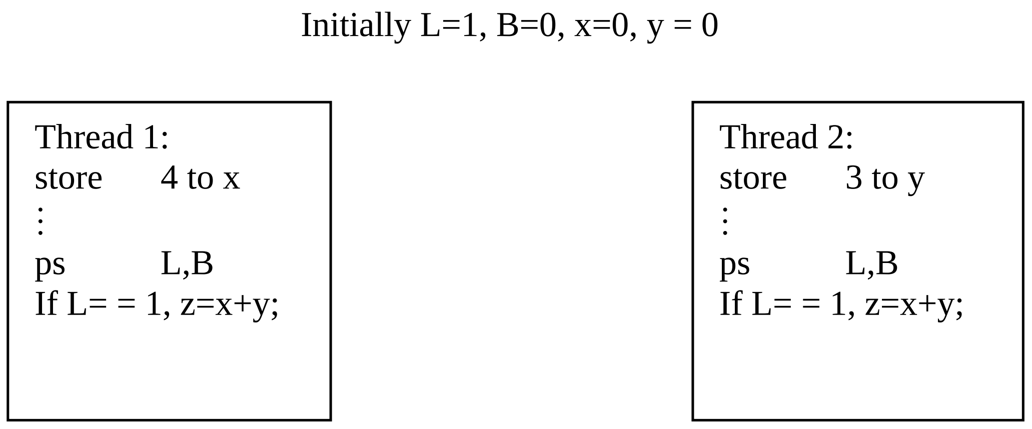

reg file
instruction broadcast bus
…
MTCU
spawn
lw rDes, m_addr
join
program
Instrcution cache for TCUs
(a) Without value broadcasting, each thread reads a parameter from memory from a *lw* instruction
rSrc
value
①
lw rSrc, m_addr
broadh rDes, rSrc
broadl rDes, rSrc
②
lui rDes, value_h
ori rDes, value_l
(b) With value broadcasting, each thread get a parameter from *lui* and *ori* two instructions

Figure 2.6: Broadcasting a value to virtual threads

When the MTCU broadcasts instructions, the *broadh* and *broadl* instructions are replaced by *lui* and *ori* instructions respectively. A 32-bit immediate value can be encoded in the *lui* and *ori* instructions and a register can be loaded with the immediate value. The MTCU composes the *lui* instruction using the value (upper 16 bits) of the MTCU register, which was loaded with the parameter by the *lw* instruction in serial mode. Similar transformation happens from *broadl* instruction to *ori* instruction. (See ② in figure.) Because both *lui* and *ori* can only carry 16 bits, two instructions are needed to update a 32-bit register.

2.3.3 Release Memory Consistency

The memory consistency model in a shared memory programming model defines how the memory system will appear to the programmer [1]. Informally, the ordering of memory accesses needs to be specified. In XMT, memory accesses by different TCUs are not subject to any ordering requirements because of IOS. Only the serial order within a thread needs to be kept. Non-blocking stores, with introduction of the store buffer, are used in uniprocessor systems to reduce the execution time of a store operation. In such a system, the latest value is used for instructions that follow by checking the store buffer before sending a load request to the memory system. In the XMT processor, this technique may not be applicable, since a read request may come from a different TCU rather than the TCU that wrote the location. If we use non-blocking stores, due to the unpredictable latency of the interconnection network for different source-destination pairs, store operations from a TCU may get delayed and result in an incorrect outcome. For example, in the program in Figure 2.7 (a fragment from tree-add program), two threads store a value to a different variable and then perform a ps operation with respect to a common base, B, which gives an ordering of two threads. The later one (that gets 1 in L after the ps operation), will add the two variables, x and y, which were written by the 2 threads earlier. A programmer expects that the later thread will read updated values for both variables, x and y. However, because the latency in the interconnection network depends on source and destination, if non-blocking stores are used, the old value may be read incorrectly. The possible values for z after the execution are 3, 4 or 7. To prevent this behavior, which is incorrect since it does not match the memory consistency model of the XMT system, a fence instruction, which blocks the TCU execution until all stores are committed to the shared cache, can be placed before the ps operation.



```
Initially L=1, B=0, x=0, y = 0

Thread 1:                    Thread 2:
store    4 to x              store    3 to y
⋮                            ⋮
ps       L,B                 ps       L,B
If L= = 1, z=x+y;            If L= = 1, z=x+y;
```

Figure 2.7: Illustration of the need for a blocking store

However, if no virtual thread reads the location after a thread writes to it during parallel mode, it is safe to use a non-blocking store without a fence instruction to improve performance of the XMT. Note that even for non-blocking stores, it is necessary that they all commit to the shared cache modules when an XMT processor switches the execution mode between parallel and serial modes. Otherwise, the interconnection network needs to be flushed prior to the switch. This is managed at the cluster level. A cluster records the number of the non-blocking store requests sent to the memory modules and also counts the acknowledgements received for non-blocking stores. A cluster flags itself as idle only when these two numbers are equal and all TCUs inside are in the idle state.

The XMT compiler is responsible for the insertion of the fence instruction. Whenever there is a synchronization between threads, for example, parent and child threads, the fence instruction needs to be inserted in the proper location. In addition to non-blocking stores, the XMT architecture also supports blocking stores, which prevent a TCU from advancing to the next instruction until it receives the acknowledgement. The blocking store instruction is functionally equivalent to a non-blocking store followed by a *fence* instruction. The basic solution for the compiler is to use blocking stores as default and only replace them with non-blocking stores, with or without a fence instruction, when it is possible.

2.3.4 Prefetch Buffer

Because there is no local cache in XMT, each and every memory access is supposed to make a round trip to shared cache through the interconnection network. This would be very expensive if the TCU stalls for this long latency operation. If a

TCU can prefetch the value in advance, then when the actual read occurs, the value will be available immediately. Prefetch operations are essentially non-blocking reads, in the sense that the TCUs can execute other instructions while the non-blocking read operations are served by the interconnection network and memory hierarchy.

Each TCU has a small number (4 in the XMT FPGA prototype) of prefetch buffers. Each buffer keeps an address and value pair. Each buffer also stores the status of the buffer to track the pending requests. When a TCU sends a prefetch request, it only sends the address of the memory location without specifying the destination register, since the value will not be written into a register. Later, when the TCU sends a normal read request, the prefetch unit will check if any prefetch buffer matches the address of the request. If one of the prefetch buffers matches the request, the TCU will get the value immediately if the value is available in the buffer. Otherwise, the request will be blocked until the value is delivered by the interconnection network. The prefetch buffer will be cleared when the XMT processor changes the execution mode to serial.

The prefetch instructions are inserted by the compiler. Instruction scheduling is a typical compiler optimization and it can be used for prefetch instruction insertion. Prefetch operations can also be used in loops by simply prefetching values for iterations ahead.

2.3.5 Read-only Buffer in Cluster

The TCUs in a cluster share one load/store port, which is connected to the interconnection network. When multiple TCUs try to read the same memory location, it is desirable to send only one request to the interconnection network. This will reduce load on both the interconnection network and the shared cache. One possible solution would be relying on the arbitration logic for the load/store port, that compares the address and combines requests if possible. The limitation of this method is that the arbitration logic is complicated because address comparison and keeping track of combinations is needed. The other limitation of relying on the arbitration logic is that it can only combine requests when they are sent at the same time or within a very short period of the time, so that they are presented to the arbitration logic at the same cycle.

Our solution to the problem is relying on both hardware and software. In hardware, each cluster has a read-only buffer, where the addresses and values are stored. The addressing of the buffer uses index and tag fields as they are used in a typical cache system, except that the read-only buffer keeps a tag per word and not per cache line. There are two different read operations, depending on whether the value can be stored in the read-only buffer, for sharing. In the *individual read* operation, the read-only buffer will not be updated when the value is returned by the shared cache, while the *shared read* operation results in an update. The compiler is responsible for choosing between individual and shared read instructions, based on whether the read is shared among the TCUs.

Before forwarding a read (either shared or individual) request to the interconnection network, the read-only buffer is first checked to find the value. If the value is found in the buffer, the result is sent back to TCU and no request is sent out to the cache system. If the value is not available, but a request for that memory location has been sent out as a shared read request, the read-only buffer stores the requestor's TCU ID for that shared read request. Again, no request needs to be sent to the cache system. If the read-only buffer does not have the value available and no shared read request has been sent out for the particular memory location, the shared read request will be sent out to the interconnection network and mark the location's status as pending.

With hardware and software co-design, the TCUs can share read operations more effectively. The read requests do not need to be issued within a limited time as the other solution requires. The arbitration network does not need to provide combination functionality, that would result in a complicated design in terms of both logic and area. The read-only buffer also provides a limited temporary locality feature as cache does, because the address and data pair will be available until they are cleared by replacement data or when the XMT processor changes the execution mode to serial.

2.4 Macro-architecture of the XMT Processor

The whole is greater than the sum of the parts. XMT as a broad concept covers the transition from how to program, to compiler optimization, and finally hardware implementation. This dissertation focuses on the last component, hardware implementation, but brief descriptions of the other components are included for completeness.

The book chapter [47] presented how to develop an efficient XMT program from concept to implementation. Figure 2.8 shows the proposed methodology. As shown in the figure ($1 \rightarrow 2 \rightarrow 3$), a PRAM program can be developed from a high-level work-depth description through a middle node, work-depth model. An XMT program need to take path ($1 \rightarrow 2 \rightarrow 4 \rightarrow 5$) or ($1 \rightarrow 4 \rightarrow 5$), in case a shortcut is possible.

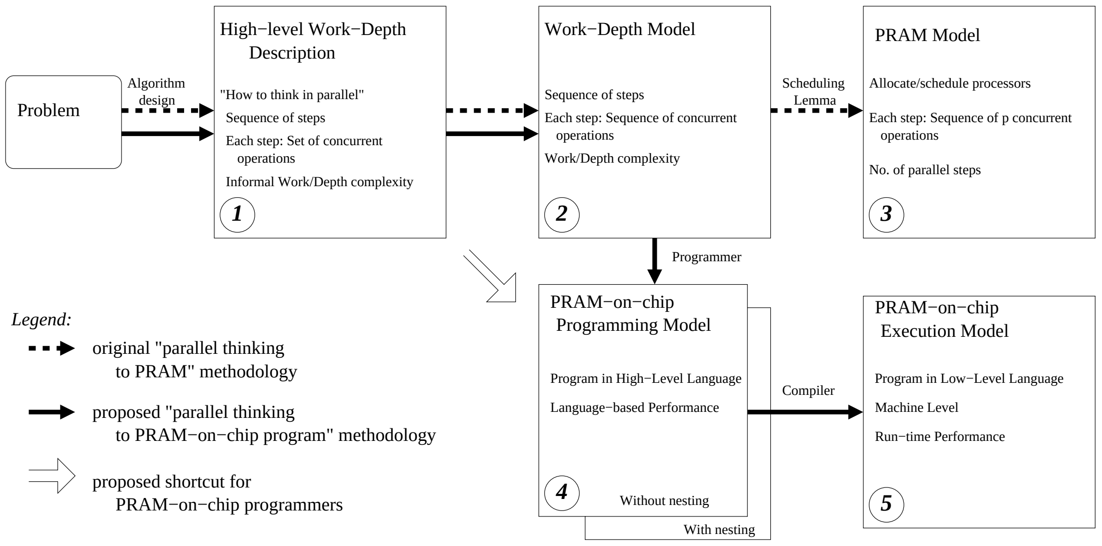

Problem
Algorithm design
High−level Work−Depth Description
"How to think in parallel"
Sequence of steps
Each step: Set of concurrent operations
Informal Work/Depth complexity
1
Work−Depth Model
Sequence of steps
Each step: Sequence of concurrent operations
Work/Depth complexity
2
Scheduling Lemma
PRAM Model
Allocate/schedule processors
Each step: Sequence of p concurrent operations
No. of parallel steps
3
Programmer
PRAM−on−chip Programming Model
Program in High−Level Language
Language−based Performance
4
Without nesting
With nesting
Compiler
PRAM−on−chip Execution Model
Program in Low−Level Language
Machine Level
Run−time Performance
5
Legend:
original "parallel thinking to PRAM" methodology
proposed "parallel thinking to PRAM−on−chip program" methodology
proposed shortcut for PRAM−on−chip programmers

Figure 2.8: Proposed methodology for developing PRAM-On-Chip programs in view of the work-depth paradigm for developing PRAM algorithms

2.4.1 Development of PRAM-On-Chip (XMT) Program

2.4.1.1 Work-depth Model

Introduced in [42], the work-depth model has been widely used in designing and reasoning about PRAM algorithms. Depth is the number of steps an algorithm takes, assuming unlimited hardware is available, and work is the number of operations performed in overall parallel steps.

High-Level Work-Depth (HLWD) description A HLWD description consists of a sequence of parallel rounds, each round being a *set* of any number of operations that can be performed concurrently. A HLWD description is an informal algorithmic description that needs to be translated to a more concrete work-depth model.

Work-Depth model In the Work-Depth model an algorithm is described in terms of successive time steps, where the concurrent operations in a time step form a sequence; each element in the sequence is indexed from 1 to the number of operations in the step. The Work-Depth model is formally equivalent to the PRAM. For example, a work-depth algorithm with $T(n)$ depth (or time) and $W(n)$ work runs on a p-processor PRAM in at most $T(n) + \lfloor \frac{W(n)}{p} \rfloor$ time steps. The work-depth model does not allow nesting of parallelism to remain unresolved.

2.4.1.2 PRAM-On-Chip (XMT) Programming Model

The PRAM-On-Chip programming model needs to solve two problems properly. (i) *Programmability:* given an algorithm in the HLWD or Work-Depth models, the programmer's effort in producing a program should be minimized; and (ii) *Implementability:* effective compiler translation of the program into the XMT execution model should be feasible.

The XMT processor uses a fine-grained SPMD (Single Program Multiple Data) programming model and it has two executing modes: serial and parallel. The two instructions, *spawn* and *join*, specify the beginning and end of a parallel section (executed in parallel), respectively. Any number of parallel threads can be declared by two integer values: low and high, which means that the thread IDs range between low and high, and all threads can be executed concurrently. Two important primitives in the XMT programming model are described below.

Prefix-sum operation The prefix-sum (ps) is essentially a multi-operand fetch-and-add operation. The primitive is especially useful when several threads simultaneously perform a ps against a common base, which will take a constant time regardless how many threads are joining the operation. When each of incremental variables have value 1, all threads will receive different return values, which can be used for (i) load balancing (threads assignment to processors), and (ii) inter-thread synchronization.

Nested parallelism A parallel thread can generate a new thread. The support of nested parallelism allows XMT programmers to take a shortcut from 1 to 4 in figure 2.8, instead of having a full stop at 2.

An XMT-C program example is shown below. XMT-C, which is used as the main performance programming language for XMT, is an extension of the standard C language. As an example, simple XMT-C code for the so-called "compaction problem" is shown below. Given an array A of size n, and a binary array B of size n, the compaction problem is to compact all elements in A[i] for which B[i] is 1 into an array D, where order in D does not matter.

```
psBaseReg x =0;
spawn(0,n-1){
    int e;
    e = 1;
    if(B[$]) = = 1){
        ps(e,x);
        D[e] = A[$];
    }
}
```

First, in serial mode, the global register x is declared and initialized to 0. Then the spawn instruction activates parallel mode where there are n threads in the range 0 to $n - 1$. $ is the thread ID and, if the corresponding value in array B is 1, the thread will do a prefix-sum operation to the global register x and then move the data in A[$] to D[e]. The threads which copy A[$] to D[e], will get a unique value from e by doing a prefix-sum operation on x with value 1.

2.4.1.3 PRAM-On-Chip (XMT) Execution Model

With the XMT execution model, the execution time of an algorithm can be analyzed. Advanced programmers may want to know about the XMT execution model to optimize their code from a high level. However, the XMT execution model is primarily used by the compiler as a guidance of optimizing. In the execution model, a program could include: (i) Prefetch instructions to bring data from the lower memory hierarchy levels either into the shared caches or into the prefetch buffers located at the TCUs; (ii) Value broadcastng: some values needed by all, or nearly all, TCUs are broadcast to all threads; (iii) Thread clustering: combining shorter virtual threads into a longer thread; and (iv) If the programming model allows nested parallelism, the compiler will use the mechanisms supported by the architecture to implement or emulate it.

A formula for estimating execution time based on these extensions is provided. The depth of an application in the XMT execution model includes the following three quantities: (i) *Computation Depth*, given by the number of operations that have to be performed sequentially, either by a thread or while in serial mode. (ii) *Length of Sequence of Round-Trips to Memory (or LSRTM)* which represents the number of cycles on the critical path spent by execution units waiting for data from memory. For example, a read request or prefix-sum instruction from a TCU usually causes a round-trip to memory (or RTM). Memory writes, in general, proceed without waiting for acknowledgments, thus not being counted as round-trips, but the end of a parallel section implies one RTM used to flush all the data still in the interconnection network to the memory. (iii) *Queuing delay (or QD)* which is caused by concurrent requests to the same memory location; the response time is proportional to the size of the queue.

We can now define the XMT "execution depth" and "execution time". XMT execution depth represents the time spent on the "critical path" (that is, the time assuming unlimited amount of hardware) and is the sum of the computation depth, LSRTM, and QD on the critical path. Assuming that a round-trip to memory takes $\mathcal{R}$ cycles:

$$Execution\ Depth = Computation\ Depth + LSRTM \times \mathcal{R} + QD \tag{2.1}$$

The equation does not count the overhead of queueing of virtual threads on the TCUs when the number of virtual threads exceeds the number of TCUs and also suppresses the overhead of starting new threads. For the full description of XMT execution model, see [47].

2.4.2 XMT Compiler

Another key component of the XMT framework, the compiler is responsible for taking advantage of features in XMT architecture. Most of the items listed below are ongoing research projects in the XMT group.

2.4.2.1 Optimization in Shortening LSRTM

Prefetching TCUs in the XMT processor have private prefetch buffers and the compiler can take advantage of them by inserting prefetch instructions. For read operations in a loop, a prefetch instruction for the next iteration can be inserted for each read operation. For other read operations, prefetch instructions are inserted as soon as the addresses are available.

Using Read-only Buffers The per-cluster read-only buffers provides two functions: (i) Combine read requests to the same location from TCUs in the same cluster and reduce load in the interconnection network and shared cache modules; (ii) Provide temporal locality for read-only memory accesses. During compiler optimization, normal read instructions are replaced with XMT-specific read instructions that make use of the read-only buffers, if the value can be guaranteed not to change during the current parallel mode. It is also possible that programmers can help the compiler to find these memory locations, by using a keyword, *constant*, in variable definitions.

Value Broadcasting The current compiler is able to identify the variables that need to be broadcast and replace the read instructions with broadcast instructions.

Release Consistency There are two types of store instructions in the XMT ISA: blocking and non-blocking. Blocking store instructions block the TCU from advancing to the next instruction until it gets an acknowledgement from the cache module. The blocking store makes sure all other TCUs will see the new value before the next instruction is executed. On the other hand, the non-blocking store will advance to the next instruction as soon as the request is accepted by the interconnection network. The XMT ISA includes a fence instruction, *sflush* (store flush), that blocks the TCU until all previous non-blocking stores have been committed to shared caches. The blocking store instruction is functionally equivalent to a non-blocking store instruction followed by a *sflush* instruction.

The compiler is responsible for selecting the proper store instructions based on the synchronization requirement of programs. One possible optimization is to use non-blocking store instructions and insert *sflush* instructions when necessary.

2.4.2.2 Nested Parallel Sections

Some PRAM algorithms can be expressed with greater clarity and conciseness if nested parallelism is supported. The XMT architecture has the capability of increasing the number of threads within parallel mode with *sspawn* and this can be used for implementing nested parallelism in the compiler.

The *sspawn* can generate one new thread at a time, but multiple threads can execute *sspawn* simultaneously and generate multiple new threads. When a parallel thread needs to generate multiple threads, a binary tree mode can be used to expedite the process. Assume that a thread i needs to generate n new threads, then it will execute a *sspawn* instruction which generates a child thread j. After this, $n-1$ needs to be generated and this can be shared by the parent thread (i) and child thread (j). Now, thread i and j will both need to generate $\frac{n-1}{2}$. This process continues until all n new threads are generated. During this process, proper initialization is needed. See section 2.2.3.

It is also possible to introduce a *kspawn* instruction that generates k threads at a time instead of one from *sspawn*.

2.4.2.3 Clustering Threads

The XMT programming model allows programmers to declare any number of threads in parallel mode, however, as explained earlier, this will result in serialization in a limited number of TCUs in hardware. In case the number of threads, N, is much larger than the number of TCUs, p, it is possible to group a few original threads to a new longer thread and reduce the number of total threads. We call this thread clustering. Thread clustering has a few advantages: (i) By having longer threads, the compiler can pipeline memory accesses and overlap latencies. This can reduce the penalty of round trips to memory (RTMs) and queueing delays (QDs) from serialization in short threads. (ii) Although the XMT is very efficient in thread handling, there is some overhead for starting a new thread. Less threads will result in reduced overhead in execution.

Clustering without Nested Spawns Suppose the program has N threads and $N \gg p$, where p is the number of TCUs available in the XMT processor. A simple way of clustering threads is grouping them into $\frac{N}{c}$ threads, where c is a function of p. In trivial cases, $c = p$.

Clustering for single-spawn and k-spawn When the number of threads is changing in parallel mode, dynamic scheduling can be introduced for clustering. The number of threads in the queue waiting for TCUs can be obtained at run time and this information can be used to determine whether a new thread will be generated or not. If the queue of waiting threads is too long, a thread can simply execute the child thread, which would be generated after it finishes the current thread. In the opposite case, when there are idling TCUs, the current thread can generate a child thread and let it share the load.

Chapter 3

XMT FPGA Prototype and Performance Evaluation

A 4-cluster 64-TCU XMT prototype system was built with 3 Xilinx FPGA chips. With the prototype system, we can evaluate the architecture by running heavier programs with large inputs. The prototype system is about 11-12K times faster than the XMT architecture simulator. In this chapter, the details of the prototype system are presented and then 8 kernels, which are used as benchmarks, are described. At the end, the performance results are presented and analyzed.

3.1 Specifications of the XMT FPGA Prototype

The XMT FPGA prototype system is built on a FPGA development board purchased from a third party and it uses 2 Xilinx Virtex-4 LX200 and 1 Xilinx Virtex-4 FX100. The board can be plugged into a PCI slot of a computer and communicates with the host computer through the PCI interface.

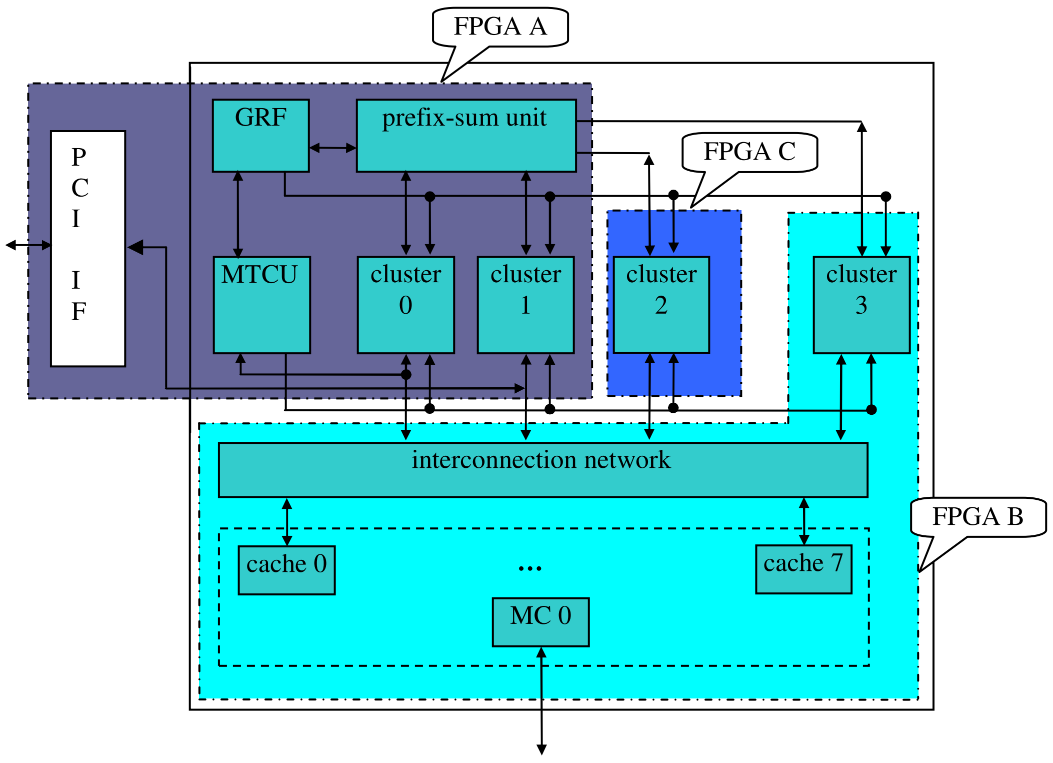
3.1.1 Partitioning the XMT Processor

Unlike an Application-Specific Integrated Circuit (ASIC), a Field Programmable Gate Array(FPGA) provides programmable logic and interconnects, which is very important for prototyping and logic verification, since potential mistakes can be fixed easily. The other advantage of FPGA prototyping is the low cost compared to an ASIC counterpart. However, the disadvantage of the FPGA implementation is slow clock speed and low logic density. The research results show that FPGA implementation is 3 to 4 times slower and takes 21 to 40 times the area of an ASIC implementation [31]. Because of the low logic density, 3 FPGAs are needed to implement a 64-TCU XMT processor, but it is reasonable that a much bigger XMT processor can be fit into a single chip if an ASIC chip is designed.

Figure 3.1 shows the partitioning of the 64-TCU XMT prototype. One should note that FPGA A and B (V4LX200) have twice the logic cells of FPGA C (V4FX100). FPGA A includes a master TCU, prefix sum unit, two clusters (32 TCUs), global register file and PCI logic blocks. FPGA B implements the interconnection network, eight cache modules, memory controller and one cluster (16 TCUs). The interconnection network provides all-to-all, full duplex communication between 4 clusters and 8 cache modules. The master TCU shares an interconnection network port with cluster 0 and the PCI interface shares a port with cluster 1. Each cache module is 32KB and all of them communicate with off-chip SDRAM through the only DDR2 memory controller. FPGA C implements only one cluster (16 TCUs).

3.1.2 Specification of the XMT FPGA Prototype

The master TCU has 8KB of local data cache, which applies a direct map, write-through policy. The hit access time is one cycle, meaning that a stream of read operations can be serviced at a rate of one read per cycle, when they result in cache hits. If a read operation turns out to be a cache miss in the MTCU local cache



FPGA A,B Virtex-4 LX200
FPGA C Virtex-4 FX100

Figure 3.1: Partitioning of the XMT processor for 3 FPGAs

(mcache), but a cache hit in the shared cache, then the access time is 25 cycles. The decomposition of the 25 cycles are shown on figure 3.2. The delay on crossing FPGA boundaries is different for each direction. This is due to the different behavior of each recipient. When read requests are ready to be sent to interconnection network from a cluster, it is possible that the buffer in the interconnection network is full and cannot accept new packets. In this case, the cluster should hold the request until the interconnection network is ready to receive packets. So, a simple handshaking is necessary. However, when the response comes from cache module through interconnection network, it is guaranteed to be accepted by the MTCU and clusters because

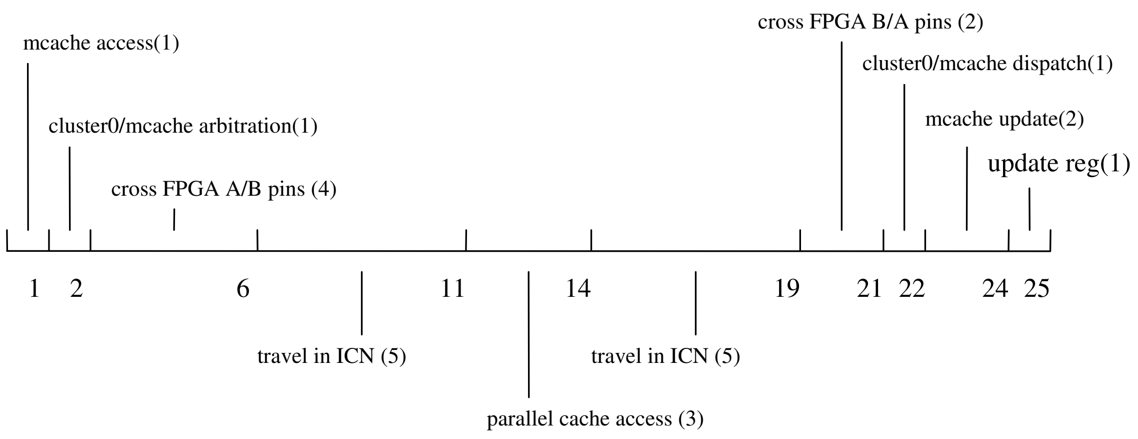


Figure 3.2: Decomposition of MTCU cache miss penalty

they are designed to process the response immediately all the time. When a response arrives at the mcache, the address of the operation needs to be retrieved using the response ID, then the value is updated to the mcache. This explains why it takes two cycles (cycle 23 and 24). The number of cycles for traveling the interconnection network is $\log_2(\#Cluster \times \#Cache) = \log_2(4 \times 8) = 5$.

There are 8 parallel cache modules in the FPGA prototype computer, which is twice of the number of clusters in the prototype. With the same number of clusters and cache modules, the cache system may not be able to match the throughput of the clusters, when cache misses are considered. By having two cache modules per cluster, the overall throughput of the cache modules are more likely to match that of clusters. As noted in above figure, the cache hit access time is 3 cycles: tag access, data access and result buffer update. The shared cache modules are designed with throughput as the first priority. These cache modules are non-blocking, meaning each of them can hold up to 64 pending cache misses accessing eight different cache lines.

When a shared cache access turns out to be a miss, the data will be fetched from SDRAM. The latency of accessing SDRAM is not deterministic, and depends on the status of the SDRAM. The eight cache modules share one SDRAM channel, so the latency will increase when multiple requests are sent to the memory controller from different cache modules simultaneously. It is always desirable to have a low latency, high throughput memory controller, but it is quite difficult to achieve both. Like the shared cache, the memory controller is designed with overall throughput as its first priority.

SDRAM chips have multiple banks that can operate in parallel to increase the data transfer rate. Each cache module is mapped to a SDRAM bank. Recall that each cache module can send requests for up to eight different cache lines, so the memory controller has 8 buffers for these different requests. SDRAM access latency and throughput is highly dependent on the order of the requests[40]. The memory controller in XMT FPGA prototype tries to reorder the requests from different banks as well as within a bank. Figure 3.3 shows the miss penalty decomposition when there is no contention. The miss penalty is 30 cycles. During cycles 5-7 the memory controller will choose among the eight possible requests, selecting the one that is best for throughput. Eight SDRAM banks share one command channel, so any SDRAM command needs to pass a 3-cycle arbitration tree (cycle 8-10, 12-14). It will take one more cycle to confirm that the ACT command has been sent (cycle 11). Then two cycles, 15 and 16, are used for SDRAM column access delay. The SDRAM data transfer rate is twice the SDRAM clock rate and quadruple that of the XMT FPGA prototype, so a burst length of 4 becomes 1 cycle in the XMT clock domain. Synchronization and buffering take another 3 cycles (18-20). After the data is transferred back to the shared cache, it will get the address from the ID (cycle 21) and then update the cache in 8 consecutive cycles(22-29). Finally, the results will be sent back to the cluster in cycle 30.

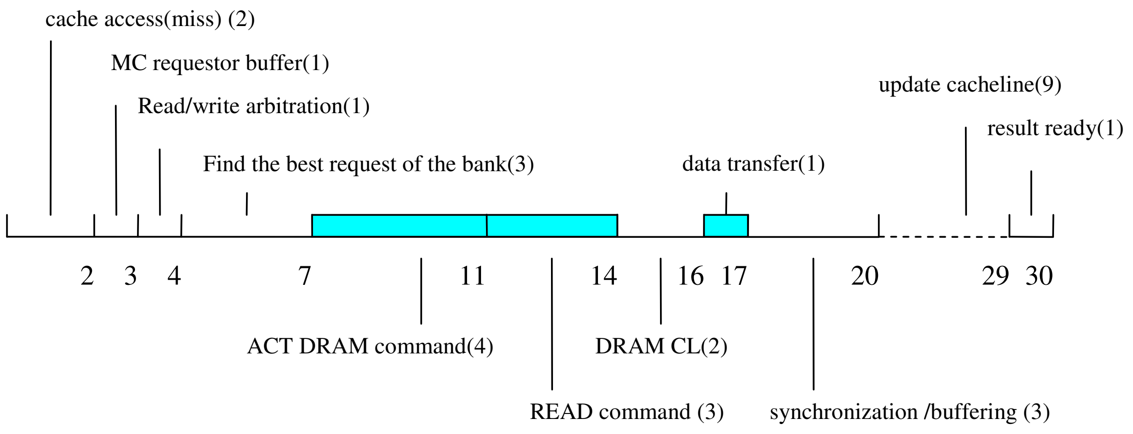
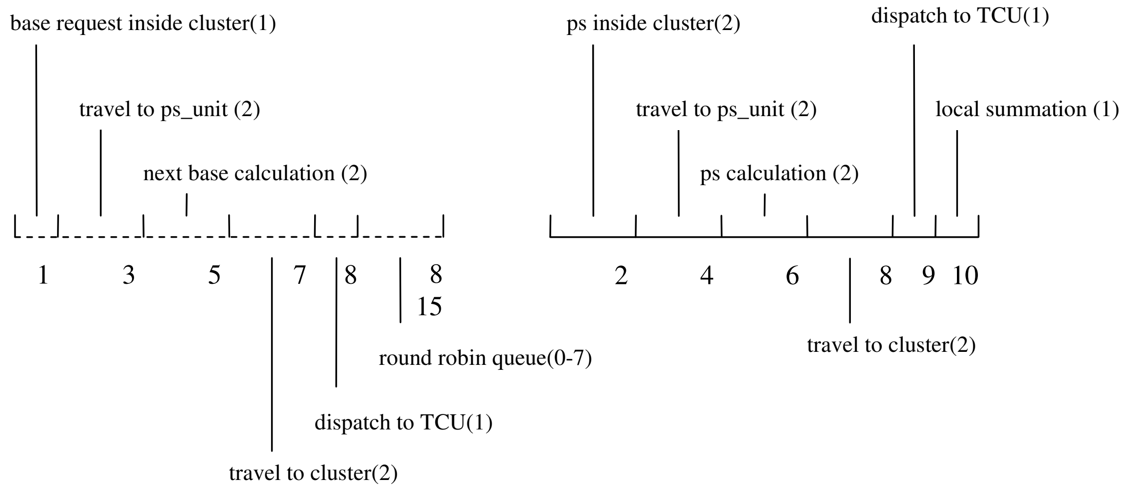


Figure 3.3: Decomposition of shared cache miss penalty

As shown in figure 3.3, multiple requests from the same bank can be overlapped. SDRAM command cycles (8-14) cannot be overlapped with other SDRAM command cycles from the same bank, which means that a bank can serve one request per seven cycles and use one cycle (17) of data bus. Considering there are 8 cache modules (or banks), it is possible to utilize the data bus to its maximum. However, SDRAM refresh and read/write change also reduce the data bus utilization. It should be noted that the 30 cycles shown in figure 3.3 is a relatively optimistic estimate, since it does not include any delays caused by contention from other banks. The ACT SDRAM command cycles can be waived if the SDRAM status permits and then the miss penalty can be 26 as the best case. For the worst case, the cache miss service time can be hundreds of cycles.

The number of cycles needed for a prefix-sum operation varies between 10 and 25, depending on the number of global registers used in the program. Recall that, the prefix sum unit is used for ps operation for the 8 global registers, but at any given cycle it can only be used for one particular global register, the *active base register*. On the other hand, a TCU can send a ps request for any of the 8 global registers, a *pending base register*. When a TCU executes a ps instruction, it waits until the active base register matches the pending base register. The prefix-sum unit monitors pending base registers from all TCUs and switches the active base register among those pending base registers. If only one global register is used in the program and the active base register of the prefix-sum unit matches that global register, a ps operation will take only 10 cycles (figure 3.4 (b)). If the active base register does not match the pending base register from a TCU, the TCU has to wait until the prefix sum unit switches the active base register to the pending base register and this procedure takes up to 15 cycles (figure 3.4 (a)). The average number of clock cycles for ps operations increases with the number of global registers used simultaneously by a program.

Most ALU instructions and data movement instructions take only one cycle. The branch instruction takes only one cycle for a branch not taken and four cycles for a branch taken, which needs to flush the pipeline. The shift instruction takes two cycles, this is only for the FPGA implementation and can be reduced to one cycle in an ASIC implementation. More detailed specification of the XMT FPGA



(a)extra cycles, when base register NOT match (b) base register matches

Figure 3.4: Prefix-sum operation

prototype system is listed in table 3.1.

3.1.3 Envisioned XMT Processor

The XMT FPGA prototype is a scaled-down version of an envisioned XMT processor [48, 36, 51], which is shown in figure 3.5. We aspire to have in the not-too-far future an XMT processor that has 1024 TCUs grouped into 64 clusters and 64 (or 128) on-chip memory modules. Each memory module consists of two levels of caches and memory access ports are shared by multiple L2 cache modules. The MTCU has local instruction and data caches for better backwards compatibility with serial programs.

Table 3.1: Specifications of the XMT FPGA prototype

Clock rate	75MHz
Number of TCU clusters	4
Number of TCU per cluster	16
Memory size	1GB DDR2
Number of shared cache modules	8
Size of each shared cache module	32KB
SDRAM. data rate	2.4GB/s
MTCU local cache	8KB
MTCU memory access local hit	1 cycle
MTCU memory access local miss, shared cache hit	25 cycle
TCU shared cache access hit	30 cycles
Shared cache miss penalty	26∼hundreds of cycles
TCU ps operation	10∼25 cycles
MTCU,TCU ALU operation	1 cycles
MTCU,TCU SHIFT operation	2 cycles
MTCU,TCU BRANCH penalty	4 cycles
MTCU multiplication	6 cycles
MTCU division	36 cycles
TCU multiplication, division sharing overhead	4 cycles
Number of multiplication/division units per cluster	1
Number of ALU,BRANCH,SHIFT units per cluster	16

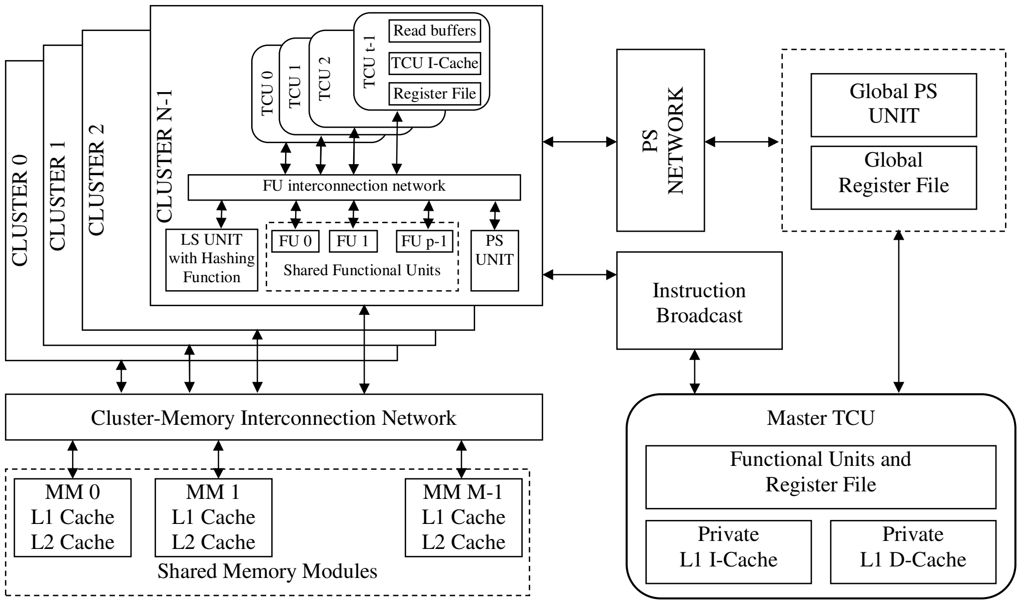


Figure 3.5: An envisioned XMT processor

3.2 Benchmarks

As a prototype, the XMT FPGA computer is not a full system, and the number of applications we can test on it is quite limited. First, it does not support floating point operations and as a result, many real applications and benchmarks, such as SPLASH, are out of consideration. The other limitation of the prototype is that the parallel section cannot have any function calls, because TCUs do not have instruction caches and the size of the instruction buffer is quite limited. And most importantly, the prototype system is not ready for an OS. These limitations are only for this version of prototype and will be eliminated in the future, when we can test the system with a broader set of applications. With the above limitations, the following eight kernel benchmarks are chosen to test the performance of the XMT FPGA prototype. Fortunately, with these integer programs, we are able to test the performance of the memory system, which is the most interesting part of the XMT architecture. The eight benchmarks consist of a variety of memory access patterns including regular (mmul, conv and add), irregular (BFS, DAG and BST) and a combination of these two (qsort and compaction).

3.2.1 Matrix Multiplication (mmul)

Problem description: Given two integer matrices X and Y, calculate matrix Z, the product of X and Y.

Algorithm: Assign a thread for each row of Z.

3.2.2 Quick Sort (qsort)

Problem description: Given an unsorted integer array X, sort the array X in an ascending order.

Algorithm: The parallel quicksort[25] used in the experiment has two phases : (1) the input array is partitioned using dual storage. Namely, rather than copying values in place, copying is done into a separate array to facilitate greater parallelism. partitioning continues until the number of blocks exceeds the number of TCUs in the system; (2) each block is sorted by a thread, which is dynamically assigned to a TCU.

3.2.3 Breadth-First Search (BFS)

Breadth-first search in parallel: Given a connected, undirected graph G(V,E) and a vertex s∈V, the breadth-first search (BFS) method visits vertices in the following order: First, visit s, then visit (in some order) all the vertices w∈V, where the edge (s,w)∈E; denote the set of these vertices by V_1, and the singleton set consisting of s by V_0; in general, V_i is the subset of vertices of V which are adjacent to a vertex in V_{i-1} and have not been visited before (i.e., they are not in any of the sets $V_0, V_1, ..., V_{i-1}$). Each set V_i is called a layer of G and i is the level of any vertex v, v∈ V_i.

The following arrays are given as the input of the BFS problem. Let m be the number of edges and n be the number of vertices.

- edges[2m][2]: the start and end vertex for each edge, in the ascending order of starting vertex ID. Each edge is recorded twice as two directions.
- vertices[n]: the index in the edges array where the edges for each vertex begins.
- degrees[n]: the degree of each vertex.

The parallel BFS algorithm solves the problem layer by layer. By tracing all edges from the starting vertex, the V_1 set can be found. Now assuming that in layer k set V_k is known, we can find set V_{k+1} in parallel. For each vertex in V_k, a thread is assigned and each thread can generate more threads if the vertex has many edges to check. The thread checks if the end point is not visited, in which case the end vertex will be added to set V_{k+1}. Note that the prefix-sum operation is used in two places: (1) determine which thread discovered a vertex in V_{k+1}. (2) make sure multiple threads can add vertices to set V_{k+1} simultaneously.

3.2.4 Finding Longest Path in DAG (DAG)

Given a directed acyclic graph (DAG), for each vertex in the graph, find the longest path from a source to the vertex. The following arrays are given as the input of the DAG problem. Let m be the number of edges and n be the number of vertices.

- inDegree[n]: the number of edges ending with each vertex.
- outDegree[n]: the number of edges starting with each vertex
- outEdges[m][2]: the start and end vertices for each edge, in ascending order of starting vertex ID.
- outVertices[n]: the index in the outEdges array where the edges for each vertex begins.

The parallel algorithm works as described below. Each vertex has a variable, current longest path, keeping the longest path among the visited incoming edges, which is initialized to 0.

1. Scan the inDegree array and collect all vertices whose inDegree is 0 into V_{done}, and the longest path for these vertices are 0.
2. Spawn thread for each vertex in V_{done} and each thread will handle all outgoing edges from the vertex.

3. If an outDegree of a vertex is high, then it may spawn a new thread and share the load among parent and child threads.

4. Each thread calculates the path length to the end vertex with this particular edge and updates current longest path with the maximum of current and new one.

5. When a thread found it has visted the last incoming edges of a vertex, the longest path to the vertex is known and it generates a new thread for that vertex.

6. When all threads finish, the longest path is calculated for all vertices.

3.2.5 Array Summation (add)

Problem description: Given an integer array X, calculate the sum of all its elements.

Algorithm: Using coarse-grained parallelism, divide the input array into m sub arrays, where m is the number of total TCUs in the XMT processor (that is 64 for the prototype). Each TCU calculates the sum of the sub arrays and we then serially add them to get the sum of the input array.

3.2.6 Array Compaction (comp)

Problem description: Given an array A = A(1), . . . ,A(n) of (any kind of) elements and another array B = B(1), . . . ,B(n) of bits (each valued zero or one). The compaction problem is to find a one-to-one mapping from the subset of elements of A(i), for which B(i) = 1, $1 \leq i \leq n$, to the sequence (1, 2, . . . , s), where s is the (a priori unknown) number of ones in B. In our testing, we generated a new array C, which consists of A(i), for which B(i)=1. The order of the elements in array A is not preserved.

Algorithm: This is an extremely fine-grained algorithm. A thread handles one element in A(i) by copying A(i) to array C if B(i) =1. The prefix sum operation is used for finding a destination index in array C, to let multiple threads copy into C simultaneously.

3.2.7 Convolution (conv)

Problem description: Given a two dimensional image array of X ($n \times n$), and another filter array of F($m \times m$), this problem uses a dot product to generate a filtered array Y. For element X(i,j), where $1 \leq i, j \leq n-m+1$, the filtering operations add the dot product of filter array F and sub array of X, X(i:i+m-1,j:j+m-1) to X(i,j).

Algorithm: The algorithm is straightforward. Spawn a thread for each row in X(i:i+m-1,j:j+m-1) to X(i,j), where $1 \leq i, j \leq n - m + 1$. The work complexity of the algorithm is $O(n^2 \cdot m^2)$.

3.2.8 Binary Tree Search (BST)

Problem description: Given a balanced binary search tree (BST) and a set of keys, search the keys in the BST. The BST has the following properties.

- The left subtree of a node contains only values less than the node's value.
- The right subtree of a node contains only values larger than the node's value.

Algorithm: Spawn a thread for each key and search the BST in parallel. Each thread starts from the root; if the key is equal to the root, the key is found, otherwise the search continues based on the comparison result. If the key is less than the root, the search continues on the left subtree of the root, otherwise the right subtree is searched. The search continues until the value is found in the tree or a leaf of the tree is reached.

3.3 Performance Analysis

The eight benchmark kernels were executed on the XMT FPGA computer and cycle counts, as well as other performance data, were collected. In this section, we report our results and examine different aspects of the system.

3.3.1 Speedup relative to Serial Execution on MTCU and AMD Opteron

For each of the eight benchmarks described in section 3.2, two different input sizes are tested. The program sizes are listed in figure 3.2. (L) represents large data size and (S) represents small data size. In the table, The last two columns are the memory space used by programs, which is collected from the compiler output.

Both serial and parallel versions of the eight benchmarks are executed on the FPGA system. The speedup of parallel versus serial is shown in figure 3.6. The

Table 3.2: Input size of the benchmarks

Application	input size	memory usage	
		parallel	serial
mmul (L)	2000x2000	48MB	48MB
mmul (S)	128x128	192KB	192KB
qsort (L)	20 million	360MB	200MB
qsort (S)	100 thousand	1.8MB	1MB
BFS (L)	V=1M, E=10M	220MB	100MB
BFS (S)	V=100K, E=1M	21.6MB	9.6MB
DAG (L)	V=1M, E=17M	368MB	160MB
DAG (S)	V=50K, E=600K	13.4MB	6.0MB
add (L)	50 million	200MB	200MB
add (S)	3 million	12MB	12MB
comp (L)	20 million	208MB	208MB
comp (S)	2 million	20.8MB	20.8MB
BST (L)	16.8M nodes, 512K keys	205MB	205MB
BST (S)	2.1M nodes, 16K key	25.3MB	25.3MB
conv (L)	image:1000x1000, filter:32x32	8MB	8MB
conv (S)	image:200x200, filter:16x16	320KB	320KB

upper bound of the speedup is 64, since a maximum of 64 threads are active for the parallel program.

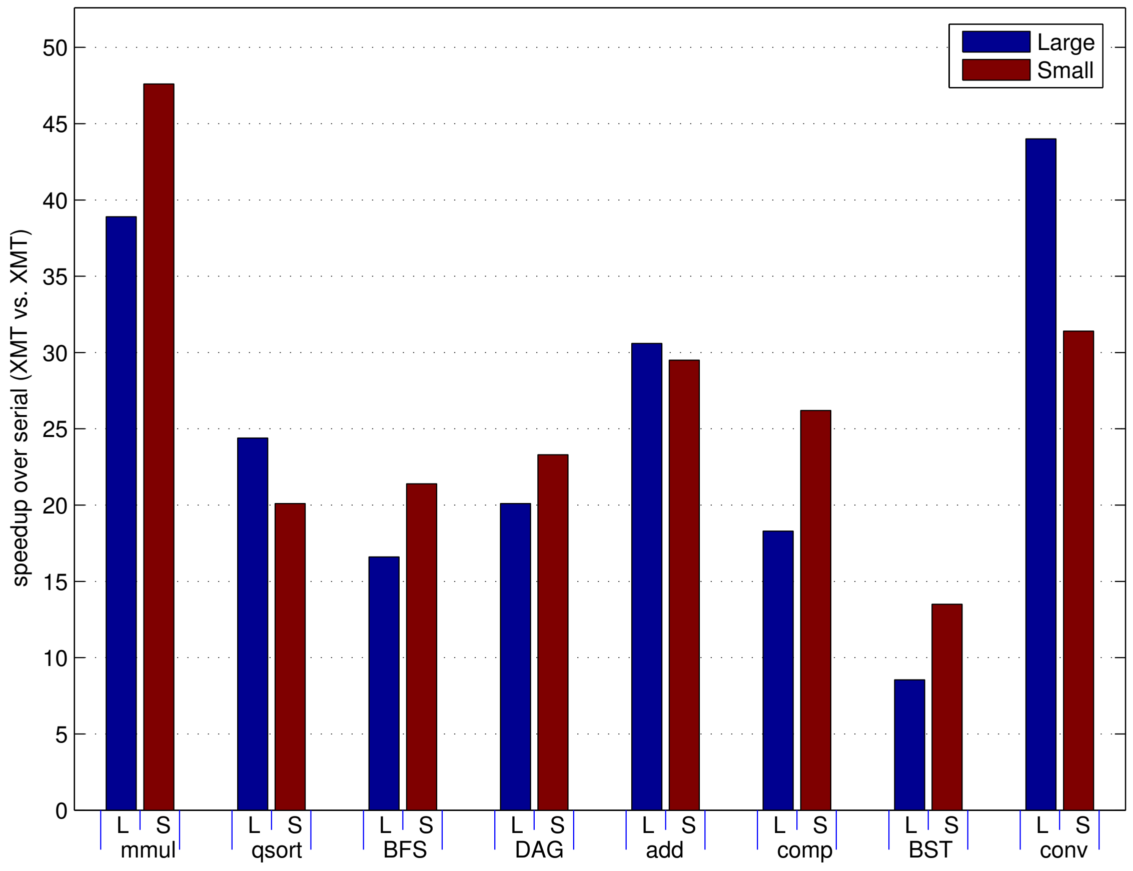


Figure 3.6: Speedups of the benchmarks (parallel Vs. serial in XMT)

Overall, the speedups shown in figure 3.6 are quite good. It should be noted that the MTCU is a 4-stage, single-issue, in-order execution processor, which is similar to one of the TCUs in the cluster. In this sense the speedup may be more appropriately be described as the efficiency of parallelism. The primary difference between the MTCU and TCU is in memory access. The MTCU has an 8KB local cache, and the cache hit latency is one (no bubble in the pipeline). Although TCUs have four prefetch buffers and can use a cluster-wide read-only buffer, they often need to make a round trip to the shared cache to access memory.

Two computational benchmarks, mmul and conv, achieved the maximum speedup. This is because these programs are highly CPU-bound and both have very regular memory access patterns, which make it easy to take advantage of software prefetch commands in XMT. Considering that there are only four multiplier/dividers in the chip, the speedups are very impressive.

On the other hand, the qsort, add and comp benchmarks, which are memory-bound programs, showed moderate speedup of 18.3∼30.6. Note that, while the FPGA computer has 64 TCUs, they are grouped into 4 clusters, and all 16 TCUs inside a cluster share one load/store port of the cluster. On the cache side, there are 8 shared cache modules, so on average, 8 TCUs share one cache module. More importantly, the FPGA computer has only one off-chip SDRAM channel shared by all 8 on-chip cache modules. For the memory bound applications, these shared resources can be saturated and prevent us from getting higher speedup numbers.

The two graph related applications, BFS and DAG, are fine-grained and their memory access pattern is irregular. XMT FPGA has speedups of 16.4∼23.3 for these two benchmarks. Note that these two programs are known to be very difficult to parallelize in the traditional coarse-grained parallel computers. Irregular memory access patterns reult in low cache hit rates in the shared cache and low speedups due to many active threads in parallel execution relative to its small on-chip shared cache in the prototype. The cache hit rate is discussed in the next section.

The BST program showed a relatively low speedup of 8.55 in the large input set, and 13.5 in the small input set. Recall that the BST is a balanced binary tree and a search proceeds from the root of the tree and advances towards leaves until the key is found or a leaf is reached. The upper part of the tree, especially the root, is accessed by all threads repetitively, which adds queuing delay to the TCUs. The read-only buffer has the ability to combine the memory read requests from a cluster, but it is still not enough. The small size of the read-only buffer (8KB) and its direct mapped scheme limited its advantage. Unlike the local cache in MTCU, which is used only by one thread, the read-only buffer is shared by all 16 TCUs in a cluster, meaning 512 bytes per thread, which is extremely small to be able to take advantage of temporal locality. The increased speedup in the small input set confirms the explanation above.

3.3.2 What is Happening inside the XMT Processor?

In the previous section, the cycle counts and speedup of the parallel versus serial versions are reported. In this section, we will investigate the performance of the XMT processor with a close look at some of the components of XMT.

3.3.2.1 Bandwidth Utilization in Memory Access

XMT is a shared memory architecture and all TCUs access the shared memory space through the interconnection network. It is interesting to understand how the interconnection network is used in each of the benchmarks. The idea of grouping 16 TCUs into a cluster and assigning one interconnection network port to a cluster is to allow efficient utilization of the interconnection network. Note that the cost of the interconnection network, both in terms of area and delay, increases significantly with an increase in the number of ports. For all benchmarks, we calculated the total number of packets delivered by the interconnection network as well as the number of requests processed by the load/store (LS) unit inside clusters. After introducing the read-only buffer in the clusters, some requests from TCUs can be processed locally without sending packets to the interconnection network. If a request cannot be served by the read-only buffer, the LS unit will decompose the request into packet(s). The number of packets that will be sent and received are listed in table 3.3.

Table 3.3: Number of packets in requests

Request	# of packets sent	# of packets received
Read	1	1
Write/PSM	2	1
SDRAM prefetch	1	0

The bandwidth utilization of the interconnection network and normalized throughput of the LS units are shown in figure 3.7. The number of total requests processed by the LS units is divided by the total number of the parallel cycles and the number of LS units in the XMT processor, which is 4 in this prototype. The total number of packets transferred from clusters to the cache modules are divided by the number of cycles in parallel mode and the number of clusters, 4.

Four benchmarks: mmul, qsort, add and comp showed a high utilization rate

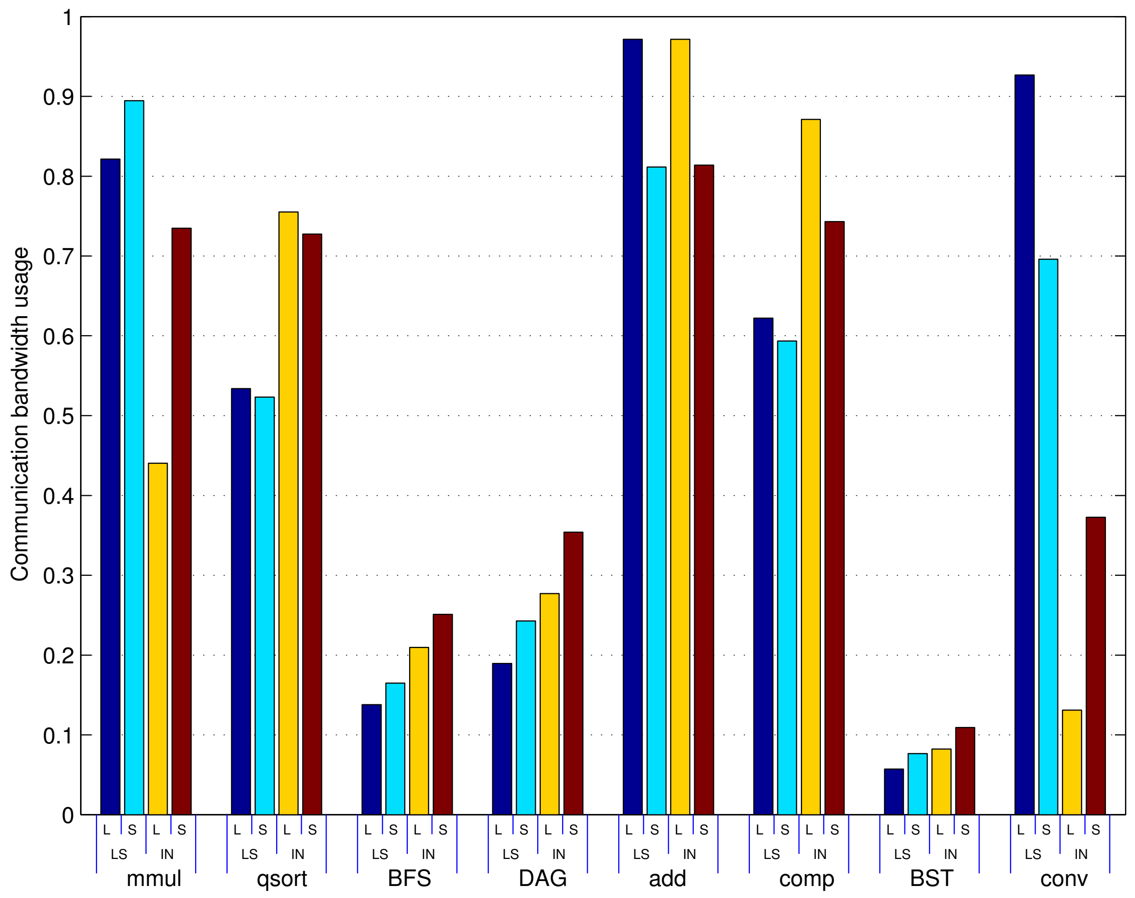


Figure 3.7: Normalized throughput of LS unit and interconnection network. 'LS' at the bottom of the bar indicates LS unit and 'IN' at the bottom of the bar indicates interconnection network.

of the LS unit or interconnection network. For mmul and conv, where the read-only buffer is extensively used, the LS unit is almost fully loaded, but the network has much less traffic. For other benchmarks, where the read-only buffer is not used extensively, the interconnection network is more crowded than the LS units, because the LS unit uses one cycle to process a write operation that will then require two packets(address and data), and, therefore two cycles in the interconnection network.

The 3 benchmarks BFS, DAG and BST used a very small portion of the bandwidth available. This is because of the long latency in cache accesses due to the low hit rate and extensively long queue in the cache module (BST).

3.3.2.2 Breakdown of Execution Cycles

It is good to know what portion of the overall XMT cycles are spent on each of the categories, such as ALU operation(alu), multiplication/division (md), memory access (mem), register based prefix-sum operation (ps), memory based prefix-sum operation (psm), idle time and bubble cycles. The idle time is the number of cycles a TCU spent in the sleep state shown in the figure 2.5. The bubble cycles are mainly for the branch misprediction, but also include cycles wasted because of an instruction buffer miss. Figure 3.8 shows the breakdown of the execution time of all benchmarks. For each benchmark, serial versions are shown on the left and parallel versions are shown on the right. The input size for benchmarks is the large one, as listed in table 3.2.

In general, memory access takes a significant portion of the overall execution time in both serial and parallel execution. This is due to the small amount of on-chip cache and large input data size. The cache hit rate is listed in table 3.4. Cache hit rate is calculated for read operations only. In serial execution only *lw* (load word) instructions are counted and the prefetch instructions are not counted. For the parallel execution, since the prefetch instructions replace some read instructions from the view of the shared cache modules, they are counted as well, but SDRAM

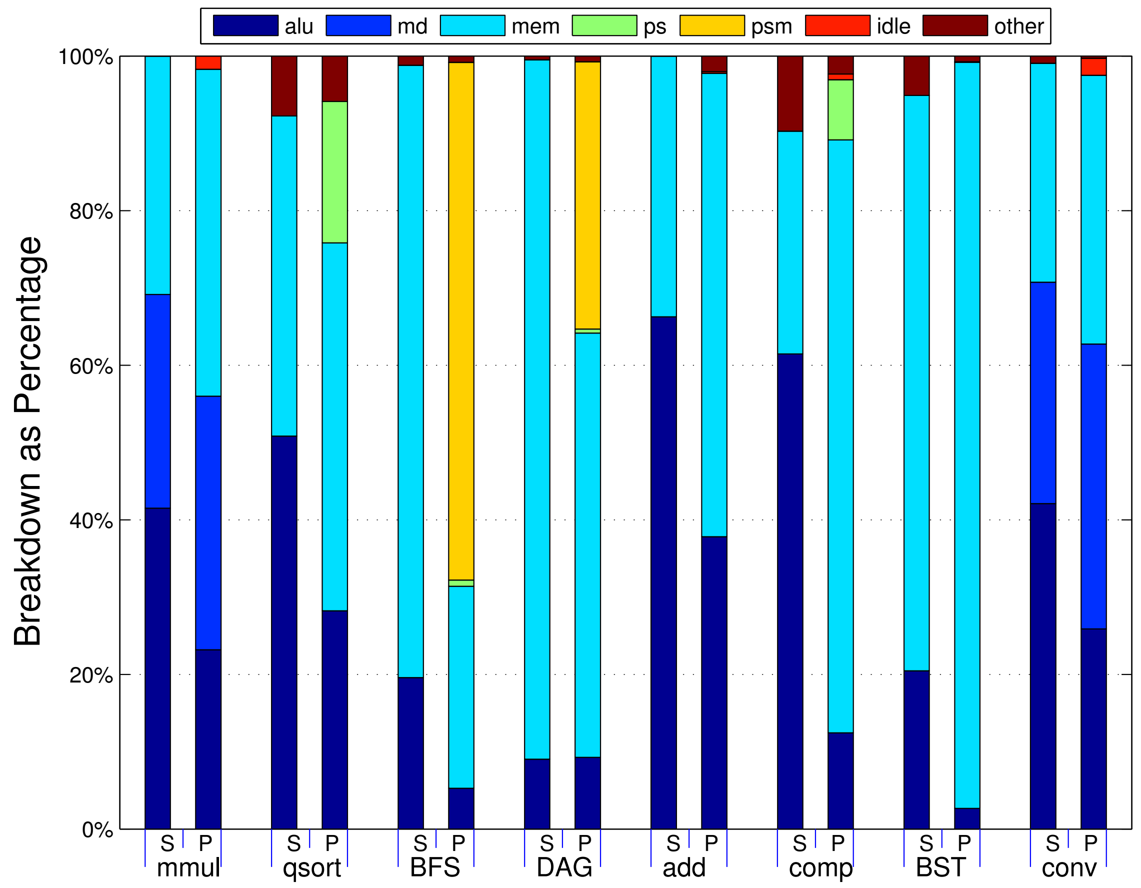


Figure 3.8: Breakdown of serial and parallel execution time in XMT FPGA prototype. For each benchmark, left bar is for serial execution and right bar is parallel execution.

prefetch is not counted. From the table we can find that the cache hit rate is low for BFS, DAG, qsort and BST. Figure 3.8 also shows that these benchmarks spend a large portion of execution time on memory accesses.

Since the interconnection network is an expensive component in the XMT processor in terms of hardware implementation cost, XMT tries to increase the utilization of the interconnection network by letting multiple TCUs share one interconnection port. As a result the individual TCUs may have longer cache access latency and

Table 3.4: Cache hit rate

execution	mmul	qsort	BFS	DAG	add	comp	BST	conv
serial	0.688	0.953	0.681	0.587	0.874	0.993	0.794	0.918
parallel	0.75	0.57	0.31	0.50	0.88	0.78	0.43	0.99

spend more time on memory access. In figure 3.8, the parallel programs spent a larger percentage of the time on memory access than their serial counterpart.

3.3.2.3 Average Latency for Read Operations

Like simultaneous multi-threaded (SMT) architectures, TCUs in the same cluster share some functional units. 16 TCUs in a cluster share one LS unit and one interconnection network port. The cache modules are also shared by all TCUs. The cache module and the LS unit are only capable of processing one requests per cycle, so when multiple requests arrive, queuing is necessary and results in a longer delay. As a result, a read operation may take extra cycles in addition to the number of cycles needed to travel the interconnection network. Prefetching may shorten the delay by sending a read request earlier than the value is actually needed by a TCU. The read-only buffer stores values that are safe to be cached and shared by TCUs, and which are identified by a special read command. Requests from other TCUs for the same memory location may be served locally by the read-only buffer, instead of fetching values from a shared cache through the interconnection network. This helps not only to reduce the traffic load in the interconnection network, but also to reduce the access latency.

Figure 3.9 shows the average latency of read operations in serial and parallel

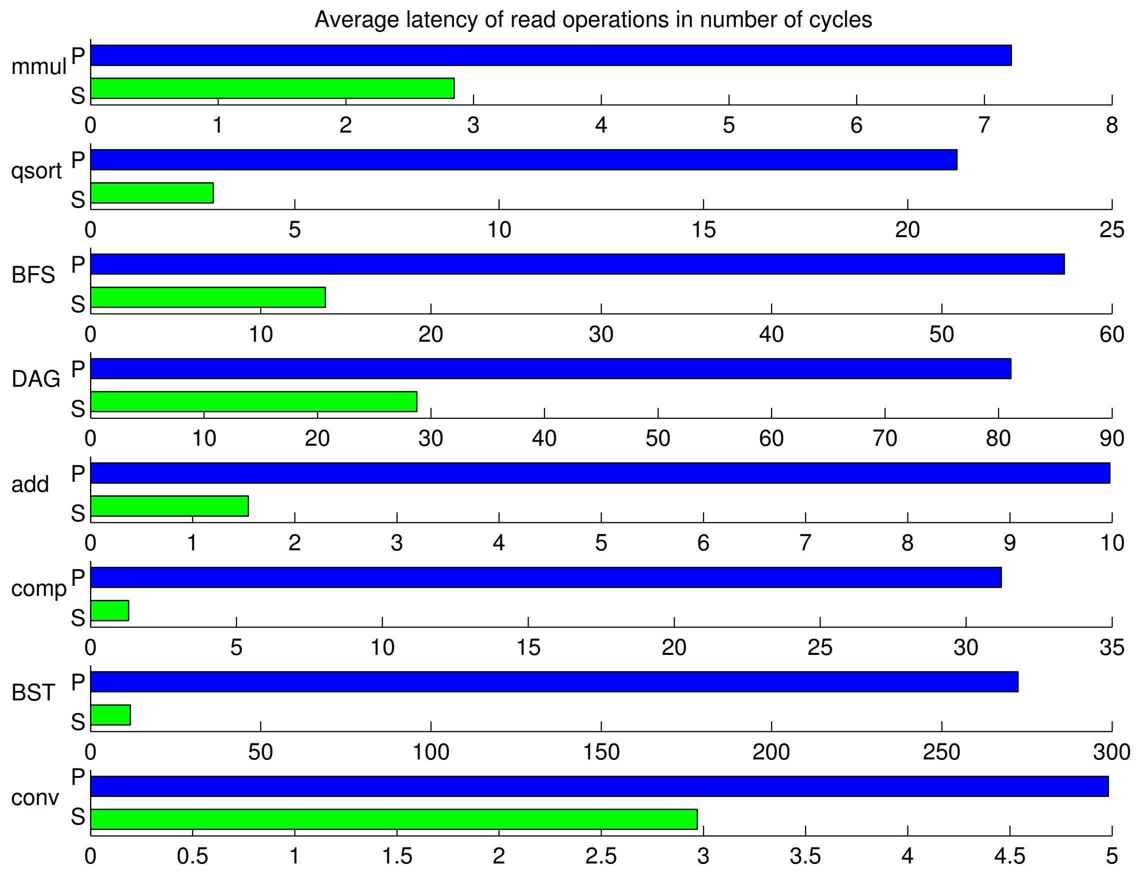


Figure 3.9: Average latency of read operations in the serial and parallel program in number of cycles

executions. In general, the read operation delay in parallel mode was larger than in serial mode. This is not surprising, because of the long latency of the round trip to memory and resource sharing among TCUs. For the benchmarks that exhibit regular access patterns, the read operation delay is significantly lower than the others. BST had the worst performance in terms of read latency, more than 10 times longer than serial execution. The low cache hit rate (see table 3.4) is one of the reasons, but more importantly, the data structure of the problem resulted in a long queue for accessing the top of the tree, particularly the root. The small size (8KB) of the read-only buffer and direct mapping caused the frequent replacement, which limited the benefit of the read-only buffer.

TCUs can send prefix-sum requests that are using different global registers. In such a case, the ps_unit chooses the base register in a round robin fashion from those that are requested. As a result, the latency of ps operation varies from 10 to 25 cycles.

The prefix-sum to memory (psm) operation does not have the limitation of the register-based ps operation. (1) The increment amount is not limited to 0 and 1; it can be any integer. (2) There is no limit to the number of bases for psm operations that can be used in the program simultaneously, while the number of global registers is limited. These advantages come at the cost of performance. The prefix-sum to memory operation is the most expensive memory access operation in terms of latency, because of the round trip to memory delay and, furthermore, the long queuing delay. Queuing of multiple prefix-sum operations to the same memory location is very likely to happen, because prefix-sum to memory operations are used to provide synchronization among multiple threads.

From table 3.5, it can be observed that the average latency of the global register-based ps operations increases as the number of global registers used in a program increases. Only two programs, BFS and DAG, used psm operations. As we expected, the delay of the psm operation is quite long, which suggests that the programmer should avoid using the psm operation if possible.

Table 3.5: Average latency of ps and psm operations

item	mmul	qsort	BFS	DAG	add	comp	BST	conv
psm lat.	-	-	194	141	-	-	-	-
ps lat.	10.0	12.0	11.8	13.1	10.0	10.0	10.0	10.0
# GR used	1	3	3	3	1	2	1	1

3.3.3 Wall Clock Time Comparison between Projected XMT Processor and AMD Opteron

In previous sections, various aspects of the performance of the XMT FPGA computer are measured and evaluated, but we are not ready to compare the performance of the XMT FPGA computer against any existing processors, because the clock rate of the XMT FPGA computer is too low. It is clear that the XMT processor with ASIC implementation can operate at a much higher clock rate, and thus achieve much better performance in terms of wall clock time. In this section, performance of an arbitrary XMT ASIC processor with a higher, yet quite modest, clock rate is projected and compared with an AMD Opteron processor. An arbitrary XMT processor with 800MHz internal clock, and 400MHz DDR2 SDRAM memory is chosen for the performance evaluation of an XMT ASIC processor (see lower part of figure 3.10). Both clock rates are quite reasonable and it is a bit conservative, considering the fact that 400MHz DDR2 SDRAM (PC2-6400) is commercially available and MIPS32® $74K^{TM}$ family cores operate at 1GHz.

If all components of the XMT FPGA computer are accelerated by the same

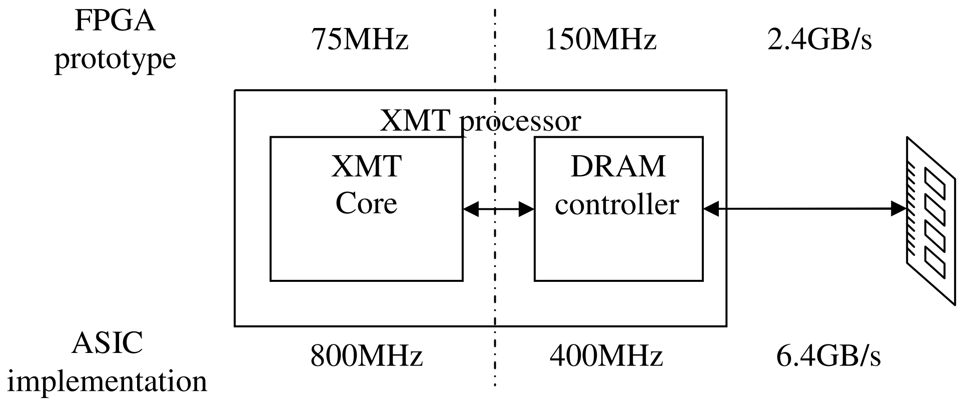


Figure 3.10: The differences in the FPGA and ASIC implementation

factor, the cycle count will not change and the wall clock time will decrease at the same ratio. Then we can easily project the performance of the XMT processor with a higher clock rate. However, this is not the case for the XMT prototype, as shown in figure 3.10. The behavior of SDRAM changes as clock rate increases, mostly because many timing constraints in SDRAM operations are given in absolute time (ns) and cycle times of different clock rates are different. In addition, the SDRAM controller in the FPGA prototype is faster than the XMT core, while it is slower than the core in the XMT ASIC counterpart. The challenge is how to apply constraints to the SDRAM controller in the FPGA prototype so that the emulator behaves in the same way as an 800MHz XMT processor in terms of cycle counts.

Based on our initial physical design with IBM technology and ARM standard cells, the 64-TCU XMT processor takes about 100 mm^2 in 90nm technology. As this was an academic project by a team of only four students and with rather limited optimization, this area usage should be interpreted as an upper bound on the area needed. Unfortunately, we were not able to find the area information about the tested 2.6GHz AMD Opteron, but a similar configuration (same cache size) AMD Opteron used 189 mm^2 in 130nm technology[21]. Although it is not fair to compare the two numbers directly, it is reasonable to believe that the XMT ASIC uses a similar silicon area as the AMD Opteron processor tested.

3.3.3.1 DDR2 SDRAM Basics

DDR2 SDRAMs are three dimensional memory arrays: bank, row and column. Accessing DDR2 SDRAM normally requires multiple steps. Banks are independent of each other and have a row buffer. A typical access cycle of the SDRAM includes three commands: ACTIVE, READ or WRITE and PRECHARGE. A row in a bank can only be accessible after it has been brought to the row buffer, which is done by the ACTIVE command. When an ACTIVE command is sent, the bank and row number should also be sent along with it to SDRAM. After tRCD, a READ or WRITE command, which specify the column address, can be sent to the active row. Data will be available after CL cycles for READ and should appear on the data bus after WL (normally WL=CL-1) cycles for WRITE command. A PRECHARGE command is needed before sending a new ACTIVE command for other rows in the same bank.

Periodical refresh is necessary for SDRAM or it will lose the data. Among the many time constraints on SDRAM operations, only part of them, which determines the performance of a SDRAM chip, are listed below. The second to last column is the specification from a Micron DDR2 SDRAM chip, MT47H128M8x8 -25E, and the last column lists the equivalent number of cycles in 800MHz.

Table 3.6: Timing constraints for MT47H128M8x8 -25E

Symbol	Parameter	time (ns)	cycle in 800MHz
t_{RAS}	ACTIVE to PRECHARGE	45	36
t_{RCD}	ACTIVE to READ/WRITE	12.5	10
t_{RC}	ACTIVE to ACTIVE(same bank)	55	44
t_{RRD}	ACTIVE to ACTIVE(different bank)	7.5	6
CL	READ to the first data	12.5[1]	10
WL	WRITE to the first data	10[1]	8
t_{RTP}	READ to PRECHARGE	7.5	6
t_{WTR}	WRITE to READ	7.5	6
t_{WR}	WRITE to PRECHARGE (write recovery)	15	12
t_{RP}	PRECHARGE period	12.5	10

3.3.3.2 Slowing Down the DDR2 SDRAM

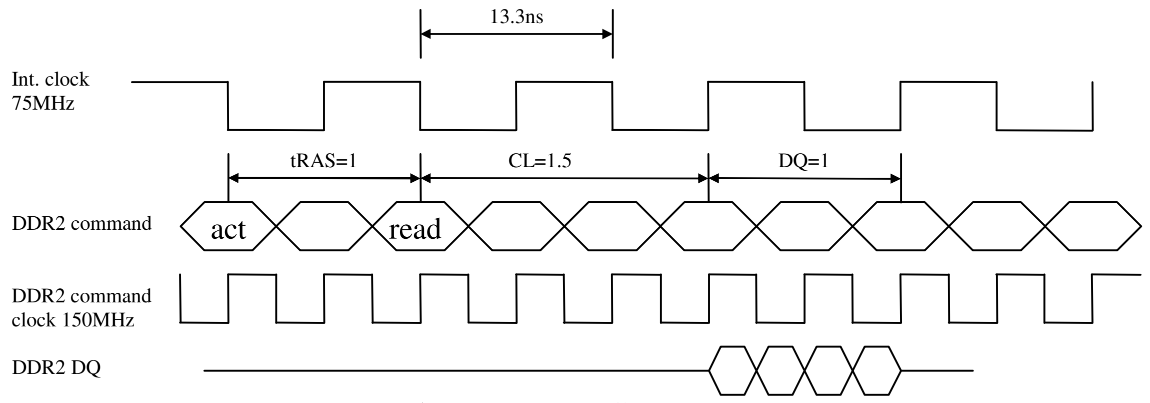
The idea of projection is to design a low clock rate system that behaves exactly as the high clock rate system in terms of cycle count. In other words, if the low clock rate system is cycle-accurate for the high clock system, the cycle count can be used for evaluating the high clock rate system. For example, t_{RCD} in table 3.6 is

[1]converted from cycle counts for 400MHz

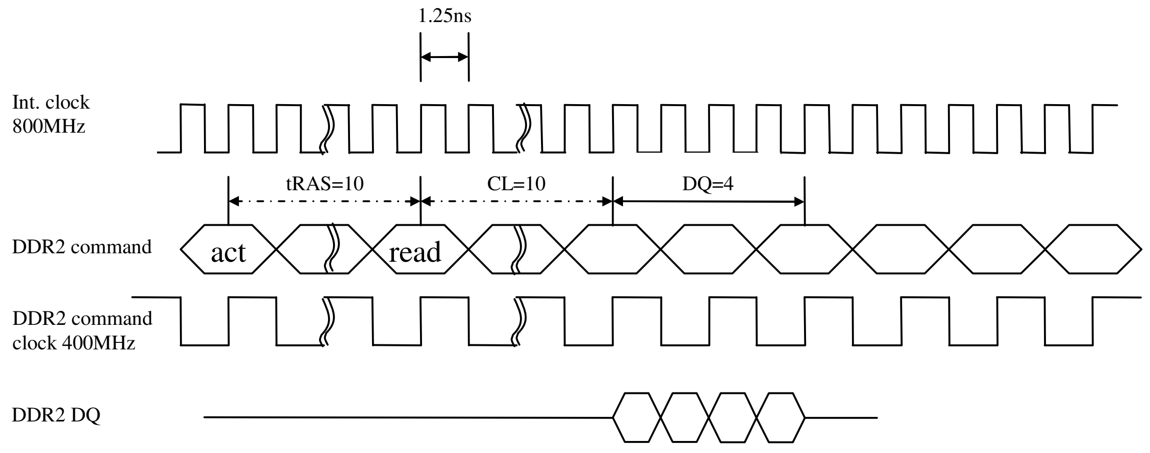
12.5ns, which is 10 cycles in 800MHz, but less than 1 cycles in 75MHz. To project a 800MHz XMT system, delay of 10 cycles needs to be used for t_{RAS} instead of 1 cycle. The number of cycles used for the XMT FPGA projection system are listed in table 3.6 with exception of CL and WL. The value of CL is limited to certain numbers and the proper number for projection, 10, is out of the range. This problem is solved by using the natural CL=3, but delaying READ or WRITE commands to accommodate the differences in the two systems. For CL and WL, 12 cycles and 11 cycles are used for delay, respectively. These delays also accommodate the difference in the data transfer rate or the burst length in cycles. A burst of 4 columns takes 1 cycle in the XMT FPGA 75MHz system, but 4 cycles in the XMT ASIC 800MHz system.

Figures 3.11 (a) and (b) show the read timing in the XMT FPGA 75MHz system and the envisioned XMT ASIC 800MHz system respectively. The t_{RCD} and CL times are converted to the number of cycles in the corresponding clock. To make the XMT FPGA 75MHz system cycle-accurate for XMT ASIC 800MHz, the DDR2 READ commands were delayed so that the last data arrives 24 cycles later as shown in figure 3.11 (b). The READ command is delayed for 12 cycles to accommodate the CL difference and data transfer rate.

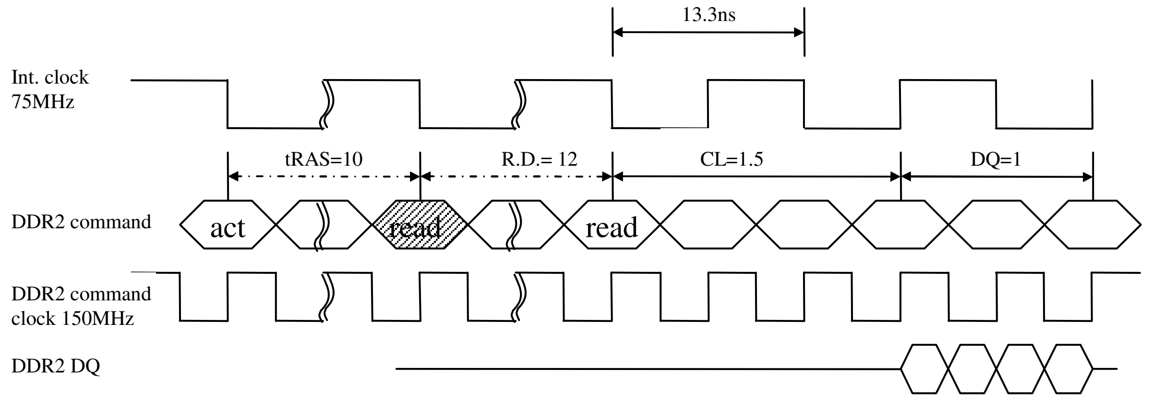
The SDRAM bandwidth is also reduced to match the imaginary 800MHz XMT system, where the SDRAM data transfer rate (after doubling the command clock rate) is the same as the internal clock rate and a READ/WRITE command can be sent to the SDRAM every 4 cycles, because the minimum burst length is 4 cycles. To emulate this, the READ/WRITE commands from all banks are queued and issued



(a) 75MHz XMT FPGA processor



(b) 800MHz XMT ASIC processor



(c) 75MHz XMT FPGA processor (cycle accurate for 800MHz)

Figure 3.11: Cycle-accurate read timing. DDR2 SDRAM commands are delayed to emulate 800MHz ASIC processor with a 75MHz FPGA processor.

to the SDRAM only every 4 cycles. This results in 3 bubble cycles in the XMT 75MHz, where a burst of 4 columns are transferred in 1 cycle, because the SDRAM transfer rate is 4 times 75MHz, or 300MHz.

Similarly, the rest of the DDR2 commands are also limited to 1 command per 2 cycles, which is done in a similar way to the READ/WRITE command. All commands other than READ/WRITE are queued in another queue and committed only after an empty cycle.

Table 3.7 summarizes major modifications applied to the prototype for a proper emulation.

Table 3.7: Modifications for performance projection

Item	75MHz	75MHz emulating 800MHz	800MHz emulated
Read latency [a] (cycle)	3.5 [b]	24.5	24.5
Maximum DRAM command per cycle	2	0.5	0.5
Peak bandwidth	2.4GB/s	0.6GB/s	6.4GB/s

[a]Latency of DRAM access depends on many factors. We only noted the latency of a read operation, under some DRAM assumptions. For those familiar with DRAMs and DRAM terminology, the assumptions are that the reading is done from a closed bank and there are no activities in other banks.

[b]The DRAM controller operates at 150MHz and one cycle in 150MHz is converted to half a cycle in 75MHz

3.3.3.3 Validation of the Projection

The XMT 800MHz was also simulated using verilog with the DDR2-800 model from Micron [53]. Since the simulation model from Micron is an accurate representation of a real DDR2 SDRAM chip, the cycle counts acquired from the verilog simulation can be used as the reference. The limitation of the simulation is the long simulation time. We chose a few programs to check the accuracy of our projection. Table 3.8 lists the number of cycles measured in both simulation and emulator for different sizes of a random memory access test. This test mixes different types of memory accesses: continuous read (CR), randomized read (RR) and randomized write (RW), in the ratio of 2:1:1 (CR:RR:RW). The total amounts of memory accessed by each test are listed in table 3.8. Table 3.9 lists the results from the eight kernel benchmarks with small input size described in table 3.2. From tables 3.8 and 3.9 we can see that the projection is quite accurate.

Table 3.8: cycle counts (million) in a random memory access program (different sizes)

test	1	2	3	4	5	6
accessed memory	4MB	8MB	16MB	32MB	64MB	128MB
simulated	4.782	10.88	23.23	48.10	97.80	197.5
projected	4.944	11.05	23.68	49.14	100.0	201.9
error	1.3%	1.5%	1.9%	2.2%	2.3%	2.2%

Table 3.9: cycle counts (million) in kernel benchmarks

Input	mmul	qsort	BFS	DAG	add	comp	BST	conv
sim.	1.111	2.311	13.03	18.92	2.043	5.410	4.690	5.328
proj.	1,126	2.271	13.20	19.12	1.988	5.470	4.750	5.334
error	1.3%	-1.8%	1.3%	1.1%	-2.7%	1.1%	1.3%	0.11%

3.3.3.4 Envisioned Performance of XMT ASIC 800MHz with DDR2-800

With the extra delays and reduced command and data rate, the XMT FPGA 75MHz system is a cycle accurate emulator of the XMT ASIC 800MHz. Although the behavior of the system is not exactly the same, it is very close in terms of cycles, because the two important key aspects of a SDRAM, latency and bandwidth, are accurately represented with the proper number of cycles. The execution time is calculated by converting the cycle count using the period of the 800MHz clock or by dividing the wall clock time by a ratio of 800/75=10.67. Figure 3.12 shows wall clock time of the 8 kernel benchmarks in AMD Opteron 2.6GHz, XMT FPGA 75MHz and envisioned XMT ASIC 800MHz. All wall clock time is normalized to the execution time of AMD Opteron 2.6GHz.

The AMD Opteron processor is operating at 2.6GHz and has 64KB (instruction) + 64KB (data) L1, and 1MB L2 cache. The system used for testing had dual channel PC-3200 DDR SDRAM, which provides bandwidth of 6.4GB/s. The envisioned XMT ASIC 800MHz system outperforms AMD Opteron for all 8 kernel benchmarks.

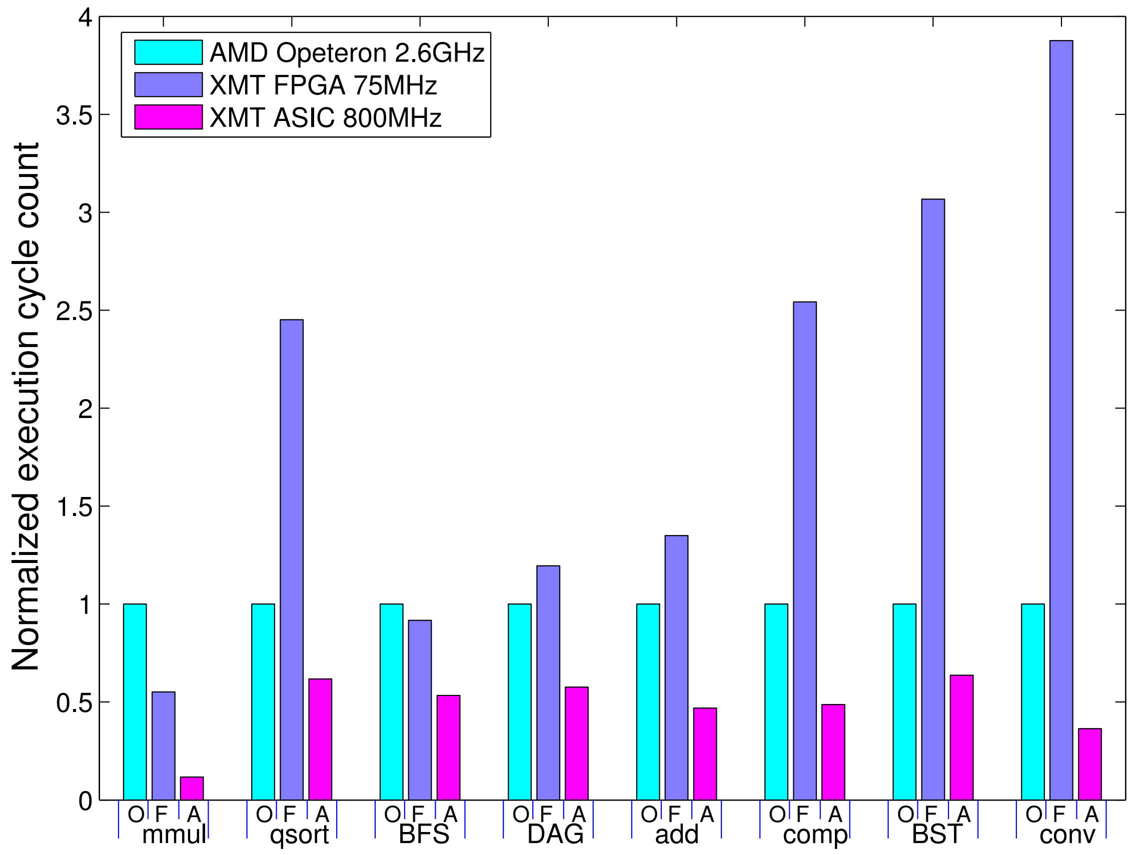


Figure 3.12: Normalized execution time. 'O','F' and 'A' at the bottom of the bars mean AMD Opteron 2.6GHz, XMT FPGA 75MHz and envisioned XMT ASIC 800MHz respectively.

It is quite impressive considering that the XMT ASIC 800MHz processor has only a total of 256 KB shared cache and operates at a much lower clock rate. The envisioned XMT ASIC 800MHz processor showed a significant advantage over the AMD Opteron in two CPU-bound benchmarks, mmul and conv.

Chapter 4

Local Caches vs Shared Caches in XMT

4.1 Why Not Use Private Caches?

The reason for not having private caches in TCUs/clusters is that XMT is a fine-grained parallel architecture, where a cache line is not the proper granularity for some problems. The aggressive number of TCUs (1024), that we are targeting, would limit the design to having cluster-level private caches, since it is prohibitive in terms of area for each TCU to have its own private cache. In this section, the current XMT design and an alternative XMT processor that incorporates cluster-level local caches are compared and analyzed.

Spacial locality - memory locations will be more likely referenced if their neighbors are referenced - is an important feature of the computer program that cache systems try to exploit. However, in XMT, this is not always true. Because the XMT is a fine-grained parallel machine, it is possible that a cache line is accessed by different TCUs from different clusters. Fetching a cache line, therefore, can create unnecessary loads to the interconnection network and cache system. Prefetch buffers in the XMT processor can be used to take advantage of spacial locality without wasting the precious bandwidth of the interconnection network, since TCUs, with help from the compiler, can prefetch only items that are guaranteed to be used.

The hardware is responsible for handling the cache coherence problem in a typical processor with multiple private caches. A directory-based system is the popular scalable solution. In a typical directory-based multiprocessor system, if a processor writes to a memory location, the cache line has to be exclusively owned by that processor. This requires that all other copies of the cache line in other processors are invalidated, resulting in non-trivial message exchanges and long latency for a write. On the other hand, if the cache line is not shared by multiple processors, private caches can improve the performance of the processor.

In the XMT processor, there are multiple shared cache modules operating independently. Any memory location is only available in one of these cache modules. The cache modules are non-blocking, meaning they can still serve the requests from TCUs while previous cache misses are pending. The latency of cache accesses ranges from 20 to 30 cycles for a cache hit, depending on the number of cache modules, the number of clusters and clock speed. However, with broadcasting, prefetch, read-only buffers and non-blocking stores, the average latency can be reduced. For example, for a read request, if a prefetch instruction is issued prior to the read, the latency can be one, similar to a processor with a local cache. It is worth noting that in early years when only multi-chip parallel computers were available, it was impossible to have a shared cache with latency similar to the XMT processor.

In chapter 2, the memory hierarchy of the XMT processor is presented. The TCUs/clusters do not have private local caches and as a result, every memory access requires a round trip to the shared cache through the interconnection network, if no other optimization, such as prefetch, is applied. With the introduction of the prefetch buffer, read-only buffer and broadcasting, the latency can be reduced, but it is interesting to know what would happen if a local private cache with a hardware cache coherence protocol was used instead. In this chapter, the XMT processor without local cache is compared to an alternative XMT processor where each cluster has a local private cache. For brevity, XMT S(hared) and XMT P(rivate) will be used to name the XMT processor without local cache and with local cache, respectively.

4.2 XMT Processor with Local Caches

The verilog HDL model is revised to analyze the XMT architecture with local caches. The model is only used for simulation and is not synthesizable. The memory hierarchy of the revised XMT P processor and original XMT S processor are shown in Figure 4.1 side by side. The two major changes in processor XMT P are: (a) the read-only buffer in the original design is replaced by a coherent private cache and (b) a directory module is attached to every shared cache module.

4.2.1 Directory

A directory-based MSI protocol is used in the XMT P for the cache coherence protocol. The directory keeps the status of each cache line. Since the memory space is partitioned among multiple shared cache modules and any memory location can only be found in one of the these cache modules. The directory is also distributed in the same way that cache lines are distributed and are placed between the intercon-

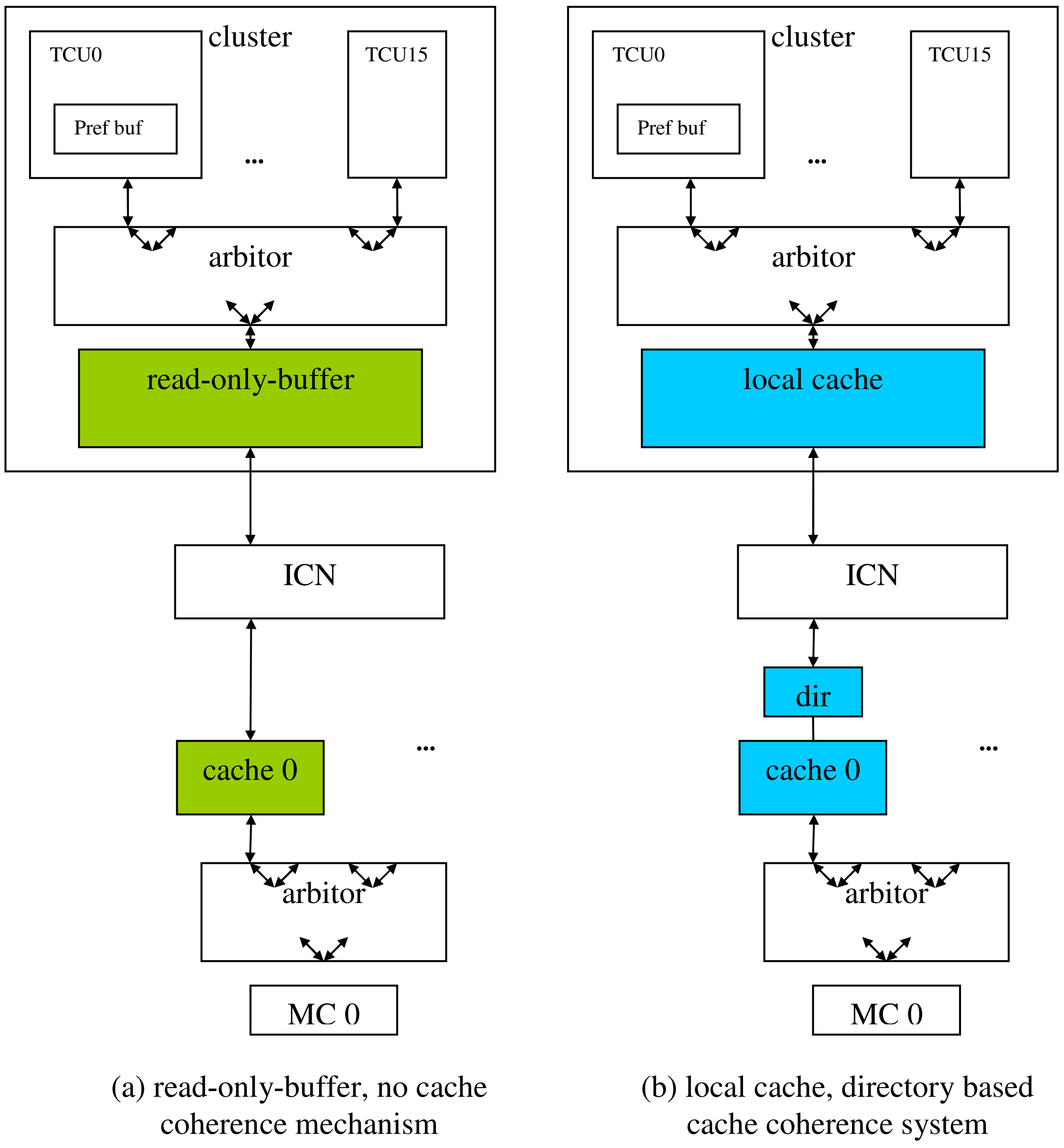


Figure 4.1: Two XMT processors side by side: (a) XMT S processor, without local cache; (b) XMT P processor with local private cache

nection network and the cache modules. Accessing shared cache modules requires checking the corresponding directory entry first, then depending on the status of the cache line, access is either permitted immediately or delayed until all necessary cache coherence messages have been exchanged.

In the directory, one bit per cache line is assigned for each of the private caches. If a cluster has a copy of a cache line, the corresponding bit is set to 1, otherwise it is set to 0. There is one extra bit for each cache line, indicating whether the cache line is exclusively owned by a private cache. Note that when the exclusive bit is set, only one of the private cache bits is 1 and the rest are all 0s. The cache line status can be

Clean None of the private caches has a copy of the cache line. Directory entry has 0 value.

Shared Exclusive bit is 0, and one or more private caches have a copy of the cache line.

Exclusive Exclusive bit is 1, and only one private cache has a copy of the cache line.

Figure 4.2 shows the state transition diagram of a cache line in the directory. The message(s) for each of the transitions are listed below, which are exchanged between private caches and directory.

① An exclusive read request from a private cache. The request will be forwarded to the shared cache, which delivers the data to the private cache.

② The private cache writes back the modified cache line to the shared cache.

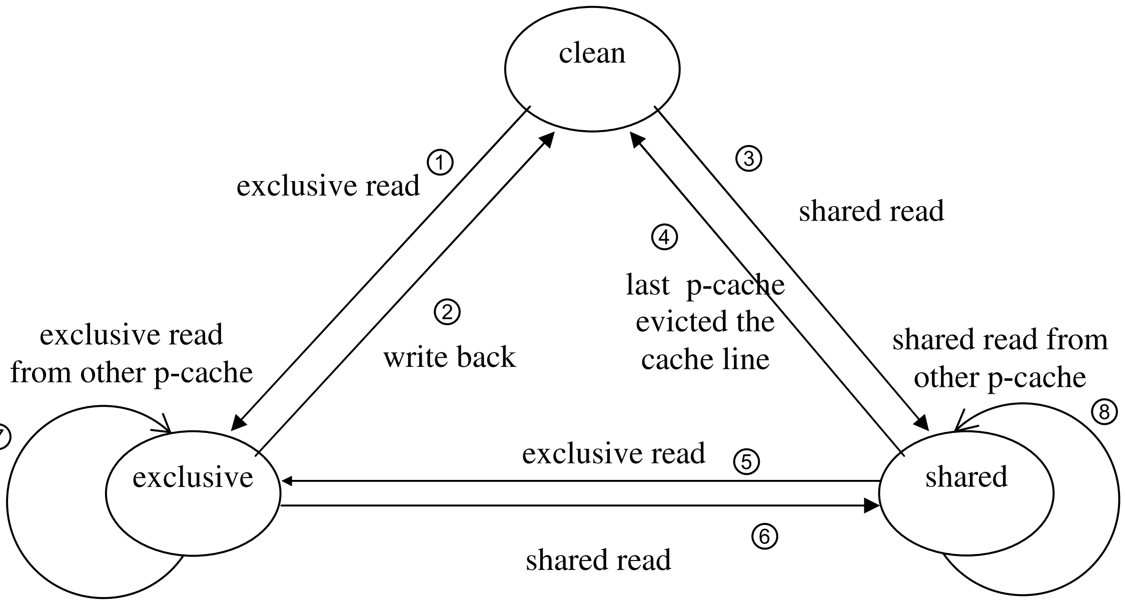

clean
exclusive read ①
② write back
③ shared read
④ last p-cache evicted the cache line
exclusive read from other p-cache
shared read from other p-cache
⑦
⑧
exclusive
shared
exclusive read ⑤
shared read ⑥

Figure 4.2: State transition diagram of a cache line in directory

③ A shared read request from a private cache. The request will be forwarded to the shared cache, which delivers the data to the private cache.

④ A cache line evicted message is received from a private cache line and no other private cache keeps this cache line.

⑤ An exclusive read request or an upgrade message is received. First, the directory will send invalidation messages to all private cache modules who have a copy of this cache line. The private cache modules invalidate the cache line and then send back acknowledge messages to the directory. After all acknowledgement messages are received, the directory is updated to an exclusive state. For an upgrade request, an approval message is sufficient, since the data is already available in the private cache, but for an exclusive read request, the shared cache module has to provide data.

⑥ The cache line is in the modified state in one of the private caches i, and a shared read request arrived from another private cache j. An invalidation message is sent to the private cache i from the directory. Private cache i invalidates the cache line and then sends back the modified cache line to the directory. Finally, the cache line is forwarded to the private cache j as well as the shared cache.

⑦ The cache line is in a modified state in one of the private caches i, and an exclusive read request arrived from another private cache j. An invalidation message is sent to the private cache i from the directory. Private cache i invalidates the cache line and then sends back the modified cache line to the directory. Finally, the cache line is forwarded to the private cache j and the directory status is updated.

⑧ The cache line is already in a shared state and a shared read request is received from another private cache. The request is forwarded to the shared cache, which delivers the data to the private cache. The directory status is updated as well.

4.2.2 Private Caches

A cache line in a private cache can be in one of the 3 states listed below:

Invalid The cache line is not available in the private cache.

Modified The cache line is exclusively owned and the content is modified; both read and write requests are permitted.

Shared The cache line is shared with other private caches and only read requests are permitted.

Figure 4.3 shows the state transition diagram. For a cache miss, the cache line is

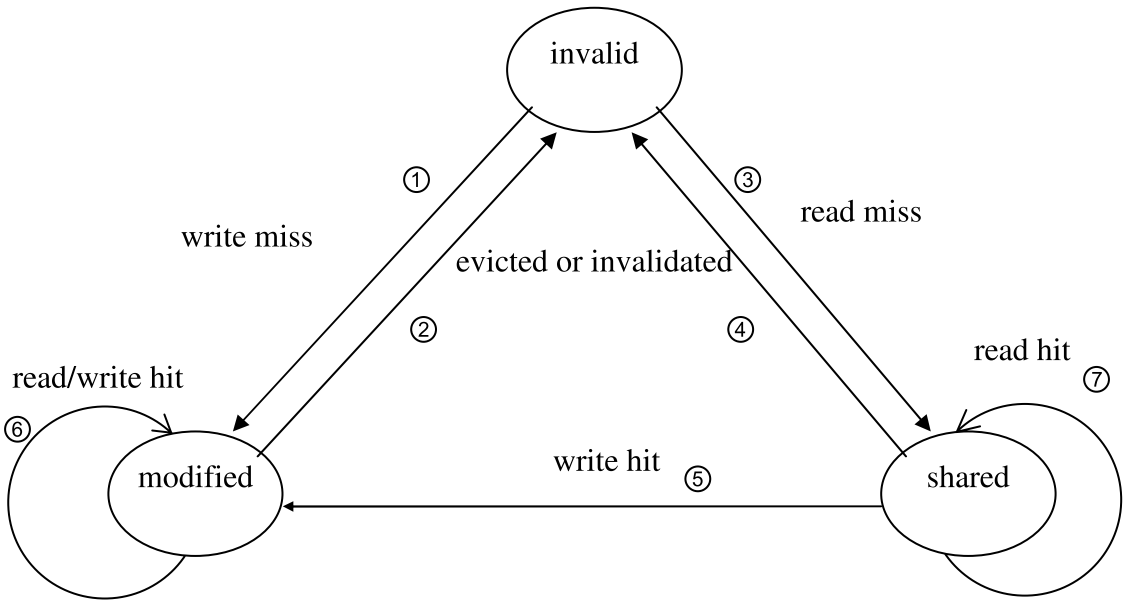


Figure 4.3: State transition diagram of a cache line in private cache

brought to the private cache from the shared cache, but marked as *shared* for a read miss and as *modified* for a write miss. The following read operations on the cache line will be served without changing the status. For the write, the operation is different based on the current cache line state. If the current state is *modified*, the write operation is processed immediately. If the write operation turns out to be a hit in a *shared* state, the cache line state must be changed to modified before the write operation can be processed. Note that for any state change, the private cache needs to exchange cache coherence messages with the directory. The messages exchanged between private cache and directory are listed below.

① An exclusive read request is sent to the directory and receives the cache line.

② This transition may occur as a result of a cache line eviction or an invalidation request from the directory. In either case, the modified cache line will be written back to the shared cache.

③ A shared read request is sent to the directory and receives the data of the cache line.

④ This transition may occur by cache line eviction or invalidation request from the directory. In either case, an invalidation acknowledgement message will be sent to the directory, so that the directory can be updated properly.

⑤ An upgrade request will be sent to the directory. The response from the directory can be (a) an approval of the upgrade request. In this case, the data in the private cache is up-to-date and only the status needs to be changed. (b) The cache line is delivered to a private cache and the cache line needs to be updated with the new data. This will happen if the data in the private cache is obsolete.

⑥ ⑦ No message is exchanged, because the state does not change.

4.3 Simulation Results

For the XMT S and XMT P processors, four different configurations, with 8, 16, 32 or 64 clusters are tested. In each case, the number of cache modules are the same as the number of clusters. In XMT S, each cluster has an 8KB read-only buffer, while a cluster in XMT P has a 32KB, 2-way associative private cache. Both XMT S and XMT P use the same size shared cache module: 32KB and 2-way associative. The bandwidth of off-chip memory is assumed to be a quarter of the total bandwidth of on-chip cache modules. A constant delay of 150 cycles are used for off-chip memory access from shared cache. Note that, as the number of clusters/caches doubles, total size of the on-chip parallel shared cache also doubles. Since XMT P can also take advantage of value broadcasting and prefetching, these two features are also incorporated in XMT P. Table 4.1 summarizes the latencies of read operations from different levels of the memory hierarchy.

The eight micro-benchmarks listed in section 3.2 are used to analyze the performance of the XMT processor with a private cache per cluster. Due to long simulation time, only small input sizes are tested. The input size and memory usages are listed in table 3.2. The basic difference in the two processors (XMT S and P) are the granularity of the data blocks transferred by the interconnection network. When private caches are used by the clusters, the minimum data transfer unit is a cache line, requiring 8 packets. Because of this, the XMT P processor has a disadvantage for extremely fine-grained parallelism and the XMT P will perform better if granularity of the program is increased. This was considered during the

Table 4.1: Read latencies in XMT S and XMT P

read from	# of clusters	XMT S	XMT P
prefetch buffer	all	1	1
RO buffer or local cache[a]	all	4	4
shared cache (SC) [b]	8	21	24
(miss in RO buffer	16	25	28
or local cache)	32	29	32
	64	33	36
off-chip memory	all	150+SC	

[a]Since RO buffer or local cache is shared by 16 TCUs in a cluster, arbitration and result delivery takes 1 cycle each.

[b]Assumed there is no contention in the interconnection network and shared cache module. XMT S needs extra 3 cycles for directory access and additional FIFOs

development of the eight parallel programs. The tasks are partitioned in coarse-grained mode whenever it is possible. For example, in the compaction application, the input array is partitioned into 1024 blocks, which is the maximum number of TCUs we simulated, and each block is assigned to a thread. With this partitioning a cache line is very likely to be read by only one thread (TCU) which minimizes unnecessary transferring of data.

Table 4.2 summarizes which features are applied to each of the eight benchmarks. While broadcasting can be used for an array address or a pointer in any program, it was not counted as an applicable feature, since in this case, broadcasting is not necessary. However, the pivot in qsort and the level number in BFS are always dynamic and use of broadcasting is beneficial. All benchmarks have loops and prefetch is used for fetching data for iterations ahead. The read-only buffer is used in five out of eight benchmarks and in the other three benchmarks, no values are shared or reused by threads, thus the read-only buffer is not used.

Table 4.2: Applicable features in benchmarks

App.	broadcasting	prefetch buffer	read-only buffer
mmul	N	Y	Y
qsort	Y	Y	N
BFS	Y	Y	Y
DAG	N	Y	Y
add	N	Y	N
comp	N	Y	N
BST	N	Y	Y
conv	N	Y	Y

In this section, the performance of XMT S and XMT P processors is analyzed based on the execution cycle count, total messages transferred by the interconnection network and the average latency of read operations.

4.3.1 Program Execution Cycle Count and Speedups

The execution time of eight benchmarks on both the XMT S and XMT P with a different number of clusters were recorded and the speedups were calculated over the eight-cluster/cache XMT S processor. The results are shown in figure 4.4. First, it is very clear that both XMT S and XMT P scaled up very well, except for

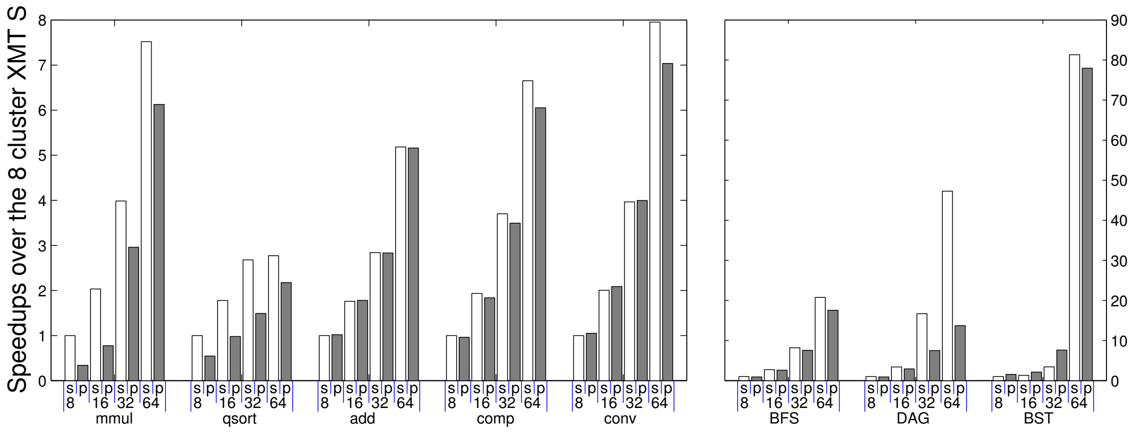


Figure 4.4: Speedups of the benchmarks over the 8-cluster XMT S. Bars are marked with ’s’ and ’p’ for XMT S and XMT P respectively. The numbers under each pair of bars are the number of clusters

the qsort benchmark. The input size of qsort is 100K and it is not sufficiently large to provide enough tasks for large-scale XMT processors, with 32 and 64 clusters. The high speedups for BFS, DAG and BST are mainly from the enlarged cache size rather than an increased number of clusters/TCUs. This is also supported by the fact that the latency of read operations in these programs decreases (see figure 4.6). For mmul, qsort, comp, BFS and DAG, XMT S outperformed XMT P for all 4 configurations of the XMT processor. In add and conv, XMT P marginally beats

XMT S in the 8 and 16-cluster XMT processors, but XMT S wins in the large-scale XMT processors of 32 and 64 clusters. This is interesting because both add and conv are coarse-grained parallel programs. XMT P had an advantage over the XMT S for the application BST in 8, 16 and especially for a 32-cluster XMT processor, where it is two times faster. In the data structure of the BST, the pointer to the left child always follows the parent node. XMT P takes advantage of this spacial locality automatically. However, for XMT S, explicit prefetch commands need to be inserted, which is not done in the simulated program because we are not sure yet that the compiler can figure out this locality.

4.3.2 Traffic Volume in the Interconnection Network

The minimum data chunk that is transferred from shared cache to the clusters is different in XMT S and XMT P. They are a cache line (eight packets) and a word (one packet), in XMT P and XMT S respectively. As a result, for the same program, the total number of packets exchanged between clusters and cache modules are different in the two processors. Figure 4.5 shows the normalized number of exchanged packets in each of the benchmarks for different sizes of XMT processors. When TCUs/clusters bring in data from cache modules, XMT S sends one packet of request and receives one packet of response. On the other hand, XMT P also sends one packet of request, but it receives a cache line, eight packets, of response. If all words of these cache line are accessed by the TCUs, XMT P has an advantage over the XMT S, since for a cache line, XMT P needs a total of 9 packets but XMT S

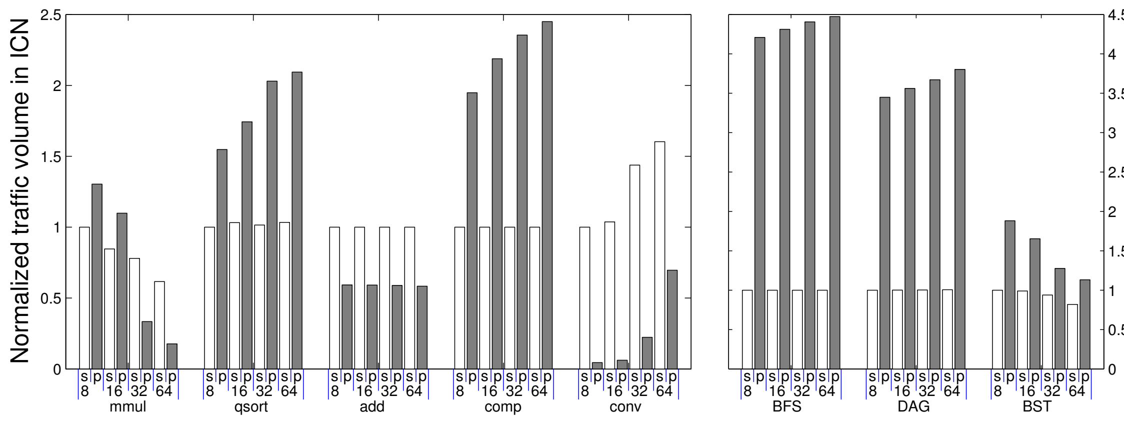


Figure 4.5: Number of packets transferred between clusters and cache modules, normalized to the 8-cluster XMT S for each applications. Bars are marked with ’s’ and ’p’ for XMT S and XMT P respectively

needs 16 packets. However, if less than 4 words in a cache line are used, then, XMT S has less communication overhead. For a write operation, the situation is similar, but it is prohibitive for false sharing [19], which happens when different clusters are writing to different words in the same cache line. For example, in the worst case, when eight different words in a cache line are written by eight different clusters, a total of up to 151 packets need to be transferred in XMT P, but only 24 packets in XMT S. The calculation follows. In XMT P, if a cache line will be brought in with 9 packets for a read operation, then the whole cache line (seven words in the cache line are unchanged) will be written back to the shared cache line, requiring 9 packets (one address and eight data). Each cluster needs to have exclusive access to the cache line, so an invalid request packet will be sent to the previous owner of the cache line. The total number of packets required is $18 \times 8 + 7 = 151$. This is quite simple in XMT S, where each write needs 2 request packets (address and data), plus one acknowledge packet, a total $3 \times 8 = 24$ packets.

For add and conv, which are coarse-grained parallel programs and where every word in a cache line is used by a TCU, XMT P has significantly less packet exchange than XMT S. That is the best scenario for the XMT P. Note that the XMT S can also increase the granularity of software prefetch and reduce the overall number of packets transferred, which is part of the future work. For comp and qsort, although the reading is done in coarse-grained mode, writing is still done in fine-grained mode. The initial experiment shows that increasing the write granularity by reserving more than one word and clearing the bubbles (unused words) in serial mode, did not improve the overall performance, because of Amdahl's law. In the BFS, DAG and BST, more packet exchange in XMT P is obvious because only part of the cache line, which is brought in regardless, is likely to be used.

In the XMT P processor, even if a cache line is accessed by different TCUs, a cluster-level private cache can still be beneficial if those TCUs are from the same cluster. In XMT (both P and S), each TCU will be assigned a natural virtual thread for the first round, which has regularity compared to the dynamic assignment from a *join* instruction. The regularity makes more efficient use of local cache in the clusters because the cache lines are more likely being shared by TCUs from the same clusters. Unfortunately, as the number of cluster increases, the probability of this lucky sharing decreases. This explains why the traffic increased in qsort, comp, BFS and DAG with an increasing number of clusters. In mmul, the matrix is 128x128 and the overall computation is partitioned into 1024 pieces. As the number of TCUs increases, the number of virtual threads that are assigned by a *join* instruction decreases, resulting in better use of local cache and reduced packet exchange. In other words, if TCUs advance synchronously and they share the same cache line, the local private cache can be used more effectively resulting in less traffic in the interconnection network as well. The same effect explains the BST application. For conv, because the program exhibits a high level of spacial locality and the local cache size is big enough to hold the working set, there is very little traffic during computation. The interconnection network traffic is mainly for loading the local caches and only one communication sequence is required for each local cache. Therefore, the traffic volume increases with the number of clusters in the conv benchmark.

In the XMT S processor, because the memory access is per-word, the traffic volume in the applications, qsort, add, comp, BFS and DAG, remains the same, with an incresing number of clusters. The synchronous execution also reduces the traffic volume in mmul and BST as in XMT P. For the conv benchmarks, because the size of read-only buffer (8KB) is smaller than the private cache (32KB) in XMT P, XMT S has to move data between clusters and the shared cache during computation, resulting in much more traffic volume.

4.3.3 Average Latency for Read Operation

The most important task of a cache is to store the data close to the processor and shorten the latency of read operations. Compared to XMT S, XMT P has local private caches which are much closer to the TCUs. It is interesting to understand how effective the prefetch buffer and read-only buffer are in shortening latencies of read operations compared to a local cache. Figure 4.6 shows the average latency of the read operations in the eight benchmarks with different numbers of clusters.

When the number of clusters increases, two factors affect the average latency of

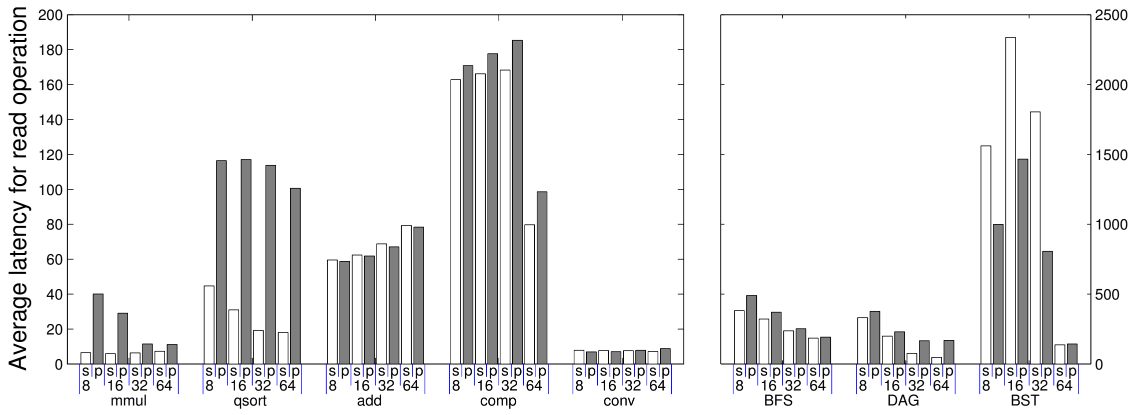


Figure 4.6: Average latency of the read operations in cycles

the read operations. First, the number of stages in the interconnection network, which is proportional to the $\log_2(\#ofcluster)$, increases, and the round trip to the shared cache takes more cycles. On the other hand, the total size of the on-chip caches increases with the number of cache modules, therefore, the cache hit rate in the shared caches increases and average latency of the read oeprations decreases.

In terms of latency of read operations, there is no clear winner between XMT S and XMT P. For mmul, qsort and DAG, the XMT S had significant advantage over the XMT P, especially for qsort. For BFS and BST (except for 64-cluster), XMT P had lower latency for read operations. The rest of the micro-benchmarks, add, comp and conv had similar latency numbers for both XMT S and XMT P processors. In general, the software controlled prefetch buffer and read-only buffer are as effective as a local cache in shortening the memory access time.

In mmul, the two processors showed a different trend when the number of cluster increases. In XMT P, the average latency decreases when the number of cluster increases. This is due to the same reason as the changes in traffic volume in the interconnection network. The synchronous execution of TCUs in the same clusters helped for high hit rate in local cache. Less traffic volume means that more read operations are served by the local cache without help from the shared cache, resulting in reduced latency. But for the XMT S, because there is no local cache, the latency instead depends on the effectiveness of the prefetch, which has longer latency for the large scale of the XMT processor.

4.3.4 Summary of XMT P and S

The different scales of the XMT S and XMT P processors are evaluated with eight kernel benchmarks. The three metrics: program execution cycle count, traffic volume in the interconnection network and average latency for read operations are used in the evaluation. XMT S, which does not have a local private cache per cluster, outperforms XMT P for the first two metrics. Although XMT P had some advantage over XMT S in the third metric for BFS and BST, XMT S had less or equal latency for other applications.

In general, XMT P is more efficient for coarse-grained parallelism than fine-grained parallelism due to the granuality of a cache line. XMT S also benefits from coarse-grained parallelism, but its impact is significantly less than in XMT P. The eight tested benchmarks are programmed in a coarse-grained manner, if possible. For example, add and conv are completely coarse-grained programs.

It is clear that XMT P is more complicated in terms of hardware implementation and takes more area than XMT S. The simulated XMT P is only a behavioral model and avoids many hardware implementation challenges. The storage for directory information is not negligible for the large scale XMT processor, which for a 64-cluster configuration reaches 25%. From figure 4.5, XMT P has more traffic volume in the interconnection network for many benchmarks, which means more power consumption in the interconnection network.

Because there is no local cache, XMT S depends on the compiler to take advantage of the prefetch buffers and read-only buffers. The challenges for the compiler are figuring out what to prefetch or store in the read-only buffer and ensuring the correctness of the program, because there is no hardware that guarantees cache coherence, like there is in XMT P.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Originated from PRAM theory, the XMT architecture is designed to efficiently execute fine-grained parallel programs based on PRAM algorithms. The XMT processor appears to programmers as a PRAM-like processor where they do not need to take care of load balancing and data locality. The XMT processor achieves this by introducing many architecture features: shared on-chip cache, high performance on-chip interconnection network, constant-time multi-operand prefix-sum unit, instruction (value) broadcasting, software/hardware co-managed prefetch buffer, and software/hardware co-managed read-only buffer. Note that the XMT processor partially relies on the compiler for optimizations that used to be part of the job of programmers.

The XMT processor has very good scalability, since the clusters and cache modules can be replicated as many times as the die area and power consumption permit. The only global resource in the XMT architecture is the interconnection network that connects clusters and cache modules. It is addressed by papers [8, 9, 7, 10]. The XMT programs are binary compatible among the different scales of XMT processors, which means re-compilation is not needed unless the program uses some specific information like number of TCUs explicitly, which is not needed in most cases.

The performance of the XMT architecture is very promising. The 75MHz 64-TCU XMT FPGA prototype outperforms a 2.6GHz AMD Opteron for 2000x2000 matrix multiplication. Note that the performance of the XMT FPGA prototype can be improved by fine tunings/optimizations that we did not do due to the limited man power. An arbitrary 800MHz XMT ASIC processor is studied with a cycle-accurate emulator, which is done by slowing down the DRAM component proportionally in the 75MHz XMT FPGA prototype. The results show that a 800MHz XMT ASIC processor outperforms 2.6GHz AMD Opteron processor for all eight benchmarks tested. Our initial results suggest that the 64-TCU XMT ASIC would take a similar area to the AMD Opteron in the same 90nm technology.

Coherent caches are used in many multicore processors and traditional massive parallel computers. In the XMT architecture, we chose to use software/hardware co-managed temporal storage as an alternative solution to the traditional coherent private caches. The choice is based on the observation that (i) The hardware implementation of a scalable cache coherence protocol is very complicated. (ii) Coherent private caches are inefficient for certain types of memory access patterns, like fine-grained parallelism. (iii) With help from the compiler, software/hardware co-managed temporal storages, like prefetch buffers and read-only buffers, provide spacial and temporal locality like a cache. The simulation results suggest that the XMT processor without coherent local caches can perform as well as an alternative XMT processor with incorporates coherent caches.

5.2 Future Work

Future work involving the XMT architecture is listed here.

- Floating point operations

  Since XMT FPGA prototype uses the MIPS I instruction set, a new dedicated register file for floating point operations needs to be added as the MIPS instruction set defines. The major work involves modifying TCUs to properly decode the new instructions and it is also needed to add shared floating point functional units to the clusters.

- Interrupt

  In serial mode, the conventional interrupt scheme can be applied to the MTCU. The challenge is in parallel mode and it can be divided into two questions. (i) how to interrupt all TCUs in parallel mode and switch back to serial mode? (ii) what kind of interrupt scheme needs to be supported by TCUs while the XMT processor is still in parallel mode? For the first question, one possible solution is as follows: When an interrupt happens, TCUs will not start a new thread but the MTCU has to wait until all TCUs finish their current thread, then the XMT processor switches to serial mode. After the MTCU returns from the interrupt service routine, it can spawn the rest of the threads that need to be executed. The potential problem of this solution is that the response time of the interrupt is undetermined and it will depend on the current active threads.

- Virtual memory

  This issue is related to interrupt handling very closely. When a page fault occurs in parallel mode, the MTCU may need to be involved in page table update, which probably needs to be handled in the form of service of an interrupt. For those TLB misses in individual TCUs, if the page table has a valid entry, it can be handled by the TCU itself by loading it from the shared memory.

- Operating System

  After introducing both interrupt handling and virtual memory, an OS can be ported to the XMT processor. Since the XMT processor is using the MIPS I instruction set, one of the portable linux OS versions is a good choice.

- Other minor improvements

  Other fine tuning/optimizations may improve the performance of the XMT FPGA prototype. The potential improvement may come from a faster clock rate in the interconnection network, optimizing the throughput of the on-chip caches. The current instruction buffers in TCUs, which limit the number of instructions in parallel sections, need to be replaced with a full instruction cache for function calls in parallel mode.

Appendix A

Introduction to the Verilog Model of the XMT Prototype

A verilog model of the XMT processor is developed for prototyping. The modules in the verilog code are explained based on the 64-TCU XMT ASIC version, but since many modules are shared by two prototypes, it is also applicable to the XMT FPGA prototype. The differences are listed in appendix G.

Figure A.1 shows the components of an XMT processor and their connections. The XMT prototype consists of clusters, an interconnection network, multiple on-chip cache modules and a master cluster.

Each chapter in appendixes explains one component from the XMT prototype shown in figure A.1 on the next page.

A.1 Clusters

The clusters are a group of TCUs and accompanying functional units. TCUs are the basic processing units of the XMT processor and each TCU executes a thread in parallel mode. TCUs have their own local registers and they are simple in-order pipelines including fetch, decode, execute/memory access, and write back stages. Every cluster has a single load/store port and a prefix-sum request port. The prefix-sum requests for the same base register from all TCUs in the same cluster are combined to one single request and sent to the global prefix-sum unit. Simple

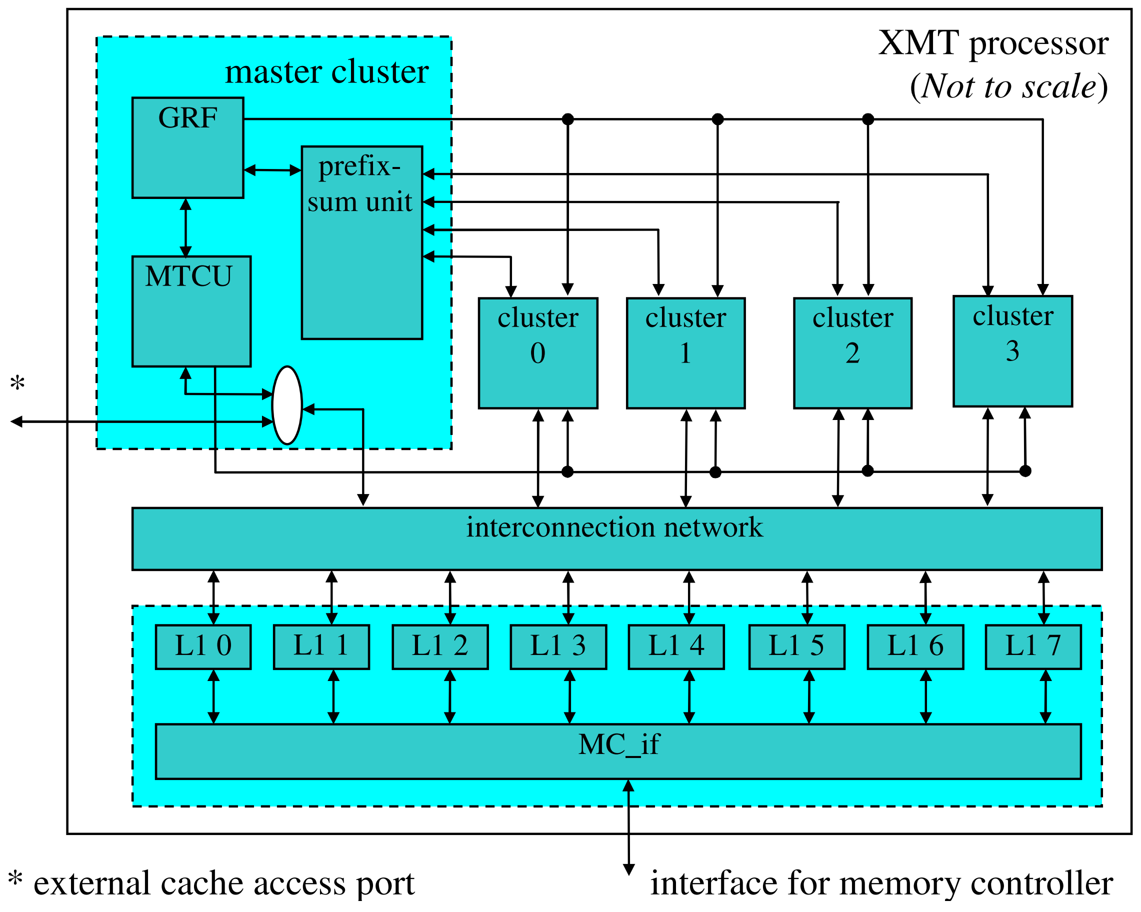


Figure A.1: XMT block diagram

functional units are dedicated to each TCU while complicated ones are shared by multiple TCUs. Arbitration of multiple requests are done fairly for TCUs.

A.2 Master Cluster

The master cluster includes a master TCU (MTCU), a global register file and a prefix-sum unit. The MTCU executes the serial portion of programs and handles the special XMT instructions such as *spawn* and *join*. The MTCU broadcasts the instructions for a parallel section to all clusters, where they are copied to a local instruction buffer and later fetched by TCUs inside clusters. The Master TCU has its own cache, L0, that is only active during serial mode and applies a write-through protocol. When the XMT processor enters parallel mode, the Master TCU discards its local cache. The overhead of the flushing the L0 cache is trivial since the write-through mechanism is chosen. When XMT operates in serial mode, the L0 cache is the first-level cache of the MTCU and multiple shared memory modules provide the lower level of the memory hierarchy (similar to a serial processor).

The prefix-sum operation is an atomic fetch-and-add computation that is very important for the XMT to achieve low-overhead synchronization between threads. The prefix-sum unit can accept binary input from multiple TCUs simultaneously and the execution will take constant time in XMT, regardless of how many threads participate.

Global registers in XMT are used primarily as base registers for prefix-sum operations in parallel mode. Among eight global registers, two are used for storing the begin (low) and end (high) thread IDs during parallel mode. The Master TCU can access global registers through special instructions that move values between global registers and local registers in the MTCU. Regular TCUs can access global registers only through prefix-sum operations.

A.3 Interconnection Network

Clusters and cache modules are connected by a high-bandwidth, low-latency interconnection network. Clusters send requests to the cache modules and receive responses. The communication between clusters and cache modules is all-to-all. When multiple clusters send requests to the same cache module, they are queued and delivered to the cache module serially.

A.4 Cache Module

The memory space is divided among multiple shared on-chip cache modules. Each cache module processes requests from TCUs independently and communicates with the off-chip DRAM through a shared DRAM access channel. To avoid an unbalanced load in different cache modules for certain patterns of memory access, hashing is used in mapping between memory address and cache modules. The shared caches are used primarily for data, since the instructions for regular TCUs are broadcast by the MTCU and stored in the instruction buffer.

A.5 Interface of XMT Processor

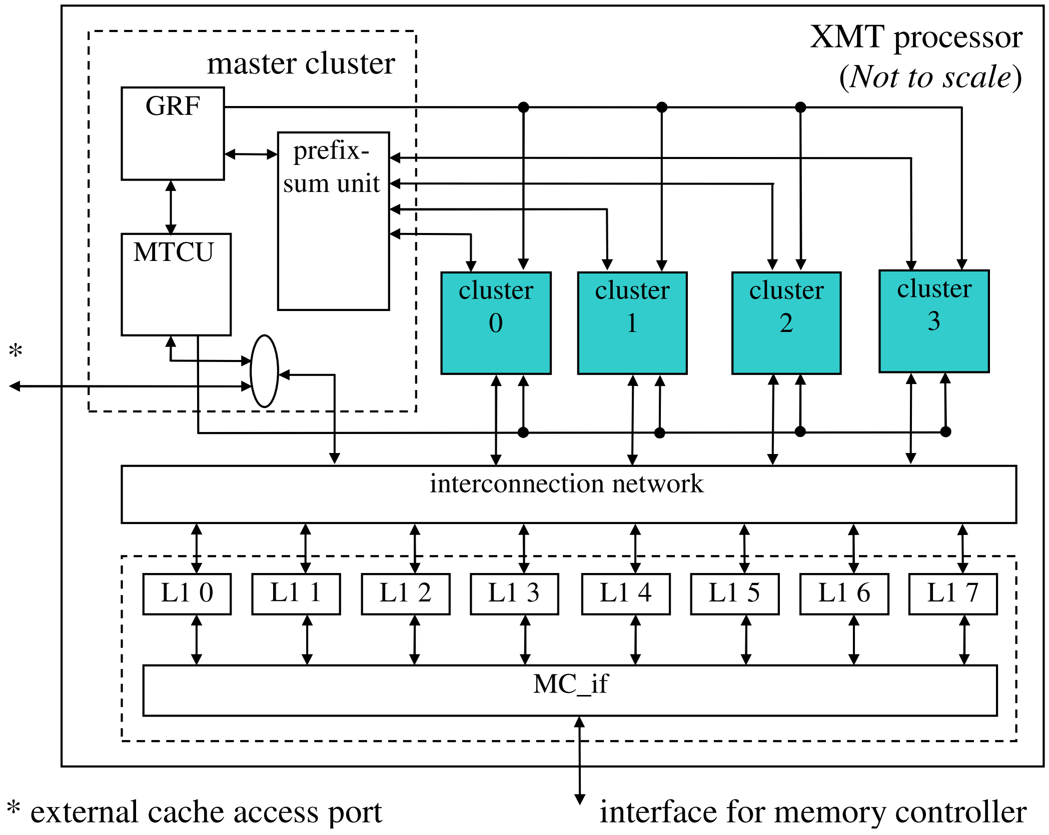
The XMT processor has two channels of interface: (1) External cache access port and (2) Off-chip DRAM controller access port. The external cache access port exposes the XMT memory hierarchy, including on-chip cache, to the outside world. The XMT ASIC prototype does not have an on-chip DRAM controller and it is supposed to be connected to an off-chip DRAM controller. The handshaking protocol is described in detail.

A.6 Differences in XMT ASIC and XMT FPGA

Unlike the XMT ASIC prototype, the XMT FPGA prototype includes an on-chip DDR2 DRAM controller. Besides the on-chip DRAM controller, other differences are also presented.

Appendix B

Cluster



B.1 Overview

A cluster is a group of 16 TCUs and accompanying functional units. The block diagram of a cluster is shown in figure B.1. TCUs have their own local registers and they are simple in-order pipelines including fetch, decode, execute/memory access, and write back stages. In parallel mode, threads will be executed by TCUs

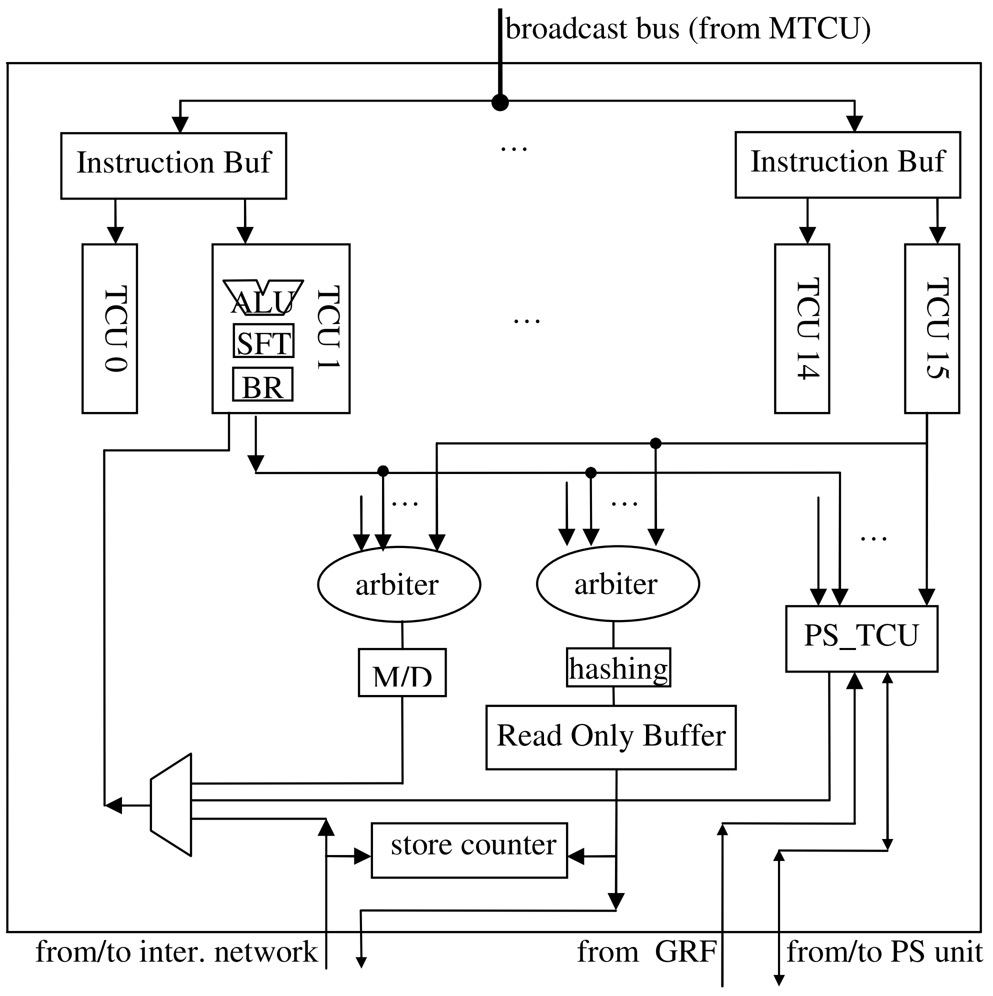


Figure B.1: Clusters

There are 16 ALUs, 16 Branch units, 16 Shifting units, 1 Multiplication/Division unit and 1 Load/Store unit in a cluster.

and when all parallel threads are finished, the XMT processor changes back to serial mode. Similar to a simultaneous multithreaded (SMT) processor, TCUs share some functional units: a Multiplication/Division (M/D) unit, read-only buffer and interconnection network port. If several TCUs assigned to a functional unit try to access it, proper arbitration is used to queue all requests. The read-only buffers in the clusters are hardware/software co-managed temporal storage for TCUs. The cluster has one load/store port to the interconnection network, which is shared by all TCUs inside the cluster. The store counter is used to flush the store operations by counting the number of pending stores.

As a key feature of the XMT processor, prefix-sum operations must be executed very efficiently. The hardware implementation of the prefix-sum unit can accept binary inputs from TCUs and the execution time does not depend on the number of TCUs that are sending requests to it. The PS_TCU module in a cluster combines all requests from TCUs within the cluster and sends one request to the global prefix-sum unit. It is also responsible for distributing the results from the prefix-sum unit to the individual TCUs. The PS_TCU module will be discussed in appendix C on page 162.

B.1.1 Interface of Cluster

The interface of a cluster module is listed in Table B.1 on the next page and Table B.2 on page 122. Parts of them are also shown in Figure B.1 on the page before.

B.2 Instruction Buffer

There are eight instruction buffers and each is shared by two TCUs. The size of an instruction buffer determines the maximum size of a parallel section of a XMT program. The instruction buffer has one write port and two read ports. A write

Table B.1: Inputs of the clusters

Name	Driver	Description
parallel	MTCU	1 in parallel mode, 0 in serial mode
cluster_id	hard-wired	id of cluster
instrBroadCast	MTCU	broadcast instructions
pNextToWrite	MTCU	destination address of the instrBroadCast
ls_back	ICN	responses of loads/stores
ls_select	ICN	acknowledge signal from ICN
writeIC	MTCU	write enable for instruction buffer
lastID	GR7	high ID of current spawn block
ps_back	PS unit	0 based prefix-sum results
ps_base	PS unit	original global register value for PS
ps_back_index	PS unit	transaction id for a ps operation
ps_base_reg	PS unit	base register and transaction id

Connections are shown in figure B.1 on page 119

port includes *addrWrite*, *write*, and *dataBusWrite*. When the *write* signal is high, the value on *dataBusWrite* will be written into the location specified by *addrWrite*. For a read operation, the data in location *addrRead* will appear on *dataBusRead* on the next clock cycle. The two TCUs can read instructions at the same time using two read ports.

Typical SRAM IPs only have two ports, but the instruction buffer uses three ports, therefore, one SRAM IP port is used as both a write and read port. A write

Table B.2: Outputs of the clusters

Name	Send to	Description
idle	MTCU	1 – idling, 0 – active
ls_send	ICN	request for load/store
ps_send	PS unit	request to PS unit
ps_Request	PS unit	indicating base addresses asked by ps operations

Connections are shown in Figure B.1 on page 119

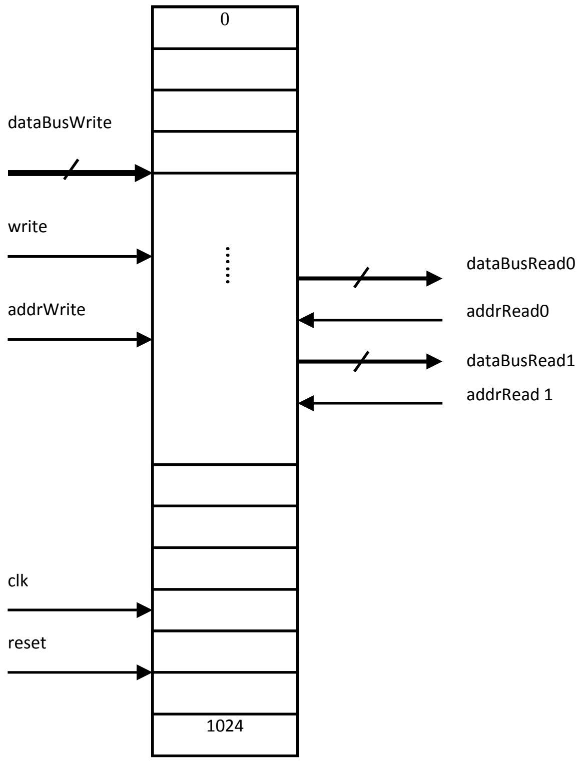


Figure B.2: Instruction cache

operation is given higher priority, thus one TCU that uses the shared read port cannot read the instruction while a write occurs. Since the instruction buffers are only updated once per parallel mode, the overhead of port sharing is negligible. The instruction buffer needs to be cleared between two parallel sections to prevent TCUs from using obsolete instructions. A register *valid_addr* is used to keep the last location where a new instruction is written. When the read address is less than or equal to this address, the instruction in the *databusRead* is valid. The size of the instruction buffer is 4KB (1024 instructions) in the FPGA prototype and 2K (512 instructions) in the XMT ASIC prototype.

The instructions buffers in clusters need to be upgraded to instruction caches to eliminate the constraint on the number of instructions in a parallel section. With an instruction cache (I-cache), the broadcasting is only used to load certain number of instructions at the beginning of a parallel section into the I-cache. When an instruction miss occurs, the I-cache will fetch instructions from the shared cache through the interconnection network as if reading data. This mechanism can also take advantage of read-only buffers, but it is still less efficient than broadcasting.

B.3 Thread Control Unit (TCU)

A TCU comprised of a simple pipeline (named as PC - Program Counter) and some dedicated local functional units, like ALU, SHIFT and BRANCH units. The block diagram of the PC is shown in figure B.3. Instructions are dispatched to the proper functional unit and the result is collected by PC after the calculation is done. The prefetch buffer needs to be checked before sending a read operation to the LS unit in the cluster. The response from cache is processed by the prefetch buffer and proper responses are delivered to the PC. The PS module calculates the sum of ps_base, ps_back and ps_value within the cluster.

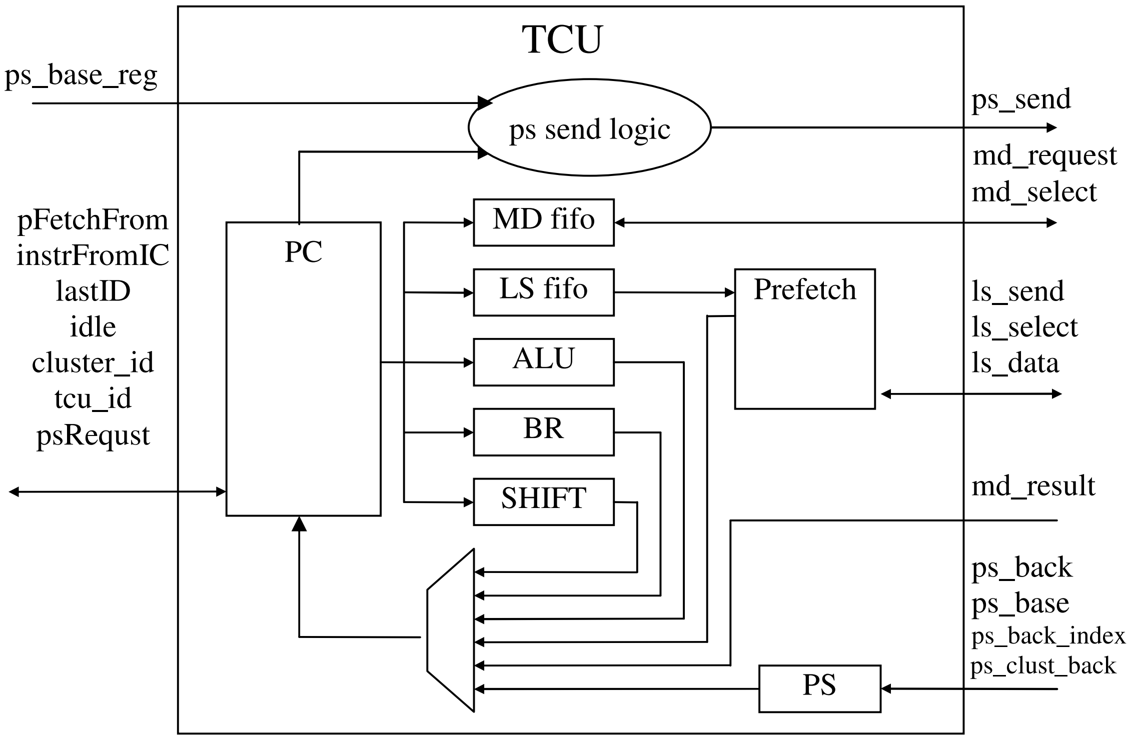


Figure B.3: Block diagram of a Thread Control Unit (TCU)

B.3.1 Interface of the TCUs

Table B.3 on the following page and B.4 on page 126 summarize the input and output ports of a TCU.

B.3.2 Program Counter (PC)

The PC module is a four-stage simple in-order processor core. This module fetches an instruction from the instruction buffer in the cluster and sends it to the

Table B.3: Inputs of the TCUs

Name	Driver	Description
cluster_id	hard-wired	Cluster ID, used by GETID command
tcu_id	hard-wired	TCU ID
instrFromIC	I-buffer	instruction from instruction buffer
lastID	GR7	The highest ID of spawn block, global register 7
ps_base	GRF	The original value of the base register
ps_back_index	PS_Unit	The index of the operation
ps_base_reg	PS_Unit	The base register number and index of the current operation
md_select	FuncMD	The MD request is accepted by the fanin tree of the FuncMD
md_result	FuncMD	The result of MD operation
ls_select	LS unit	The ls request is accepted by fan-in tree for load/store
ls_data	LS unit	The response of the LS request
ps_clust_back	PS_cluster	PS result within a cluster
ps_back	PS_Unit	The result of the PS operation

Connections are shown in figure B.4 on page 127.

proper functional unit. A virtual thread is executed by PC. Figure B.4 on page 127 shows the pipeline stages and the connections of the registers in PC module.

Table B.4: Outputs of the TCUs

Name	Send to	Description
pFetchFrom	I-buffer	Program Counter
idle	Cluster	set if current ID greater than lastID
md_request	FuncMD	MD request
ls_send	LS unit	load/store request
psRequest	PS_Unit	Identify which base register is used
ps_send	PS_Unit	PS operation binary input with valid bit

Connections are shown on figure B.4 on the next page.

The behavior of the pipeline is described below.

IF stage -

The description of the main registers and signals are as follows.

- *PC* [8:0]

  $\Leftarrow$ PC + 1 if previously fetched instruction is valid and no exception.

  $\Leftarrow$ *exception_pc* if exception happened.

- **instruction_pc**[6:0] - the program counter of the fetched instruction.
- **isInstrValid** - wire indicating if fetched instruction is valid (1) or invalid (0).
- **instruction** - wire connected to the read bus of the instruction buffer.
- **pre_pc** - When a fetched instruction is invalid, pFetchFrom will be assigned from this register, not from **pc**.

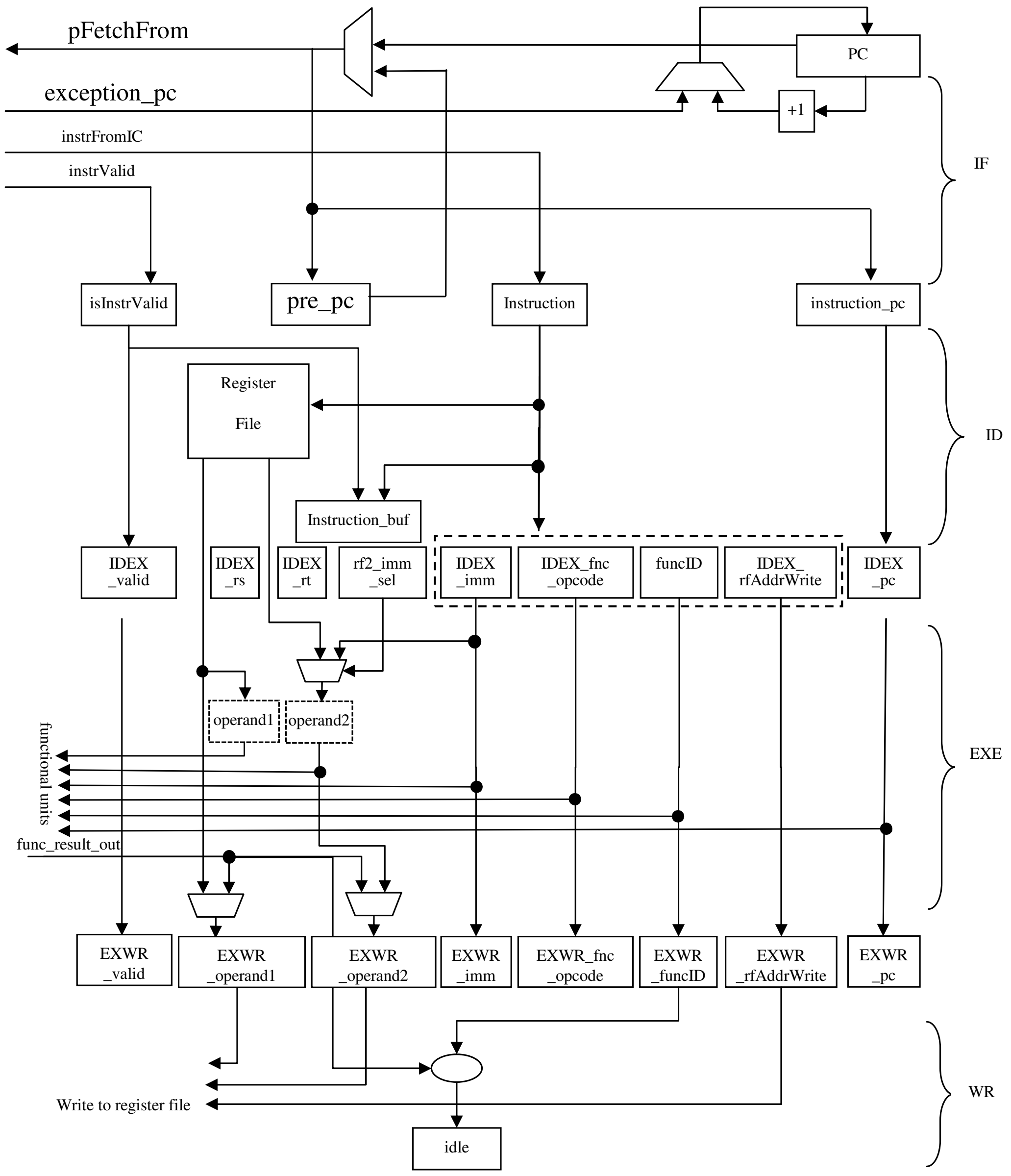


Figure B.4: Structure of the TCUs

ID stage - The ID/EX registers, which are *IDEX_valid, IDEX_pc, IDEX_rs, IDEX_rt, rf2_imm_sel, IDEX_imm, IDEX_fnc_opcode, funcID* and *IDEX_rfAddrWrite*, will be updated when the following conditions are met.

- No exception happened. (If exception occurs, ID/EX registers will be cleared)

AND EX/WR registers will be updated with current ID/EX registers or current *IDEX_valid* is 0 indicating ID/EX registers are invalid.

The description of the main registers and signals are as follows.

- **register file** - 32 32-bit local registers with two read ports and one write port.
- **IDEX_valid** - indicates the status whether the fetched instruction in ID/EX stage is valid. 1 is valid and 0 is invalid.
- **IDEX_pc** - PC of the decoded instruction.
- **IDEX_rs** - the address of register file for operand1.
- **IDEX_rt** - the address of register file for operand2.
- **rf2_imm_sel** - the selection bit whether the operand2 is from immediate value or register file.
- **IDEX_imm** - the immediate value of a instruction.
- **IDEX_fnc_opcode** - the functional opcode of the instruction decoded.
- **funcID**[6:0] - the ID of the functional unit needed for this instruction. Only one bit of **funcID** can be 1 and from bit 6 to 0 indicate move, LOAD/STORE, Prefix-sum, MD,BR, shift and ALU respectively.
- **IDEX_rfAddrWrite** - the destination register of the instruction, if no register will be updated, *IDEX_rfAddrWrite* will be 0.

- **instruction_buf** - the decoded instruction.

EX stage - The EX/WR registers, which are *EXWR_valid*, *EXWR_operand1*, *EXWR_operand2*, *EXWR_imm*, *EXWR_fnc_opcode*, *EXWR_funcID*, *EXWR_rfAddrWrite* and *EXWR_pc*, monitor the *func_result_out* port. If the result is available (*isResultForMe*==1), it is written to the register file. If an instruction in the ID/EX registers is chkid and if the branch is taken, *idle* will be set to 1, otherwise 0. If the instruction is not a *chkid* instruction, *idle* will not change. The description of other registers and signals follows.

- *operand1*[31:0] *wire*
  - ⇐ from *func_result* when forwarding is needed.
  - ⇐ from register file.
- *operand2*[31:0] *wire*
  - ⇐ from *func_result* when forwarding is needed.
  - ⇐ from *lastID* when instruction is chkid.
  - ⇐ from NATURAL_ID when instruction is getid, it is the same as the TCU hardware ID.
  - ⇐ immediate value from instruction, for I-type instructions.
  - ⇐ from instruction for some shifting instructions.
  - ⇐ from register file.
- **EXWR_valid** - valid bit in EXWR stage.
- **EXWR_imm** - the immediate value in EXWR stage.

- **EXWR_pc** - the program counter in EXWR stage.
- **EXWR_fnc_opcode**[3:0] - the operation type of the functional unit in EXWR stage.
- **EXWR_funcID**[6:0] - the functional unit needed for the instruction in EXWR stage. bit definition is the same as **funcID**.
- **EXWR_rfAddrWrite**- the destination register of the current instruction in EXWR stage.

WR stage - This stage has the idle register.

- **idle** - the TCU status. 1 is idle and 0 is busy.

The register file is updated by **rfWrite**, **EXWR_rfAddrWr**, and **rfBusWrite** when the result of an instruction is ready.

B.3.3 Performance Counter Register Files

Each TCU has two Performance Counter Register Files (PCRF). The *instruction count PCRF* is used to count the number of instructions executed by a TCU and the *cycle count PCRF* is counting the number of cycles used for instructions. Both PCRFs keep the accumulative counts. Some instructions share one register in PCRFs and they are shown in table B.5 on the next page. The size of PCRF is 16(depth)x64(width) and 64-bit registers (not 32-bit) are used to count longer programs. When an instruction is executed the instruction count PCRF will increment the corresponding register specified by its group. It is only incremented by one even

Table B.5: PCRF instruction categories

Group	instructions
0	ALU instructions
1	SFT instructions
2	BR instructions
3	mult,multu,div,divu
4	ps
5	lw
6	instructions processed by PC, without using FU
7	mfhi,mflo,mthi,mtlo
8	sw
9	psm
0xa	pref 8
0xb	swp
0xc	pref 9
0xd	lwbuf
0xe	DRAM prefetch
0xf	all other instructions.(used for cycle count PCRF)

if the instruction takes multiple cycles. On the other hand, the cycle count PCRF is incremented by one for every cycle for the corresponding register specified by its group. The reset will not clear the contents of the PCRFs. During serial mode, both PCRFs in TCUs will not be active and values remain unchanged.

In addition to two PCRFs, the MTCU has a 64-bit register that counts the total number of cycles the MTCU spent since last reset. This *cycle count register* is cleared to 0 when XMT is reset.

B.3.3.1 How to use PCRFs?

The instructions, *mvfp* and *mvtp*, are introduced for users to access PCRFs. The formats of these two instructions are listed in table B.9 on page 143.

mvfp rt,rp,imm

- rt is the destination register number in the TCU.
- rp is a register in the TCU and the content of it is used to address the PCRF. This means the PCRF is using indirect addressing. The 16x64 registers are accessed as 32x32 registers, so two mvfp are needed to read a 64-bit value from the PCRF. For example, to read count value for ps operation, the content of rp should be 8 and 9, for low and high (32-bit values), respectively.
- imm can be either 1, 2, 4, or 8. If imm is 1 or 2, the mvfp instruction operates on cycle count PCRF or instruction count PCRF, respectively. If imm is 8, the mvfp is not reading from PCRFs, but disables both PCRFs, meaning the PCRFs stop counting. Note that imm=4 is only valid for MTCU. If imm is 4, mvfp reads the total number of cycles the MTCU spent since the last reset (reset clears this to 0).

mvtp rp,rs,imm

- rs is the source register number in the TCU.
- rp is a register in the TCU and the content of it is used to address the PCRF. This means the PCRF is using indirect addressing. Unlike mvfp, mvtp clears the upper 32-bit to 0 and set lower 32-bit with content of rs.

- imm can be either 1, 2, or 8. If imm is 1 or 2, the mvtp instruction operates on cycle count PCRF or instruction count PCRF, respectively. If imm is 8, the mvtp is not writing to PCRFs, but enables both PCRFs, meaning the PCRFs begin counting. Note that using 4 for imm in a mvtp cannot change the cycle count register in MTCU.

By using these two instructions, these performance counter registers can be dumped into the XMT memory and that can be accessed from the host computer through the external cache access port of the XMT processor. Typical pieces of assembly code for initializing and dumping to the memory are shown below.

Initializing with 0

```
mvfp $0 , $0 , 8           #disable PCRFs
addi $2 , $0 , 32          #number of iterations
addi $1 , $0 , 0           #start from register 0 at PCRFs
clean_pcm :
mvtp $1 , $0 , 1           #clear address $1 in cycle count PCRF
mvtp $1 , $0 , 2           #clear address $1 in instruction count
                           PCRF
addi $1 , $1 , 2           #increase the pointer in PCRFs, each mtpf
                           clears 64-bit register
bne $2 , $1 , clean_pcm    #clear all 16 registers in PCRFs
```

Dumping PCRFs to memory

```
mvfp $0 , $0 , 8           #disable PCRFs
addi $2 , $0 , 32          #number of iterations
addi $1 , $0 , 0           #start from register 0 at PCRFs
read_pcm :
mvfp $3 , $1 , 1           #read from cycle count PCRF
sw $3 , 0 ( $29 )          #store it to memory
mvfp $3 , $1 , 2           #read from instruction count PCRF
sw $3 , 128 ( $29 )
addi $29 , $29 , 4         #update memory pointer
addi $1 , $1 , 1           #increase the pointer in PCRFs
bne $1 , $2 , read_pcm     #iterate
```

B.4 Arbiters

B.4.1 Basic Arbiter

In XMT, arbiters are used to deal with resource contentions. For example, the MD unit is shared by several TCUs or multiple clusters send memory requests to the same on-chip cache module. Arbiters should be fair for every input.

A parameterized arbiter is designed and used for XMT verilog description as a building block. Inputs and outputs are listed in table B.6.

Table B.6: Inputs and outputs of the arbiter

Name	Port type	Width
in0	input	Parameter wire_width
in1	input	Parameter wire_width
in0_select	output	1
in1_select	output	1
out	output	Parameter wire_width
out_select	input	1

Parameters - Wire width of the inputs and outputs are parameterized. The inputs are assumed to have a valid bit indicating whether the request is valid. The position of the valid bit is specified by a parameter valid_bit.

Register - A one bit history register *his* is used to record previous arbitration. *his* is used to choose one of the two inputs if they both have a valid input and *his* will be updated to ∼*his*(negative of previous value), so that next time the arbiter will choose the other input given both inputs are valid.

Arbitration - The behavior of an arbiter is shown in table B.7.

Table B.7: Truth table of arbiter

input			output			
in0	in1	his	in0_select	in1_select	out	next his
valid	valid	0	out_select	0	in0	1
valid	valid	1	0	out_select	in1	0
valid	invalid	X	out_select	0	in0	his
invalid	valid	X	0	out_select	in1	his
invalid	invalid	X	0	0	0	his

B.4.2 Basic Arbiter Tree

A balanced binary tree is used to build arbiters whose number of input ports is larger than two. When arbiters are cascaded, the timing is a concern for implementation, since the signal *out_select* is supposed to be transferred to *in*_select* passing through log_2^N layers. One solution for this problem is to place a buffer between two layers of basic arbiters as shown in figure B.5 on the following page.

To prevent the signal from traveling through multiple layers, data can be registered to buffers only if the destination register is empty. This results in half of the transfer rate at output. If there are two buffers for each node, the transfer rate

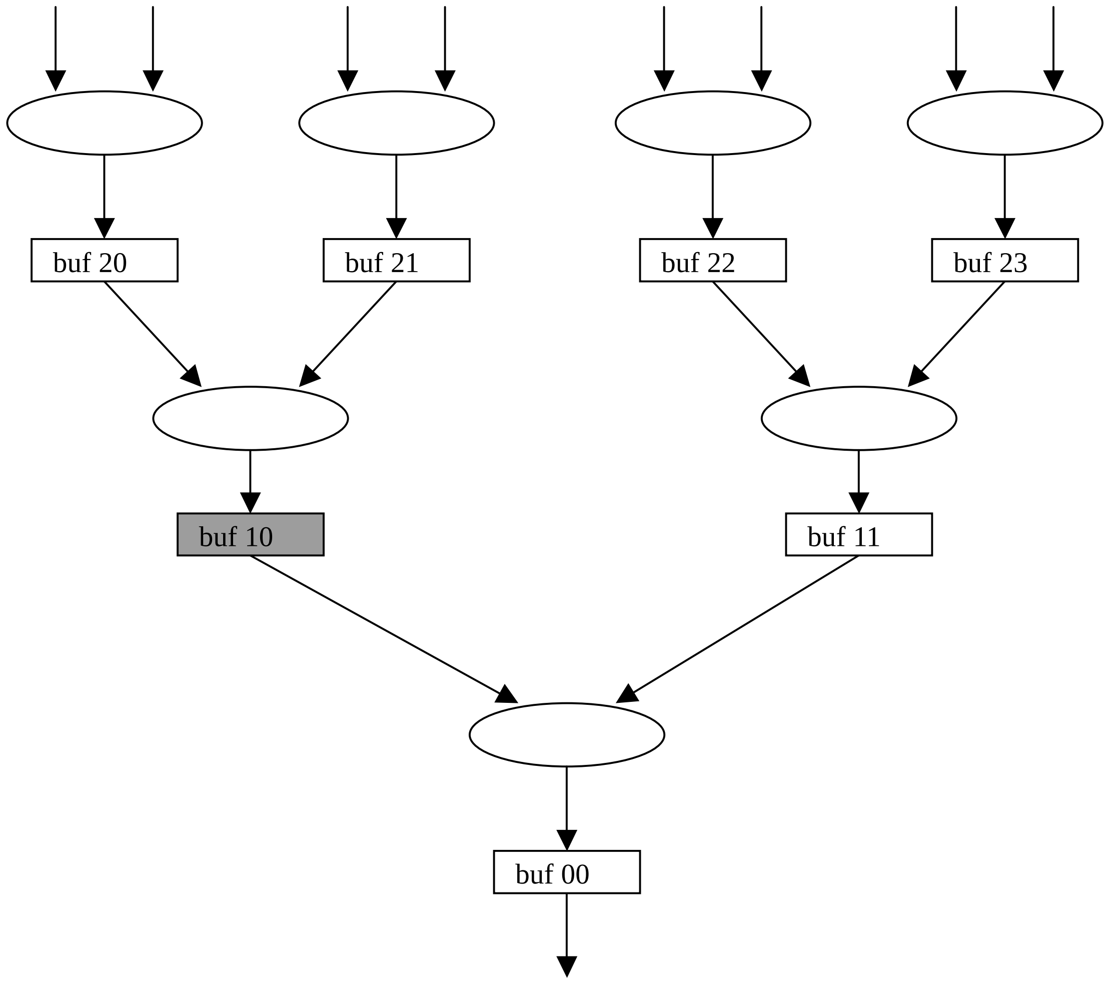


Figure B.5: One buffer arbiter tree

can be one packet per cycle. Figure B.6 on the next page has 8 inputs and 1 output with a two-entry buffer.

Two registers in a node share common input and output ports and only one of two registers is connected to them. When a new valid packet comes and the current input port is empty, the packet will be registered to the input register. When a valid packet is taken from the next level arbiter, the current output register will be

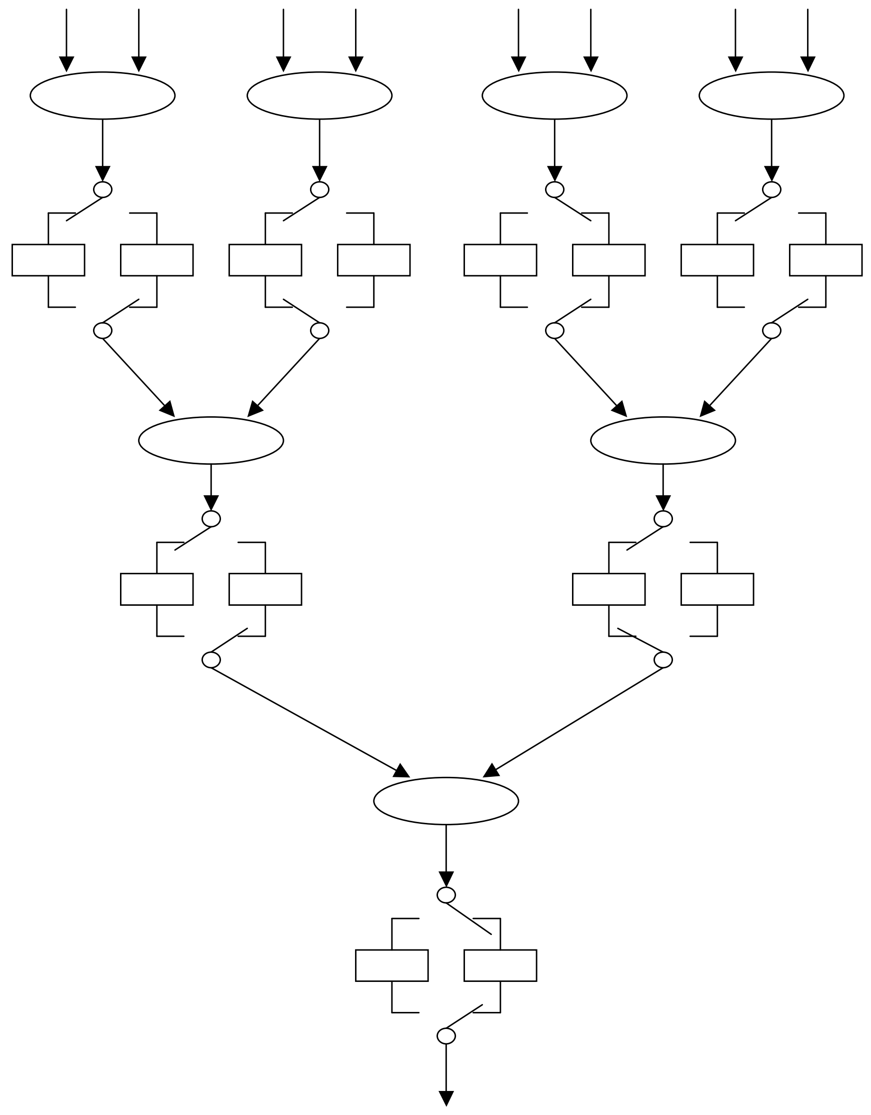
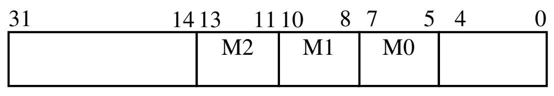
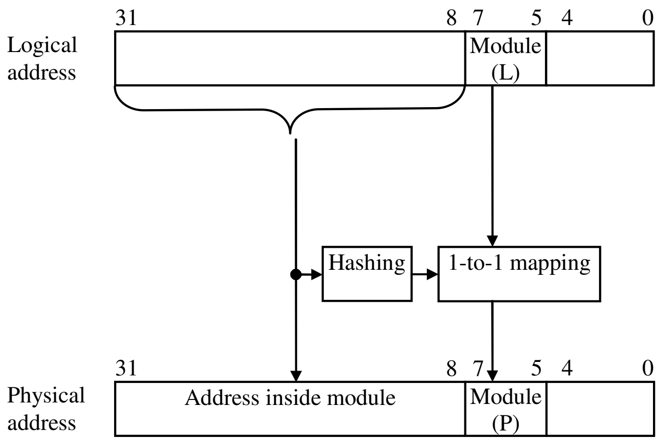


Figure B.6: Two entry buffer arbiter tree

cleared and the output will be connected to the other register. The two-entry buffer is essentially a FIFO (First In, First Out).



(a) address field definition



(b) physical address calculation

Figure B.7: Memory address hashing

B.5 Memory Address Hashing

The memory space is evenly divided among memory modules. Each memory request from a cluster will be dispatched to one of those memory modules. One possible implementation of the allocation of the memory address to memory modules is outlined below. Some bit fields of the address can be used to determine the cache module, such as the Module(L) field consisting of bit 5 through 7, as shown in Figure B.7, but then code exhibiting certain regularities could result in unbalanced accesses to the cache modules. Hashing is used for module indexing in the following manner. The module index (see Module(P) in Figure B.7) is constructed using both the upper address field (bits 8 through 31 and the natural module index (Module(L)). Applying a permutation to the bit field of Module(L) will result in having memory locations that have the same upper address field, but different Module(L) values map to different cache modules. Mixing such a permutation with both XOR and ADD operation on higher bit fields can be used to change the natural ordering for the module index, but each of them can only provide 8 different orders for 3 bit module index field. Let M0 denote Module(L) (bit fields 5 through 7). For example, given a non-zero value M1 (bit fields 8 through 10), M1 XOR M0 will change the original sequence of 0,1,. . ., 7 to a different sequence, and the overall number of potential sequences remains 8. An extra XOR with M2 (bit fields 11 through 13) will still result in one of those 8 possible sequences. An ADD operation also has similar limitations, but when these two are combined, the number of potential sequences become much larger than 8. The proposed hashing can be represented by the formula:

$$Module(P) = ((M_0\ XOR\ M_1) + S)\ XOR\ M_2.$$

Figure B.7 on the preceding page (a) depicts bit fields M0, M1 and M2. S in the formula can be any number from 0 to 7, and 3 was chosen for current implementation. Adding a number to $M_0\ XOR\ M_1$ results in a cyclic shifting of the sequence. Applying XOR M2 to the shifted sequence will generate new sequences. This hashing scheme showed good performance in tentative studies. It needs only a relatively simple hardware implementation: only two 3-bit XOR units and one 3-bit adder. The mapping needs theoretical analysis and a much more elaborate quantitative study. This scheme solves contention from a regular memory access pattern while keeping spatial locality. All logical addresses with identical upper 24 bits will be evenly distributed to the 8 memory modules.

B.6 Functional Units

The output of ALU, BRANCH and SHIFT units has the same format.

	35	34:32	31:0
func_result:	VALID	EXCEPTION	VALUE

The meaning of the exception field in func_result is listed in table B.8.

Table B.8: Definition of the Exceptions

code	Source	Description
000	–	No exception
001	PC	Jump instruction
010	BR	branch instructions
011	BR	checkid instruction
101	Spawn/Join	SPAWN instruction
110	Spawn/Join	JOIN instruction
111	PC	HALT

Some functional units also have exception_pc port, which specifies the destination address for an exception.

Some instructions do not need any functional units for execution, like data movement between two registers. These instructions are handled directly by the PC module and they are listed in table B.9 on the next page.

Table B.9: Instructions handled by PC

instruction	instruction format					opcode in PC
	6 bits	5 bits	5 bits	5 bits	11 bits	4 bits
j target	0x02	target				1
jal target	0x03	target				2
lui rt,imm	0xf	0	rt	imm		3
jr rs	0	rs	0		8	4
jalr rs,rd	0	rs	0	rd	9	5
mvfp rt,rp,imm	0x1c	rp	rt	imm		8
mvtp rp,rs,imm	0x1d	rs	rp	imm		9
sflush	0x2d	don't care				0xa
mflo rd	0	0		rd	0x12	0xc

B.6.1 ALU

Input

- forALU:

68	67:64	63:32	31:0
VALID	OPERATION	OPERAND1	OPERAND2

  OPERATION bits is defined on table B.10 on page 145

Output

- *func_result*
- *exception_pc* This is NOT used currently, but it is reserved for future use.

B.6.1.1 Instructions Executed by ALU

Figure B.10 lists instructions executed by the ALU and the opcode inside the ALU are listed on the right most column. These values are specified in *IDEX_fnc_opcode* after an instruction is decoded.

Table B.10: Instructions executed by ALU

instruction	instruction format					opcode in ALU
	6 bits	5 bits	5 bits	5 bits	11 bits	4 bits
getid rd, rs	0x19	rs	rt	don't care		0
add rd,rs,rt	0x00	rs	rt	rd	0x20	0
addi rt,rs,imm	0x08	rs	rt	imm		0
addu rd,rs,rt	0	rs	rt	rd	0x21	1
addiu rt,rs,imm	0x09	rs	rt	imm		1
sub rd,rs,rt	0	rs	rt	rd	0x22	2
subu rd,rs,rt	0	rs	rt	rd	0x23	3
and rd,rs,rt	0	rs	rt	rd	0x24	4
andi rt,rs,imm	0x0c	rs	rt	imm		4
or rd,rs,rt	0	rs	rt	rd	0x25	5
ori rt,rs,imm	0x0d	rs	rt	imm		5

Table B.10: Instructions executed by ALU

instruction	instruction format					opcode in ALU
	6 bits	5 bits	5 bits	5 bits	11 bits	4 bits
nor rd,rs,rt	0x27	rs	rt	rd	0x27	6
xor rd,rs,rt	0	rs	rt	rd	0x26	9
xori rt,rs,imm	0x0e	rs	rt	imm		9
slt rd,rs,rt	0	rs	rt	rd	0x24	0x0a
slti rt,rs,imm	0x0a	rs	rt	imm		0x0a
sltu rd,rs,rt	0	rs	rt	rd	0x2b	0x0b
sltiu rt,rs,imm	0x0b	rs	rt	imm		0x0b

ALU operations take one cycle.

B.6.2 Shift Functional Unit(SFT)

Input

- forSFT:

39	38:37	36:5	4:0
VALID	OPERATION	OPERAND1	OPERAND2

OPERATION bits is defined on table B.11

Output

- *func_result*

B.6.2.1 Instructions Executed by SFT

Table B.11 lists instructions executed by the shift functional unit.

Table B.11: Instructions executed by SFT

instruction	instruction format						opcode
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	2 bits
sll rd,rt,shamt	0	0	rt	rd	shamt	0	1X[1]
srl rd,rt,shamt	0	0	rt	rd	shamt	2	00
sra rd,rt,shamt	0	0	rt	rd	shamt	3	01
sllv rd,rt,rs	0	rs	rt	rd	0	4	1X[1]
srlv rd,rt,rs	0	rs	rt	rd	0	6	00
srav rd,rt,rs	0	rs	rt	rd	0	7	01

Shifting operations take 2 cycles and there is no exception. The FPGA prototype needs 2 cycles to meet timing constraints. In an ASIC implementation, a 2 cycle delay may not be necessary.

[1]X means don't care

B.6.3 Branch Functional Unit(BR)

Input

- forBR:

96	95:80	79:68	67:64	63:32	31:0
VALID	OFFSET	PC	OPERATION	OPERAND1	OPERAND2

OPERATION bits is defined on table B.12 on the following page

Output

- *func_result*
- *exception_pc*

B.6.3.1 Instructions Executed by BR

Table B.12 on the next page lists instructions executed by the branch functional unit.

chkid is an instruction introduced for implementing the *spawn* instruction. Each TCU needs to compare it's own ID($) with the high ID(GR7) and if $ is less than GR7, the TCU will advance to the next instruction, otherwise *chkid* will be executed again (jumping to itself). Since the high ID (GR7) is broadcast to the TCUs all the time, the *chkid* can catch any increase of GR7 from the *sspawn* instruction.

Table B.12: Instructions executed by BR

instruction	instruction format						opcode in BR
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	3 bits
beq rs,rt,label	0x04	rs	rt	offset			0
bne rs,rt,label	0x05	rs	rt	offset			1
chkid rs	0x18	0	0xff				2
blez rs,label	0x06	rs	0	offset			6
bgtz rs,label	0x07	rs	0	offset			7
bltz rs,label	0x01	rs	0	offset			8
bltzal rs,label	0x01	rs	0x10	offset			8
bgez rs,label	0x01	rs	1	offset			9
bgezal rs,label	0x01	rs	0x11	offset			9

B.6.3.2 Behavioral Description of BR

Operations in the branch unit take two cycles.

B.6.4 Multiplication/Division Functional Unit(MD)

Input

- forMD:

71	70:67	66:64	63:32	31:0
VALID	TCU_ID	OPERATION	OPERAND1	OPERAND2

OPERATION bits will be defined on table B.13 on the following page

Output

- *func_result*

B.6.4.1 Instructions Executed by MD

Table B.13 lists instructions executed by the multiplication/division (MD) functional unit.

Table B.13: Instructions executed by MD

instruction	instruction format						opcode in MD
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	4 bits
mult rs,rt	0	rs	rt	0		0x18	0
multu rs,rt	0	rs	rt	0		0x19	1
div rs,rt	0	rs	rt	0		0x1a	2
divu rs,rt	0	rs	rt	0		0x1b	3
mfhi rd	0	0		rd	0	0x10	4
mthi rs	0	rs	0			0x11	5
mtlo rs	0	rs	0			0x13	7

B.6.4.2 Behavioral Description of MD

In the MIPS ISA, there are two special registers, HI and LO, for multiplication and division operations. The result of these two operations are stored in HI and LO. For multiplication, the upper and lower 32 bits of the product are stored in HI and LO, respectively. In division, HI is for the remainder and LO register is for the quotient. To match the MIPS ISA, a pair of HI and LO registers are instantiated per TCU in the MD unit. The MD unit stores the result in the corresponding HI and LO for each TCU. The MD unit is fully pipelined and it can serve one request per cycle. The latency of multiplication and division in FPGA prototype is 6 cycles and 36 cycles, respectively. The latency of multiplication and division in the ASIC implementation is 6 cycles and 11 cycles, respectively.

mult or *div* instructions only store the results in the HI and LO registers within the shared MD unit. A *mfhi* or *mflo* is needed to move the results to the register file in a TCU. To reduce the overall latency of a multiplication or division operation, the LO register is replicated on the TCU side, so the *mflo* can get the value of the LO register locally. Both LOs in the MD unit and in the TCU are updated at the same time for any instruction that changes the LO register.

B.6.5 Prefix-sum

The prefix-sum of 16 TCUs is calculated inside clusters and the sum of ps requests is sent to the PS unit to get the global prefix-sum results. Prefix-sum operations are described in appendix C on page 162.

Table B.14: Instruction executed by PS_unit

instruction	instruction format					
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
ps rt, gr	0x16	gr	rt	0		

B.6.6 Load/Store

Each cluster has one Load/store port and the requests may be queued at the port. Currently there are four layers of arbiters with two-entry buffers to solve the contention from 16 TCUs. Detailed information about arbiters can be found in section B.4.2 on page 136. Table B.15 listed different types of memory access instructions. *lw* and *sw* are load and store instructions and they match the definition from the MIPS ISA. The four instructions: *lw*, *lwbuf*, *pref 8* and *pref 9* are explained in Table B.18. *swp* is explained in section B.7.4. The *psm* instruction is similar to ps but the base is a memory location and the increment value can be any unsigned integer. The following two operations happen atomically for a *psm* instruction.

memory[imm(rs)] $\Leftarrow$ memory[imm(rs)] + rt

rt $\Leftarrow$ memory[imm(rs)]

The register rt will get the original value in memory[imm(rs)].

pref 10 is a DRAM prefetch instruction that asks a cache module to bring in a cache line from the off-chip DRAM. The cache module will not send any acknowledgement to the TCUs for *pref 10*.

Table B.15: Instructions executed by LS unit

instruction	instruction format						opcode in LS
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	4 bits
lw rt, imm(rs)	0x23	rs	rt	imm			0
sw rt, imm(rs)	0x2b	rs	rt	imm			1
lwbuf rt, imm(rs)	0x27	rs	rt	imm			2
psm rt, imm(rs)	0x17	rs	rt	imm			3
pref 8,imm(rs)	0x33	rs	8	imm			4
swp rt, imm(rs)	0x2c	rs	rt	imm			5
pref 9,imm(rs)	0x33	rs	9	imm			6
pref 10,imm(rs)	0x33	rs	0xa	imm			0xe

B.7 Four Enhancements

B.7.1 Broadcasting

Value broadcasting is used when all or most of the threads read the same variable. The shared variable will be read by the MTCU and the value will be broadcast through the instruction broadcasting bus. Figure B.8 (copy of Figure 2.6) shows how it works. (a) shows when there is no broadcasting mechanism, all TCUs will send a *lw* request to the same memory location and result in more traffic in the interconnection network, as well as a long queue in the shared L1 cache module. (b) shows what happens when a broadcasting mechanism is used.

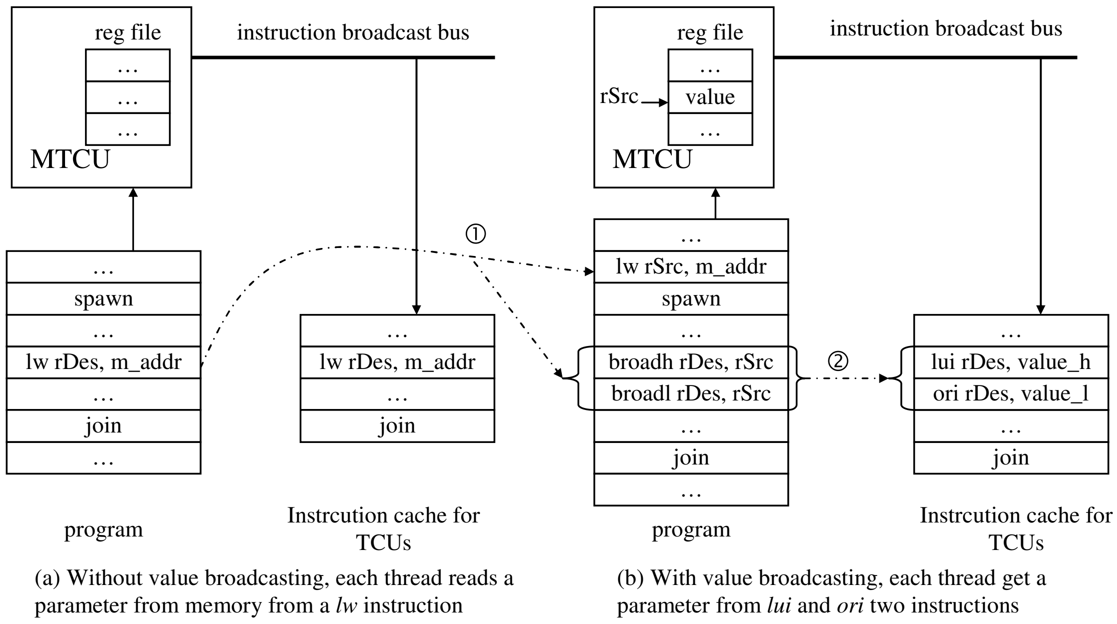

reg file
instruction broadcast bus
MTCU
...
spawn
lw rDes, m_addr
join
①
lw rDes, m_addr
join
program
Instrcution cache for TCUs
(a) Without value broadcasting, each thread reads a parameter from memory from a *lw* instruction
reg file
instruction broadcast bus
rSrc
value
MTCU
lw rSrc, m_addr
spawn
broadh rDes, rSrc
broadl rDes, rSrc
join
②
lui rDes, value_h
ori rDes, value_l
join
program
Instrcution cache for TCUs
(b) With value broadcasting, each thread get a parameter from *lui* and *ori* two instructions

Figure B.8: Value broadcasting

The broadcasting requires support from both hardware and software(compiler).

Software

- A *lw* instruction is inserted before the spawn instruction. The value will be stored in a register in the MTCU, rSrc.
- Replace the original *lw* instruction with two instruction: *broadh* and *broadl*. Note that the *broadh* instruction should be always placed before *broadl*, since the *broadh* instruction will clear the lower 16 bits to 0.

Hardware When instructions are broadcast at the beginning of a parallel section, normal instructions are broadcast without any change, but these two instructions, *broadh* and *broadl*, are handled differently. *broadh* is replaced by *lui* and the immediate field is filled with the upper 16 bits of value in the MTCU register rSrc. *broadl* is replaced by *ori* and the immediate field is filled with the lower 16 bits of value in the MTCU register rSrc.

After broadcasting is used, the TCUs get the value from two instructions, *lui* and *ori*, instead of reading it from memory. Since the broadcasting uses a register in MTCU, the number of values that can be broadcast is limited.

B.7.2 Prefetch Buffer

Each TCU has four prefetch buffers and they provide support for software prefetch. Note that hardware does not guarantee the coherence of these prefetched values.

In each prefetch buffer, there is an address and data field. Depending on the status of the address and data field, The prefetch buffer has 3 states:

Invalid The prefetch buffer is not used at all. Both the data and address are invalid.

Pending A prefetch command is sent out, but the data is not received yet. The address field is valid, but the data field is not valid.

Valid The prefetch buffer has valid data that can be used by the TCU immediately. Both address and data are valid.

The last case, invalid address and valid data, is a non-reachable state. When a TCU sends a prefetch command, one of the 4 prefetch buffers will be used, chosen in a round-robin fashion. If the particular buffer is in the pending state, the prefetch command has to wait until the response comes back, otherwise the prefetch buffer will be confused by the two possible responses. For the other two cases, the prefetch command will be sent out immediately.

Both normal read and write for a prefetched memory location will invalidate the prefetch buffer. For the read, if the buffer is in the pending status, it will wait until the response is returned. For the write, the buffer is marked as "write invalidate" and the write request will be sent out. When the response comes back, the buffer enters the state of *invalid*. When the XMT processor enters serial mode, the prefetch buffers are cleared and all entries enter the *invalid* state.

B.7.3 Read-Only Buffer

Each cluster has an 8KB read-only buffer (ROB), which is a hardware/software co-managed storage. Unlike regular cache, ROB is supposed to store values that will not be changed during a particular parallel section to avoid the cache coherence problem. The other difference is that there is no cache line and each word has it's own tag. If the data can be safely stored in the ROB, the compiler should use *lwbuf* instruction for a read operation and *pref 9, addr* for prefetch command. These two instructions are referred to as ROB commands and other read commands are referred to as non-ROB read commands.

The ROB uses direct map, so when a ROB command or any read command arrives at ROB, only one entry of the ROB needs to be checked. When a ROB command turns out to be a miss, then the request will be forwarded to the shared cache. It will take time to get the response back from the shared cache through the interconnection network, and it is desirable to keep the old tag and data pair if they are valid, because other TCUs may be able to use the old data. Therefore, each ROB entry has an extra address field where the address of the pending request is kept.

The behavior of the ROB is simple for the non-ROB read commands.

Hit If the current address of the ROB matches the request and ROB has valid data, then ROB can serve the request locally.

Miss If the address of both the current and pending addresses do not match the request or they are invalid, then the request is forwarded to the shared cache through the interconnection network.

Pending If the pending address of the ROB matches, a request for the same location has been sent out and the data will be available later. Therefore the TCU ID will be recorded and as soon as the data is available, the response will be delivered to the TCU.

Table B.16 summarize the behavior of ROB for *lw* and *pref 8, addr*.

For a ROB command , the response of the ROB is different. The response is described below and summarized in the table B.17.

Hit If the current address of the ROB matches the request and ROB has valid data for the location, then ROB can serve the request locally.

Table B.16: Response of ROB for *lw* or *pref 8*

current entry in ROB	pending entry in ROB	Response
hit	-	NOT send request to shared cache, send result to TCU
miss	invalid or mismatch	send request to shared cache
	valid and match	NOT send request to shared cache, add the TCU ID to the waiting list

Miss When the current address of the ROB entry is not the same as the request, the pending address field needs to be checked. When the pending address field is **empty**, **mismatch** or **match**, the response of ROB is:

Empty The address of the request is stored in the pending address field and the request is sent to the shared cache. When the response comes back the data and address are updated to the current address and data field.

Mismatch If the pending address is not empty but it is different from the request, the request will be replaced with a corresponding non-ROB command and sent to the shared cache.

Match If the pending address is the same as the request, the TCU ID of the request is recorded so that when the response arrives from the shared cache, the result will be delivered to the TCU.

The ROB is cleared when the XMT processor enters serial mode. Since each entry of the ROB take one cycle to clear, the number of cycles needed for clearing the ROB is the same as the size of the ROB. For the size of 8KB, it is 2048 cycles. This procedure starts immediately after the XMT processor switches to serial mode, and will continue in the following parallel mode, if the serial mode is not long enough. When the clearing continues into the next parallel mode, all requests will bypass the ROB. This only affects the performance of a program with a short serial section followed by a short parallel section that uses ROB commands. A potential solution for this issue is using two (or more) sets of valid bit arrays and alternate between them, because this effectively reduces the cycles needed for clearing.

When a TCU ID is added to a waiting list, 6-bit information needs to be stored and therefore, a total of 96 bits are needed for 16 TCUs in a cluster. Instantiating a waiting list for every entry of ROB is a huge overhead and this is avoided by using a pointer. The size of the waiting list is limited by the number of TCUs and maximum number of pending requests from a TCU. In the current design, there are 16 TCUs in a cluster and each TCU can have a maximum of 16 pending requests. Therefore the size of the waiting list can be 256, which is much less than the number of entries in the ROB, 2048. Instead of having a waiting list for every entry, each entry has a pointer to the waiting list array as shown in figure B.9. By fixing the size of the waiting list to 256, the overhead of the ROB is reduced significantly.

Until now, we presented how the prefetch and read-only buffers are implemented in hardware. Table B.18 summarizes four different read instructions available in XMT.

Table B.17: Response of ROB for lwbuf or pref 9

current entry in ROB	pending entry in ROB	Response
hit	-	NOT send request to the cache, send result to TCU
miss	invalid (empty)	send request to shared cache, record the address in pending address field
	valid but mismatch	send non-ROB counterpart of the request to shared cache
	valid and match	NOT send request to shared cache, add the TCU ID to the waiting list

Table B.18: Four different memory read instructions

instruction	destination	update ROB
lw	register	no
lwbuf	register	yes
pref 8	prefetch buffer	no
pref 9	prefetch buffer	yes

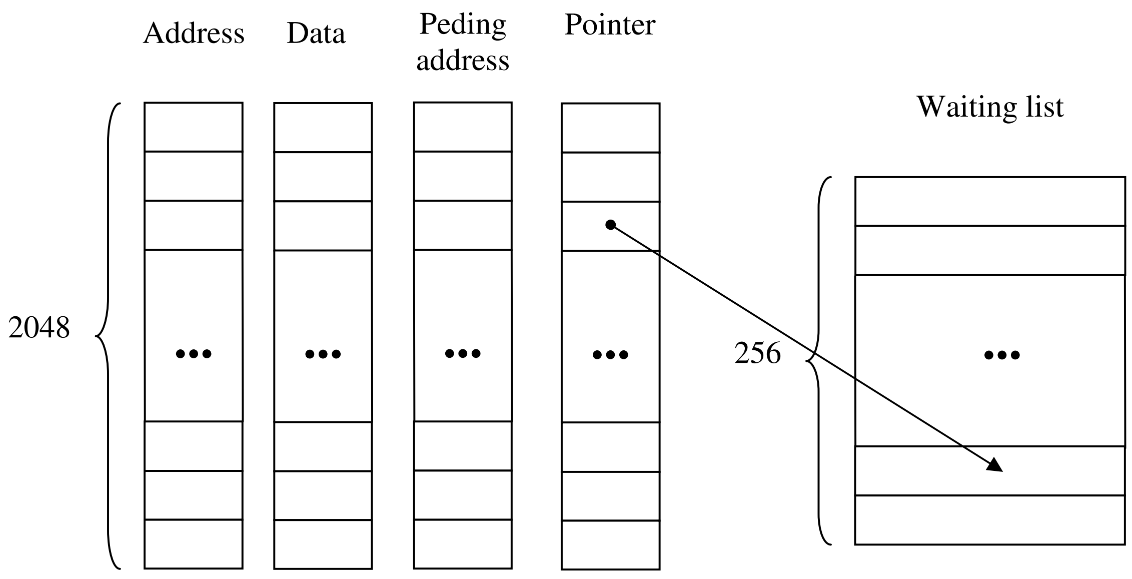


Figure B.9: Using pointer in ROB for waiting list

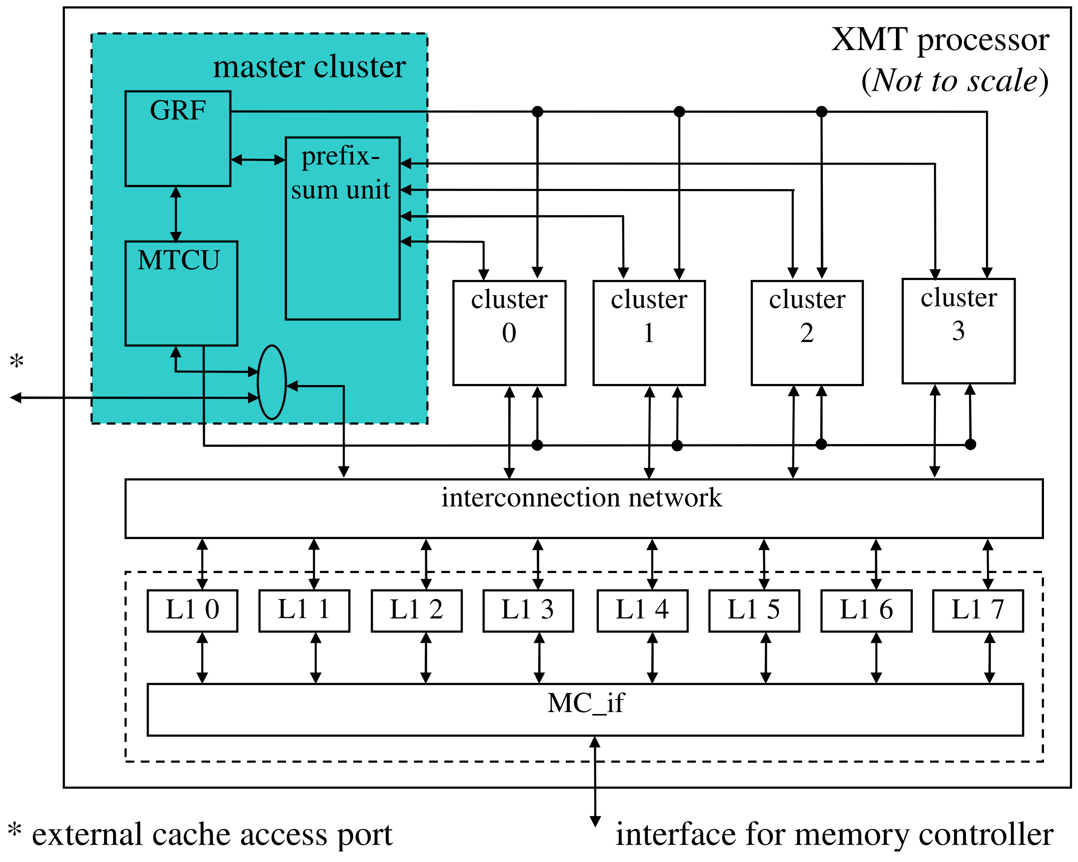
B.7.4 Non-Blocking Store

There are two kinds of stores in XMT: (i) blocking store, *sw*, where a TCU waits until the store operation is acknowledged by the shared cache, and (ii) non-blocking store, *swp*, where a TCU will advance as soon as the store requests are accepted by LS unit in the cluster. The behavior of the parallel cache is the same for the two store instructions, acknowledgement will be sent back to the TCU for both store commands. The compiler should choose the non-blocking store whenever it is safe to use.

Each TCU has two counters to store the number of non-blocking stores sent out and the number of non-blocking store acknowledgements received. When the XMT processor changes to serial mode, the TCU will make sure all non-blocking store instructions have been committed to the shared cache by making sure the two counters are the same. A *sflush* instruction blocks the TCU from advancing to the next instruction until the two counters in the TCU have the same value. Essentially, a blocking store is the same as a non-blocking stored followed by a *sflush* instruction.

Appendix C

Master Cluster



The master cluster includes a global register file (GRF) and a prefix-sum (PS) unit in addition to the MTCU. The MTCU is a serial processor with the capability of handling XMT-specific instructions. The MTCU has an integer ALU, a shifting unit, a branch unit, a multiplication/division unit, a spawn/join functional unit, local private cache and a load/store port.

Some instructions are only for the Master TCU and they are not supported by regular TCUs. Table C.1 lists instructions executed by the MTCU only.

Table C.1: Instructions executed by MTCU only

instruction	instruction format					
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
spawn	0x14	0				
join	0x15	0				
broadh rt, rs	0x34	rs	rt	0		
broadl rt, rs	0x35	rs	rt	0		
mvfg rt, gr	0x1a	gr	rt	0		
mvtg gr, rs	0x1b	rs	gr	0		
halt	0x3f	don't care				

Some instructions are supported in regular TCU, but they are not supported in MTCU. They are:

- pref
- ps rt, gr
- lwbuf rt, imm(rs)

C.1 Master TCU

The interface of the Master TCU is listed below, which is slightly different from the regular TCUs.

Table C.2: Inputs of the MTCU

Name	Driver	Description
ls_back	ICN	response from cache module
ls_select	ICN	acknowledgement from interconnection network
spawnDone	Clusters	set 1, when all TCUs idle
grBusRead	GR	read bus of global register file

Table C.3: Outputs of the MTCU

Name	Send to	Description
grAddrRead	GR	read address port of GR
grAddrWrite	GR	write address port of GR
grBusWrite	GR	value to be written into the global register file
grwrite	GR	write enable signal
ls_send	ICN	LS request to the cache
instrBroadCast	Clusters	instruction broadcasting bus
pNextInSpawn	Clusters	location of the broadcasted instruction
parallel	Clusters	set to 1 during parallel section
spawnblock	Cluster	1 during instruction broadcasting
stop	System	set to 1, when halt command is executed

Compared to regular TCUs, the master TCU has a few special registers and extra interfaces for global register operations.

Additional registers

- *spawnblock* set to 1 during master TCU broadcasts instructions
- *parallel* set to 1 during regular TCUs are active. *spawnblock* and *parallel* are set to 1 in the same cycle, but *parallel* will not be cleared until the execution of spawn block finishes, while *spawnblock* will be cleared as soon as master TCU finishes instruction broadcasting.

Interface for global registers

- Master TCU can read and write global registers by using move instructions.
- *spawn* instruction only changes the execution mode to the parallel, the two registers: GR_high and GR_low need to be initialized by using the *mvtg* instruction.

When the XMT processor exits from reset, the MTCU will start from the instruction in address 0. The MTCU has a 16KB instruction buffer, which will be loaded with the contents of the memory address from 0 to 16KB. The MTCU starts executing instructions as soon as they are copied into the instruction buffer from the memory space. After instructions are saved in the instruction buffer, any consequent changes in the memory will not affect the instructions in the buffer.

C.1.1 Spawn/join Functional Unit(SJ)

Unlike TCUs in the clusters, all functional units for the MTCU are dedicated. The MTCU does not have a PS_unit, but it has a SJ unit which a normal cluster does not have. The SJ_unit handles the spawn/join instruction.

Input

- forSJ:

14	13:2	1:0
VALID	PC	OPCODE

- *spawnDone* 1 bit, 1 – means finished spawn block, 0 – means not finished. Checked when the XMT processor switches to serial mode.
- *mcacheidle* 1 bit, 1 – means mcache is idling. Checked when XMT processor switches to parallel mode.

Output

- *func_result*
- *exception_pc*

C.1.1.1 Behavioral Description of SJ

Spawn and join instructions take 1 cycle and the behavior is summarized in table C.4. The *spawn* and *join* instructions act like a branch instruction based on spawnDone and mcacheIdle respectively. These two instructions also update a special register *spawnblock*, in the Master TCU with the VALUE in table C.4.

Table C.4: Operation of SJ

instr.	Input			Output		
	OPCODE	*spawnDone*	*mcacheidle*	EXCEP.	VALUE	*exception_pc*
spawn	1	X	1	0	1	X
	1	X	0	5	1	self
join	0	0	X	6	0	self
	0	1	X	0	0	X

C.1.2 Local private cache for MTCU

- Size of 8KB
- 32-byte cache lines (eight 32-bit words per line)
- 2 cycles access time
- Direct-Mapped
- Write-through
- No write-allocate
- blocked on a read cache miss.

The Master TCU has a local private cache. The shared caches act as a level two cache for the *mcache*, as all cache misses in the Master TCU's cache (*mcache*) must go through the interconnection network and bring in the missed cache line from parallel caches. Thus, the MTCU has two levels of on-chip cache: its own unified cache and the shared caches.

When switching from serial to parallel mode, the entire tag array of the *mcache* is marked invalid. Whenever the *mcache* is modified during serial mode, the shared cache modules are also updated using a write-through policy, to minimize the overhead of a transition from serial mode to parallel mode.

The implementation of the *mcache* is much simpler compared to the shared cache modules. The *mcache* has two SRAMs: tag and data. When a request comes from the master program counter (MPC), both tag and data arrays are read using index field of the request. The behavior of the cache depends on whether the access is cache hit or miss, as well as the request type (read or write), which are shown in table C.5.

When a read request from MPC is a cache miss, the cache line will be brought in from the shared cache module. The shared cache modules expect a per-word operation, therefore, the *mcache* will send eight requests for the cache line with the critical word first. After the first word arrives from the shared cache module, it will be forwarded to the MPC immediately, but *mcache* needs another seven cycles to update the rest of seven words. During this time, the *mcache* will not accept any new request from MPC.

Table C.5: Behavior of MCache

hit	read	the value in the data SRAM will be sent to the MPC next cycle.
miss	read	mark *mcache* as busy and send the requests to the shared cache module and wait.
hit	write	the value will be written into data SRAM next cycle and an acknowledge will be sent to MPC.
miss	write	the request will be forwarded to the ICN and an acknowledgement will be sent to MPC. No missed cache line is brought in (no write allocate)

C.1.3 Instruction Broadcasting

Following the execution of a spawn instruction, the MPC starts broadcasting the instructions between spawn and join instructions. Most instructions are broadcast as is, but there are three instructions that are changed during broadcasting. They are *broadh*, *broadl* and *join*. The transformation of *broadh* and *broadl* is described in B.7.1 on page 152.

The *join* instruction in TCUs is a jump instruction that jumps to the *chkid* instruction in the spawn-join section. When instructions are broadcast, they are monitored by the MTCU. If a broadcasted instruction is a *chkid* instruction, then the program counter of the *chkid* is stored in a register. When a join instruction is fetched by MTCU, it will be replaced with a *j* instruction that jumps to the *chkid* using the program counter stored in the register.

C.2 Global Register File (GRF)

- Global register file has one read port and one write port.
- There are 8 global registers, GR0 $\sim$ GR7
- Master TCU can access all global registers by using special transfer instructions.
- In parallel mode, from GR0 to GR7 are used for base register of prefix -sum operation.
- GR6 stores the low ID of the spawn block. If a regular TCU finishes a virtual thread, it will perform a prefix-sum operation with GR6 as the base register and get a new ID.
- GR7 stores the high ID of the spawn block. If a regular TCU executes a prefix-sum on this global register, it will increase the high ID and therefore generate a new virtual thread.

C.2.1 Interface of the GRF

Basically, the global register file has one read and one write port. Write ports have a one bit write enable and the content of the grBusWrite will be registered to the global register addressed by grAddrWrite only if the one-bit write enable is high.

C.2.2 Serial Mode

Global registers can be accessed by the master TCU using transfer instructions. The *mvtg* and *mvfg* instructions are for MTCU use only. The "*mvtg gr, rs*" instruction moves the data from *rs* in the MTCU register file to the global register *gr*. The "*mvfg rt, gr*" instruction moves the data from the global register *gr* to *rt* in MTCU register file.

C.2.3 Parallel Mode

If a regular TCU finishes a virtual thread, it will perform a prefix-sum operation on GR6 and get the next virtual ID, and, if it is less than GR7, the TCU executes the new virtual thread; otherwise, it will flag itself as idling. A *sspawn* instruction executed by a regular TCU will increment GR7 and this is implemented by a prefix-sum operation on GR7 with the incremental value of 1. Because of *sspawn*, a virtual thread may become active anytime, so idling TCUs still need to keep checking the last ID(GR7).

C.3 Prefix-sum Unit

A Prefix sum unit executes multi-operand prefix-sum operations from different TCUs in a constant time.

- Binary prefix-sum(input: 0 or 1)
- Compute the base-zero prefix-sum

- Prefix sum of the TCUs inside a cluster is calculated locally
- Sum of binary input of the TCUs inside a cluster is used for a global prefix-sum operation
- Serve different base registers in a round-robin fashion

The prefix sum calculation has two steps: in the first step, the prefix-sum of the TCUs within the clusters is calculated; in step two, the sum is sent to the global prefix-sum unit. After the global prefix-sum results are sent back to the clusters, each TCUs will calculate their own prefix-sum value. In this section, we will first look at the interface between clusters and PS unit, then the implementation inside the cluster, and finally the PS unit itself.

C.3.1 Interface between Cluster and PS Unit

Figure C.1 on the following page shows the connections between clusters and the PS unit. *ps_base_reg* specifies the base register number for the current cycle along with a 6 -bit number which is a time stamp that is used to identify the result. *ps_send* is the sum of the binary inputs from TCUs in a cluster. The PS unit will send the calculation results to the clusters through *ps_back* and *ps_base*. *ps_back* is a unique value for each cluster and *ps_base* is shared for all clusters as it is the original value of the base register.

psRequest has 8 bits information, each bit representing a global register. Each bit is set to 1 if there is a request from a TCU for the corresponding global register. In the worst case, the prefix-sum operation may take 2 round trips to the PS unit.

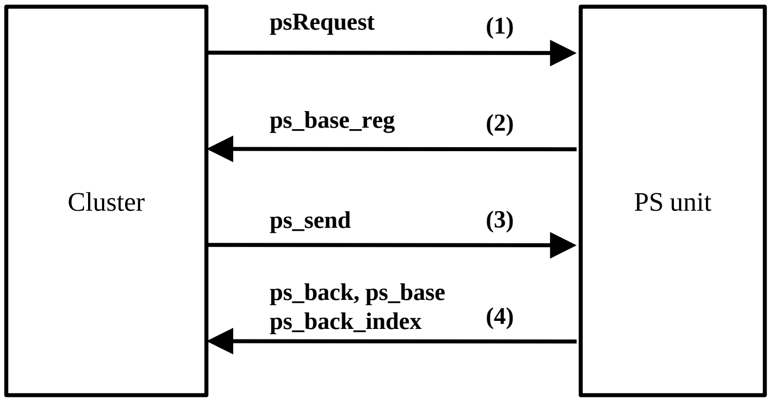


Figure C.1: Interface between clusters and PS unit

The first notifies the PS unit about the base register, and the second sends the actual request when the PS unit serves that particular base register.

C.3.2 Interface of PS Unit

PS requests from TCUs are combined in each cluster before sending to the PS unit and the PS unit sends the results to each cluster.

Table C.6: Inputs of the PS unit

Name	Bit width	Driver	Description
psRequest	8	clusters	specify the base register of ps
ps_send_vec	24	clusters	requests of ps operations
Z0	32	GR	initial value of base register

Table C.7: Outputs of the PS unit

Name	Bit width	Send to	Description
ps_base_reg	10	clusters	base register of accepting ps operation, time stamp
GR_base_read	4	GRF	base register of reading ps operation
GR_base_write	4	GRF	base register of writing ps operation
grwrite	1	GRF	write enable signals for GRF
Zn	32	GR	new value for global register
W	32	Clusters	original value in base register
ps_back_out	48	clusters	Zero-base prefix-sum

C.3.3 Base Register Selection

Considering the cost of hardware, the XMT processor cannot have one PS unit per global register. The PS unit must be used for multiple base registers and give the same priority for every global register. A simple and efficient policy for choosing base register is round-robin.

Bit 4 of the *ps_base_reg* specifies that the global register will be defined as a *global_id*. If a request from a TCU is using the register that the *global_id* specifies, then the request will be accepted. If a request from a TCU is using a different register than the *global_id* specified, the TCU will keep the request until the *global_id* matches. The PS unit will change *global_id* only if there is a prefix-sum operation request for another *global_id*.

Every cycle, the PS unit will check to see if the current global register (specified by *global_id*) is requested (specified by psRequest). If it is not, then the PS unit will check if any other global register is requested in round-robin order, starting from the current *global_id*. Although actual hardware will evaluate this in parallel during one cycle, it is easy to understand by describing it step by step.

1. Make a 16-bit (twice of psRequest) wire, twoPsRequest, by repeating psRequest twice.

2. if (*global_id*+1)th bit of the twoPsRequest is 1, increase global_id by 1, otherwise, next step

3. if (*global_id*+2)th bit of the twoPsRequest is 1, increase global_id by 2, otherwise, next step

4. ...

5. if (*global_id*+7)th bit of the twoPsRequest, increase global_id by 7, otherwise, keep *global_id*.

This policy has the following properties:

Fair: *global_id* moves towards one direction(larger), this means there is no starvation.

Efficient: *global_id* will change only if there are other requests, this guarantees global_id will never change to a new global register on which there is no request.

C.3.4 Zero-base Prefix-sum Computation

Figure C.2 on the next page shows a PS unit with 8 binary inputs. This module is used for both prefix-sum operation of TCUs inside clusters and global ps_unit. The result of a zero-base prefix-sum is the prefix-sum result when the original value of the base register is zero (0).

- First stage of the PS is computing $\frac{N}{2}$ prefix-sum of two inputs. N inputs are grouped to $\frac{N}{2}$ sets, $S_i = \{input(x)|x = 2i + j, j = 0, 1\}, i = 0 \cdots \frac{N}{2} - 1$. In each set, the prefix- sum of two elements are computed and save the result on registers. In figure C.2 on the following page, the prefix-sum of input 0 and input 1 is calculated as well as the other 3 set of prefix-sum.

- Second stage is computing $\frac{N}{4}$ prefix-sum of four inputs. N inputs are grouped to $\frac{N}{4}$ sets, $S_i = \{input(x)|x = 4i + j, j = 0, \cdots, 3\}, i = 0 \cdots \frac{N}{4} - 1$. In each set, the prefix-sum of four elements needs to be calculated. The first half is ready from the previous stage and only second half needs to be computed. Computation of the second half can be done by adding the sum of the first half that is provided by the previous stage. In figure C.2 on the next page, 8 inputs are divided to 2 sets, this step calculates the prefix-sum of each set, input 0,1,2 and 3 are the first set and input 4,5,6 and 7 are the second set.

- Repeat until the prefix-sum of N inputs is obtained. The i_{th} stage will get prefix-sum of 2^i inputs, so the total number of stage will be log_2^N.[1]

- Stage i needs $\frac{N}{2}$ of adder with i bits wide.
- TCU 0 will always get 0 and TCU i(i>1) will get the sum of the first i-1 inputs.

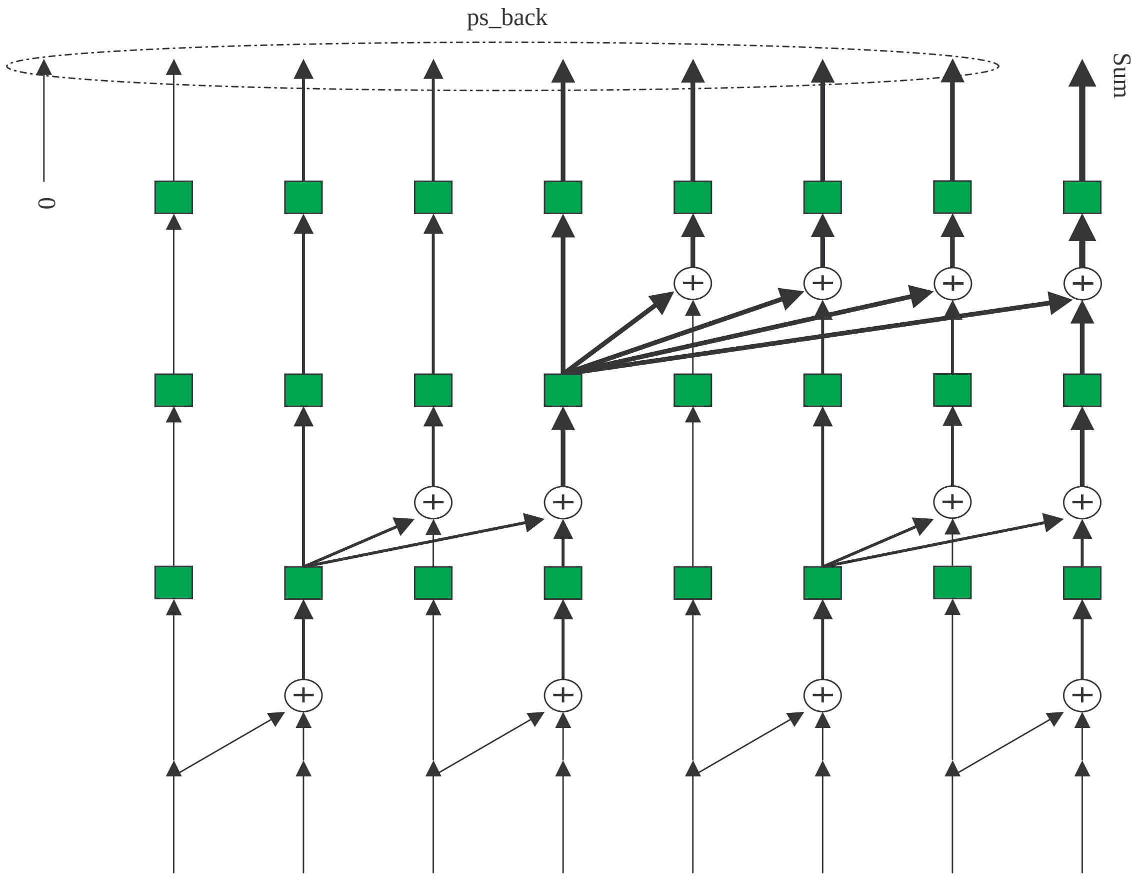


Figure C.2: Prefix-sum tree for 8 inputs

C.3.5 Putting it All Together

The components that serve the PS operations have been described. Figure C.3 on the following page shows the whole system.

[1]Here stage is counted from 1

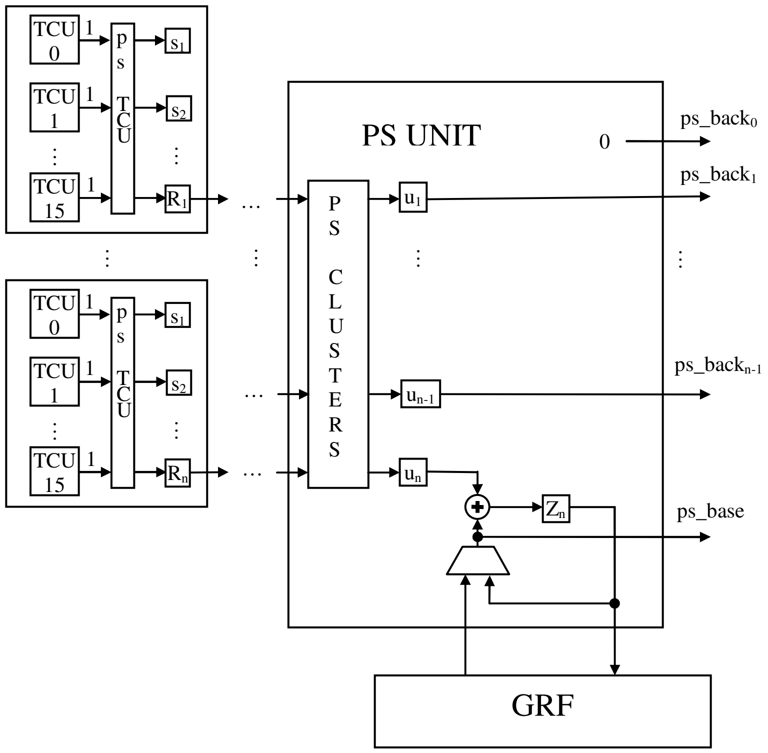


Figure C.3: PS unit and clusters

Not all signals are shown.

PS_TCUs calculate zero-based prefix sum inside a cluster and the results and the time stamp that are assigned by the PS_UNIT are kept in local registers $s_1 \ldots s_{15}$. When the result with the corresponding time stamp comes back it can calculate the final prefix- sum results by adding s_i,*ps_back* and *ps_base*.

The number of cycles needed for a prefix-sum operation varies between 10 and 25, depending on the number of global registers used in the program. Recall that, the prefix-sum unit is used for PS operations to the 8 global registers, but at any given cycle, it can only be used for only one particular global register, *active base register*. On the other hand, a TCU can send a prefix-sum request for any global register, the *pending base register*. When a TCU executes a ps instruction, it waits until the active base register matches the pending base register. The prefix-sum unit monitors pending base registers from all TCUs and switches the active base register among those pending base registers. If only one global register is used in the program and the active base register of the prefix-sum unit matches that global register, a ps operation will take only 10 cycles (figure C.4 (b)). If the active base register does not match the pending base register from a TCU, the TCU has to wait until the prefix-sum unit switches the active base register to the pending base register and this procedure takes up to 15 cycles (figure 3.4 (a)).

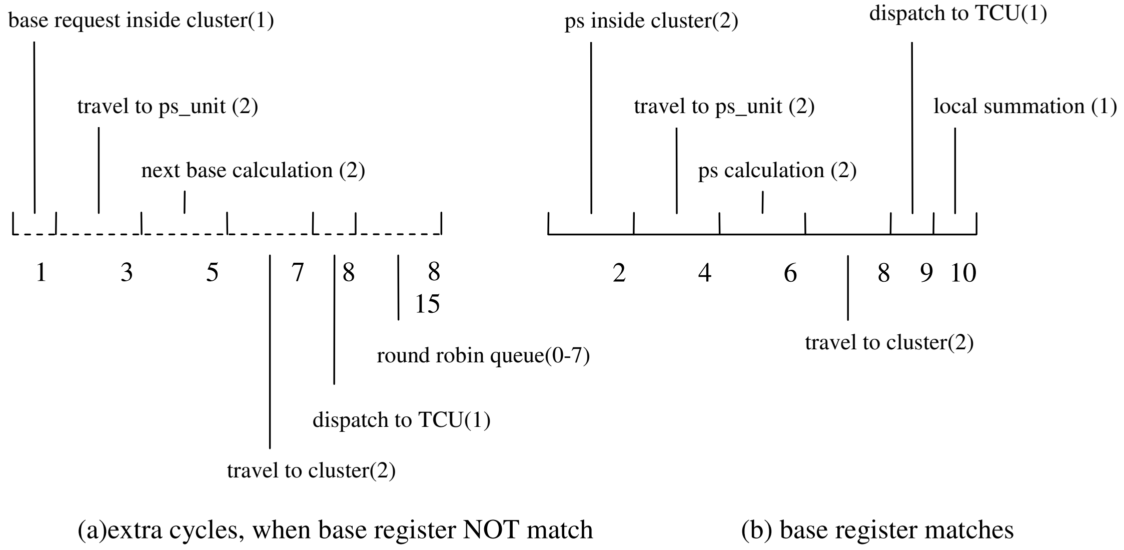
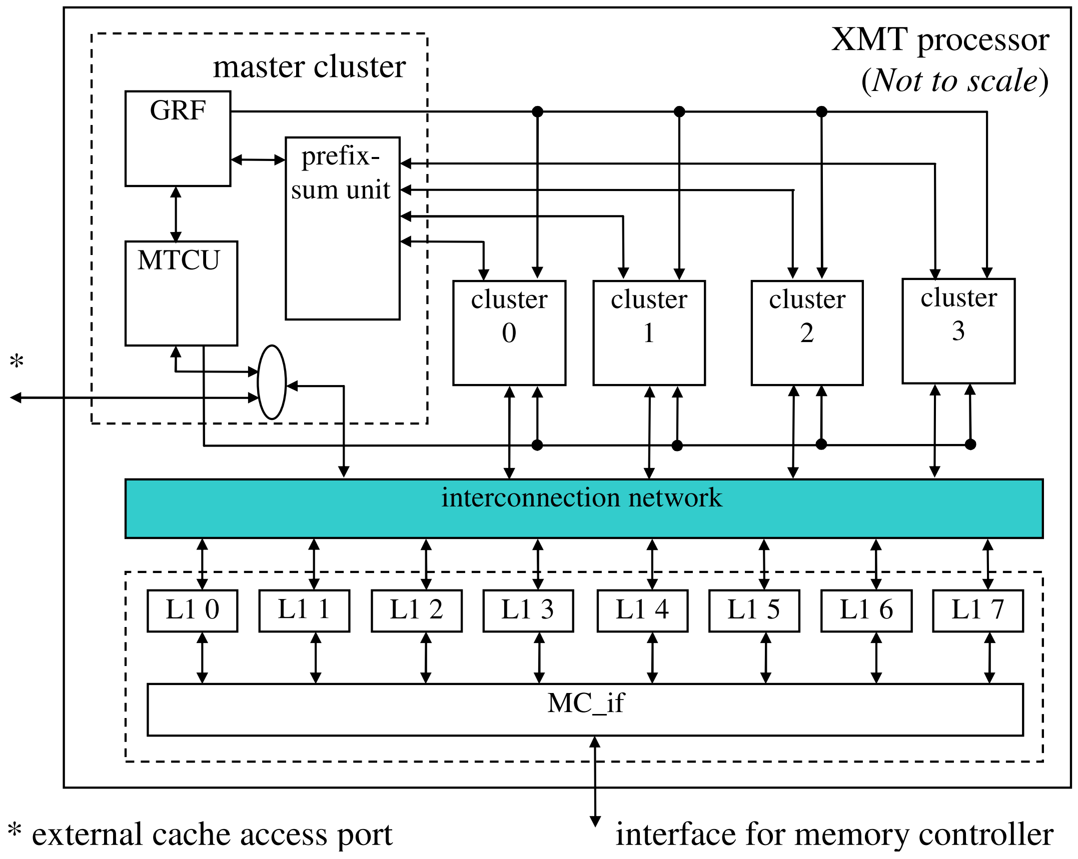


Figure C.4: Prefix-sum operation

Appendix D

Interconnection Network



The interconnection network provides all-to-all communication paths between clusters and on-chip cache modules. Figure D.1 on the next page shows components inside the interconnection network. The shaded part is based on papers [8, 9, 7] and it is designed by Aydin Balkan. The shaded part is integrated into the XMT processor during place and route as a soft IP.

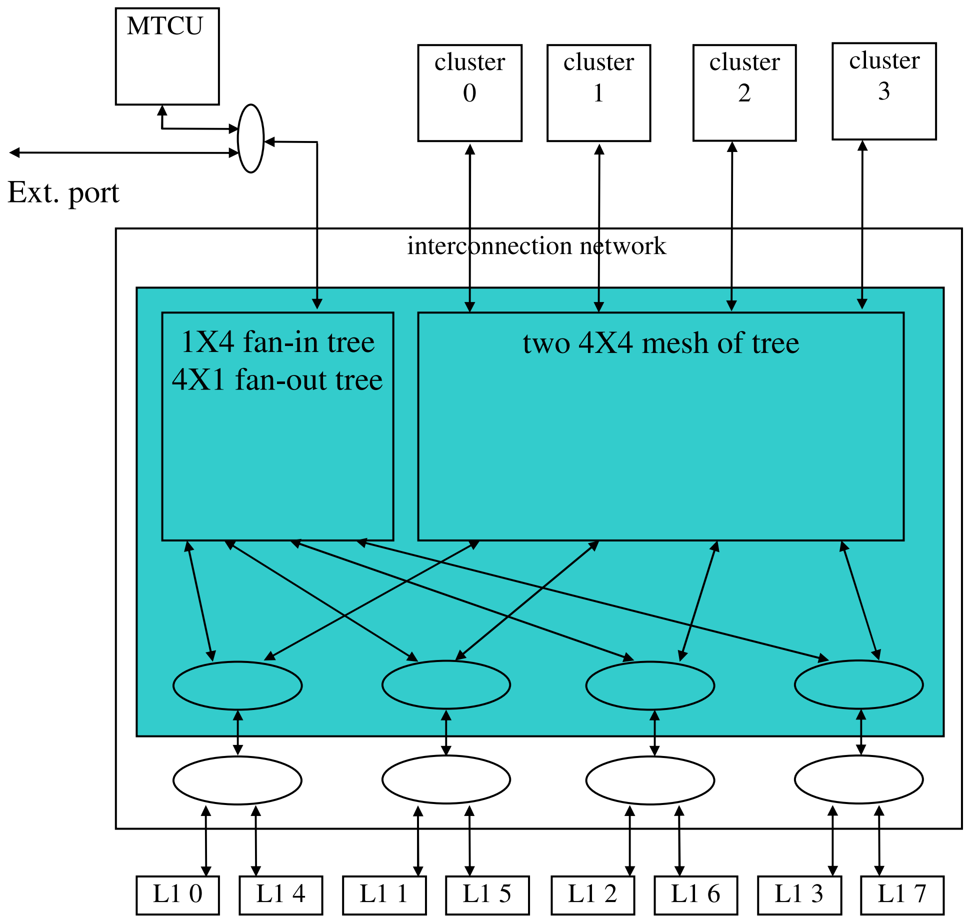


Figure D.1: interconnection network

D.1 Interface of the interconnection network

Table D.1 on the following page lists the interfaces of the interconnection network. The interconnection network provides a full duplex communication path between clusters and cache modules. Requests enter the interconnection network from *ls_send_vec* or *ls_send* and out from the interconnection network through *cache_request_vec* to cache modules. Responses from caches enter from *cache_response_vec* and out through *ls_back_vec* or *ls_back_e*. Except *ls_back_vec*, each request or response port is paired with an acknowledge port indicating the packets are accepted by the receiver.

The *ls_back_vec* does not have an associated acknowledge port, because the response packets from the interconnection network are guaranteed to be accepted by clusters for every cycle.

Table D.1: Interface of the Interconnection Network

Name	Width	type	Description
ls_send ls_send_select_e	50 1	packet ack	input of request port for MTCU and external port
ls_send_vec ls_send_select	200 4	packet ack	input of request port for clusters
cache_request_vec cache_request_select_vec	200 4	packet ack	output of request
cache_response_vec cache_response_select_vec	208 4	packet ack	input of response from cache modules
ls_back_e ls_back_e_select	52 1	ack packet	output port of response from cache module
ls_back_vec	208	packet	output port of response from cache

D.2 Packet formats

For write requests from clusters, both address and data need to be transferred to cache modules while a read request only needs to send an address. Having an interconnection network that is wide enough for both address and data in one packet is not a good solution, since the bandwidth will be wasted for read requests. A packet in the current interconnection network can only carry either an address or data. Therefore a write request needs two packets and a read request needs one packet.

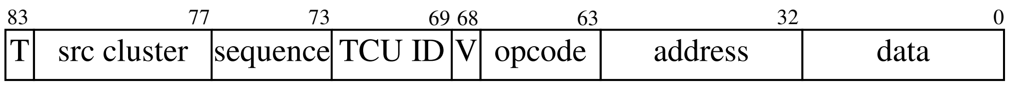
Read requests expect data packets from cache modules and write requests expect acknowledgements from cache modules.

Figure D.2 on the next page shows a bit field definition for packets. (a) shows the request packet from a TCU, which is generated for a load/store command and also used by shared cache modules. (b) and (c) are packets split from (a), data packet is only relevant for a write request. (d) is the format of the response packet from cache modules. The fields of a load/store request are explained in table D.2. Note that since the XMT chip is expected to have 64 clusters and 64 cache modules, 6 bits are used for identifying a cluster or a cache module. When an 84-bit memory request is split into two packets: address and data packets, some fields are replicated in both packets as proper head information is needed for each packet.

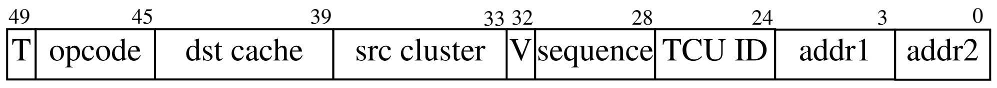
As shown below, the address field of a memory request is partitioned into 3 fields : dst cache, addr1 and addr2. The rest of field in address packet have the same definitions as in a memory request shown in table D.2 on page 185.

dst cache	address[10:5]
addr1	address[31:11]
addr2	address[4:2]

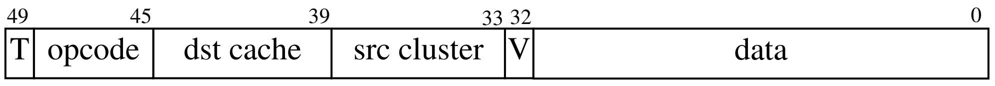
The fields in a data packet have the same definitions as in a memory request



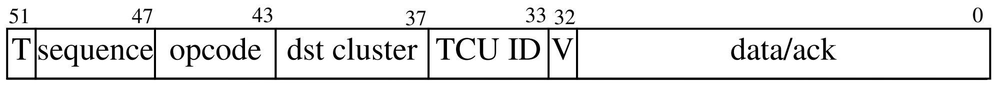
(a) request from TCUs



(b) address packet



(c) data packet



(d) response packet

Figure D.2: Packet bit definition

shown in (a) table D.2 on the following page.

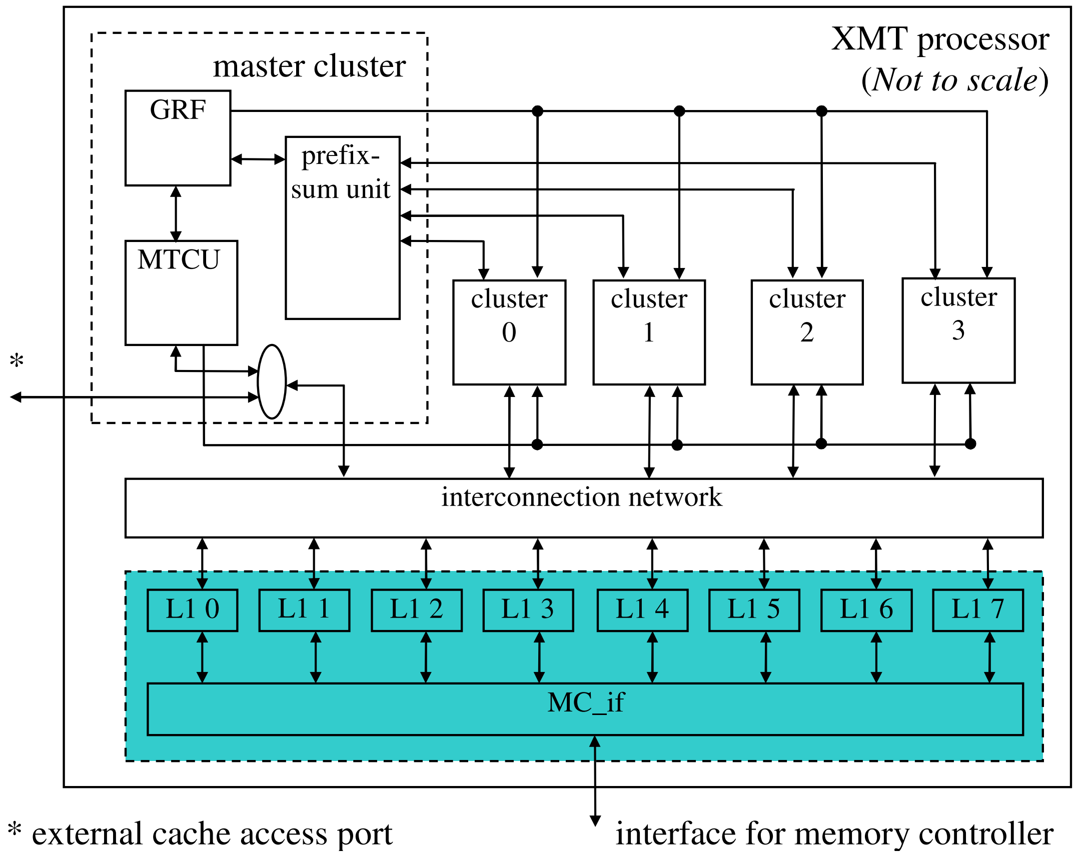
When packets arrive at cache modules, they are restored to the 84-bit request packet (Figure D.2(a)). The response from cache modules use some fields from a request packet. For example, the *dst cluster* is used for routing in the interconnection network and that is the copy of the *src cluster* field in the request. Depending on the request type, data or an acknowledgement is sent back to the TCUs. Currently, the acknowledgement packet has 0s in the field of *ack* in Figure D.2 (d).

Table D.2: Field definition for a load/store request

Field	Width	Description
T	1	specifies cluster type: 0 - parallel clusters, 1 - master cluster
V	1	valid bit: 0 - invalid, 1 - valid
src cluster	6	specifies the request is from which cluster
sequence	4	each TCU can send multiple requests and they are identified by a sequence number
TCU ID	4	specifies the request is from which TCU
opcode	4	opcode for memory operation. see E.14 on page 198
address	32	address of the memory request
data	32	data for a write request

Appendix E

On-chip Cache



Key Features

- Non-blocking cache, can hold up to 64 misses for 8 different cache lines
- 2-way associative
- Write allocate

- 8 cache modules share one memory access port
- Cache line size: 32 bytes, 8 word(32-bit)

E.1 Overview of the Cache Hierarchy in the XMT Prototype

The XMT processor has eight on-chip cache modules, they are connected to the four clusters with an interconnection network. The cache modules handle 3 categories of commands: read, write, and psm. A value is sent back to the requestor for read and psm, and an acknowledgement is sent to the requestor for a write operation.

The cache modules are connected to (i) an interconnection network, through which they get requests from the clusters and send back responses to the clusters. (ii) a shared memory controller(MC) through a mc_if (memory controller interface) module, which provides arbitration for requests from eight cache modules.

The cache module is a non-blocking cache and it can respond to new requests until there are up to 64 pending cache misses for 8 different cache lines. A cache access time is 3 cycles for a hit. The cache module is able to handle out-of-order responses from the MC, which means the responses can arrive in a different order than they were sent.

E.2 Interface of a Cache Module

Table E.1 on the next page and E.2 on the following page list the input and output of a cache module, respectively. In the interface names, L2 refers to the lower level cache in the memory hierarchy and in the current design, since there is no second level cache, these ports are connected to the memory controller through the mc_if module.

Table E.1: Inputs of a parallel cache module

Name	Bit width	Driver	Description
ackFromInter	1	ICN	acknowledgement to the cache response
ackFromL2	1	mc_if	acknowledgement of L1 request to mc_if
rqstToCacheIn	84	ICN	request from clusters and mcluster
rspsFromL2	36	mc_if	response from mc_if

Table E.2: Outputs of a parallel cache module

Name	Bit width	Sent to	Description
ackToInter	1	ICN	acknowledgement to the cache request
ackToL2	1	mc_if	acknowledgement of mc_if response
result	52	ICN	response to the clusters and mcache
rqstToL2	38	mc_if	request to mc_if
idle	1	Not used	

E.3 Components of a Cache Module

Figure E.1 on the next page shows submodules inside a cache module. The connections among them are suppressed since it is very complicated and may not help in understanding the big picture of a cache module. A brief description of the submodules is listed in Table E.3.

Table E.3: Sub-modules of the L1 Cache

Module Name	Purpose
cache	Actual data-storage
tag	The information to distinguish cache misses from cache-hits
newTag	On a cache miss, this is a temporary location to store the tag of the new address for the new cache line, this is also used to check whether this is the first miss for the cache line
PendingRqsts	A buffer that stores requests that are misses
pendQue	A stack for managing the PendingRqsts buffer
L2RqstBuf	FIFO, acts as a buffer for requests to the MC on a cache miss
L2RspsBuf	FIFO, used to store responses from the MC

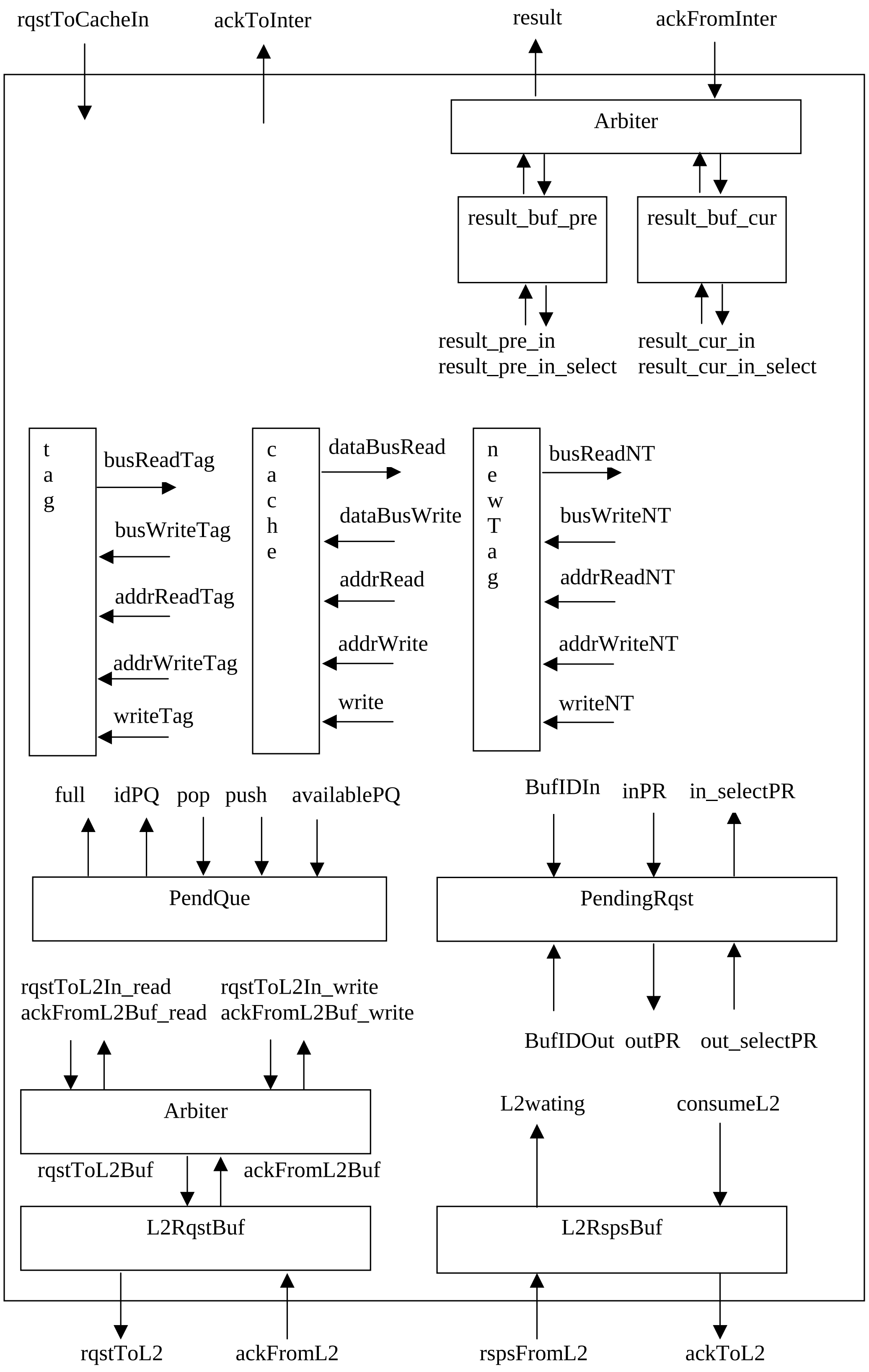


Figure E.1: Components of an L1 Cache Module

E.3.1 Tag

The *tag* module is used to store the tags for the cache module. Each cache module is 32KB and 2-way associative, so the width of the index is 12 bits. Therefore the width of the *tag* is $16 = 32 - 3(8 cachemodules) - 12(index) - 2(4 - byteword) + 1(validbit)$. The *tag* module has one write port and 2 read ports. If both the read and write ports try to access the same word, the old value will be read from the read port. Compared to the other option, where read port always gets the new value, the hardware implementation is simple, since some extra logic is not needed. In the cache module, since this case (read and write the same location in *tag*) is an exception and the value in the read port will not be used anyway, this simple implementation is chosen. The inputs and outputs for the *tag* are shown in table E.4 and E.5.

Table E.4: Tag Inputs

Name	Bit width	Description
busWriteTag	16	The tag value
addrReadTag1	10	The index for which tag to read
addrReadTag2	10	The index for which tag to read
addrWriteTag	10	The index for which tag to write
writeTag	1	Write enable signal

Table E.5: Tag Outputs

Name	Bit width	Description
busReadTag1	16	The tag value
busReadTag2	16	The tag value

E.3.2 NewTag

The *newTag* module is used to store tags of previously missed requests. When a request turns out to be a cache miss, before sending a request to the MC, we need to check if there is a previously missed request that already asked for the same cache line and waiting for the response from the MC. If the cache line has been requested before, no request needs to be sent to the MC. The *newTag* module stores tags of these pending cache lines and a queue ID within the *PendingRqst*(see below). Since there are eight queues within a *PendingRqst*, *newTag* is 3 bits wider than *tag*. The *newTag* module has one write port and 2 read ports. If both the read and write ports try to access the same word, the updated value will be read from the read port. The inputs and outputs for the *newTag* are shown in tables E.6 and E.7.

E.3.3 Data Array

The *cache* module is a simple memory array with one read port and one write port. The depth of this memory array is 8192 and width is 32-bit. If both the read and write ports try to access the same word, the updated value will be read from read port.

Table E.6: NewTag Inputs

Name	Bit width	Description
busWriteNT	19	The new tag value, lower 3 bits indicate which buffer in PendingRqsts are used
addrReadNT1	10	The index for which new tag to read
addrReadNT2	10	The index for which new tag to read
addrWriteNT	10	The index for which new tag to write
writeNT	1	Write enable signal

Table E.7: NewTag Outputs

Name	Bit width	Description
busReadNT1	19	The new tag value
busReadNT2	19	The new tag value

Table E.8: Cache Inputs

Name	Bit width	Description
dataBusWrite	32	new value need to be written into
addrRead	13	the index for which to read
addrWrite	13	the index for to write
write	1	write enable signal

Table E.9: Cache Outputs

Name	Bit width	Description
dataBusRead	32	the word at the location specified by *addr-Read* in previous cycle

E.3.4 Pending Requests

When there is an L1 cache miss, an entire cache line will be brought in. The address of the requested word, and the TCU that requested it is placed in one of 64 entries in the *PendingRqsts* module. There are eight identical queues (called *PendRqstInL1*) inside the *PendingRqsts* module, each of which can be allocated to service one cache line at a time. Each such queue is eight entries deep, and each entry can be for any member of the cache line, and for any TCU. After the responses from the MC arrive, the missed requests in the queue are processed and the queue can be used for another missed request.

E.3.5 Pending Queue

Because the cache module does not require the MC to process requests in-order, the queues in the *PendingRqsts* module need to be managed properly. A stack, *PendQue*, is used for tracking the usage of the queues in the *PendingRqsts* module. This stack has entries that indicate which queue in the *PendingRqsts* module to allocate. When this stack receives a *pop* command (when there is a new cache miss for an as-yet not serviced cache line), an empty queue ID is presented

Table E.10: Inputs of Pending Requests

Name	Bit width	Description
bufIDIn	3	specifies into which queue the new request (inPR) will be inserted
inPR	57	the new request to be inserted
out_selectPR	1	acknowledgement to the outPR, indicating the outPR can be dequeued
bufIDOut	3	specified the request from which queue will be processed

Table E.11: Outputs of Pending Requests

Name	Bit width	Description
in_selectPR	1	acknowledgement to the inPR, indicating it will be enqueued
outPR	57	output of the queue that is specified by bufIDOut in previous cycle

for the cache line and the pointer will be moved to the next available queue ID. On a push command (when a buffer becomes empty after an entire cache line has been served), the pointer on the stack is moved in the other direction, and the entry in the stack is updated to reflect the newly available buffer.

Table E.12: Inputs for Pending Queue

Name	Bit width	Description
pop	1	Indicates a buffer in the Pending Requests has just become in-use
push	1	Indicates a buffer in the Pending Requests has just become available
availablePQ	3	Indicates which buffer in the Pending Requests become available

Table E.13: Outputs for Pending Queue

Name	Bit width	Description
idPQ	3	Indicates which buffer in the Pending Requests is available
full	1	Indicates if there are any free buffers to service additional cache line misses. Logic 1 means all queues are used and cannot hold any new cache miss
idlePQ	1	Indicates if all queues are empty(no cache line is pending)

E.3.6 FIFOs and Arbiters

E.3.6.1 MC Request and Response Buffers

These two modules are FIFOs and used to achieve better timing by having registered outputs. As the name indicates, these two modules also provide extra storage for buffering requests and responses. Each of them are 8 entries deep, enough for a full cache line.

E.3.6.2 Result Arbiter and FIFOs

The TCUs expect value or acknowledgement from cache modules. For each request, the return message can be generated at two different times and they are fed to the result port through different ports of the arbiter. The "current" port (marked as result_cur_in in figure E.1) is used for requests that turn out to be cache hits. The "previous" port (marked as result_pre_in in figure E.1) is used for requests that turn out to be cache misses. Both "current" and "previous" ports have a FIFO to provide buffering and reduce the stalls in the cache modules.

E.4 How a Cache Module Works

The cache modules accept three kinds of commands: read, write, and psm. The cache module returns a value for a read or a psm command and an acknowledgement for a write command. The supported commands are listed in the table E.14 The command category is determined by the 2 LSBs from the binary encoding. In

Table E.14: Supported cache access commands

Command	Binary	Description
READ	0000	Read a word to a register, NOT cache the value into ROB(read-only-buffer), blocking
READ_BUF	0010	Read a word to a register, cache the value into ROB, blocking
FETCH_TCU	0100	Read a word to a prefetch buffer, NOT cache the value into ROB, non-blocking
FETCH_TCU_BUF	0110	Read a word to a prefetch buffer, cache the value into ROB, non-blocking
DPF	1110	DRAM prefetch
WRITE	0001	Write a word to cache, blocking
WRITE_POST	0101	Write a word to cache, non-blocking
PSM	0011	prefix-sum to memory
INSTR	0100	Read a word to the instruction buffer
CACHE_PCM	1100	Read and write the performance counters in the cache modules, explained in the chapter F

addition to the command, the requests also contain information about (i) point of origin (source cluster ID, TCU ID). (ii) the sequence ID of the request, which is

Table E.15: Categories of cache access commands

Category	Binary
read	xxx0
write	xx01
psm	xx11

used to identify different prefetch instructions. (iii) memory address, data value for write and psm.

A cache access is processed in a pipeline and it takes at least three stages: tag access, read data, and write data. Some commands will pass some pipeline stages without actually performing any operations. For example, for a write command, the read data stage is not needed and the request will simply moved to the write stage. In the first cycle, the tag information is retrieved from the *tag* module and it will be used to check whether the access is a hit or miss. Based on the cache hit or miss, the cache module will operate differently and they are presented in the following sections.

E.4.1 The Life-Cycle of a Cache Hit

If the tag information retrieved in the first cycle resulted in a cache hit, the set number will be determined and the *cache* module is accessed for read and psm commands. For a write, the request as well as the set information will move to the next stage. After the read data stage, the return value (or acknowledgement) is presented to the "current" port of the result arbiter where it will be sent to TCUs through the interconnection network.

E.4.2 The Life-Cycle of a Cache Miss

For a cache miss, the processing time can be partitioned into 3 parts: (i) cache miss detection and status update, (ii) update tag and data array with the response from the MC, and (iii) process missed requests in the queue in the *PendingRqst* module.

Cache miss detection

If the tags do not match, there are two different cases: (i) First miss. This is the first request for the missed cache line. The cache line needs to be brought in from the off-chip memory to the chip. (ii) Following misses. This is not the first request for the missed cache line. The cache line has been requested by another previously missed request.

- First miss

  For the case of a first miss, the *PendQue* module will assign an empty queue from *PendingRqst* for the missed cache line and the missed request will be kept in the queue. At the same cycle, a request to the MC will be sent out. The *newTag* module will be updated with the tag of the new address and the queue ID assigned by the *PendQue* module. The information in *newTag* is used to check if a missed request is a first miss or a following miss.

- Following miss

  In this case, since a request to the MC has been sent out by a previously missed request for the same cache line, no request to the MC needs to be sent out. The missed request will be appended to the end of the queue in the *PendingRqst* module, and wait for the response from the MC. The queue ID can be found from a *newTag* module, since the first miss of the cache line will store the queue ID to the *newTag* module.

The *newTag* module is used to determine whether a cache miss is a first miss or following miss, since any first miss will update it with the new tag information which can be compared with the request.

Updating tag and data array with response from the MC

The MC will respond to a request from a cache module with the cache line, eight words, as a chain of 8 packets. Each response identifies itself with a 3-bit ID, that is also the same as the queue ID in the *PendingRqst* module. The cache module will pick the victim set and update *tag* and *cache* module with the new cache line. If the victim cache line is a dirty block, the data will be sent back to the MC, otherwise, it will be simply discarded.

During this 8-cycle period, both sets (2-way associative) of the cache for the same index address are locked, since the values are changing and it will result in an error if a wrong value is used. The write port of the *cache* module will be used for updating with the new values, but the read port may not be needed if the victim cache line is not a dirty block. In this case, the read port can be used by new requests from the interconnection network.

Processing a queue in the *PendingRqst* module

After the *tag* and *cache* modules are updated with the responses from the MC, the previously missed requests that are stored in a queue can be processed. For a read and psm request, the value is read from the *cache* module and it will be presented to the "previous" port of the result arbiter. For a write, only the *cache* needs to be updated, since the acknowledgement has been sent to the TCU when the request is stored in the queue.

The number of cycles needed for this stage is the same as the number of requests in the queue. Since the depth of the queues in the *PendingRqst* module is eight, this stage can take up to eight cycles. During this period, the two sets of the cache line for the same index address are locked for the same reason as above.

E.4.3 Cache Module Status

Based on previous discussion, we find that the read and write ports of the *cache* module can be used in three ways: (i) request that is a cache hit, (ii) updating with new data from the MC and write back the victim cache line, and (iii) processing previously missed requests from the *PendingRqst* module.

To make the assignment simple, the cache module operates in three states as shown in Figure E.2 on page 204.

- S1

  Process requests from interconnection network. If the request is a cache hit, the port of the *cache* module will be assigned properly. When a response from MC arrives the cache module changes to S2 state.

- S2

  Update data array (*cache*) with the responses from the MC. After all updates are finished, the cache module enters S3 state.

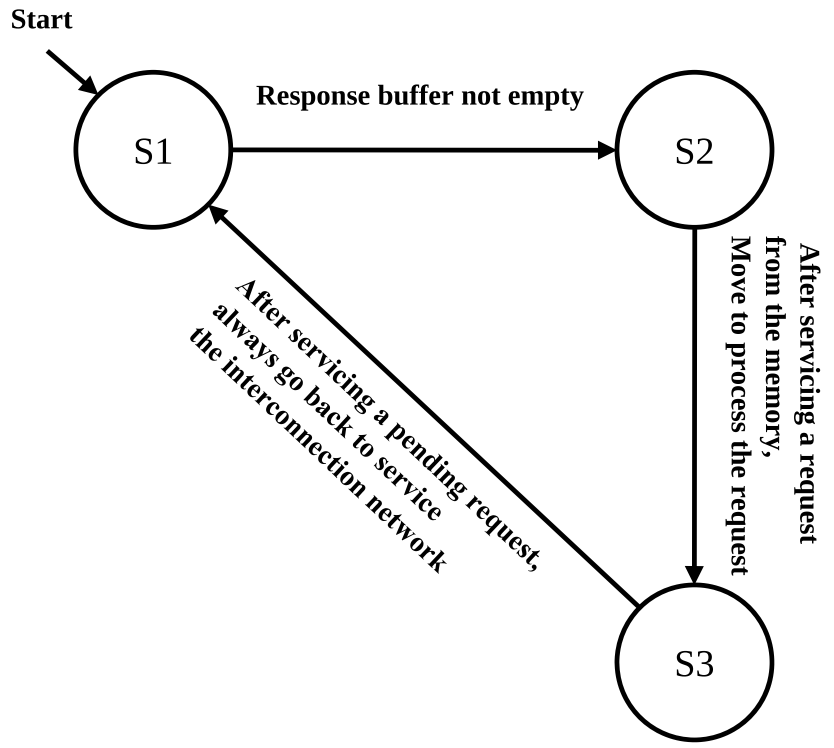
- S3

  Process pending requests (missed) in the *PendingRqst* module. After all pending requests are processed, the cache module enters S1 state.

E.5 Sharing a Memory Controller Channel

The eight cache modules in the XMT prototype share one memory controller interface as shown in figure E.3 on page 205.

For each cache module, there is a bank controller (BC) which handles the requests from cache modules. BCs act as an interface between two clock domains and provides a data packet conversion between the cache module and the off-chip memory controller. The XMT processor uses two synchronous clocks: a slow clock for IO operation and a fast clock (4$\times$ of slow clock) for internal components. The BCs are used where the two clock domains meet and the *response buffer* and *request*


Start
Response buffer not empty
S1
S2
S3
After servicing a request from the memory, Move to process the request
After servicing a pending request, always go back to service the interconnection network

• S1 : Process requests from the interconnection network.
•S2 : Process a response from the memory – that is copying data into cache from the memory
•S3 : Process the pending requests by scanning through pending requests buffer and sending the appropriate words from the cache into the interconnection network.

Figure E.2: State transition diagram for L1 shared cache

que in Figure E.3 on the next page are working at the slow clock. The memory controller interface can transfer one cache line per clock (slow), but cache modules can only process a word per fast clock period. BCs also take care of combining and splitting of a 256-bit cache line.

The requests from cache modules are sent to an off-chip memory controller through the *request que* and responses are buffered in the *response buffer*, then it is broadcast to all of the BCs. Read requests only carry an address, but write requests have 256-bit data along with an address, therefore there are read and write queues separately. The data bus between the XMT processor and MC is a two-way bus, and only one can be the bus master at a certain time. When the XMT processor is the master of the memory controller data bus, the write queue is processed; otherwise, the read queue is processed. The bus control logic is also part of the *request que* and is described in detail in Chapter F on page 212. An arbiter chooses one of the requests from BCs and puts it in the proper queue. The *request que* also checks RAW (Read After Write) hazards for cache lines and, if detected, the read request will be blocked until the write request has been committed to the memory controller.

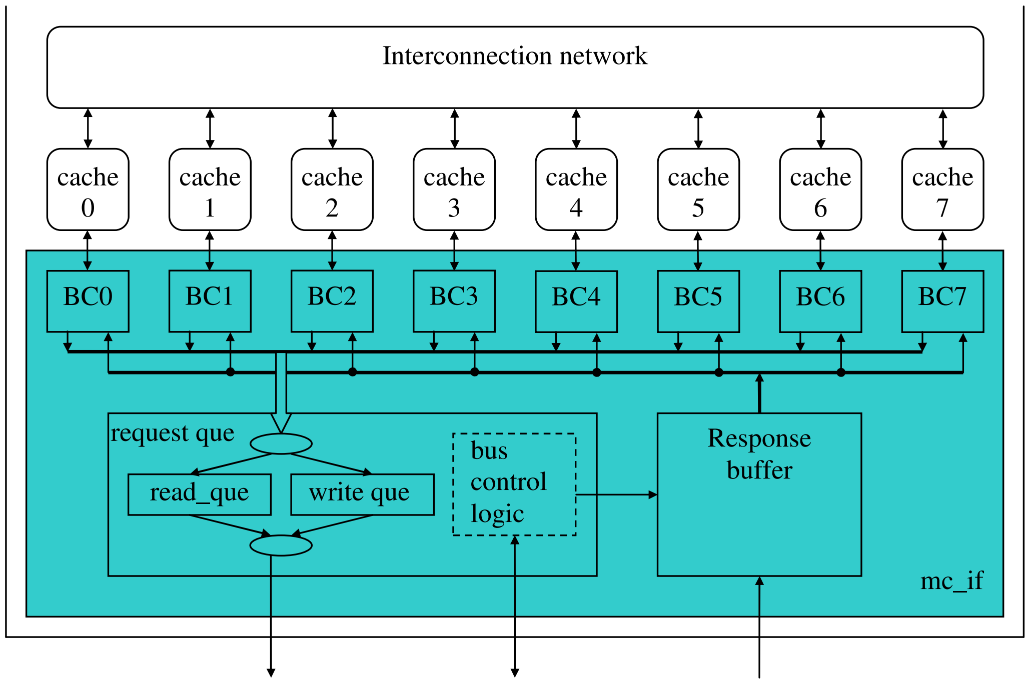


Figure E.3: Eight cache modules share one memory controller interface

E.6 Performance Counter and Special Registers

Each cache module has three performance counter register files (PCRFs) for evaluating behavior of the cache module. The three PCRFs are *hit*, *miss*, and *all* and they count the number of cache requests that are cache hits, misses, and total, respectively. Actually, the *total* is the sum of *hit* and *miss*. Recall that a TCU memory request has a 4-bit opcode field, so there are 16 types of memory requests. The PCRFs count every type of request separately so each PCRF is a 16x32 register file.

There are also counters for DRAM read and write requests that can be used to find how many DRAM operations are performed from each cache module.

Since the SRAMs are prone to manufacturing error, a limited correction mechanism is introduced in the current design using two special-purpose registers (SRs). The first one is to limit the cache module to use only one of the two sets by disabling half of the SRAMs. This is controlled by a 2-bit register, *one_set* in every cache module and either set can be disabled. The second one is using the built-in redundant rows and columns in the Artisan SRAM IPs, which are controlled by a 40-bit register, *RB*. To use the built-in redundancy rows and columns in SRAM IPs, a 40-bit register is needed per IP and we decided to apply the second solution only to the largest SRAM IP, data array, in the cache module, since it takes more than half of the total area taken by all SRAM IPs.

To access these PCRFs and special purpose registers, a memory access request can be sent to cache modules through the external cache access port. The opcode for cache PCRF/SR access request is 0xc as shown in table E.14 on page 198. The opcode 0xc belongs to the read type command, but this command is also used to clear PCRFs and update SRs as shown below. Figure E.6 and table E.16 on the next page shows how the address field is used in a PCRF/SR request.

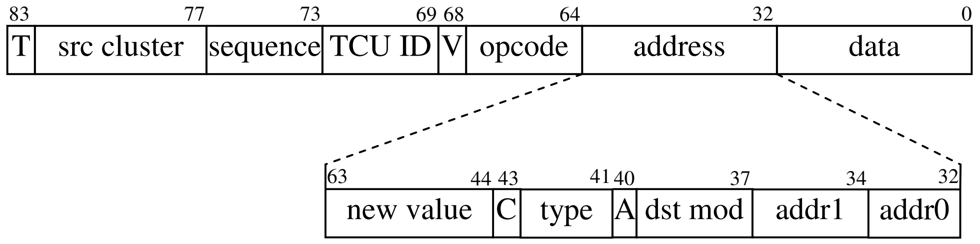


Figure E.4: Cache PCRF/SR access request

For the three PCRFs, the A and addr1, total of 4 bit is used to addressing the PCRF. The location is the same as the definition of the opcode as shown in table E.14 on page 198. When *type* is 3, the request is for SRs. The meanings are different for read and write, which are listed below.

Table E.16: Address field in a cache PCRF/SR request

Field	Width	Description
addr0	2	Not used
addr1	3	lower 3-bit of address in a PCRF (or RF addressing for type=3)
A	1	MSB of address in a PCRF (or RF addressing for type=3)
dst mod	3	the destination cache module, 0-7
type	2	define which PCRF/SR is accessed. 0 - hit, 1 - miss, 2 - total, 3 - SRs
C	1	whether update the selected PCRF/SR. 0 - not update, 1 - update
new value	20	when C is 1 and SRs are selected, this new value is written into one of the SR specified by addr1

2 LSBs of addr1	C	What happens?
00	0	returns number of cache lines read from DRAM
01	0	returns number cache lines written to DRAM
10	0	returns current one_set value in bit 1:0 and upper 8 bits of RB in bit 31:24
11	0	returns lower 32 bits of RB
00	1	reset both DRAM access counts to 0
01	1	update lower 20 bits of RB using new value
10	1	update one_set with the 2 LSBs in new value

The one_set is a special register that configures which set among the 2 sets in the cache modules is used. Among the SRAM IPs used in the cache modules, tag, newTag, and cache are controlled by one_set. The meaning of the one_set is listed below.

Value	Meaning
0x	Both sets are enabled.
10	set 0 is enabled, but set 1 is disabled.
11	set 1 is enabled, but set 0 is enabled.

The RB is used to configure how the two redundant rows and columns are used. The meaning of the bits are defined in table E.17

Table E.17: Bit definition for RB

Bit(s)	Name	Description
39	RRE2	Row redundancy enable fuse for second redundant row, if it is blown(1) then row redundancy is ON for first row.
38:30	FRA2	Faulty row address fuse for second redundant row, Bus with 9 bits This has the logical address of the row. These are the higher A[8:0] bits of address bus that select this row.
29	RRE1	Row redundancy enable fuse for first redundant row, if it is blown(1) then row redundancy is ON for first row.

Continued on Next Page…

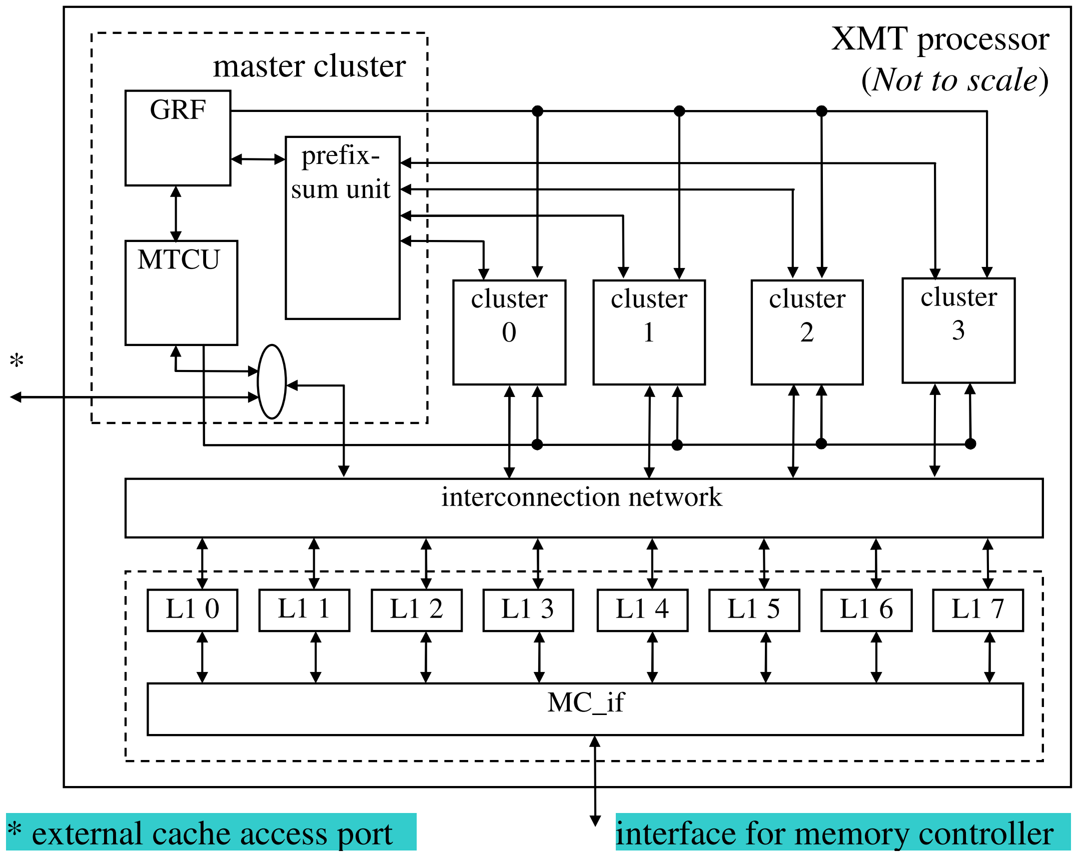
Bit(s)	Name	Description
28:20	FRA1	Faulty row address fuse for first redundant row, Bus with 9 bits This has the logical address of the row. These are the higher A[8:0] bits of address bus that select this row.
19	CRE2	Column redundancy enable fuse, if it is blown(1) then second column redundancy is ON.
18:14	FBA2	Faulty Bit Address fuse for second column, this fuse value tell the index of the BIT in the word that is faulty. For example for 9 BIT memory if the first bit is faulty the FBA will have value 4'b0000, if the 9'th bit is faulty it will have value 4'b1000.
13:10	FCA2	Faulty Column Address fuse for second column, this fuse value tell the logical address of the faulty column in the faulty bit. The width of this FUSE is same as the width of the lower address bits.
9	CRE1	Column redundancy enable fuse, if it is blown(1) then first column redundancy is ON.

Continued on Next Page…

Bit(s)	Name	Description
8:4	FBA1	Faulty Bit Address fuse for first column, this fuse value tell the index of the BIT in the word that is faulty. For example for 9 BIT memory if the first bit is faulty the FBA will have value 4'b0000, if the 9'th bit is faulty it will have value 4'b1000.
3:0	FCA1	Faulty Column Address fuse for first column, this fuse value tell the logical address of the faulty column in the faulty bit. The width of this FUSE is same as the width of the lower address bits.

Appendix F

Interface of XMT Prototype



The XMT processor core has two interface ports to the outside world. One is the interface to the off-chip memory controller and the other one is for users to access the memory space of the XMT processor, including the on-chip caches. The interface of an XMT processor core is shown in Figure F.1.

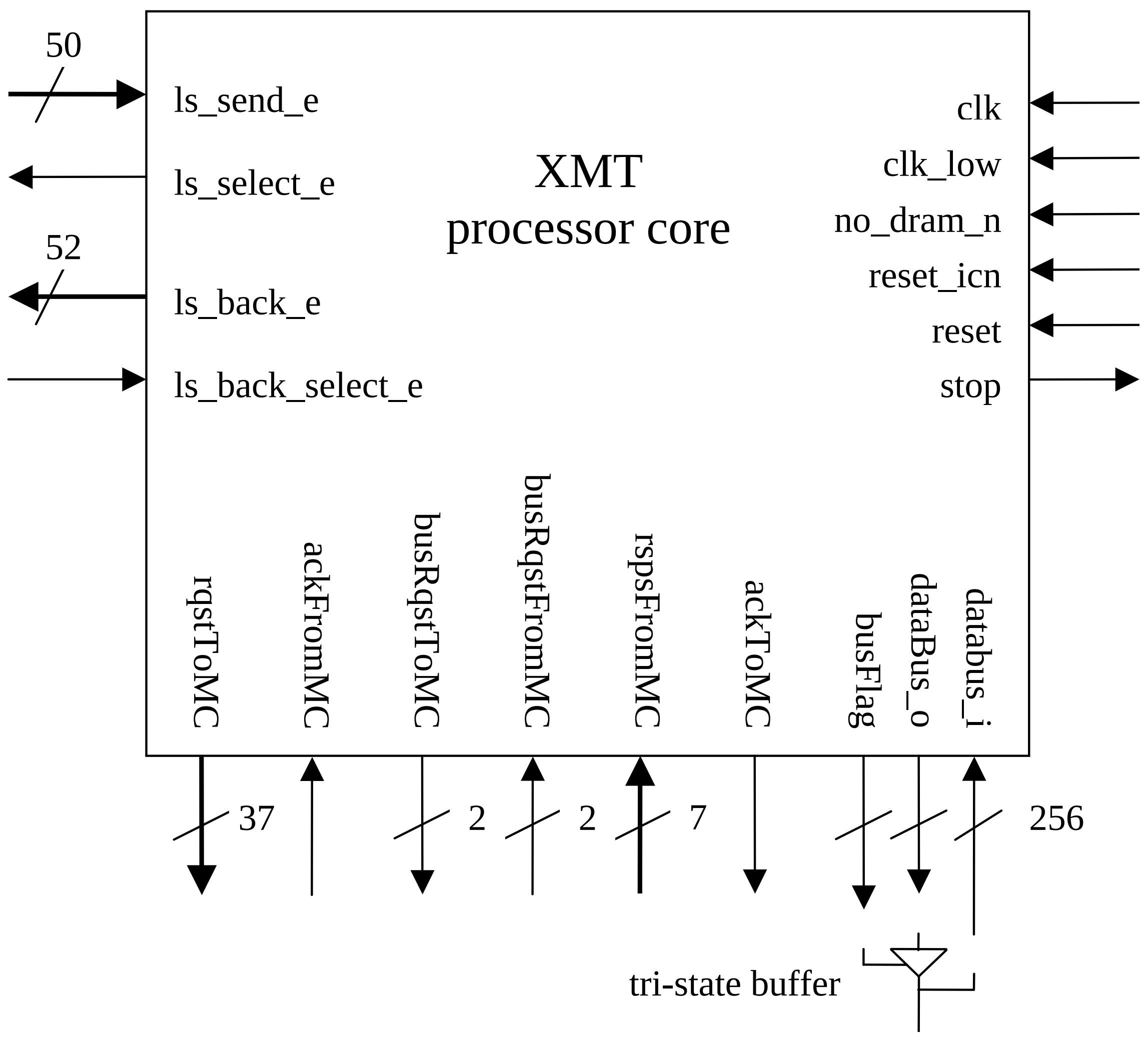


Figure F.1: Interface of an XMT processor core

F.1 clock, reset, stop and no_dram_n

The XMT processor core uses two clock signals, *clk_low* and *clk*. *clk* is generated internally from clk_low by a PLL. The *clk* is 4 times faster than *clk_low* and they are positive-edge aligned.

Both *reset* and *reset_icn* pins are active-high. The *reset_icn* signal is used to reset the interconnection network and eight on-chip shared cache modules, as well as the interface to the MC. After *reset_icn* is cleared (logic 0), the cache modules will clear their contents and then accept requests. If the *reset* pin is cleared (logic 0), then the MTCU will start executing instructions in the memory, starting from address 0.

Typically, both *reset_icn* and *reset* need to be asserted high (logic 1) for resetting the processor, then *reset_icn* cleared (logic 0). Then a program and data can be written into the memory of the processor through the external memory access port. After all initial data and programs are in place, the *reset* pin can be cleared to start the execution of the processor. If the processor reaches a *halt* instruction, the processor flags itself as "done" using the *stop* pin, setting it to high. To execute another program, the *reset* pin needs to be set, which clears the *stop* signal. A new program begins after clearing *reset*. The *reset_icn* signal can stay at logic 0, unless the cache modules need to be cleared. Note that *reset_icn* only clears the on-chip cache, but not off-chip memory.

The no_dram_n pin is for testing the XMT processor without connecting to an off-chip memory controller. If it is tied to logic 0, the off-chip memory controller is not used. If a program does not need to access off-chip DRAM (data and program fit in on-chip cache), it should run correctly. During normal operation mode, this pin should be tied to logic 1.

F.2 External Memory Access Port

Both the *ls_send_e* and *ls_back_e* ports have the same bit definition as the interconnection network (see chapterD). Both the *ls_send_e* and *ls_back_e* ports have an associated acknowledge pin, which are driven by receivers. Logic 1 means the receiver will store the value in the next positive edge of *clk_low*, otherwise not. For a write command, two packets need to be sent to the processor and the two packets have to arrive in consecutive cycles. The response message can arrive in a different order than requests were sent, but each response will be marked with the sequence number presented in the request so that responses can be paired with the request.

This port is also used to access performance counters in cache modules and some special purpose registers introduced for testing. Special registers are a 2-bit direct map register and 40-bit redundancy bus (RB). For detailed binary encoding information see section E.6.

F.3 Interface to an Off-chip Memory Controller

The XMT processor communicates with an off-chip memory controller with this interface. The data is transferred through a 256-bit bus between XMT processor and off-chip memory controller.

- rqstToMC (output)

  The XMT processor sends requests to off-chip memory controller through this port.

Bit index of rqstToMC	Description
36	opcode: 0-read 1-write
35:33	request ID, it is the same as the ID of the queue in *PendingRqst* where the missed requests are stored
32	valid bit
31:0	address of the request

- ackFromMC (input)

  Acknowledgement from the MC for the request presented on *rqstToMC*.

- rspsFromMC (input)

  Header part of the response from the off-chip memory controller.

Bit index rspsFromMC	Description
6	valid bit
5:3	destination cache ID, identify which cache module among those eight modules sharing one off-chip memory controller
2:0	queue ID, the copy of 35:33 in the *rqst-ToMC*

- ackToMC (output)

  Acknowledgement to the MC for the response presented on *rspsFromMC*.

- dataBus_i, dataBus_o, and busFlag

  These ports are supposed to be connected to tri-state buffers. The busFlag will be set to logic 1s, for enabling the output of the tri-state buffers. The width of these ports are 256 bits, a cache line.

- busRqstToMC (output)

  Only one of the XMT processor and memory controller can be the bus master at a given time. Therefore a handshaking protocol is needed between the XMT processor and the memory controller. This port is used to notify the current status of the XMT processor to the off-chip memory controller. The value of this 2-bit wide port is the same as the binary encoding of the state machine inside the XMT processor.

value	Description
00	XMT is the bus master and will keep the bus for next cycle
01	XMT is NOT the bus master
11	XMT is the bus master, but transfer the control to the memory controller next cycle
10	Impossible value, will never present this value

- busRqstFromMC (input)

  When the memory controller is the bus master it will notify the XMT processor about the status of the memory controller. The value of this 2-bit wide port is the same as the binary encoding of the state machine inside the memory controller.

value	Description
00	MC is the bus master and will keep the bus for next cycle
01	MC is NOT the bus master
11	MC is the bus master, but transfer the control to the memory controller next cycle
10	Impossible value, will never be this value

The protocol between the XMT processor and the MC for switching bus master is shown in figure F.2. The two states at the top and right-middle state is for the XMT processor and the rest of states are for the off-chip memory controller. Initially, the MC is the bus master and the XMT processor is the slave. The bus master continues to use the bus until there is no need for the bus or the buffer in the slave is full, then it will enter the transition state. When the slave sees the master is in the transition state, it enters master state next cycle and the original master now becomes the slave. During bus master transition, the data bus is not used for one cycle, which results in one bubble cycle for every bus master change.

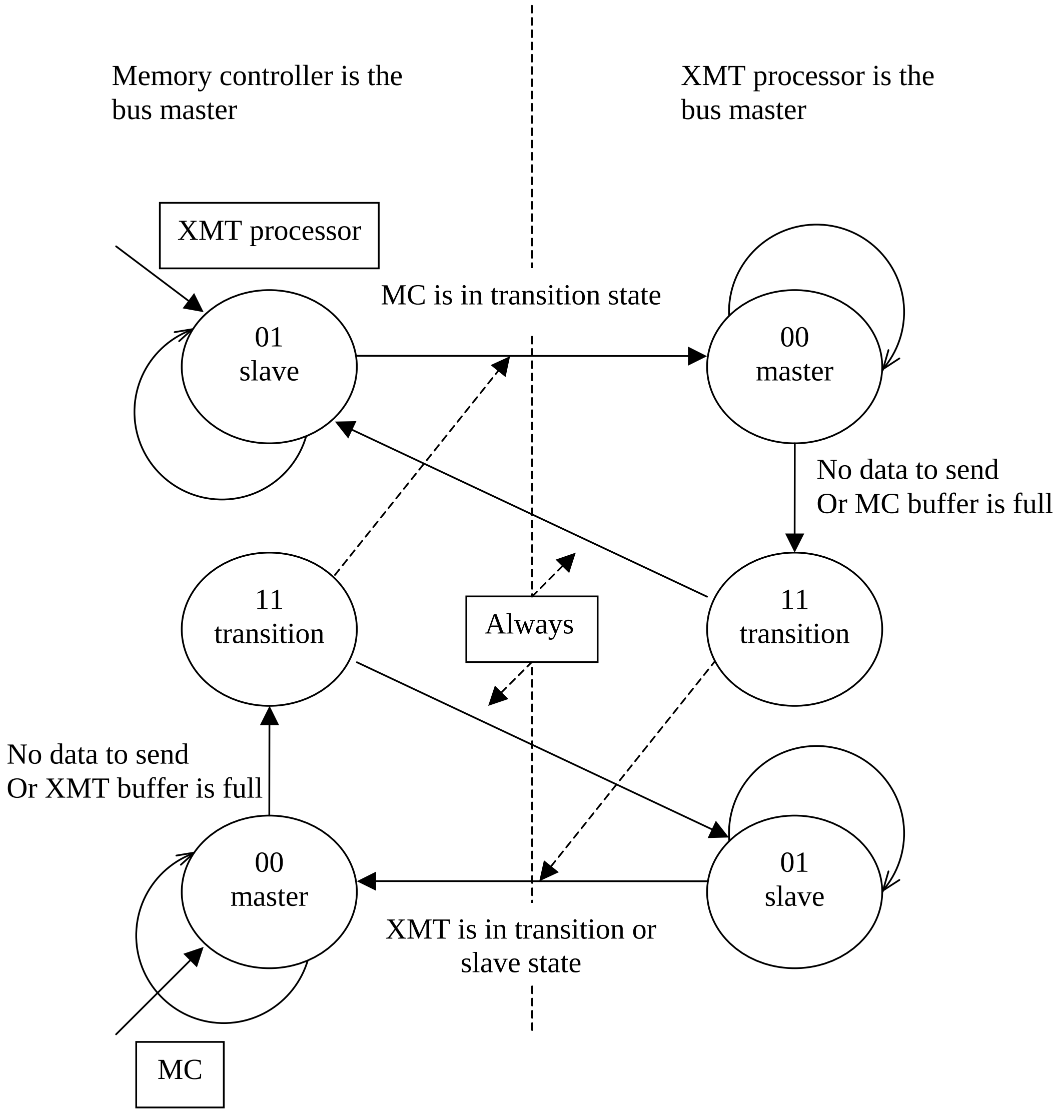

Memory controller is the bus master
XMT processor is the bus master
XMT processor
MC is in transition state
01 slave
00 master
No data to send Or MC buffer is full
11 transition
Always
11 transition
No data to send Or XMT buffer is full
00 master
01 slave
XMT is in transition or slave state
MC

Figure F.2: State transition diagram of XMT processor and MC

Appendix G

XMT FPGA Prototype - Paraleap

The XMT FPGA prototype - Paraleap[1] shares most of the Verilog code with the XMT ASIC version, but there are some differences between these two prototypes. For example, the FPGA prototype includes a PCI interface that is connected to the external port of the XMT processor core.

- PCI interface

  The FPGA prototype uses the PCI bus to communicate with the host computer and a pci_if module provides the connection between PCI bus and the external memory access port of the XMT processor.

- On-chip DDR2 DRAM controller

  The FPGA prototype has an on-chip DDR2 DRAM memory controller shared by eight on-chip cache modules.

- Other minor differences

  There are no redundance bus(RB) and cache module configuration registers. The size of instruction buffer in the clusters is different.

[1] A naming contest for the XMT FPGA prototype held by the University of Maryland got nearly 6000 submissions. The name Paraleap was selected.

G.1 PCI Interface Module

The DN8000K10PCI board, which is used in the XMT FPGA prototype, has a dedicated 64-bit PCI bridge, QL5064, connected to the FPGA A. Dini Group, the manufacture of the board, provides a sample code for simple PCI interface and it can be found in the accompanying CD. Through the module, ql5064_interface, the registers in FPGA A can be accessed. The ql5064_interface (QL) module is also capable of DMA(direct memory access), but it is not used in the Paraleap prototype. For normal reads and writes, ql5064_interface can access eight bars (term used for PCI), but the Paraleap uses only bar 2.

G.1.1 Interface of pci_if Module

The interface of the pci_if module is listed in table G.1. The pci_if module connects the XMT processor core and the ql5064_interface module that was provided by Dini group.

Table G.1: interface of pci_if module

Name	width	Port	Con. to	Description
bar2_read_enable	1	in	QL	QL read request
bar2_write_enable	1	in	QL	QL write request
target_write_data	64	in	QL	data value to be written to pci_if
bar2_read_data	64	out	QL	data from pci_if as the response of a read
bar2_read_data_valid	1	out	QL	indicating bar2_read_data is valid
target_address	64	in	QL	address of read and write operation
ls_send_e	50	out	XMT	request to XMT
ls_select_e	1	in	XMT	acknowledgement of ls_send_e
ls_back_e	52	in	XMT	response from XMT
stop	1	in	XMT	set to logic 1 when a program finishes
reset_icn	1	out	XMT	reset interconnection network and cache modules
reset_xmt	1	out	XMT	reset XMT processor core, active high. Connected to the reset pin of the XMT processor

G.1.2 How does the pci_if Module Work?

G.1.2.1 Two Buffers in pci_if Module

The pci_if has two 256×32 buffers. When the host computer writes the XMT processor's memory, it first uploads the data into the buffers and then the pci_if module will send the data to the XMT processor through the external port.

A simple procedure of writing the XMT processor's memory is described below, which uses only one buffer.

1. Write data to the buffer chosen until the buffer is full or no more data to write.
2. Specify the starting address in the XMT processor memory space.
3. Notify pci_if to transfer data to the XMT processor memory space through the external port.
4. Wait until it finishes. If there is more data to write, repeat these steps until all data is transferred.

Because there are two buffers, it is possible to overlap uploading and transferring of the data as shown in figure G.1(a).

To read the XMT processor's memory, the following steps are needed.

1. Specifies the starting address in the XMT processor's memory space.
2. Notify pci_if to transfer data to the XMT processor memory space through the external port.

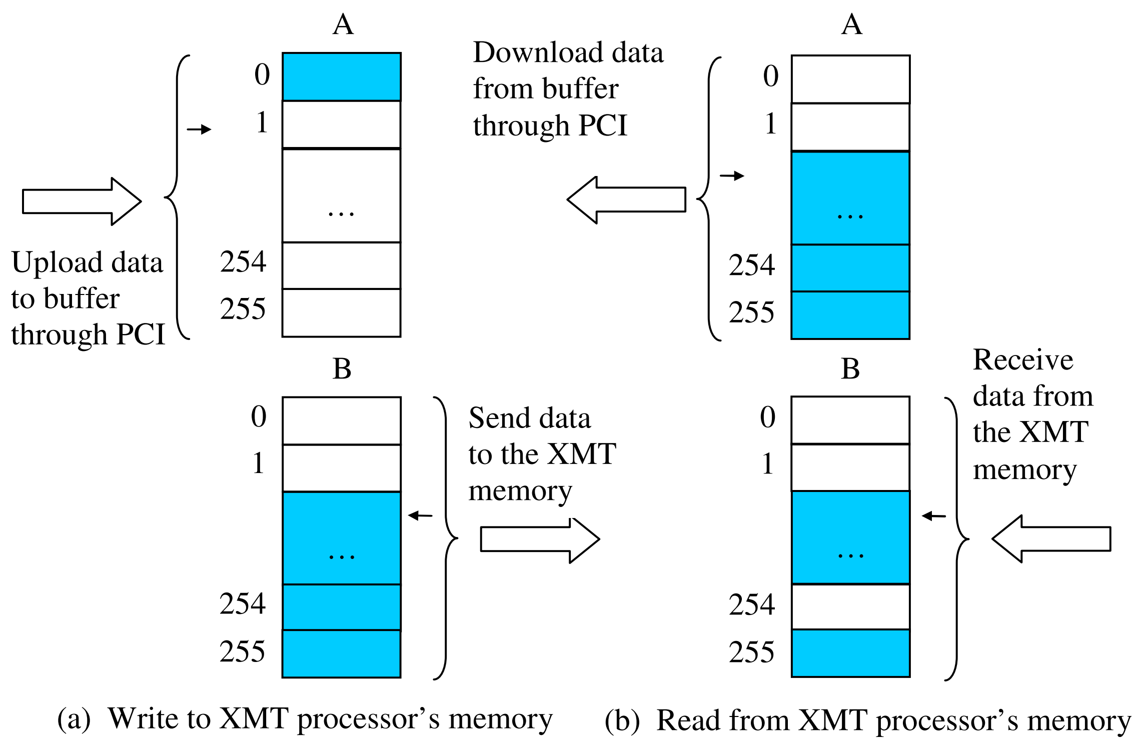

A
0
1
…
254
255
Upload data to buffer through PCI
B
0
1
…
254
255
Send data to the XMT memory
(a) Write to XMT processor's memory
A
0
1
…
254
255
Download data from buffer through PCI
B
0
1
…
254
255
Receive data from the XMT memory
(b) Read from XMT processor's memory

Figure G.1: Access the memory space of the XMT processor through the PCI bus

3. Wait until all data are received from the XMT processor. Note that the data may arrive out of order.

4. Read data from the buffer through the PCI interface.

Similar to the writing procedure, it is possible to overlap receiving and downloading of the data as shown in figure G.1 (b).

In verilog, these two buffers are actually lower and upper parts of a 512×32 memory. The MSB, bit 8, is used to choose which half to use. The depth of 512 is chosen to use exactly one BRAM (16Kb) in the Xilinx Virtex-4 FPGA.

G.1.2.2 Address Mapping for the Host Computer

The address mapping is listed in table G.2. Because the PCI data bus is 64-bit, the 3 LSBs are not used in the PCI operations.

Table G.2: Address mapping of pci_if

Addr.	Op.	Description
0x0	write	The address in the XMT processor memory space that will be read from or written to. This value will be changed by pci_if during sending requests to the XMT
0x0	read	The current address in the XMT processor memory space that pci_if is reading from or writing to
0x8	write	Notifying pci_if to send READ (from XMT memory) requests to the XMT processor. Also clears the read counter. the target_write_data[13:0] is used. 0 - defines which half buffer is the source. 9:1 - define number of word to read. 13:10 - which read commands to use, 0(READ) or 0xc(CACHE_PCM).
0x8	read	Returns the upper 20 bits of the request that pci_if sends to the XMT processor. 83:64 part of the cache request vector.

Continued on Next Page...

Addr.	Op.	Description
0x10	write	Notifying pci_if to send WRITE (to XMT memory) requests to the XMT processor. The target_write_data[9:0] is used. 0 - defines which half buffer is the destination. 9:1 - define number of word to write.
0x10	read	Undefined
0x18	write	Update the register which stores the 31:0 of the last valid response packet from the XMT processor
0x18	read	The 31:0 of the last valid response packet from the XMT processor
0x20	write	Clears cycle count register to 0. Also set reset and reset_icn of the XMT processor. The target_write_data[1:0] is used. bit 0 - reset_xmt, bit 1 - reset_icn
0x20	read	7:0 are defined. 0 - reset_xmt, 1 - reset_icn, 2 - stop signal from the XMT processor. Bits 7:3 represent current buffer state. one hot state, from bit 3 to bit 7: idle, sending read requests, waiting for response of read, first write request, sending write request.

Continued on Next Page. . .

Addr.	Op.	Description
0x28	write	Define which half of the buffer will be accessed by the host computer. The target_write_data[9:0] is used. 0 - defines which half buffer is accessed. 9:1 - initialize the pointer with this value (normally 0).
0x28	read	Read from buffers in the pci_if module. The starting address is determined by what is written to address 0x28. Every read operation automatically increase the pointer by 1, so, to read multiple words from the buffer, only need to read address 0x28 for multiple times.
0x30	write	Write to the buffer in the pci_if module. Every write operation automatically increase the pointer by 1, so, to write multiple words to the buffer, only need to write into address 0x30 consecutively. The starting pointer can be initialized by writing to address 0x28.
0x38	read	Read counter, showing how many response packets are received for this round (since 0x8 is written to start reading).
0x40	write	Write counter
0x40	read	Write counter

Continued on Next Page...

Addr.	Op.	Description
0x48	read	Bit 31 specifies which buffer is used for the XMT processor. Bit 30 specifies which buffer is accessed for host pc. Bit 29 is always 0. Bit 28:20 is the length of the current round of transfer. Bit 19:17 are always 0. Bit 16:8 transfer count(how many read or write requests are sent to XMT processor for this round. Bit 7:0 pointer of the buffer.
0x58	read	40-bit cycle count. Since XMT FPGA operates at 75MHz, the maximum execution time is 14660 seconds or slightly more than 4 hours.

G.2 Memory Controller

When a memory request has turned out to be a cache miss in a shared cache module, the cache line needs to be brought in from the off-chip DDR2 DRAM. The Paraleap has an on-chip DDR2 DRAM memory controller as shown in figure G.2.

Each parallel cache module is connected to a bank controller (BC) that handles cache miss requests. The eight BCs share one DDR2 DRAM channel. The DDR2

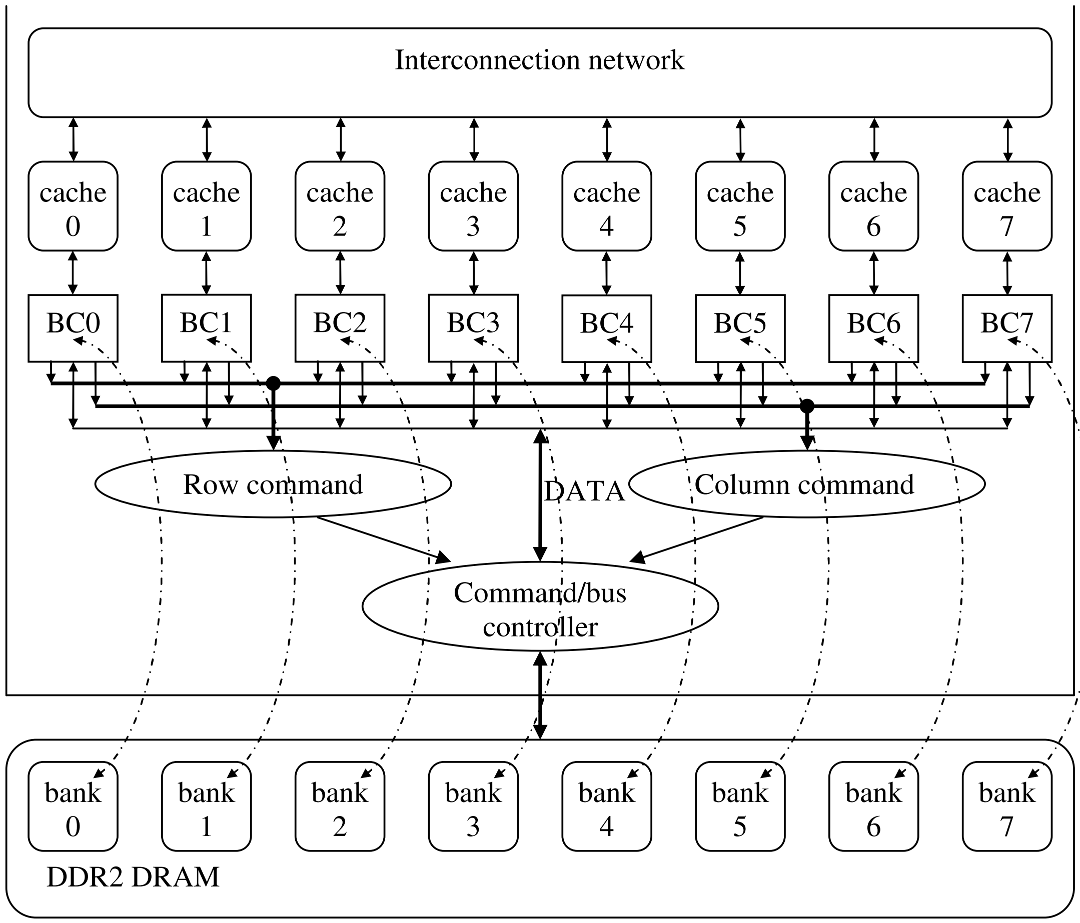


Figure G.2: Memory modules

DRAM module used in the Paraleap has exactly 8 banks and each of them is assigned to a shared cache and BC as shown with dotted lines in figure G.2.

G.2.1 DRAM Architecture and Organization

Before presenting the MC in the Paraleap, the basic concepts of DDR2 DRAM are briefly reviewed in this section as background. Each address in DRAM is specified by a triple – bank, row, and column – and DRAM is accessed like a three dimensional array. The DDR2 DRAM module used in the Paraleap, MT16HTF12864HY-

667, has 8 banks, 16k rows per bank, and 1k columns per row. Also, each bank has its own row buffer, the function of which is explained below.

Accessing an address in DRAM can require up to three operations. First, ACTIVE: a row in a bank must be copied into the bank's row buffer for the row to be accessed. Second, the READ/WRITE command accesses a column in the row buffer. Third, PRECHARGE: before another row in the same bank can be activated, the row buffer must be copied back into memory. In the event of a READ/WRITE to a new row, the bank must be first PRECHARGED and then the new row can be ACTIVATED before the READ/WRITE occur. But it should be noted that subsequent READ/WRITE commands to an active row require no additional operations.

The fourth important command of a DRAM is REFRESH. Periodical REFRESH is necessary for DRAM or it will lose its data. The micron MT16HTF12864HY-667 requires a REFRESH for every 7.8125μs.

G.2.2 Bank Controller (BC)

A bank controller accepts cache miss requests from the paired cache module and generates a sequence of DDR2 DRAM commands to access the off-chip DRAM module. A BC keeps the state of the corresponding bank and attempts to send commands through the shared command/data controller (CBC). Because the minimum clock rate for the MT16HTF12864HY-667 is 133MHz and the XMT FPGA operates at 75MHz, a 2x clock is used for the CBC. The CBC has two 150MHz cycles for every cycle of 75MHz. For convenience of bus arbitration, the CBC processes the column commands only in the cycle where two positive edges (75MHz and 150MHz) occur. READ and WRITE commands are column commands and they are using the DRAM data bus. ACTIVE, PRECHARGE, and REFRESH are not using the data bus and they are row commands.

The DRAM commands are sent in the following order:

- If the bank is not active, send the ACTIVE command
- If the bank is active but the row does not match with the request, then send PRECHARGE command
- READ or WRITE command depending on request

The REFRESH command operates on all banks and all banks need to be in the idle state. When a REFRESH command is necessary, the CBC will request all banks to remain in or change to the idle state. If a row in a bank is opened (ACTIVE), then a PRECHARGE command will be sent to change the bank to the idle state. After all banks are ready for a REFRESH command, the CBC will send a REFRESH command to the DRAM.

There are many time constraints on the DRAM operations. Generally, every DDR2 DRAM command takes multiple cycles. Especially, opening or closing a row is expensive. Also, switching from READ to WRITE – effectively changing the direction of traffic – also requires some extra cycles. The exact numbers of cycles(150MHz) are listed below for the micron MT16HTF12864HY-667 module.

Table G.3: Timing constraints for MT16HTF12864HY-667

Symbol	Parameter	value (ns)	cycles 150MHz
t_{RAS}	ACTIVE to PRECHARGE	40	6
t_{RCD}	ACTIVE to READ/WRITE	12	2
t_{RC}	ACTIVE to ACTIVE(same bank)	55	9
t_{RRD}	ACTIVE to ACTIVE(different bank)	7.5	2
CL	READ to the first data	-	3
WL	WRITE to the first data	CL-1	2
t_{RTP}	READ to PRECHARGE	7.5	2
t_{WTR}	WRITE to READ	7.5	2
t_{WR}	WRITE to PRECHARGE (write recovery)	15	3
t_{RP}	PRECHARGE period	12	2

Because of the timing constraints, BCs have to keep the banks' state. For example, a row can be ACTIVATED only if the bank has been PRECHARGED. The following registers are used to keep bank information.

- bk_state, 13 bits, only 1 bit can be 1 and there are 13 state.
- rowStatus, 15 bits, if bit 14 is 1, a row is open and the row number is in the bit field 13:0
- bk_state_counter, 8 bits, cycle count since the bank entered this state, increase

1 at 2x clock

- bk_last_act, 4 bits, cycle count since last ACTIVE command to this bank, increase 1 at 2x clock

A BC makes sure all timing constraints are met before a command is sent to the CBC using current bank state and global information from the CBC. The following information will be broadcasted from the CBC to all banks.

- ACTIVE

  indicating whether a ACTIVE command is permitted. Constrained by t_{RRD}

- READ

  indicating whether a READ command is permitted. Not permitted when data bus is used or will be used for WRITE commands.

- WRITE

  indicating whether a WRITE command is permitted. Not permitted when data bus is used or will be used for READ commands.

- REFRESH request

  request of REFRESH will be presented.

- REFRESH commit

  notifying banks about the commitment of a REFRESH command.

- bus direction

  indicating current direction of data movement.

- last DRAM command

  this is monitored by banks to confirm that a posted command has been committed.

The figure G.3 shows the block diagram of a bank controller. Each BC has eight buffers for request from a cache module. A stack tracks the usage of buffers, because the requests in these eight buffers are not processed in the order they arrived. The reordering of the requests is to achieve a better off-chip memory bandwidth utilization and it is described after this brief overview of BC. There are two IDs in bank controller and they need to be clarified. The *request ID* is the ID assigned by a cache module to distinguish multiple pending requests, which also indicates which queue in the *PendingRqst* is used for this cache line. The *request buffer ID* indicates which one among the eight request buffers in the BC is used for the request.

When a cache module send a read request to its BC, the request will be saved in one of the eight request buffers, which is allocated by the empty buffer ID stack. When the read request is selected by the BC, the request ID (buffer ID of *PendingRqst* in cache module) of the read request will be pushed in the ReadID FIFO and it will be paired with the data from DRAM. The FIFO is used because the data will arrive at the BC a few cycles later and the latency can vary. After the data from DRAM is paired with the request ID, it can be delivered to the cache who requested this cache line.

While a read request from a cache has only one address packet, a write request has a total of nine packets: one address packet followed by eight data packets.

Similar to read request, the address packet goes to one of the eight buffers allocated by the empty buffer ID stack. The eight data packets will be saved into a register file (Write buffer) using the request buffer ID of the address packet and packet sequence number. When the write request is processed by the BC, the buffer ID is used to retrieve data from the buffer. For the same reason as in the ReadID FIFO, a WriteID FIFO is used to keep the ID number of the request. Note that the write port and read port of the Write buffer have a different width: 32-bit for write and 256-bit for read. It takes 8 cycles to store a cache line, but only one cycle to read.

After a BC processes a request, the corresponding buffer ID will be pushed into the empty buffer ID stack, so the buffer will be reused. The BC also checks for RAW (Read After Write) hazards, and the read request that would cause RAW will be blocked until the write request is processed by the BC.

The BC reschedules the requests to achieve better bandwidth utilization and less overall latency. As described in section G.2.1, a typical DRAM access needs 3 commands: ACTIVE, READ/WRITE, and PRECHARGE. When a row is opened(ACTIVE), multiple column requests (READ or WRITE) can be sent to DRAM. This will result in less commands and achieve better performance. When a DRAM switches the direction of the data movement in the bus, the data bus cannot be used during the transition. The BC in the Paraleap attempts to minimize both the row open/close operations and bus direction changes. The BC has a request buffer that holds 8 commands, this allows it to look ahead 8 steps and find the best one. The bank

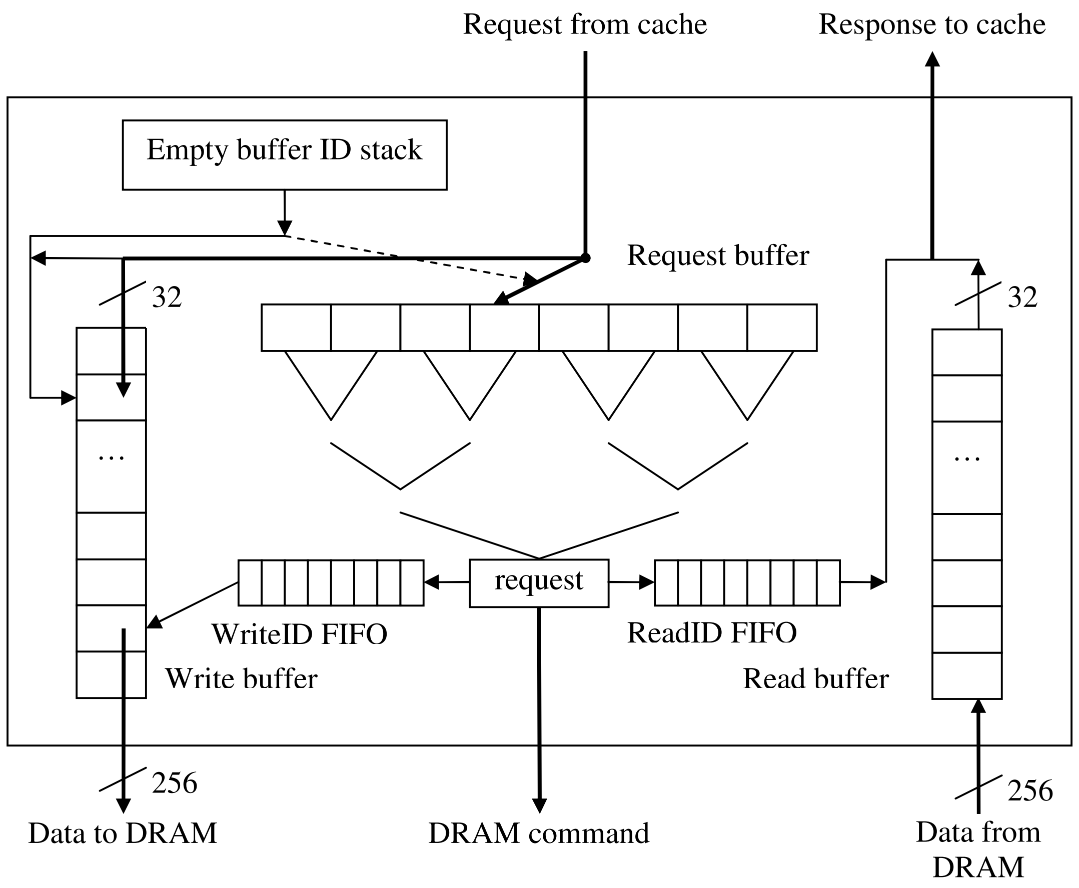


Figure G.3: Bank controller

controller chooses 1 command out of the 8 by three rounds of binary comparison. The algorithm for binary comparison operates as follows: at the first level where one command is deemed better than another, the algorithm stops. Two commands will only be compared at step n if they are equivalent for the $n-1$ previous comparisons.

1. Take a valid request over an invalid request.
2. Take a request which has been in the buffer for a very long time (more than a certain number of cycles).
3. Take a request which matches the current traffic direction over a request which

goes against traffic.

4. Take a request to the current open row over a request to an inactive row.

5. Take a row that has more pending commands. Two requests to the same row are only considered as pending if both are in the same traffic direction.

6. Take the request that was received earlier.

G.2.3 Command and Bus Controller(CBC)

The eight BCs send DDR2 DRAM commands to the CBC and the CBC sends these commands to the off-chip DRAM. The functions of the CBC are:

- DDR2 DRAM initialization
- posting DRAM commands from BCs
- periodical DRAM refreshing
- presenting data to the DRAM bus for writing
- collecting data from the DRAM for reading
- providing BCs with global information

G.2.3.1 Interface of the CBC

The interface of the CBC is listed in table G.4 and G.5. The brief description of each port is also listed in tables.

Table G.4: interface of the command/bus controller (BC side)

Name	Width	Port	Description
clk_low	1	input	75Mhz clock signal
clk_in	1	input	150Mhz clock signal, pos edge aligned with clk_low
clk_90	1	input	150MHz, 90 degree a head of clk_in
clk_180	1	input	150MHz, 180 degree a head of clk_in
clk_90_n	1	input	150MHz, 270 degree a head of clk_in
bks_cmd	184	input	DRAM commands from BCs
bks_busy	8	input	busy banks will present 1 at corresponding bit, not used now
bks_open	8	input	Used for making sure the banks are ready for refresh, need to be all 0
ack_bk_cmd	8	output	acknowledgement to the DRAM commands from BCs
mem_access_r	8	output	the corresponding bit for a BC will be 1 if the data is presented in the bus for read from DRAM

Continued on Next Page...

Name	Width	Port	Description
mem_access_w	8	output	the corresponding bit for a BC will be 1 if the data will be consumed for a write to DRAM
write_bk_id	3	output	this is used to select a 256-bit data bus from 8 BCs
rqst_r	8	input	each bit indicating whether there is a read request from corresponding BC
rqst_w	8	input	each bit indicating whether there is a write request from corresponding BC
switch_dir_ok	8	input	each bit indicating whether the corresponding BC is ready for a traffic direction change
globalInfo	7	output	Information about DRAM state. bit 0 to 2 : indicate whether active , read, and write is permitted. bit 3: refresh request to all banks. bit 4: refresh command has been committed. bit 5: the current traffic direction in the DRAM data bus. bit 6: not used

Continued on Next Page…

Name	Width	Port	Description
write_data	256	input	the data will be written to DRAM, selected from BCs using write_bk_id
read_data	256	output	the data from DRAM will be broadcasted and mem_access_r is used to specify the destination BC
mem_cmd_info	8	output	the last DRAM command committed. bit 7: valid, bit 6-4 bank ID, bit 3-0 DRAM command

G.2.3.2 How Does the CBC Work?

The DDR2 DRAM needs to be initialized and the steps are defined in the chip manual. The CBC sets CL as 3 and burst length for 4.

The CBC operates at twice the rate of the internal clock. For convenience of bus control, the column commands are processed in cycles where the positive edges of 2× and 1× clocks overlap. Name this cycle as column command cycle and the other as row command cycle. During column command cycle, a row command can be sent to the DRAM if there is no column command. The CBC uses a 36-bit register, dq_reserve, to track the bus usage for column commands. An associated register bk_reserve stores the 3-bit bank IDs for column commands.

Table G.5: interface of the command/bus controller (DRAM side)

Name	Width	Port	Description
Cal_clk	1	input	200MHz reference clock for IDELAY in Xilinx Virtex-4 FPGA
Clk	1	output	clock for DRAM, 150MHz
Clkn	1	output	negative of Clk
CKE	2	output	should keep high for normal DRAM operation
CSn	2	output	chip select, low effective
RASn	1	output	row access strobe
CASn	1	output	column access strobe
WEn	1	output	write enable
DM	8	inout	not used in
BA	2	output	bank address
A	14	output	address pins
DQ	64	inout	data bus, tri-state bus
DQS	8	inout	data strobe, tri-state bus
DQSn	8	inout	negative of DQS, tri-state bus
ODT	2	input	on die termination

The CBC also decides when to switch the traffic direction in the DRAM data bus. Each BC presents the current request type through *rqst_r* and *rqst_w* and the CBC will change when a direction change is needed. To minimize the number of direction changes, the CBC will only change the direction when multiple BCs request a change or any of the BC has been waiting for a change for a long time. This behavior prevents starvation while reducing the number of direction changes.

G.3 Other Differences in XMT FPGA Prototype

- Instruction Buffer for the MTCU

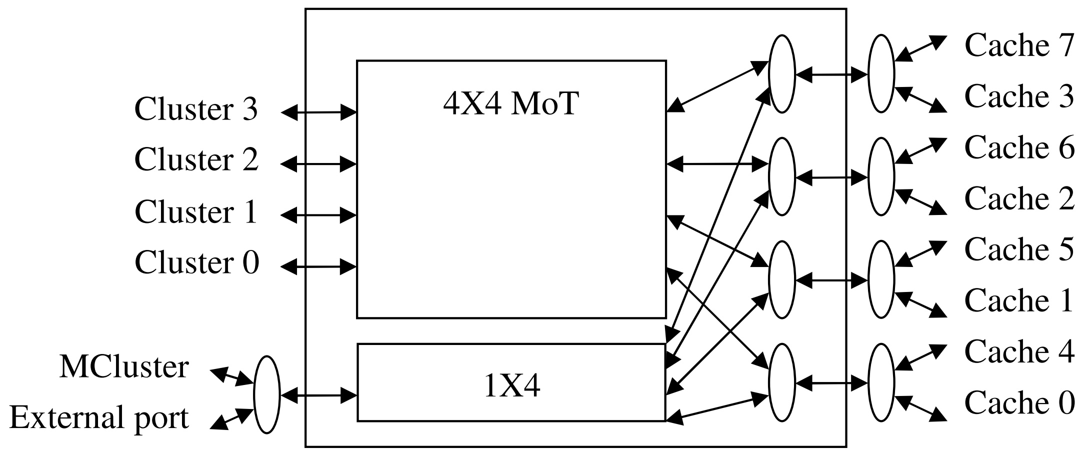
  The maximum size of the XMTC program is limited by the size of the instruction buffer in the MTCU. The maximum program size for the XMT FPGA prototype is 64KB, while it is 16KB for the XMT ASIC.

- Instruction Buffer for TCUs

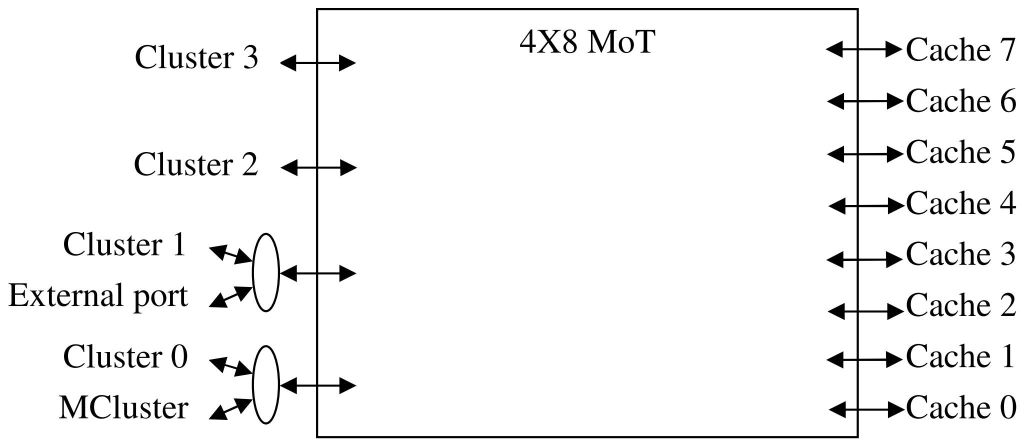
  The size of each instruction buffer in the clusters is different. XMT FPGA prototype is 4KB and can hold up to 1024 instructions, however, the XMT ASIC has half of it, 2KB or 512 instructions.

- Interconnection Network

  Figure G.4 shows the differences in the XMT ASIC and the XMT FPGA prototype. In the XMT ASIC, separate trees are used for MCluster and external ports as shown at (a). This will reduce the latencies of cache accesses from MCluster and external ports. In the XMT FPGA prototype, the FPGA IO pins are limited and cannot afford to have a dedicated port for MCluster as in the XMT ASIC. Therefore the MCluster and Cluster 0 share one port and the External port shares with Cluster 1.



(a) Interconnection network in XMT ASIC



(b) Interconnection network in XMT FPGA prototype

Figure G.4: Interconnection network topology

Appendix H

Verilog Module Hierarchy

Figure H.1 on the following page lists module names, and normally they are defined in files with the same name and .v extension. Modules whose file name is different from the module name are followed by a number. (1) block.v (2) pre-PAR module (3) mc_if.v (4) Synopsys DesignWare library module.

```
xmt_s-|-reset_cell
      |-xmt_part1-|-mcluster---|-mtcu------------------|-micache
      |           |            |                       |-mpc------------|-rfile
      |           |            |                       |                |-tcu_pcm
      |           |            |                       |-ls_addr
      |           |            |                       |-mcache---------|-mcache_rf
      |           |            |                       |                |-mtagBRAM
      |           |            |                       |                |-cacheline_rqst
      |           |            |                       |-FuncALU
      |           |            |                       |-FuncSFT
      |           |            |                       |-FuncBR
      |           |            |                       |-FuncMul
      |           |            |                       |-FuncSJ
      |           |            |                       |-packet_split
      |           |            |-ps_request_or(1)
      |           |            |-GRs
      |           |            |-ps_unit(1)------------|-pstree_B(1)
      |           |-simpPipe
      |           |-cluster----|-tcu-------------------|-pc-------------|-rfile
      |           |            |                       |                |-pc_monitor
      |           |            |                       |-prefetchTCU
      |           |            |                       |-ls_addr
      |           |            |                       |-FuncALU
      |           |            |                       |-FuncSFT
      |           |            |                       |-FuncBR
      |           |            |                       |-FuncPS
      |           |            |                       |-MultiClockBuf
      |           |            |-ps_cluster(1)
      |           |            |-FuncMul---------------|-hi_low
      |           |            |                       |-div_mtcu
      |           |            |                       |-DW_div_pipe(4)
      |           |            |                       |-DW02_mult_5_stage(4)
      |           |            |-addrHash
      |           |            |-read_only_buf---------|-ntagBRAM_RAW
      |           |            |                       |-nRequestor
      |           |            |                       |-mcache_rf
      |           |            |                       |-seqRecord
      |           |            |                       |-MultiClockBuf
      |           |            |                       |-addrRecord
      |           |            |-sahred_icache
      |           |            |-mul_tree(1)-----------|-MultiClockBuf
      |           |            |                       |-CombArbitor
      |           |            |-SimpPath
      |           |            |-stree(1)--------------|-MultiClockBuf
      |           |            |                       |-CombArbitor
      |           |            |                       |-Shrink_E(1)
      |           |            |-pkt_split
      |           |            |-ps_request_cluster(1)
      |           |-external_port
      |           |-MultiClockBuf
      |           |-part1_logic
      |           |-pkt_arbitor
      |           |-reset_cell
      |
      |-xmt_part2-|-icn(2)
                  |-reset_cell
                  |-icn_split_i
                  |-icn_split_o
                  |-mem_access(3)-|-pkt_combine
                  |               |-reset_cell
                  |               |-level_one_cache-------|-pendQueL1
                  |               |                       |-MultiClockBuf
                  |               |                       |-CombArbitor
                  |               |                       |-pendingRqsts
                  |               |                       |-cache_pcm
                  |               |                       |-cache_rf
                  |               |                       |-next_rf
                  |               |                       |-dirty_rf
                  |               |                       |-Cache1RF
                  |               |                       |-tagBRAM
                  |               |                       |-tagBRAM_RAW
                  |               |
                  |               |-mc_if(3)--------------|-bkController(3)|-MultiClockBuf
                  |               |                       |
                  |               |                       |-rqst_que(3)----|-MultiClockBuf
                  |               |                       |                |-CombArbitor
                  |               |                       |                |-wr_que(3)
                  |               |                       |                |-IO_buf_send
                  |               |                       |
                  |               |                       |-Rsponse_buf(3)-|-IO_buf_receive
```

Figure H.1: Verilog Module Hierarchy

Appendix I

XMT ISA

instruction	instruction format					
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
j target	0x02	target				
jal target	0x03	target				
lui rt,imm	0xf	0	rt	imm		
jr rs	0	rs	0			8
jalr rs,rd	0	rs	0	rd	0	9
mvfp rt,rp,imm	0x1c	rp	rt	imm		
mvtp rp,rs,imm	0x1d	rs	rp	imm		
sflush	0x2d	don't care				
sll rd,rt,shamt	0	0	rt	rd	shamt	0
srl rd,rt,shamt	0	0	rt	rd	shamt	2
sra rd,rt,shamt	0	0	rt	rd	shamt	3
sllv rd,rt,rs	0	rs	rt	rd	0	4
srlv rd,rt,rs	0	rs	rt	rd	0	6
srav rd,rt,rs	0	rs	rt	rd	0	7
getid rd, rs	0x19	rs	rt	don't care		

Continued on Next Page...

instruction	instruction format					
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
add rd,rs,rt	0x00	rs	rt	rd	0	0x20
addi rt,rs,imm	0x08	rs	rt	imm		
addu rd,rs,rt	0	rs	rt	rd	0	0x21
addiu rt,rs,imm	0x09	rs	rt	imm		
sub rd,rs,rt	0	rs	rt	rd	0	0x22
subu rd,rs,rt	0	rs	rt	rd	0	0x23
and rd,rs,rt	0	rs	rt	rd	0	0x24
andi rt,rs,imm	0x0c	rs	rt	imm		
or rd,rs,rt	0	rs	rt	rd	0	0x25
ori rt,rs,imm	0x0d	rs	rt	imm		
nor rd,rs,rt	0x27	rs	rt	rd	0	0x27
xor rd,rs,rt	0	rs	rt	rd	0	0x26
xori rt,rs,imm	0x0e	rs	rt	imm		
slt rd,rs,rt	0	rs	rt	rd	0	0x2a
slti rt,rs,imm	0x0a	rs	rt	imm		
sltu rd,rs,rt	0	rs	rt	rd	0	0x2b
sltiu rt,rs,imm	0x0b	rs	rt	imm		
beq rs,rt,label	0x04	rs	rt	offset		
bne rs,rt,label	0x05	rs	rt	offset		

Continued on Next Page…

instruction	instruction format					
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
chkid rs	0x18	rs	0	0xffff		
blez rs,label	0x06	rs	0	offset		
bgtz rs,label	0x07	rs	0	offset		
bltz rs,label	0x01	rs	0	offset		
bltzal rs,label	0x01	rs	0x10	offset		
bgez rs,label	0x01	rs	1	offset		
bgezal rs,label	0x01	rs	0x11	offset		
mult rs,rt	0	rs	rt	0		0x18
multu rs,rt	0	rs	rt	0		0x19
div rs,rt	0	rs	rt	0		0x1a
divu rs,rt	0	rs	rt	0		0x1b
mfhi rd	0	0		rd	0	0x10
mthi rs	0	rs	0			0x11
mtlo rs	0	rs	0			0x13
mflo rd	0	0		0	rd	0x12
lw rt, imm(rs)	0x23	rs	rt	imm		
sw rt, imm(rs)	0x2b	rs	rt	imm		
lwbuf rt, imm(rs)	0x27	rs	rt	imm		
psm rt, imm(rs)	0x17	rs	rt	imm		

Continued on Next Page...

instruction	instruction format					
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
pref hint,imm(rs)	0x33	rs	hint	imm		
swp rt, imm(rs)	0x2c	rs	rt	imm		
ps rt, gr	0x16	gr	rt	0		
spawn	0x14	0				
join	0x15	0				
broadh rt, rs	0x34	rs	rt	0		
broadl rt, rs	0x35	rs	rt	0		
mvfg rt, gr	0x1a	gr	rt	0		
mvtg gr, rs	0x1b	rs	gr	0		
halt	0x3f	0				

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, 1990.

[3] E. Anderson, J. Brooks, C. Grassl, and S. Scott. Performance of the CRAY T3E multiprocessor. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17, New York, NY, USA, 1997. ACM.

[4] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through EPI throttling. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department University of California, Berkeley, 2006.

[6] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, and C. Lichtenau. Building the 4 processor SB-PRAM prototype. In *Proceedings of the 30th Hawaii International Conference on System Sciences: Advanced Technology Track - Volume 5*, 1997.

[7] A. O. Balkan, M. N. Horak, G. Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *HOTI '07: Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*, pages 21–28, Washington, DC, USA, 2007. IEEE Computer Society.

[8] A. O. Balkan, G. Qu, and U. Vishkin. Arbitrate-and-move primitives for high-throughput on-chip interconnection networks. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2004.

[9] A. O. Balkan, G. Qu, and U. Vishkin. A mesh-of-trees interconnection network for single-chip parallel processing. In *ASAP '06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 73–80, Washington, DC, USA, 2006. IEEE Computer Society.

[10] A. O. Balkan, G. Qu, and U. Vishkin. An area-efficient high-throughput hybrid interconnection network for single-chip parallel processing. In *IEEE 45th Design Automation Conference (DAC)*, Anaheim, CA, USA, 2008.

[11] E. Berkovich, J. Nuzman, M. Franklin, B. Jacob, and U. Vishkin. XMT-M: A scalable decentralized processor. Technical Report CS-TR-4061, University of Maryland Institute for Advanced Compuer Studies, 1999.

[12] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[13] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[14] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc, 1998.

[15] S. Dascal and U. Vishkin. Experiments with list ranking for explicit multi-threaded (XMT) instruction parallelism. *J. Exp. Algorithmics*, 5:10, 2000.

[16] A. Formella, J. Keller, and T. Walle. Hpp: A high performance PRAM. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 425–434, London, UK, 1996. Springer-Verlag.

[17] L. Geppert. Sun's big splash niagara microprocessor chip. *IEEE Spectrum*, 42:56–60, 2005.

[18] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: accounting for contention in parallel algorithms. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 638–648, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[19] J. R. Goodman and P. J. Woest. The wisconsin multicube: a new large-scale cache-coherent multiprocessor. *SIGARCH Comput. Archit. News*, 16(2):422–431, 1988.

[20] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing an mimd shared memory parallel computer. *IEEE Trans. Computers*, 32(2):175–189, 1983.

[21] M. Gotuaco, P. Huebler, H. Ruelke, C. Streck, and W. Senninger. Implementation of CVD low-k dielectrics for high-volume production. *Solid State Technology*, 47:60–62, 2004.

[22] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extermely fine-grained chip multiprocessor. *Journal of Embedded Computing, Special Issue on Embedded Single-Chip Multicore Architectures and Related Research - from System Design to Application Support*, 2006.

[23] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The stanford hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[24] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[25] P. Heidelberger, A. Norton, and J. T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Trans. Comput.*, 39(1):133–138, 1990.

[26] H. P. Hofstee. Power efficient processor architecture and the cell processor. *hpca*, 00:258–262, 2005.

[27] J. JáJá. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[28] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[29] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[30] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):162–173, 2007.

[31] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2006. ACM.

[32] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical constraints. In *High-Performance Computer Architecture*, 2006.

[33] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. *SIGARCH Comput. Archit. News*, 28(2):161–171, 2000.

[34] D. Naishlos, J. Nuzman, C. Tseng, and U. Vishkin. Evaluating multi-threading in the prototype XMT environment, 2000.

[35] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Evaluating the XMT parallel programming model. In *HIPS '01: Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 95–108, London, UK, 2001. Springer-Verlag.

[36] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. *Theory of Computing Systems. Springer*, 36:521–552, 2003.

[37] D. A. Patterson and J. L. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[38] W. J. Paul, P. Bach, M. Bosch, J. Fischer, C. Lichtenau, and J. Röhrig. Real PRAM programming. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 522–531, London, UK, 2002. Springer-Verlag.

[39] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge. The limits of instruction level parallelism in spec95 applications. *SIGARCH Comput. Archit. News*, 27(1):31–34, 1999.

[40] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA*, pages 128–138, 2000.

[41] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Comput. Archit. News*, 31(2):422–433, 2003.

[42] Y. Shiloach and U. Vishkin. An o(n2 log n) parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, 1982.

[43] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[44] U. Vishkin. From algorithm parallelism to instruction-level parallelism: an encode-decode chain using prefix-sum. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 260–271, New York, NY, USA, 1997. ACM.

[45] U. Vishkin. A no-busy-wait balanced tree parallel algorithmic paradigm. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 147–155, New York, NY, USA, 2000. ACM.

[46] U. Vishkin. Two techniques for reconciling algorithm parallelism with memory constraints. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 95–98, New York, NY, USA, 2002. ACM.

[47] U. Vishkin, G. Caragea, and B. Lee. *Handbook of Parallel Computing: Models, Algorithms and Applications*, chapter Models for advancing PRAM and other algorithms into parallel programs for a PRAM-On-Chip platform. CRC press, 2007.

[48] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (XMT) bridging models for instruction parallelism. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures SPAA*, 1998.

[49] D. W. Wall. Limits of instruction level parallelism. Technical Report WRL-93-6, HP labs, 1993.

[50] X. Wen and U. Vishkin. PRAM-On-Chip: first commitment to silicon. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 301–302, New York, NY, USA, 2007. ACM.

[51] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-On-Chip processor. In *ACM Computing Frontiers, Ischia, Italy, May 5-7*, 2008.

[52] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[53] Micron http://www.micron.com/ddr2. DDR2 SDRAM data sheet.