Asymmetric Coherent Configurable Caches for PolyBlaze Multicore Processor

by

Ziaeddin Jalali
B.Sc., Sharif University of Technology, 2010

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Applied Science

In the
School of
Engineering Science

SIMON FRASER UNIVERSITY
Fall 2014

APPROVAL

Name:	Ziaeddin Jalali
Degree:	Master of Applied Science
Title of Thesis:	Asymmetric Coherent Configurable Caches for PolyBlaze Multicore Processor
Examining Committee:	**Dr. Ivan Bajic** Associate Professor, School of Engineering Science Chair
	Dr. Lesley Shannon, P.Eng. Associate Professor, School of Engineering Science Senior Supervisor
	Dr. Alexandra Fedorova Associate Professor, School of Computing Science Supervisor
	Dr. Fabio Campi Lecturer, School of Engineering Science Internal Examiner
Date Defended:	28 November 2014

Partial Copyright Licence

Abstract

Modern computing systems gain performance by several means such as increased parallelism through using Chip-level Multiprocessor (CMP) systems. Symmetric Multiprocessor (SMP) systems use uniform processing cores to form a CMP in which all cores are identical in every aspect. Conversely, Asymmetric Multiprocessor (AMP) systems consist of processing cores with variable configurations such as different cache configurations, co-processors, and cache sizes. AMP systems coupled with such smart scheduling algorithms can improve resource utilization while maintaining overall system performance because real-time profiling in a computing system using light-weight hardware profilers can help smart scheduling algorithms make meaningful decisions. In other words, the vision into an application's behavior helps in the decision making process on how to allocate available resources for different applications without penalizing the performance by putting too much overhead on the system. Currently, there is no AMP research framework available that allows us to look into asymmetry in processing systems.

In this thesis, we present an extension on PolyBlaze framework for asymmetric coherent Level-1 (L1) caches. Our implementation in this work includes other arbiter and prefetching units as well. We measure data cache read miss rates and application run-times for select benchmarks from SPEC CPU2006 executed in a Linux environment on top of a variety of cache configurations. In the scope of this work, we manually assign applications to cores to take advantage of AMP configurations. Our results show that in a AMP system, different applications can benefit from various configurations to complete their work faster using less resources.

To Yasaman.

Acknowledgments

First, I would like to thank my supervisor, Dr. Lesley Shannon who patiently provided me guidance, support, and encouragement, particularly when the project was not going well. My main takings from this degree are the many lessons I learned from her.

Additionally, I would like to thank Dr. Alexandra Fedorova, Dr. Fabio Campi, and Dr. Ivan Bajic for serving on my defense committee.

Also, I want to thank my friends and colleagues in the RCL lab, especially Eric Matthews and Nicholas Doyle who helped, supported and answered my many questions. I had the most wonderful time there.

Finally, to my lovely wife, Yasaman, and my parents whose support throughout this journey was priceless. I will never be able to thank you enough.

Contents

List of Tables

List of Figures

Glossary

ABACUS hArdware Based Accelerator for Characterization of User Software. 13, 29, 34, 40, 63, 64, 69, 78

AMP Asymmetric Multiprocessor. iv, 1–3, 6, 27, 28

BRAM Block RAM. 10, 31, 37, 38, 50–61

CLB Configurable Logic Block. 10

CMP Chip-level Multiprocessor. iv, 1

COMA Cache-Only Memory Architecture. 7

DMA Direct Memory Access. 25, 26

DSP Digital Signal Processing. 10

FF Flip-Flop. 10

FIFO First-In-First-Out. 23, 35, 41, 47, 48, 66, 67

FPGA Field Programmable Gate Array. viii, 2, 3, 10–14, 26, 28, 37, 51, 62, 63, 67

FPGAs Field Programmable Gate Arrays. 5

FPU Floating Point Unit. 11, 14, 27

FSL Fast Simplex Link. 23, 48

GTS Global Task Scheduling. 28

I/O Input/Output. 6, 10

IC Integrated Circuit. 10

IP Intellectual Property. 10

ISA Instruction Set Architecture. 1, 3, 6, 28

L1 Level-1. iv, viii, xii, 3, 4, 7, 9, 27, 30, 35–38, 41, 42, 44–50, 57, 60, 61, 63, 66–68, 78

L2 Level-2. viii, xii, 7, 23–25, 32, 33, 37, 44–48, 58, 60, 62, 63, 66–68, 78, 79

LCC Library Cache Coherence. 9, 10

LMB Local Memory Bus. 18, 20, 23, 52, 53, 55, 60

LRU Least Recently Used. 32, 39, 56–60, 63, 64, 69, 70, 72–74

LUT Look-up Table. 10, 31, 38, 51–61

LWX Load Word Exclusive. 17, 18, 25, 32

MMU Memory Management Unit. 13, 14, 17, 20, 22, 42
MPMC Multi-Port Memory Controller. 23, 35, 41, 44, 48, 54, 57, 63, 65–67
MSR Machine Status Register. 18

NPI Native Port Interface. xii, 44, 48, 57, 58, 60, 63, 67, 68
NUMA Non-uniform Memory Access. 7

OS Operating System. 2, 3, 6, 7, 12, 13, 28, 29, 38, 43, 64, 78, 79

PBML PolyBlaze Memory Link. xii, 4, 33–37, 40, 41, 44–46, 48, 57, 60, 61, 63, 66, 67
PLB Processor Local Bus. 23, 52, 53, 55
PVR Processor Version Register. 13

RAMP Research Accelerator for Multiple Processors. 11
RISC Reduced Instruction Set Computing. 13, 14

SMP Symmetric Multiprocessor. iv, 1, 6, 13, 28
SOCs Systems-on-Chip. 5
SWX Store Word Exclusive. 17, 18, 25, 26, 32, 36, 47

TLB Translation Look-aside Buffer. 17

UMA Uniform Memory Access. 7

WDC Write to Data Cache. 22, 31, 34, 35, 40
WIC Write to Instruction Cache. 20, 40, 41

XCL Xilinx CacheLink. 16, 18, 20, 23, 35, 41, 48, 54, 57, 61

1 Introduction

Over the past decades, performance in modern computing systems has increased by several means including increasing operating frequencies. However, as processor frequencies reach their limits, a popular solution to increase performance has been adding more processors to increase parallelism. Today, a Chip-level Multiprocessor (CMP) provides two or more processing cores on a single chip that permit even greater performance due to reduced memory latency.

CMPs fall into two main categories: Symmetric Multiprocessor (SMP) and Asymmetric Multiprocessor (AMP). A SMP architecture comprises identical cores with equivalent frequencies, Instruction Set Architecture (ISA), cache sizes, functions, etc. Processing cores in a AMP architecture still have a common ISA, but they can utilize different configurations in their caches (different cache sizes, replacement policies, cache-line sizes, etc.), different operating frequencies, and reduced instruction support (e.g. no floating point operations).

Research on AMP systems is difficult as they are not widely available. Most commercial processors are often SMP with different cores possibly clocking at different rates. Moreover, detailed information about the internal architecture in AMP systems is sometimes proprietary. Also, these systems are not necessarily configurable. Therefore, system research has to rely on high-level information gathered from limited resources available about real-world behaviour of these processors. To solve this problem, we need a framework that provides configurable processor and cache architecture.

1.1 Motivation

An Operating System (OS) uses different methods to balance its workload based on the demands of running applications and available resources (processors, their local cache memories, and the amount of physical memory). For instance, when assigning tasks to available processors, it is important to balance the workload as assigning too many tasks to a processor will impose an overload from context-switches; conversely, assigning not enough tasks would waste processor cycles while the processor is idle. Maintaining this balance is even more complicated in an AMP environment since some tasks can utilize certain resources better than others, e.g. some applications can utilize processor cache better than others. In such cases, analyzing system behavior in presence of multiple tasks requires insight as to the nature of the current tasks and the availability of resources in the system.

Modelling an AMP system on a FPGA can provide a rich framework for more research in this area. Having access to such a system with fully configurable caches allows the user to gather lots of data on complex interactions while having negligible impact on the software. Achieving this goal is not necessarily feasible on commercial processors or system simulators. Commercial multicore processors often include hardware registers that can be used to measure different events in the life time of an application. For instance, by measuring the number of cache hits and memory requests, we can obtain the hit ratio of an application. However, the number of hardware registers in commercial processors is limited and the measurements have to be taken from multiple independent executions of the application. Conversely, system simulators would execute the application in an emulated environment and capture the required information while running the application. Therefore, they can give us as much visibility as we need in the runtime of an application, but they impose a lot of overhead.

Developing an AMP framework comprising cores with identical ISA and asymmetric configurable caches that supports an OS, provides real-time visibility into how the system is actually working. Therefore, it allows us to investigate cache asymmetry and its impact on a multiprocessor system.

1.2 Objective

PolyBlaze is a configurable research framework for systems research [2]. In this thesis, our objective is to develop an asymmetric Level-1 (L1) cache extension for the PolyBlaze research framework. In order to achieve this goal, we need modules such as configurable caches with a built-in coherency mechanism. Arbitration modules are also needed to provide the interconnection between processors and main memory and support the coherency mechanism. Additionally, implementing optional prefetching units can provide better insight into this AMP system. Since PolyBlaze is a FPGA system, we can create the whole hardware system with the desired configuration and run the OS on this platform. Then we can execute benchmark applications such as SPEC CPU2006 [3] to measure the impact of different configurations (symmetric and/or asymmetric) on the overall system performance.

1.3 Contributions

The main contribution of this thesis is the development of the necessary hardware infrastructure for the PolyBlaze framework to support L1 caches that may be asymmetrically configured. The proposed infrastructure addresses the necessary requirements for integrating caches into the PolyBlaze framework. Our cache infrastructure is also designed to be as configurable and scalable as possible. The following modules are presented in this thesis:

- Configurable asymmetric level-1 data and instruction caches,
- L1 arbiter,
- PolyBlaze Memory Link (PBML), and
- Prefetching units.

1.4 Thesis Organization

The remainder of the thesis is structured as follows. Chapter 2 describes the background on multicore system architectures, memory and cache coherence mechanisms, and PolyBlaze system. The new cache coherent architecture for the PolyBlaze processor is presented in Chapter 3. Chapter 4 outlines the memory architecture of the PolyBlaze processor and the roles of the added modules in the processor's memory path. Chapter 5 discusses the experimental framework and the experiments run to validate and evaluate our system. Finally, Chapter 6 concludes the thesis and outlines future work.

2 Background and Related Work

In 1965, Gordon E. Moore presented his observation, later known as Moore's law, that the number of transistors on integrated circuits doubles approximately every two years [4]. Later on, Intel executive David House predicted that chip performance, being a combination of the increase in the number of transistors and their operating frequency, would double every 18 months. On the other hand, in 1974, Robert H. Dennard et al. [5] stated that as transistors get smaller their power use stays constant (power use stays in proportion to area). This statement, later know as Dennard Scaling, was true until about 2005-2007. As such, while it is possible to shrink transistor sizes and put more and more transistors on the die, it is not possible to drop the voltage and the current these transistors need to operate reliably at the same rate. Therefore, it is necessary to come up with other solutions to enable continued performance growth.

The remainder of this chapter presents the relevant background material and previous work for this thesis. First, in Section 2.1, we discuss multicore computer architectures, their memory infrastructure, memory coherency and synchronization in these systems. Then, we provide an overview of Systems-on-Chip (SOCs), Field Programmable Gate Arrays (FPGAs), soft processors and processor emulators in Section 2.2. Next, in Section 2.3, we describe the PolyBlaze framework and its development from the MicroBlaze soft processor and its internal architecture. Afterwards, we provide a brief overview of cache and scratchpad memories in different processors and the use of asymmetric caches in current processors.

2.1 Multicore Computer Architecture

One of the solutions to the Dennard Scaling problem is to use parallel processing. Increased use of parallel computing in the form of multicore processors is one of the approaches that has been pursued to improve overall processing performance. Parallel processing in its simplest form is the use of two or more processing cores to execute simultaneous instructions in a single computer system [6] and share some or all of available memory and Input/Output (I/O) facilities in the system [7].

Multiprocessing systems are often designed in two different methods: 1) shared memory architecture vs. 2) distributed memory architecture. In shared memory models, there is one common shared memory for all cores and processors. This memory is usually very large since it will be the main memory in the system. In distributed memory models, on the other hand, each processor has its own, local memory. In this case, the content of each memory is not necessarily replicated anywhere else.

Shared memory architecture designs usually follow two different processor architectures. The first method is SMP in which all of the processing units are identical, i.e. they all have the same ISA, frequency, caches, memory, etc. In SMP systems, the OS can treat all the cores equally and if there are multiple processes running in parallel, the OS can run them on different processor cores.

The second method, AMP, is slightly different from SMP. The cores in these processors, still share the same ISA and operating frequency, but some configurations such as cache sizes or cache line sizes can be different. In these systems, the OS can still ignore the difference between processing cores and assign tasks to them. However, it is possible to take advantage of these different configurations and gain better performance from the whole system.

Chapter 2. Background and Related Work

Distributed memory architecture designs have a different processing architecture method. Different processing units in these systems, use message passing mechanisms to communicate with each other. Therefore, they can have different processor architectures and even different formats for the methods as long as the interconnect network can translate the messages for the corresponding processors.

In this thesis, we focus on shared memory architectures. Globally shared memory architectures often use one of the following three organizations: 1) Uniform Memory Access (UMA) in which all the processors share the physical memory uniformly [8], 2) Non-uniform Memory Access (NUMA), which is similar to UMA except memory access time depends on the memory location relative to a processor [9, 8], and 3) Cache-Only Memory Architecture (COMA), in which the local memories for the processors at each node is used as a cache memory [8].

In shared memory architecture systems, the communication between processors is done by reading and writing memory locations [10]. However, there are two key problems regarding the scalability of a shared memory architecture system:

1. **Performance Degradation**: when several processors try to access the same memory location, we will have contention on the memory interface.
2. **Coherency**: modifications often cause different cached memories to have different values and loose consistency between different copies.

To address the performance degradation problem, we often try to balance the tasks assigned to each processor such that they can take advantage of their local private memories, e.g. L1 caches and sometimes L2 caches, so that the need to access the shared main memory is reduced. Typically, this problem is handled in the OS using different software algorithms, making it outside of the scope of this thesis. Instead, we focus on the second problem, i.e. memory coherency. This is often handled in

different layers of software and hardware, depending on the system architecture and consistency model. Sections 2.3.3 and 3.1 will discuss how we handle this problem in more detail withing different modules.

2.1.1 Memory and Cache Coherence

When two or more processors or cores share a common area of memory, they have to consider the coherence of memory regions shared between themselves. This issue does not exist in a single-core system because when a value is modified by the processor, all subsequent reads of that memory location will see the updated value, regardless of the data being cached anywhere or not.

In a multicore system, many processors can access the same location of shared memory. In this case, so long as none of them tries to modify this location and everyone is only reading from it, they can share it indefinitely. However, as soon as one of them updates the shared location, the others will have to be notified in order to use the updated value. Otherwise, the other processors might go on using their out-of-date copy of the data that could reside in their local caches.

Overall, in a multicore system, the time window during which different processors might have inconsistent views of the shared memory could vary. This time could be as low as a few cycles, even zero in some cases, to an indeterminate length of time. The size of this window usually depends on the coherency protocol implemented in the system. Systems that have strict memory consistency between processors and will not allow any sort of caching, will always have a coherent memory. Systems that allow simple caches with write-through policy, i.e. every write to the cache will be sent to memory right away, usually need a few cycles for the memory synchronization to complete [11]. In these systems, because of the write-through policy, all the store operations will be passed to the main memory right away. The SPARC T1 [12],

with its L1 caches, is an example of this style of architecture. An alternate system architecture uses write-back caches and has more sophisticated caching techniques, which requires more complex coherency mechanisms as well. In these systems with write-back caches, memory will not be updated with the most up-to-date data until that data is evicted from cache and written back into memory. Intel and AMD processors have L1 caches that use a Write-back policy. In addition, these systems require extra software support to force coherency when atomic instructions are issued [13, 14].

Coherence Mechanisms

There are four different coherency mechanisms: 1) Snooping (or *write-invalidate*) [15], 2) Snarfing (or *write-update*) [15], 3) Directory-based [16, 15], and 4) Library Cache Coherence (LCC) [17]. In systems that use snooping, individual cache controllers will continuously monitor the memory accesses from every other processor. When a memory location is modified, the cache controller will invalidate its own copy of that location if it has a copy. This protocol is sometimes called a *write invalidate protocol* [18]. The second mechanism, Snarfing, is similar to Snooping in behaviour. In this mechanism, the cache controller will look at address and data. When another processor tries to update a memory location, the cache controller will grab the new data and update its own copy if it has any. Therefore, it will use the updated copy next time the processor asks for that cache line.

In systems with directory-based coherency mechanisms, a common directory will be used to keep track of data shared between caches to maintain coherency. When a processor needs to access shared data, it will ask the directory for permission and then loads the data entry from main memory into its cache. Upon modification of shared data in one of the caches, the directory will notify the other caches and either

update or invalidate their data.

The LCC mechanism is based on directory-based methods, but with less broadcasted messages. In this mechanism, libraries are sets of timestamps that are used to auto-invalidate shared cache lines. These timestamps are also used to prevent writes on the lines from happening until all shared copies expire. There are a few advantage to use LCC methods instead of directory-based mechanisms. One of advantages is elimination of need to broadcast a lot of messages and therefore simplifying the interconnection network. Additionally, LCC also allows reads on a cache block to take place while a write to the block is being delayed while memory consistency is still in tact [17].

2.2 Soft Processors and Processor Emulators

FPGAs are Integrated Circuit (IC) devices that consist of arrays of logic resources that are designed to be configured by customers after manufacturing. For instance, a typical FPGA introduced by Xilinx [19], consists of several components including Configurable Logic Block (CLB) connected by an interconnection network of wiring channels and routing blocks (switch matrices).

FPGAs are heterogeneous arrays of configurable logic blocks (CLB) and other Intellectual Property (IP) blocks that can be programmed by customers to implement their desired circuits after manufacturing. Each CLB in turn is comprised of logic cells containing Look-up Table (LUT)s and registers or Flip-Flop (FF). The other IP cores commonly found in modern FPGAs include processors, Digital Signal Processing (DSP) blocks, Block RAMs (BRAMs), and I/O blocks. Developers can combine these components with user-defined modules to build complicated designs.

Along with the hard processors embedded in some FPGA fabrics, we can also instan-

tiate processors in the reconfigurable fabric. Reconfigurable fabric in FPGA enables us to investigate architectural and simulation research questions. For instance, we can accelerate the simulation of traditional architectures as done in [20] or open more research areas into soft processor architectures [21]. Additionally, soft processors allow us to create far better emulation platforms for the design of commercial multicore processors [22, 23].

The complexity of the tasks for soft processors in a system will define the complexity of the processor itself or the features that it might have. The PicoBlaze [24], for instance, is a simple 8-bit soft processor developed by Xilinx [19] that is suitable for designing simple state-machines. Xilinx's other soft processor, the MicroBlaze [1], and Altera Inc.'s NIOS II [25] are two other examples of commercial soft processors that are designed specifically to target their company's FPGA architecture.

Although hard processors on FPGAs support higher operating frequencies than soft processors, their configurability is limited compared to soft processors. For example, if for a particular task, we have to do some floating point calculations, we can simply add a Floating Point Unit (FPU) to the processor.

Additionally, soft processors open areas of research into multicore processor architecture. The Research Accelerator for Multiple Processors (RAMP) project [26] as one of the works done in multicore processor architecture research areas, focuses on multiple lightweight processors on tens and hundreds of FPGAs [27]. The Beehive Project enables multiple lightweight processors on a single FPGA[28]. The project Hthreads provides support for threads in hardware and software along with a hardware based OS [29]; Xilinx provides a lightweight xilkernel based multicore MicroBlaze system [30]. Also, on a larger scale, there are some emulation platforms for design of commercial multicore processors, e.g. a FPGA-synthesizable Intel®Atom™processor core, a

FPGA-synthesizable Intel®Nehalem™processor core, etc. [22, 23, 31]

One of these research areas looks into load balancing and workload behaviour in multicore architectures. Approaches taken by some multicore processor emulators based on FPGA platforms such as OpenSparc and LEON3 [32, 33] take advantage of access to the hardware implementation of the emulator. In these cases, the hardware implementation provides cycle-accurate visibility into the microachitecure of the system. On the software side, the capacity to run a full OS provides full access to the features that might be of interest in the operating system.

The OpenSPARC based systems support a simple version of the Linux distribution (based on Ubuntu Linux distribution) for a single-core or dual-core setup. These systems, however, have limited scalability. The memory hierarchy in dual-core systems requires a single MicroBlaze to process all memory requests. Moreover, if we want a multicore system, we have to use separate FPGA boards, which in turn can impose other restrictions on the system. Another SPARC-based platform is the LEON3. It supports up to a dual-core setup. However, the memory controller support for different boards is incomplete. Furthermore, due to the hardware resource usage, LEON3 has a large processor design such that we can only fit two processors on a Virtex 5 110t FPGA from Xilinx.

There are other soft processors that have Linux support. NIOS II from Altera Inc. is one of these soft processors [25]. Xilinx's MicroBlaze is another soft processor that supports Linux [1]. PetaLogix [34], an embedded Linux solutions provider that was acquired by Xilinx, has been providing Linux support for MicroBlaze soft processor and PowerPC processors since the 2.6.30 version of the Linux kernel. The latest version of this distribution was released in February 2014. A multicore version of MicroBlaze is PolyBlaze [2]. We can fit up to an eight core PolyBlaze (four cores

with medium-sized caches) in a Virtex 5 110t FPGA from Xilinx.

2.3 PolyBlaze

PolyBlaze is a 32-bit Reduced Instruction Set Computing (RISC) embedded multicore soft processor based on the MicroBlaze core. It consists of several instances of slightly modified MicroBlaze cores and a few new modules. These new modules exist to provide the required hardware support for sharing resources in a multicore processor. Without this hardware infrastructure, it is still possible to run applications on this processor, but we cannot support booting an SMP OS and run general applications [2, 35]. The modifications on the processor are ranged from minor additions, such as processor identification through existing Processor Version Register (PVR), to more extensive modifications on interrupts, timers, atomic operations and Memory Management Unit (MMU) [2].

2.3.1 Overall Architecture

Figure 2.1 provides an example PolyBlaze system that consists of multiple processor cores with symmetric or asymmetric configurations, multiple levels of cache memory, and potential custom hardware accelerator cores. Symmetric processor cores by themselves allow a simple SMP system, while asymmetric processor cores and custom hardware accelerator cores enable a more complex heterogeneous system. Multiple levels of cache memory are common in systems, especially when some of them are shared while the rest are private. On top of these, the hArdware Based Accelerator for Characterization of User Software (ABACUS) hardware profiler can hook into many of these modules and gather useful information about their operation.

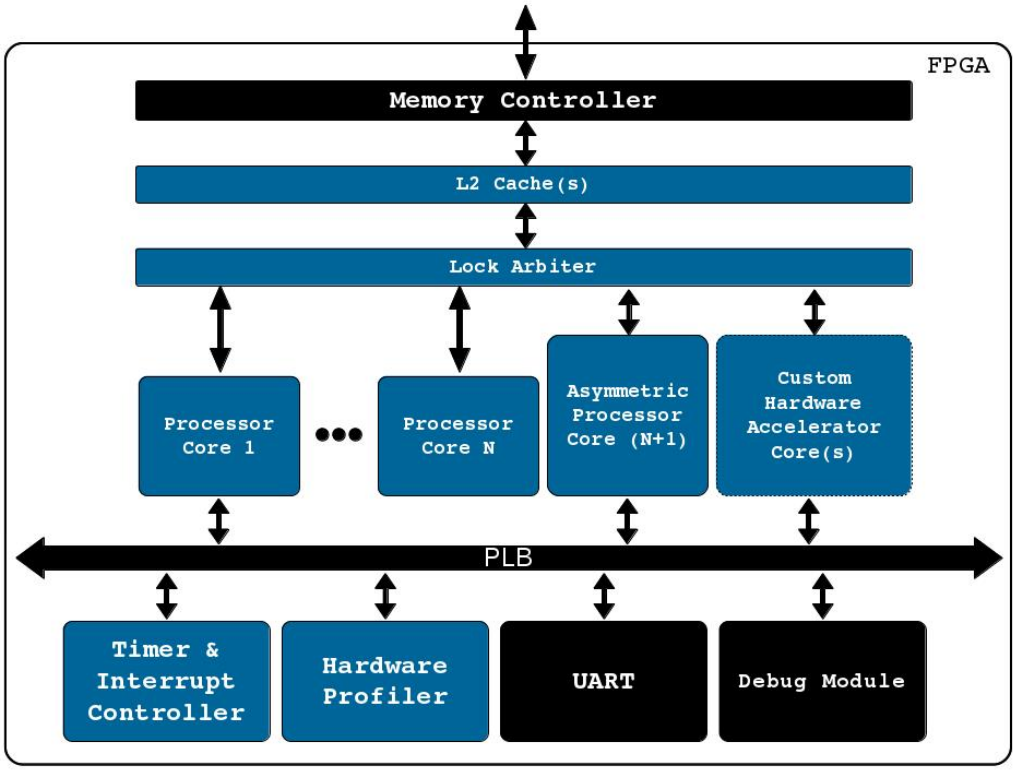

Figure 2.1 PolyBlaze: the multiple MicroBlaze platform

2.3.2 MicroBlaze

MicroBlaze is a 32-bit RISC embedded processor soft-core. It is optimized for implementation in Xilinx FPGAs [1] and is highly configurable. Features such as MMU, FPU, instruction and data caches, dedicated hardware multiplier and divider, hardware barrel shifter and several bus connections are just some examples of the options that we can use for an instance of MicroBlaze soft processor [1].

MicroBlaze instruction execution is pipelined. There are two configurations for the processor pipeline:

- Three-stage pipeline: if we optimize the processor implementation to reduce the hardware cost, we will have three stages in the pipeline: 1) Fetch, 2) Decode, and 3) Execute [1]. Figure 2.1 illustrates a three stage pipeline example with two stall slots.
- Five-stage pipeline: if we optimize the processor implementation to maximize performance, we will have five stages in the pipeline: 1) Fetch (IF), 2) Decode (OF), 3) Execute (EX), 4) Access Memory (MEM), and 5) Writeback (WB) [1]. Figure 2.2 illustrates a five stage pipeline example with two stall slots.

Table 2.1 MicroBlaze core's three-stage pipeline optimized for minimum area/hardware cost [1]

cycle number	1	2	3	4	5	6	7
instruction 1	Fetch	Decode	Execute				
instruction 2		Fetch	Decode	Execute	Execute	Execute	
instruction 3			Fetch	Decode	Stall	Stall	Execute

Table 2.2 MicroBlaze core's five-stage pipeline optimized for maximum performance [1]

cycle number	1	2	3	4	5	6	7	8	9
instruction 1	IF	OF	EX	MEM	WB				
instruction 2		IF	OF	EX	MEM	MEM	MEM	WB	
instruction 3			IF	OF	EX	Stall	Stall	MEM	WB

In any case, most of instructions require one clock cycle in each pipeline stage. There are also a few instructions that require multiple clock cycles (in execution stage) to complete. In those cases, bubbles are used to stall the pipeline.

Additionally, there are some instructions that require multiple cycles in other stages, e.g. memory stage. Load instructions fall into this last category. This additional latency can significantly affect the pipeline efficiency especially while reading instructions from external memory. In order to mitigate this problem, MicroBlaze implements an instruction prefetch buffer. This prefetch buffer can significantly reduce the impact of such multi-cycle instruction memory latencies.

Overall Architecture

Figure 2.2 illustrates a functional block diagram of the MicroBlaze core. As we can see in this figure, MicroBlaze has a Harvard memory architecture. Modules shown with darker background are optional and can be excluded from the processor if not necessary. The processor has up to three interfaces for memory accesses:

- Local Memory Bus (LMB)
- Advanced eXtensible Interface (AXI4) or Processor Local Bus (PLB)
- Advanced eXtensible Interface (AXI4) or Xilinx CacheLink (XCL)

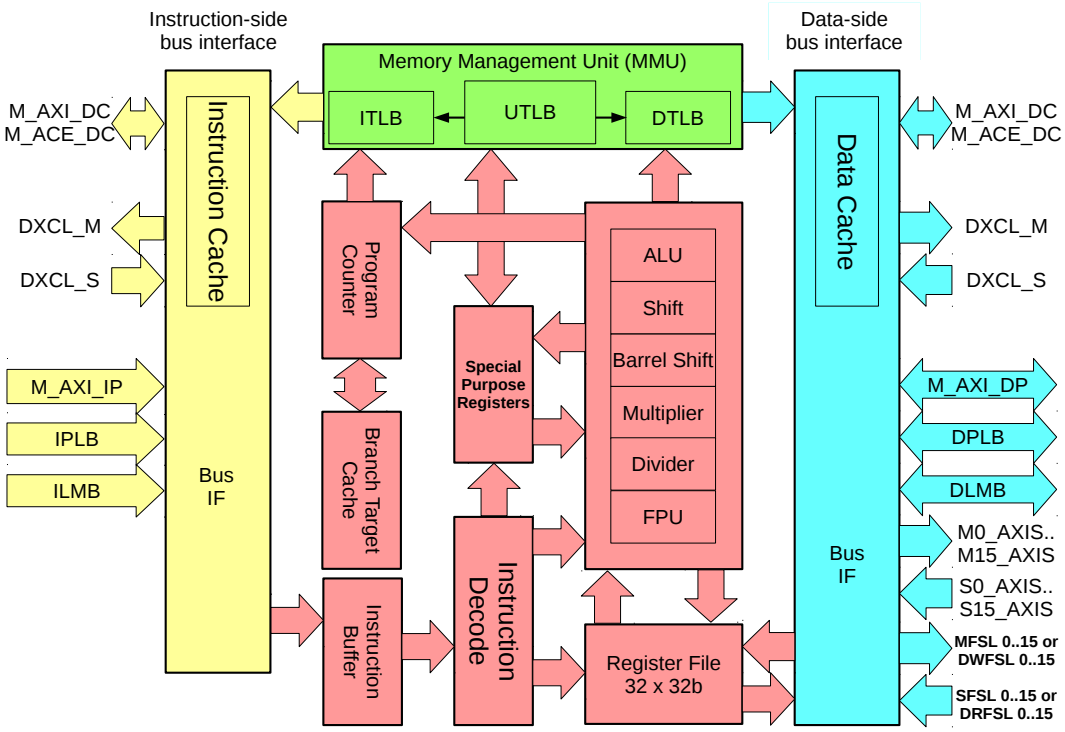

Figure 2.2 MicroBlaze Core Block Diagram [1]

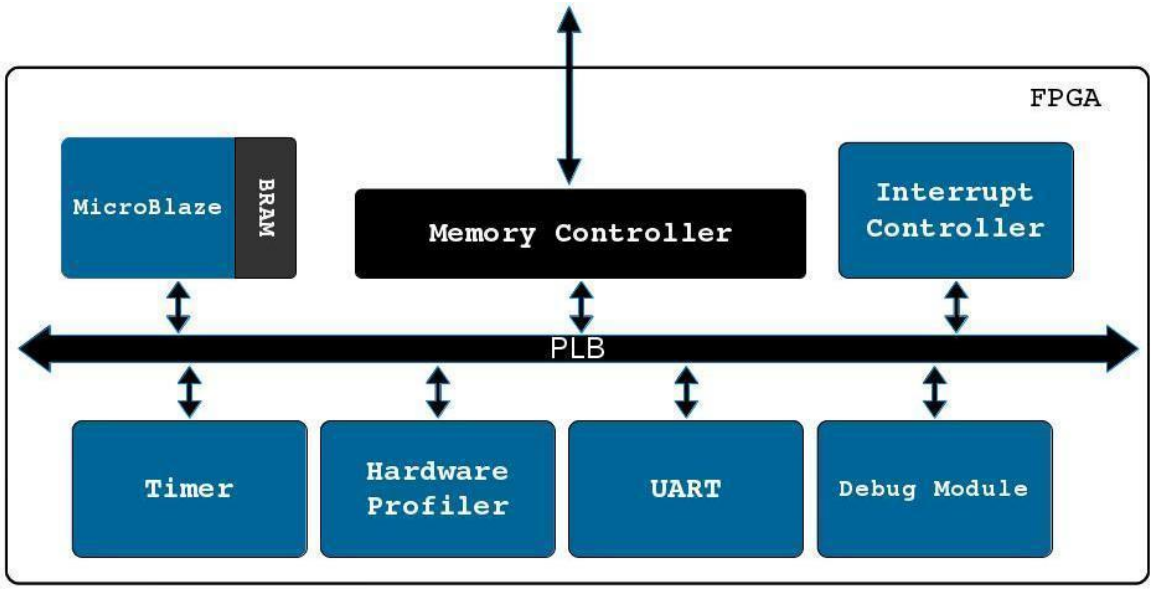

Figure 2.3 A block diagram of a Linux capable MicroBlaze system.

Moreover, MicroBlaze has a MMU that is based upon the PowerPC 405 [1] MMU. The MMU in MicroBlaze is software managed, i.e. on Translation Look-aside Buffer (TLB) misses, software subroutines will add or remove entries in TLB.

Semaphore Synchronization

In a system based on MicroBlaze soft processors, programs can implement semaphore operations such as spin locks, test and set, compare and swap, exchange memory, and fetch and add, using Load Word Exclusive (LWX) and Store Word Exclusive (SWX) instructions.

In order to implement an atomic operation, the program should first use LWX to load a semaphore from memory and set an internal reservation bit in the processor. Then the program tries to write the result back to the same memory location using a SWX instruction. If the internal reservation bit is set, the SWX will go through. Otherwise, if for any reason this internal reservation bit has been cleared, the SWX

will fail to go through, which means the atomic operation has failed. The internal reservation bit can be cleared on different conditions such as an exception, interrupt or another condition store to the same location. The success or failure of a SWX instruction will be stored in the carry bit in Machine Status Register (MSR) [1].

There are some limitations for semaphore synchronization in MicroBlaze, however. The first limitation is that we can only maintain one reservation at a time. So if we want to change the address associated with the reservation, we have to use another LWX instruction. Also, executing a SWX always clears the reservation bit regardless of what the associated address is, i.e. no address matching check happens.

Instruction Cache

MicroBlaze has an optional instruction cache. This instruction cache can be used for improved performance when the processors is executing code that resides outside the Local Memory Bus (LMB) address range[1]. As mentioned before, MicroBlaze is designed to be very configurable. To that end, its instruction cache is also very configurable. Some of the features in the instruction cache are as follows [1]:

- Direct mapped (1-way associative)
- User selectable cacheable memory address range
- Configurable cache, tag, and cache-line (4 or 8 words) sizes
- Caching over AXI4 interface or XCL interface
- Optional stream buffers to improve performance by speculatively prefetching instructions
- Optional victim cache to improve performance by saving evicted cache lines upon replacement and reduce the number of conflict misses

[1]The access time for reading from or writing into LMB is just one cycle.

The following Figure 2.4 illustrates the organization of instruction cache in MicroBlaze. The cacheable instruction address consists of two parts: 1) the cache address, and 2) the tag address. Since the instruction cache can be configured from 64 bytes to 64 kB, a cache address will be between 6 to 16 bits. In addition to that, the tag address together with the cache address should match the full address of cacheable memory. Since MicroBlaze instructions are single words (four bytes), then instruction cache will ignore the two least significant bits [1].

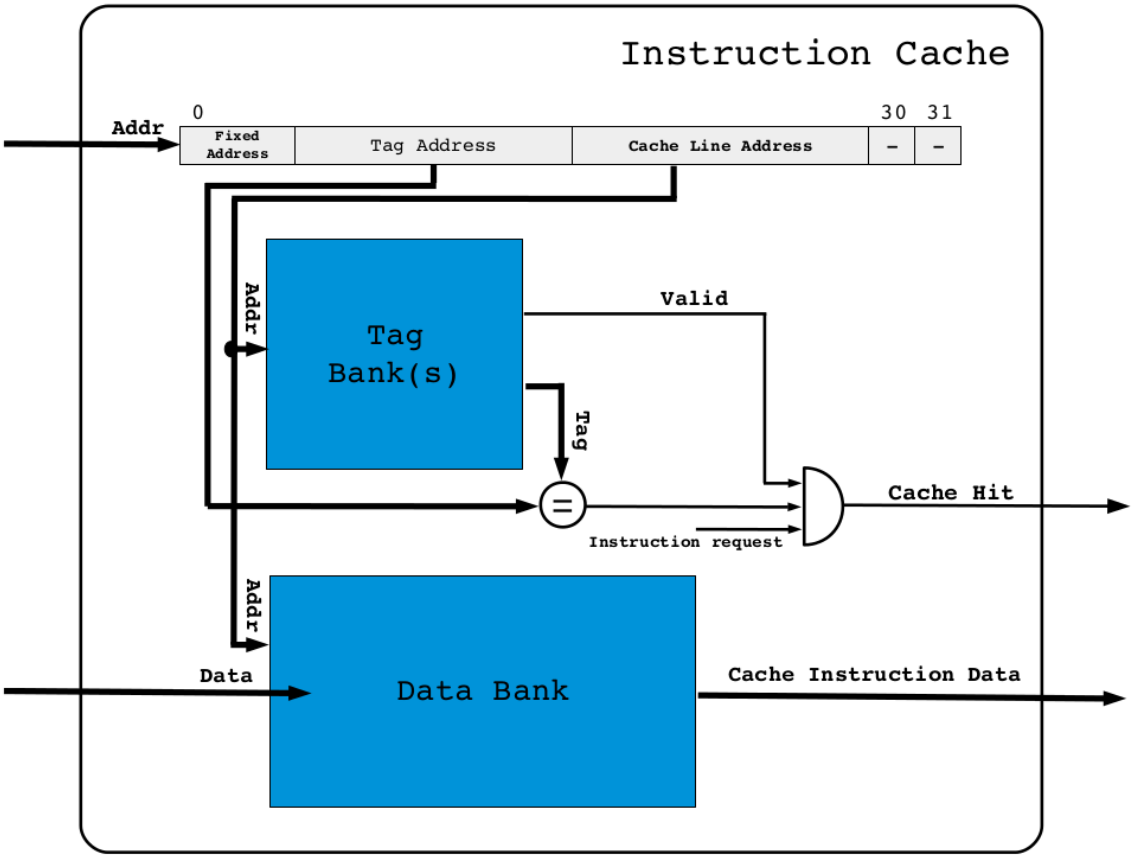

Figure 2.4 Instruction Cache Organization in the MicroBlaze [1]

The instruction cache in the MicroBlaze has optional stream buffers. When the stream buffers are enabled, the cache will fetch subsequent cache lines in advance,

until the stream buffer is full (up to two cache lines). If the processor requests for subsequent instructions from a cache line, they will be immediately available [1].

There is an optional instruction, Write to Instruction Cache (WIC), that applications can use to invalidate cache lines in the instruction cache from within the software. WIC will be considered as a privileged instruction when MicroBlaze is configured to use MMU [1].

Data Cache

MicroBlaze has an optional data cache. This data cache can be used for improved performance when the processors is accessing data that resides outside the LMB address range[2]. As mentioned before, MicroBlaze is designed to be very configurable. To that end, its data cache is also very configurable. Some of the features in the data cache are as follows [1]:

- Direct mapped (1-way associative)
- Write-through or Write-back
- User selectable cacheable memory address range
- Configurable cache, tag, and cache-line (4 or 8 words) size
- Caching over AXI4 interface or XCL interface
- Optional victim cache to improve performance by saving evicted cache lines

Figure 2.5 illustrates the organization of data cache in the MicroBlaze. Similar to the instruction cache, the cacheable data address consists of two parts: 1) the cache address, and 2) the tag address. The data cache can be configured from 64 bytes to 64 kB, so a cache address will be between 6 to 16 bits. In addition to that, the tag address together with the cache address should match the full address of cacheable memory [1].

[2]As mentioned before, the access time for reading from or writing into LMB is just one cycle.

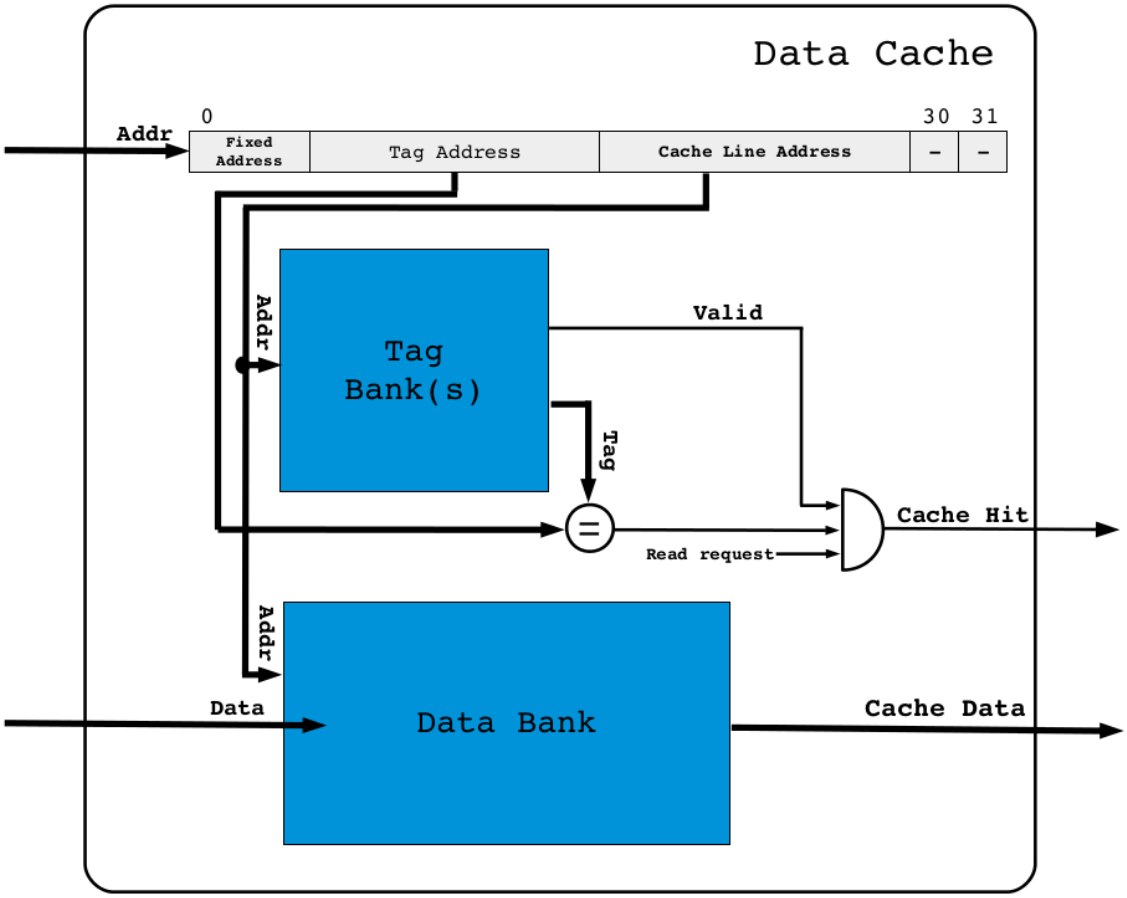

Figure 2.5 Data Cache Organization in the MicroBlaze [1]

When the data cache is configured to use the write-through protocol, a store to an address within the cacheable range generates an equivalent write request over the data interface to external memory. In case of a write cache-hit, i.e. the target address word is in the cache, the write also updates the cached data. A write cache-miss does not load the associated cache line into the cache [1].

When the data cache is configured to use the write-back protocol, a store to an address within the cacheable range always updates the cached data. Therefore, if the target address word is in the cache, i.e. a write cache-hit, the data will be updated in the

cache. In this case, no equivalent write will be sent over the data interface to external memory. On the other hand, if the target address word is not in the cache, i.e. a write cache-miss, the data will be updated in the cache, the address is first requested over the data interface from external memory and written into the cache. Then the cache will update the data as if it is a write cache-hit. In case of a write cache-miss, if the cache line that is being brought to the cache is already occupied by another line of data that has already been modified, the cache has to evict the occupying line and write it to the external memory before overwriting it with the incoming data [1].

In both write-through and write-back protocols, when processor wants to read some data from cacheable address range, the cache triggers a check to determine if the requested data is currently cached. If the requested data is caches, then we will have a read cache-hit and the requested data will be retrieved from the cache. Otherwise, the address will be requested over the data interface from the external memory on a read cache-miss. In this case, the processor pipeline has to be stalled until the cache line is returned from the external memory controller [1].

Similar to instruction cache, there is an optional instruction, Write to Data Cache (WDC), that applications can use to invalidate cache lines in data cache from within the software. When MicroBlaze is configured to use MMU, WDC will be treated as a privileged instruction. Additionally, when the data cache is using a write back protocol, this instruction can be used to force cache line flushes [1].

Over all, the instruction and data caches in MicroBlaze behave similarly in many ways. However, there are some key differences between them. The first difference is that instruction cache is read only whereas data cache can be modified through the applications[3]. The second difference is that instruction cache works with both virtual

[3]It is possible to update instruction cache through the application (self-modifying code), but in order to actually update the instruction cache, we have to invalidate the desired cache lines and force

and physical addresses while data cache only works with physical addresses.

XCL

XCL is a high performance interface for external memory accesses. MicroBlaze's XCL interface is designed to connect directly to a memory controller such as Multi-Port Memory Controller (MPMC) that supports integrated First-In-First-Out (FIFO)s (Fast Simplex Link (FSL) buffers).

The XCL interface is only available on MicroBlaze when at least one of the caches is enabled. Memory locations that reside outside the cacheable range will be accessed over other bus interfaces such as Processor Local Bus (PLB) or LMB. The XCL controllers can handle 4 or 8-word cache lines. Moreover, they can be configured to use either critical word first or linear fetch depending on the selected protocol. When selected protocol is critical word first, when processor is requesting a second word in a cache line, that word is what will be brought into the cache through XCL first and the rest of the words follow it. For example. when the processor request the third word in a cache line, the words fetched for the cache line will arrive in this order: 3, 4, 1, 2. Conversely, in a linear fetch protocol, the words in a cache line will always get to the cache in order from the first to the last word. Regardless, the separation of XCL from PLB bus reduces contention for non-cached memory accesses.

2.3.3 L2 Arbiter

L2 Arbiter is an important module in PolyBlaze's memory hierarchy. This module has three main purposes: 1) receive all the requests and pass them on to memory in order, 2) broadcast invalidation requests to maintain memory coherency, and 3) handle reservations and conditional operations (which is done by Lock Arbiter as

the program to continue executing from there. Then the instruction cache will read the updated values from memory.

described in Section 2.3.4).

As mentioned before, the first purpose of the L2 Arbiter is to receive requests from all of the connected processors and hardware accelerators and pass them on to memory in order. In order to achieve this goal, the L2 Arbiter scans its input ports one by one using a round robin method so that every connected processor gets their fair share of requests. On any given cycle, if the selected processor does not have any requests, the L2 Arbiter will consider other ports that come after the selected processor until it detects a request or until it determines that no one has any requests. This method allows a fair memory access for every port, so no port will be waiting for its turn while other ports keep sending memory requests. Also, if one port does not have any request on its turn, another port can use the cycle to send a request to memory, hence the cycle will not go to waste.

The second purpose of the L2 Abiter is to broadcast invalidation requests to maintain memory coherency between connected ports. Since the L2 Arbiter is the central point through which all memory requests are sent, it can broadcast invalidation requests correctly. To achieve this goal, the L2 Arbiter will generate an invalidation packet whenever it is issuing a write request on behalf of a port. These invalidation packets will be broadcast to every other port. When the broadcasting is complete and the write request has been written to memory queues, the L2 Arbiter considers the write operation successful and removes it from processor's queue.

The third role of the L2 Arbiter is to handle reservations and conditional requests. Each port in the L2 Arbiter has its own reservation bit and reservation address register. Upon receiving a conditional load request from a processor, the L2 Arbiter sets the corresponding reservation bit and registers the address of load operation into the corresponding reservation address register. In addition, the load request is passed

on to memory. Upon receiving a conditional store request, the L2 Arbiter checks the reservation bit and looks for a match between reserved address and conditional store's address. If the address matches and the reservation bit is set, then conditional store request is considered successful; otherwise it is considered as a failed request. Successful requests are passed on to memory and failed requests will be dropped. Additionally, the L2 Arbiter sends a notification to the corresponding processor and notifies them of the result of the conditional store. There are other features in later versions of the L2 Arbiter including burst writes and Direct Memory Access (DMA) support. However, they are not required in this work.

2.3.4 Semaphore Synchronization

As discussed earlier, the MicroBlaze performs atomic operations with conditional load/store instructions: LWX and SWX. First LWX is used to load from memory, e.g. a semaphore's location, and set the internal reservation bit. Then, a SWX will be used to write a new value to that location if the reservation bit is still set. If this bit is cleared for any reason, interrupts, exceptions, or other conditional stores to any memory location, the conditional store will not happen [1]. This design is perfectly suitable for a single-core system, but it cannot guarantee that no other processor will modify the memory location. In other words, this behaviour will not be able to guarantee the atomicity of operation.

In a multicore system, several memory requests can be in progress at a given time. Therefore, PolyBlaze has to take a slightly different approach to this problem. In PolyBlaze, the Lock Arbiter, as shown in Figure 2.1, will take care of reservation handling for each processor and acts as the central synchronization point between all the processors. In this system, all the store operations will be passed to the Lock Arbiter along with extra signals indicating that they are regular or conditional stores.

Then the Lock Arbiter will decide whether to put the write request to main memory or to drop the request because the reservation bit has been cleared before the store reaches the Lock Arbiter. Then the Lock Arbiter will notify the processor of the result of its conditional store via some other extra signals. The result of this operation is then fed back into the processors logic the execution of the SWX instruction will be complete [2].

Beside moving the reservation bit to the Lock Arbiter, PolyBlaze improves the performance of lock arbitration by adding one reservation bit per processor and a matching address for each reservation bit. The extra logic used here helps with performance and scalability of the Lock Arbiter. Since each processor has its own reservation bit and reservation address registers, multiple processors can work with different semaphores and locks at the same time without interferreing with each other. However, it is worth noting that in this system, when we have any store operation to memory, if someone else has tried to set a reservation on that location, we will clear their reservation bit so they will fail and have to try again [2].

2.4 Cache and Scratchpad Implementation on FPGAs

Memory hierarchies in modern multicore computing systems are based on one of the following schemes: 1) multi-level coherent caches, or 2) scratchpad memories with DMA support. Systems that use caches are often general purpose systems as the software programmer does not need to know where the data is actually stored or how the hardware is handling the data movements in the system. Such data movements are indirect results of cache misses and coherence events. In contrast, software developers for systems that use scratchpads, need to know how the data is supposed to be stored in memory [36]. This knowledge, obviously, can lead to much better performance. Moreover, the user control over the scratchpads eliminates the

need for coherency mechanisms between multiple scratchpads and, therefore, such systems require less additional inter-processor communications. These optimizations become especially important in large-scale systems since coherency mechanisms, due to their need for inter-processor communications, can have dramatic effects on the scalability of a system [37].

Many of the soft processors mentioned in Section 2.2 either do not implement L1 caches at all or implement a single core system, without any coherency mechanisms. MicroBlaze [1] and NIOS II [25] only provide single core systems. These processors are highly configurable but their L1 cache implementations use only a direct-mapped approach. Also, since they only support a single core system, they do not implement any coherency mechanisms [1, 25].

Conversely, OpenSPARC T1[32] supports coherent L1 caches, but it is not very configurable. However, an OpenSPARC core is large enough to use almost all of available resources in a Virtex 5 110LXT. Moreover, the memory controller in OpenSPARC system is emulated using the firmware on a MicroBlaze processor. Therefore, the setup for a multicore OpenSPARC system requires multiple FPGA boards due to the size of the processor and limits the system scalability. Another SPARC-based system supporting multicore setup uses LEON3 soft processors [33], but board memory controller support for LEON3 processor is limited.

2.4.1 Configurability

As mentioned earlier, MicroBlaze and NIOS II are two highly configurable soft processors [1, 25]. The built-in configurability in these cores permits more options for an AMP system. For instance, we can turn on a FPU in one of the cores, use hardware multipliers and dividers in another, and keep the rest simple to reduce the resource usage in the whole system. However, not all soft processors, regardless of being a

single-core or multicore processor, support enough configurability for our purpose. Nikiforos et al. present a local memory design that can be configured to behave as a cache and a scratchpad at the same time to support implicit communications via caches and explicit communications via scrachpads. In their work, it is possible to lock parts of cache and prevent that part from evicting data, hence treating it as an scratchpad memory [38].

2.5 Asymmetric Caches in Multiprocessors

Most of the work done on caches either target single core computing systems or SMP systems. ARM's *little.BIG* architecture is one of the commercial movements towards AMP systems. In this architecture, two types of processors exist. "big" processors are designed to provide maximum compute performance while "LITTLE" processors are designed for maximum power efficiency [39]. Since both of these processor types use the same ISA, the OS running on the system does not necessarily need to be aware of the difference between processors. However, the Global Task Scheduling (GTS), which is in fact a patch on the scheduling mechanisms in the OS, is aware of the difference between processors and tries to get better performance by assigns tasks to different processors based on the needs of applications. Besides this architecture, there has not been any work on asymmetric caches on FPGAs to the extent of author's knowledge. Moreover, the ARM processor cores available on some FPGAs are hard cores and are not part of the reconfigurable fabric of the FPGA.

3 An Asymmetric Cache Coherent Architecture for PolyBlaze System

A single-core MicroBlaze system with its instruction and data cache can boot an OS or run any generic application without any problems. A multicore system based on MicroBlaze, however, cannot necessarily do that just by replicating MicroBlaze cores and their caches. Since there is no coherency mechanism implemented for MicroBlaze's caches, without proper measures, the memory consistency will become a problem, the OS will fail to even boot, and generic applications will fail to execute correctly. Therefore, the designed caches for PolyBlaze include a coherency protocol handler. Additionally, these caches follow the original MicroBlaze caches and use as many parameters as possible to have high configurability, which in turn enables asymmetric architectures. Besides, ABACUS interface allows ABACUS hardware profiler to easily connect to PolyBlaze caches and gather data about their behaviour.

The remainder of this chapter will present the proposed coherent cache memory architecture. This includes the necessary modifications on instruction and data caches to support a simple *write invalidation protocol*. In addition, we will talk about possible configurations of different parameters in the system and how they will affect system behaviour. Finally, we will discuss the involvement of caches in synchronization and conditional instructions.

3.1 Level-1 Data Cache

MicroBlaze has an optional data cache (we can remove it from the system without loosing functionality). However, the data cache is mandatory in PolyBlaze's current implementation as it is involved in handling conditional instructions as well as data access to memory. So in a PolyBlaze system, it is possible to disable the data cache, but its memory interface has to remain in the system. Figure 3.1 highlights the internal architecture of PolyBlaze's data cache. The blue modules exist in MicroBlaze's data cache and the green modules are added for PolyBlaze's data cache. A detailed explanation of each module's role in the cache follows.

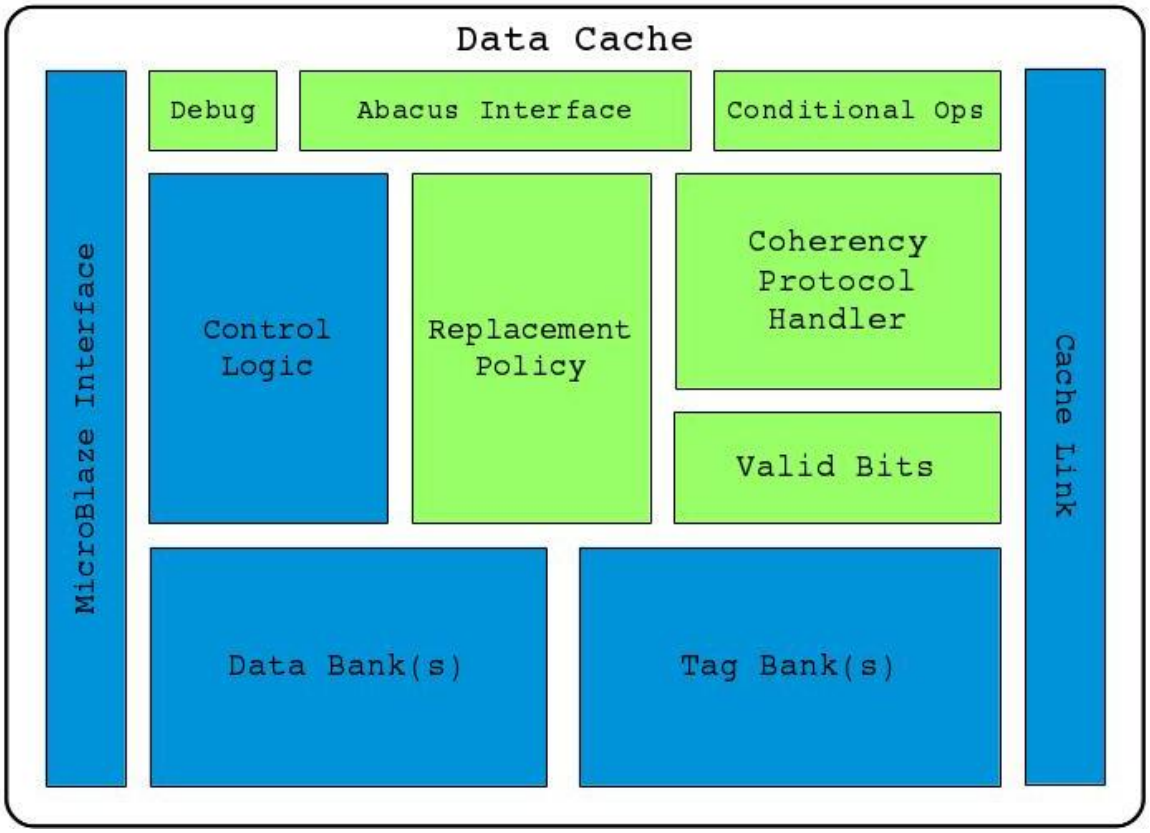

Figure 3.1 PolyBlaze's Data cache overview

Data Bank(s)

The data bank in MicroBlaze's data cache is just used to keep the data. It has several parameters such as the size of cache line or type of memory used to store the data (BRAM or LUTs). In PolyBlaze's data cache, we have the same parameters. However, depending on the replacement policy, the number of used memory blocks can be different.

Tag Bank(s)

The tag bank in MicroBlaze's data cache is used to keep the tag values and the valid bits. The available parameters for tag bank are similar to data bank's parameters: size of cache line and type of memory used to store values (BRAM or LUTs). In PolyBlaze's data cache, we have the same parameters for tag bank. However, the number of used memory blocks can be different depending on the replacement policy. Additionally, valid bits in PolyBlaze's data cache are stored in a separate bank.

Valid Bits

As mentioned before, valid bits in MicroBlaze's data cache are kept along with tag values and are only updated when there is either a cache miss or the processor has issued a WDC instruction to invalidate a cache line in data cache. In the PolyBlaze, we have moved the valid bits into a separate memory bank. The main reason for breaking this logic into two separate parts is to localize the logic that accesses the valid bits since it is part of the critical path in the processor. This logic consists of accesses for cache hits, updates to valid bits upon cache misses, invalidations due to WDC instructions, and invalidation logic from coherency protocol. Additionally, the replacement policy module needs access to valid bits as well, depending on the replacement policy used in the system.

Replacement Policy

MicroBlaze only implements a direct-mapped replacement policy. Therefore, the logic for this module is embedded with the rest of logic in cache. In PolyBlaze, we implement a dedicated module and make replacement decisions in that module in order to be able to define different replacement policies such as direct-mapped, Least Recently Used (LRU), Clock, or pseudo-random replacement policy[1]. This module will also allow for asymmetric caches in a PolyBlaze system since we can choose different replacement policies for different cores.

Conditional Operations

Conditional load operations in MicroBlaze are unrelated to the data cache, except that the processor will request data from the data cache, just like any other load operation. In the PolyBlaze, when a processor tries to execute a LWX instruction, instead of setting an internal reservation bit, it will notify the data cache. The data cache in turn will force[2] a cache miss and send a memory request to the L2 Arbiter (a module that includes the Lock Arbiter), which handles reservations. The rest of what happens for the LWX instruction is the same as regular loads operations of MicroBlaze's data cache behaviour: when the data cache receives the data from memory, it will return the data to the processor just like any other load instruction.

Similar to conditional loads, conditional store operations in MicroBlaze do not effect the data cache except to execute a store operation if the store operation is supposed to go through upon a successful conditional operation. Conversely, in PolyBlaze, on a SWX instruction, a write request will be sent to the L2 Arbiter. Then, the L2 Arbiter

[1] In this thesis, we only implement direct-mapped and LRU replacement policies.

[2] We force a cache miss because we have to send a request to the Lock Arbiter to make sure that another core is not trying to access the same lock. So if the cache line happens to be in the cache, we will mark it as invalid and send a request to memory to fetch the data from there.

will decide whether the write is supposed to go through all the way to memory or not. If the operation is successful, meaning the reservation bit in the L2 Arbiter is set and the write address matches the reserved address, the L2 Arbiter will pass on the write request to memory. In the mean time, the L2 Arbiter notifies the processor of the result of conditional store via sending a single bit through PBML. If the operation is successful, the value of this bit will be *'1'*, otherwise it will be *'0'*. When the data cache receives the result of conditional store, it will pass the result to the processor. In addition to that, upon a successful conditional store, the data cache will update its content as if there has been a hit on the requested address.

Coherency Protocol Handler

The coherency protocol handler in PolyBlaze's data cache receives the invalidation requests from the L2 Arbiter. Since the implemented method is *write invalidation protocol*, this module will look at invalidation address in each packet and invalidate the corresponding cache line if the tag from invalidation address matches the tag value read from tag bank.

The *write invalidation protocol* is one of the simplest coherency protocols. More sophisticated methods that are not part of this work, will keep track of which processor is accessing what parts of memory and notify other processors if necessary. Some even mark memory blocks as their own and respond to requests from other processors instead of main memory. Many of these protocols are designed for write-back caches though and are not necessarily applicable to write-through caches. Moreover, these complicated methods would require a more sophisticated interconnect than what we currently have in the L2 Arbiter.

ABACUS Interface

We can connect ABACUS [35] to the data cache through the ABACUS interface. Using ABACUS, we can gather information about behaviour of data cache such as number of read requests, cache read misses, cache read hits, write requests, cache write hits, cache write misses, invalidation hits for coherency protocol, total number of incoming packets for coherency protocol, and so on. The gathered information can later be used to analyze the performance of an application. Moreover, it is possible to add more signals to this interface depending on the type of data that we want to gather.

Debugging Data Cache

During the design process, we have embedded some signals that are useful for debugging purposes. These signals range from key signals in the cache control logic to the PBML connection signals to hardware counters and check-sum registers. If necessary, we can hook these signals into the ChipScope logic analyzer [40] to gather more information about system behaviour. For example, we can use the ChipScope logic analyzer to gather information such as memory latency. When not being used, we can remove this logic by setting the *C_PB_DCACHE_USE_DEBUG* parameter of the PolyBlaze processor to '0'.

Control Logic

Control logic is the heart of data cache. It connects different modules, keeps track of timing for different signals and multiplexes the addresses for different modules such as data bank, tag bank, or valid bits. Also, the implemented logic for handling invalidation instructions, i.e. WDC, is part of control logic. These instructions are often handled in a single cycle. The WDC instructions provide an address for invalidations. However, the data cache only looks at the cache line address and ignores the

tag value. Therefore, a WDC instruction will invalidate the cache line regardless of a hit or miss on the tag part of the address.

PolyBlaze Memory Link (PBML)

MicroBlaze uses XCL for direct connection to MPMC. XCL can be used to stream data to processor and is arbitrated on the slave-side. PolyBlaze, on the other hand, requires a more complicated memory interface. There are several modules in PolyBlaze's memory architecture and they are connected to each other via different links, generally called PBML. We will further discuss the internals of a PBML link in Section 4.3. The PBML for the data cache consists of four different queues as shown in Figure 3.2. Data cache PBML connections to the L1 Arbiter are colored turquoise/-green.

1. Request Queue: the first queue is used to send request information to the L1 Arbiter. Each request is sent as a packet that contains 32 bits for the address of the operation, one bit for the type of the operation ('0' for a write operation and '1' for a read operation), 32 bits for outgoing data if the request is for a write operation, four bits representing the byte-enable bits for a write operation, and one bit indicating a conditional versus regular operation. There are a few other bits that will be ignored in the later modules since their functionality is not required yet, hence they are not implemented [3].
2. Data Queue: the second queue is used to bring in the incoming data from memory to data cache. The incoming data will be transferred through the L1 Arbiter. The width of this FIFO is 32 bits to bring in one word at a time. The incoming words will come in order – meaning that if we are reading the line containing the address 0x50000088 from memory, the first word we receive will

[3]These bit may be removed if we decide to take a different approach to handle them in the future.

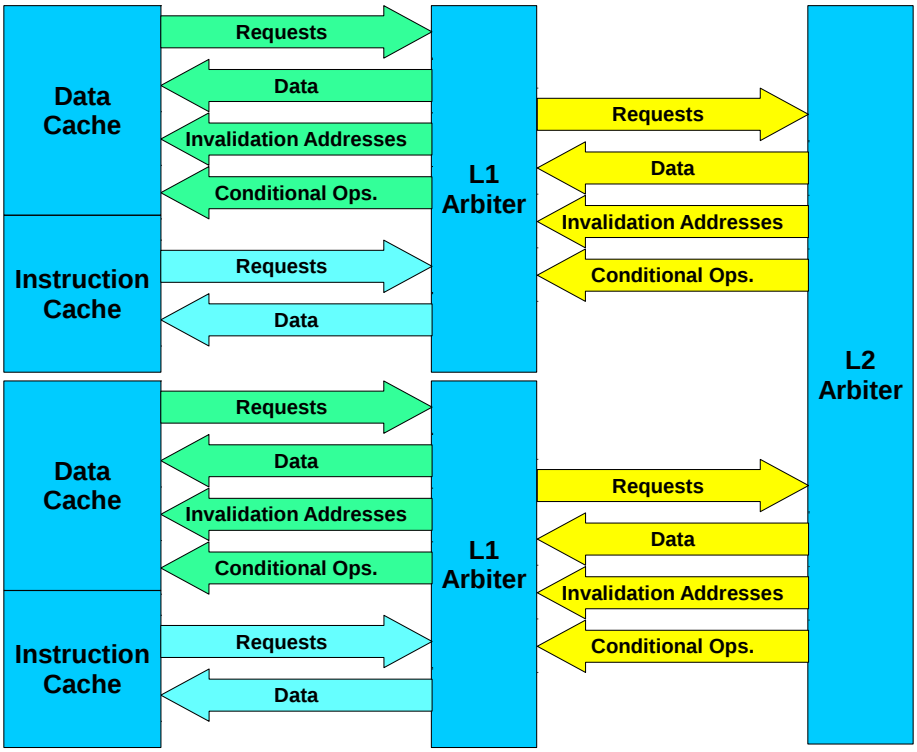

Figure 3.2 PolyBlaze's PBML connections

be 0x50000080, then 0x50000084, 0x50000088, and 0x5000008C, respectively.

3. Invalidation Queue: the third queue is used by coherency protocol handler. The coherency packets are issued by the Level-2 Arbiter and are transferred by L1 Arbiter. The size of teach packet is 32 bits at the moment and can be reduced to 30 bit later on.
4. Conditional Operation Results: the forth and the last queue carries the result of conditional operations. This queue is just one bit wide. A '1' value means a successful conditional operation, i.e. the SWX has gone through to memory, and a '0' value means that the conditional store operation has failed.

MicroBlaze Interface

The MicroBlaze interface is the set of internal signals to/from the processor's logic. The behaviour of these signals is very similar to MicroBlaze's original data cache. The small differences in these signals' behaviour are due to the difference in handling conditional operations.

3.1.1 Design Process

The presented memory architecture and caches in this thesis is the third generation of cache design for PolyBlaze. The first generation was a simple cache design with custom connections to memory for each cache, similar to MicroBlaze's behaviour. The main goal for this generation of PolyBlaze caches was to obtain the correct behaviour for MicroBlaze interface and the internal connections of caches and other parts of processor. The second generation of caches introduced PBML queues and the first versions of the L1 and the L2 Arbiters. Finally, the third generation was designed based on the extracted characteristics and behaviours of the system from the previous two generations. This generation uses a much clearer design flow that is easier to understand, expand and evaluate.

In addition, the optimizations that are added in the third generation allow us to build a quad-core PolyBlaze system with 4KB instruction and data caches that is able to run at 100MHz. These optimizations are mostly done by taking the original source code of MicroBlaze and following the same structures and methods used there to get better performance. Moreover, we have used modules such as BRAMs and comparators that are implemented in MicroBlaze targeting different FPGA architectures designed by Xilinx Inc. These modules will be optimized for different FPGA architectures and using them will allow us to get better resource usage and performance when working with MicroBlaze cores on Xilinx FPGAs.

3.2 Level-1 Instruction Cache

The MicroBlaze has an optional instruction cache, but in contrast with data cache, the existence of instruction cache is not necessary in PolyBlaze even if we want to boot an OS[4]. However, we have implemented a configurable instruction cache for the PolyBlaze that has a few features used in an asymmetric system. For instance, similar to data cache, we can choose different replacement policies. Figure 3.3 highlights the internal architecture of PolyBlaze's instruction cache. The blue modules exist in MicroBlaze's instruction cache. The green modules are added for PolyBlaze's instruction cache. A detailed explanation of each module's role in the cache follows. Since both data and instruction cache have been designed with similar goals in mind, their behaviour is very similar to each other.

Data Bank(s)

Similar to the data cache, the data bank in the MicroBlaze's instruction cache is just used to keep the instructions. It has several parameters such as the size of cache line or type of memory used to store the data (BRAM or LUTs). In the PolyBlaze's instruction cache, we have the same parameters. However, depending on the replacement policy, the number of used memory blocks can be different.

Tag Bank(s)

The tag bank in the MicroBlaze's instruction cache is used to keep the tag values and the valid bits. The available parameters for tag bank are similar to data banks: size of cache line and type of memory used to store values (BRAM or LUTs). In the PolyBlaze's instruction cache, we have the same parameters for tag bank. However, the number of used memory blocks can be different depending on the replacement

[4]Since the coherency of instruction caches between different cores are handled in software, we can use the original instruction cache of MicroBlaze if we want.

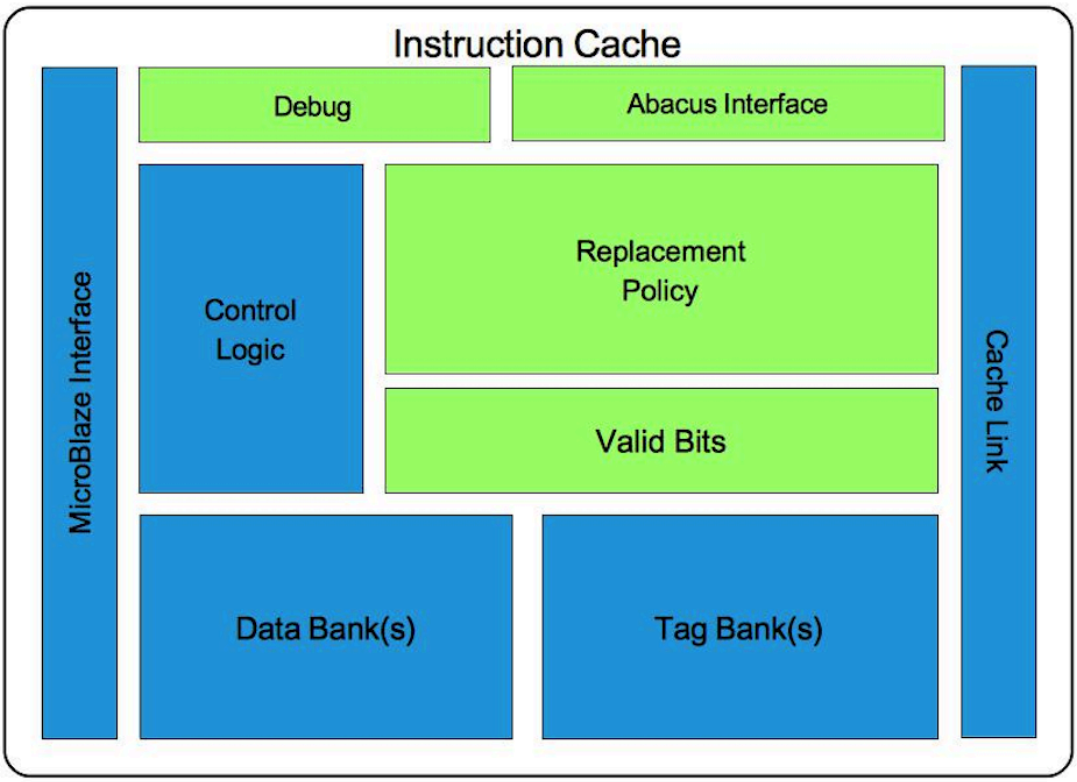

Figure 3.3 Instruction cache's overview in PolyBlaze

policy.

Replacement Policy

The MicroBlaze only implements a direct-mapped replacement policy, therefore, the logic for this module is embedded with the rest of logic in cache. In the PolyBlaze, we implement a dedicated module and make replacement decisions in that module in order to be able to define different replacement policies such as direct-mapped, LRU, Clock, and random replacement policy [5]. This module will also allow for asymmetric caches in a PolyBlaze system, since we can choose different replacement policies for different cores.

[5]In this thesis, we only implement direct-mapped and LRU replacement policies.

Valid Bits

As mentioned before, the valid bits in the MicroBlaze's instruction cache are kept along with tag values and are only updated when there is either a cache miss or the processor has issued a WDC instruction to invalidate a cache line in data cache. In the PolyBlaze, we have moved the valid bits into a separate memory bank. The main reason for separating this logic is to localize the logic that accesses the valid bits because it is part of the critical path in the processor. This logic consists of accesses for cache hits, updates to valid bits upon cache misses, and invalidation due to WIC instructions. In addition, the replacement policy module needs access to valid bits as well depending on the replacement policy used in the system.

ABACUS Interface

We can connect ABACUS to the instruction cache through the ABACUS interface. Using ABACUS, we can gather information about behaviour of instruction cache such as number of read requests, cache read misses, cache read hits, and so on. The gathered information can later be used to analyze the performance of an application.

Debugging Instruction Cache

Similar to data cache, in the design process, we have embedded some signals that are useful for debugging purposes. These signals range from key signals in the cache control logic to the PBML connection signals to hardware counters and checksum registers. ChipScope logic analyzer [40] can connect to these signals and gather information. When not being used, we can remove this logic by setting the *C_PB_ICACHE_USE_DEBUG* parameter of the PolyBlaze processor to '0'.

Control Logic

Control logic is the heart of instruction cache. It connects different modules, keeps track of timing for different signals and multiplexes the addresses for different modules such as data bank, tag bank, or valid bits. Moreover, the implemented logic for handling invalidations instructions, i.e. WIC, is part of control logic. These instructions are often handled in a single cycle. WIC instructions provide an address for invalidations. However, data instructions only looks at the cache line address and ignores the tag value. Therefore, a WIC instruction will invalidate the cache line regardless of a hit or miss on the tag part of the address.

PolyBlaze Memory Link (PBML)

The MicroBlaze uses XCL for direct connection from instruction cache to MPMC. However, as mentioned before, the PolyBlaze requires a more complicated memory hierarchy. For consistency with data cache, instruction cache uses the same memory hierarchy. The PBML for instruction cache consists of two different FIFOs. Instruction cache's PBML connections to the L1 Arbiter are colored cyan/blue in Figure 3.2.

1. Request Queue: the first FIFO is used to send request information to the L1 Arbiter. Each request is sent as a packet that contains 32 bits for the address of the operation. Since the instruction cache will only issue regular read operations, we do not need anything else in these packets.
2. Data Queue: the second FIFO is used to bring in the incoming data from memory to instruction cache. The incoming data will be transferred through L1 Arbiter. The width of this FIFO is 32 bits to bring in one word at a time. Similar to the data cache, the incoming words will come in order.

MicroBlaze Interface

The MicroBlaze interface is the set of internal signals from/to the processor's logic. The behaviour of these signals is very similar to MicroBlaze's original instruction cache.

3.3 A comparison of L1 Data and Instruction Caches

The general architecture of the data and instruction caches are very similar to each other; for example, the way they connect to replacement policy module is identical. However, there are certain features that are implemented quite differently. In the remainder of this chapter, we will talk about the differences between instruction and data caches, the reasons of such differences, and the implication of them.

Physical Address vs. Virtual Address

Since the requests to the data cache in the MicroBlaze, and similarly in the PolyBlaze, are initiated in EX stage of processor's pipeline, the input address to the cache has had enough time to be translated by the MMU. Therefore, all the addresses that go into data cache are physical addresses. However, the addresses that go into instruction cache could be physical or virtual addresses since they are initiated in IF stage. Due to this key difference, the data cache can always ignore the fixed part of memory addresses in tag comparison for a match whereas the instruction cache has to take them into account. Besides that, in presence of virtual memory, the instruction cache has to keep track of process identification and the type of cached address (virtual or not) as well in order to generate correct match signals.

Coherency Protocol

The data cache in the PolyBlaze is involved in a *write invalidation protocol* to maintain coherency between all data caches in a PolyBlaze system. In contrast, the instruction cache's coherency is not handled in the hardware. In a PolyBlaze system, if we need to maintain the coherency of instruction caches' contents, we will handle it in the software, i.e. OS [35].

4 Memory Architecture in PolyBlaze System

PolyBlaze, as a multicore processor system, requires a more complicated memory hierarchy than a single-core system based on MicroBlaze. There are several requirements for the design of this new memory hierarchy such as scalability, the ability to maintain memory coherency, and to provide a better performance for the entire system. Figure 4.1 illustrates an overview of memory architecture in a PolyBlaze system. As is apparent in this figure, each processor is connected to a L1 Arbiter. The L1 Arbiters, along with any hardware accelerator cores, and other modules that require access to coherent memory are all connected to the L2 Arbiter. From that point onward, all requests to main memory are merged into sequential access. Afterwards, we have a simple NPI module that simply translates the requests for MPMC. It is possible to have a shared/private Level-2 cache instead of the NPI module. Also, it is possible to have a Level-3 cache as well (and higher levels of cache), but that is outside of the scope of this thesis. The focus of this thesis is the L1 Arbiter, the PBMLs and their interface with the L2 Arbiter as well as the NPI interface. The rest of this chapter will discuss the proposed memory architecture with details explanations about different modules in this architecture, their roles, and their impacts over the entire system.

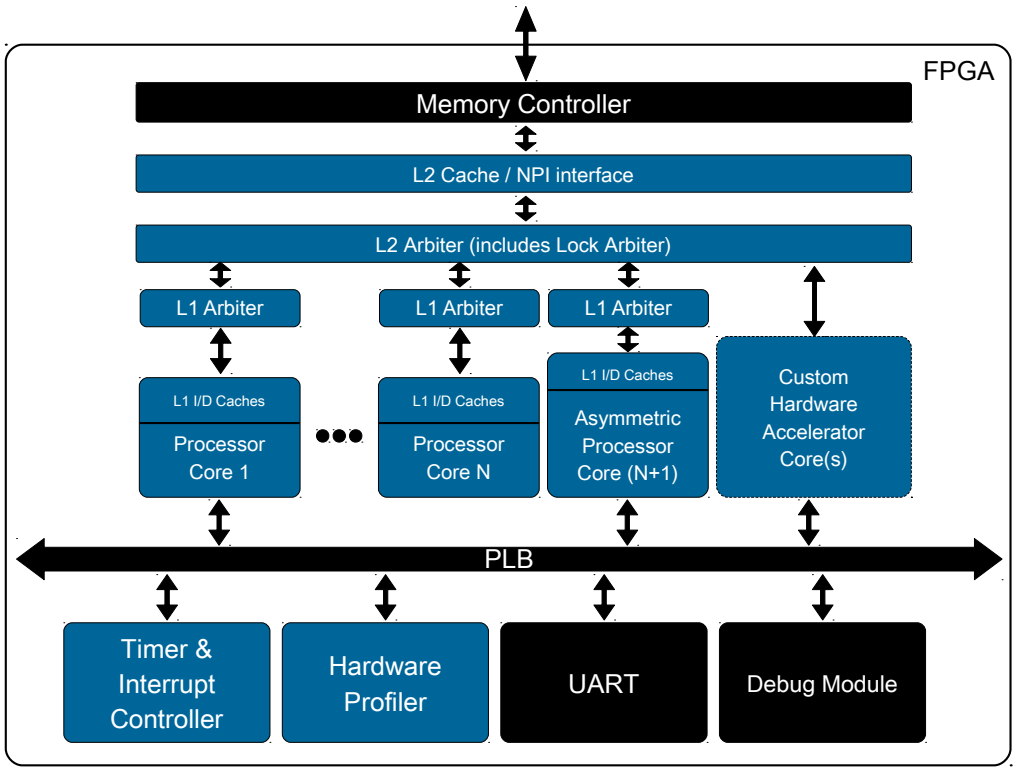

Figure 4.1 Memory Architecture in a PolyBlaze System

4.1 Level 1 Arbiter

The L1 Arbiter is a simple module due to its simple role in the system. The main reasons for its existence in the PolyBlaze system are compatibility and scalability. Specifically, the L1 Arbiter is used to multiplex the requests coming from instruction and data cache. The requests from each cache are fed into this module through their respective PBMLs. Then the Level-1 Arbiter makes a request from the L2 Arbiter and when data comes back, the L1 Arbiter marshals the data into the appropriate instruction or data cache. The unified requests of the L1 Arbiter are similar to those of the data cache since instruction cache requests can be a subset of data cache requests

(in format), as the instruction cache always reads data and never writes back (i.e. no self-modifying code). However, the L1 Arbiter has to tag its requests so that it knows to whom the incoming data belongs. For this purpose, a single bit is defined in the request packets: '1' for data cache requests and '0' for instruction cache requests.

The unification of outgoing requests from the L1 Arbiter helps with the compatibility of ports that go into L2 Arbiter. So the L2 Arbiter uses a series of generic ports that can be connected to L1 Arbiters, hardware accelerators, or any other module that requires to be part of coherent memory in the system.

In addition, it is possible to connect data or instruction caches directly into L2 Arbiter given that the ports on the L2 Arbiter are generic. For such cases, we can hardwire the extra logic. However, doing so would require two ports on the L2 Arbiter per processor, which would impact the scalability of the system, especially with the selection algorithms implemented in the L2 Arbiter, e.g. round-robin selection between ports. Moreover, the L1 Arbiter adds two extra cycles to memory latency. As such, we must try to address the trade-off between better scalability and lower latency for misses in the L1. Additionally, we have implemented prefetching units that are located in the L1 Arbiter. These prefetching units can help to minimize the increased latency added by the L1 Arbiter. The prefetching unit will be discussed later on in Section 4.4.

The L1 Arbiter connects to other modules via PBMLs, as was shown in figure Figure 3.2. Four queues connect L1 Arbiter to data cache:

1. Request Queue: the first queue is used to send request information from the data cache to the L1 Arbiter. Each request is sent as a packet that contains 32 bits for the address of the operation, one bit for the type of the operation ('0' for a write operation and '1' for a read operation), 32 bits for outgoing data

if the request is for a write operation, four bits representing the byte-enable bits for a write operation, and one bit indicating a conditional versus regular operation.

2. Data Queue: the second queue is used to send the incoming data from memory to the data cache. The incoming data will be transferred through the L1 Arbiter. The width of this FIFO is 32 bits to bring in one word at a time. The incoming words will come in order.
3. Invalidation Queue: the third queue is used by the coherency protocol handler. Coherency packets are issued by the L2 Arbiter and are transferred by the L1 Arbiter. The size of each packet is 32 bits at the moment and can be reduced to 30 and less bits if required in the future. The number of bits used in the cache to perform invalidation requests from coherency protocol, depends on a few architectural factors. In our implementation, we ignore the two least significant bits (bit indecies 1 and 0) since they are used for addressing bytes in each word. Additionally, we ignore a few more bits (2 or 3 bits) based on the cache line size since we are invalidating cache lines and not just words.
4. Conditional Operation Results: the forth queue carries the result of conditional operations. This queue is just one bit wide. A '1' value means a successful conditional operation, i.e. the SWX has gone through to memory, and a '0' value means that the conditional store operation has failed.

There are two more queues that connect the L1 Arbiter to instruction cache: 1) a request queue, and 2) data queue. Additionally, five queues connect the L1 Arbiter to the L2 Arbiter: 1) address request queue, 2) data write queue, 3) data read queue, 4) invalidation address queue, and 5) conditional operation results queue. Notice that requests from instruction and data caches will be divided into two parts and go through address request queue and data write queue to the L2 Arbiter.

4.2 Level 2 Cache/Memory Interface

This module is located between the L2 Arbiter and main memory. In this work, we have used a simple module that connects to the MPMC using a NPI interface. This module simply receives the requests from the L2 Arbiter and processes them. The NPI interface is a simple FIFO-based port on MPMC that accepts up to four outstanding read operations. For more details on this interface, please refer to LogiCORE IP Multi-Port Memory Controller documents at [41].

4.3 PolyBlaze Memory Link

PolyBlaze Memory Links consist of asynchronous or synchronous FIFOs with different depths and widths. These FIFOs allow different modules to work independently. Furthermore, asynchronous FIFOs allow different modules to work at different clock rates. For instance, the processor cores can operate at 100 MHz while the arbiters and memory interfaces can operate at 200 MHz. Additionally, different modules can work at different frequencies, which provides more options for an asymmetric system.

For each processor in the PolyBlaze framework, we can have up to 11 PBMLs as shown in yellow in Figure 3.2. If a core uses caches, four PBMLs will connect data caches to the L1 Arbiter, two links will be between instruction caches and the L1 Arbiter, and five links will connect the L1 and L2 Arbiters. In addition, there are four more links that connect L2 Arbiter to L2 Cache or NPI interface.

In MicroBlaze systems, XCL uses integrated FSL/FIFO connections to connect the processor to the MPMC. In PolyBlaze we have similar FIFOs, but they exist in NPI. PBML, however, provides its own FIFOs. Since these FIFOs are asynchronous, we can get modules such as L1 Arbiter, L2 Arbiter, and NPI to work at higher frequencies and service more processors in less time.

4.4 Prefetching Unit

The prefetching unit is a part of L1 Arbiter. In its simplest form, the L1 Arbiter only handles requests from data and/or instruction cache. When an optional prefetching unit exists in the memory path of either caches, after every read request from cache (e.g. address 0x1000), the prefetching unit will submit another request for the subsequent cache line (e.g. address 0x1001) and store it locally in a register. If the next read request from the cache happens to be for that line, then prefetching unit can return the data quickly and submit another request for the cache line after that (e.g. address 0x1002).

If the requested cache line does not exist in prefetching unit or the existing cache line in prefetching unit is not what the cache requests, a proper request will be submitted to memory. Additionally, the data in prefetching unit will be dropped and another request for the subsequent cache line will be submitted.

There are more complicated implementations for prefetching units. The algorithm used in L1 Arbiter is the simplest version (stride-1) which will assume we always want to read from subsequent addresses. This implementation is specially beneficial for instruction cache since program instructions are often in contiguous memory locations.

5 Evaluation and Experimental Results

In this chapter, our focus is to demonstrate the evaluation platform for PolyBlaze's L1 caches. First, in Section 5.1, we present the resource usage of different modules and systems. Later in Sections 5.2 and 5.3, we talk about system properties such as memory latency and operating frequency. Afterwards, we demonstrate the performance of PolyBlaze's multiple cores over MicroBlaze's single core.

5.1 Resource Usage

Table 5.1 presents the total resource usage of a single core MicroBlaze processor system with 4, 8, and 16 kB data caches all while the instruction caches in these three systems are 4kB in size. Table 5.2 shows the resource usage of single core MicroBlaze systems with 4kB data caches, and 4, 8, and 16kB instruction caches respectively. These two tables can be used as base-line measurements for resource usage presentation of different PolyBlaze configurations. Please note that the reported numbers are derived from place and route reports. Please note that the numbers are reported for the entire system and not just the processors or caches because in placement and routing phases, the optimizations are run in a global scale.

As we see in Tables 5.1 and 5.2, the number of BRAMs used in a system will grow with cache size. From the detailed numbers in resource usage logs from each systems (also from analyzing the numbers presented in these tables) we see that for each 4

Table 5.1 Post place and route resource usage for different data cache configurations with 4kB direct-mapped instruction cache in a single core MicroBlaze system

Data Cache Size	Line Size	Registers	LUTs	BRAMs
4 kB	4	7617	6518	26
	8	7672	6556	26
8 kB	4	7620	6523	27
	8	7674	6559	27
16 kB	4	7618	6520	29
	8	7673	6557	29

Table 5.2 Post place and route resource usage for different instruction cache configurations with 4kB direct-mapped data cache in a single core MicroBlaze system

Instruction Cache Size	Line Size	Registers	LUTs	BRAMs
4 kB	4	7617	6518	26
	8	7672	6556	26
8 kB	4	7620	6526	27
	8	7685	6561	28
16 kB	4	7618	6521	29
	8	7683	6556	29

kB of cache, we are using one instance of available BRAMs in the FPGA. Another interesting point is that line size (i.e. the number of words in each cache line) does not increase the number of BRAMs used in the system. With more words in a cache lines, we reduce the number of lines in each cache, so we can still use the same number of BRAMs for each instance of data cache. Obviously, cache size does not affect the rest of system so the number of LUTs and registers are not very different between different table entries. Moreover, the difference between the same data or instruction cache configurations in these two tables is negligible (most of the times zero). That is due to the fact that the data and instruction caches' structure and behavior are very similar to each other.

To get a better idea about system resources, we can look at the break down of resources used by different modules in a single core MicroBlaze system presented in

Table 5.3 Post place and route resource usage for different modules in a single core MicroBlaze system with 4kB direct-mapped (4 words per line) instruction and data caches.

Module	Registers	LUTs	BRAMs
MicroBlaze	2639	3080	5
Memory Controller	3679	2387	17
RS232 Uart	368	388	0
SysACE Compact Flash	199	95	0
Clock Generator	4	1	0
Debug Module	125	120	0
Data LMB	1	0	0
Data LMB Controller	2	6	0
Instruction LMB	1	0	0
Instruction LMB Controller	2	2	0
LMBs BRAMs	0	0	4
PLB	58	55	0
Reset Module	33	23	0
System	0	1	0
Interrupt Controller	506	360	0
Total	7617	6518	26

Table 5.3. As we can see, modules like LMBs (data or instruction), LMB controllers, clock generator, debug module, and others do not consume much resources compared to interrupt controller, memory controller and of course the MicroBlaze processor itself.

Table 5.4 Post place and route resource usage for different modules in a single core MicroBlaze system with several cache configurations (4kB direct-mapped, 4 words per line).

Module	Both Caches			DC only			IC Only			No Cache		
	Registers	LUTs	BRAMs	Registers	LUTs	BRAMs	Registers	LUTs	BRAMs	Registers	LUTs	BRAMs
MicroBlaze	2707	3076	5	2457	2880	3	2576	2829	3	2325	2659	1
Memory Controller	3682	2393	17	3614	2343	17	3533	2321	17	3283	2136	13
Interrupt Controller	506	360	0	506	360	0	506	360	0	506	360	0
RS232 Uart	368	387	0	368	386	0	368	391	0	368	387	0
SysACE CompactFlash	199	95	0	199	95	0	199	95	0	199	95	0
Clock Generator	4	1	0	4	1	0	4	1	0	4	1	0
Debug Module	125	120	0	125	120	0	125	120	0	125	120	0
Data LMB	1	0	0	1	0	0	1	0	0	1	0	0
Data LMB Controller	2	6	0	2	7	0	2	6	0	2	6	0
Instruction LMB	1	0	0	1	0	0	1	0	0	1	0	0
Instruction LMB Controller	2	2	0	2	2	0	2	2	0	2	2	0
LMBs BRAMs	0	0	4	0	0	4	0	0	4	0	0	4
PLB	80	210	0	80	210	0	80	210	0	80	210	0
Reset Module	33	23	0	33	23	0	33	23	0	33	23	0
System	0	1	0	0	1	0	0	1	0	0	1	0
Total	7710	6674	26	7392	6428	24	7430	6359	24	6929	6000	18

Table 5.5 Post place and route resource usage for MicroBlaze caches with 4kB direct-mapped 4 words per lines compared to the no cache baseline

Configuration	Registers	LUTs	BRAMs
Instr. Cache Only	10.8% (251)	6.4% (170)	200% (2)
Instr. Cache Only	5.8% (132)	4.6% (221)	200% (2)
Both Caches	16.4% (382)	15.7% (417)	400% (4)

Table 5.4 shows the break down of different modules in a single core MicroBlaze system with four different cache configurations where: 1) we do not have any caches in the system, 2) we only have instruction cache in the system, 3) we only have data cache in the system, and 4) we have both data and instruction caches in the system. In these systems, each cache, if present would be 4 kB direct-mapped with 4 words per cache line. As we can see in the table, the resource usage of most of the modules does not change (or the difference is negligible). The modules that change a lot are the MicroBlaze and the MPMC. The change in resource usage of MPMC is because of using XCLs for caches.

Table 5.5 shows the resource usage of caches for a single core MicroBlaze system with 4 kB direct-mapped caches (4 words per cache line). These percentage numbers are the difference between a baseline system with no caches and systems with different cache configurations. As we can see, when both caches are present in the system, the number of added resources is almost equal to the sum of components for individual caches. Numbers in parentheses are the differences of each resource.

Furthermore, Table 5.6 shows the resource usage of all modules in a single core MicroBlaze system with 4 kB direct-mapped instruction cache (4 words per cache line) and 4 kB or 16 kB direct-mapped data cache (4 words per cache line). As we can see, only resource usage of MicroBlaze changes. Even there the change to number of registers or LUTs is negligible and we only see an increase in the number of used

Table 5.6 Post place and route resource usage for different modules in a single core MicroBlaze system with 4 kB or 16 kB direct-mapped (4 words per line) data caches.

Module	4 kB			16 kB		
	Registers	LUTs	BRAMs	Registers	LUTs	BRAMs
MicroBlaze	2707	3076	5	2708	3080	8
Memory Controller	3682	2393	17	3682	2393	17
RS232 Uart	368	387	0	368	388	0
SysACE CompactFlash	199	95	0	199	95	0
Clock Generator	4	1	0	4	1	0
Debug Module	125	120	0	125	120	0
Data LMB	1	0	0	1	0	0
Data LMB Controller	2	6	0	2	6	0
Instr. LMB	1	0	0	1	0	0
Instr. LMB Controller	2	2	0	2	2	0
LMBs BRAMs	0	0	4	0	0	4
PLB	80	210	0	80	210	0
Reset Module	33	23	0	33	23	0
System	0	1	0	0	1	0
Interrupt Controller	506	360	0	506	361	0
Total	7710	6674	26	7711	6680	29

BRAMs.

Tables 5.7 to 5.10 present similar data to Tables 5.1 and 5.2 respectively, but for dual and quad core PolyBlaze systems. If we compare a dual core PolyBlaze system with a single core MicroBlaze system, both with 4 kB instruction and data caches, the resource usage of the dual core PolyBlaze system is almost twice the amount of resources used in a single core MicroBlaze system. This observation matches what we expect to see since not every system module is duplicated in a dual core system (e.g. clock generators, memory controllers, interrupt and timer handlers, etc.) In contrast to MicroBlaze, bigger cache size in PolyBlaze increases the number of used LUTs and registers. The difference is due to the modifications in cache structure and behavior that are necessary for new features in the system, especially for cache

Table 5.7 Post place and route resource usage for different data cache configurations with 4kB direct-mapped instruction cache in a dual core PolyBlaze system.

D$ Rep. Policy	Size	Line Size	Registers	LUTs	BRAMs
Direct-Mapped (1-way LRU)	4 kB	4	12814	16578	45
		8	12997	16658	45
	8 kB	4	12193	14057	55
		8	13579	19231	45
	16 kB	4	12213	14097	55
		8	12444	14401	63
2-way LRU	4 kB	4	13336	17187	45
		8	13848	17173	45
	8 kB	4	14501	20269	45
		8	14101	19573	45
	16 kB	4	14261	16063	65
		8	15763	25230	45
4-way LRU	4 kB	4	14539	17692	45
		8	15302	18901	45
	8 kB	4	15554	20526	45
		8	15807	20289	45
	16 kB	4	18887	26793	45
		8	16815	25083	45

coherency. Please note that for quad core systems, some numbers are missing. In these configurations, the tools have failed to build the system for our Virtex 5 board due to the available resources on the board not being enough to support everything.

The anomalies in the Tables 5.7 and 5.8 are often due to the heuristics used in different algorithms used by tools used to build systems, e.g. synthesis, mapping, placement and routing tools. Playing with random seeds used in these algorithms, we can get better results for resource usage of a system. The numbers presented here in Tables 5.1, 5.2, 5.7 and 5.8 are all generated with a fixed random seed.

Table 5.11 shows the break down of different modules in a dual core PolyBlaze system with two different data cache configurations: 1) configuration A has 4 kB direct-

Table 5.8 Post place and route resource usage for different instruction cache configurations with 4kB direct-mapped data cache in a dual core PolyBlaze system.

I$ Rep. Policy	Size	Line Size	Registers	LUTs	BRAMs
Direct-Mapped (1-way LRU)	4 kB	4	12814	16578	45
		8	12761	15713	53
	8 kB	4	13441	18285	45
		8	13074	16633	53
	16 kB	4	14675	21672	45
		8	13692	18319	53
2-way LRU	4 kB	4	13343	17254	53
		8	13089	16100	69
	8 kB	4	14482	19610	53
		8	13603	17379	69
	16 kB	4	16747	24110	53
		8	14731	19666	69
4-way LRU	4 kB	4	14007	17620	69
		8	13498	16201	101
	8 kB	4	15537	20095	69
		8	14263	17658	101
	16 kB	4	18836	25787	69
		8	15788	20148	101

mapped data caches with 4 words per cache line, and 2) configuration B has 16 kB 4-way LRU data caches with 4 words per cache line. In both configurations, processors have 4 kB direct-mapped instruction caches with 4 words per cache line. Again, as we can see in the table, the resource usage of most of the modules does not change (or the difference is negligible). The added modules such as L1 Arbiter, different PBML queues. and NPI interface are independent of cache configurations. In addition, they are negligible in size compared to the processor itself. The modules that change a lot are the MicroBlaze and the MPMC. The change in resource usage of MPMC is because of using XCLs for caches and of course MicroBlaze's resource usage changes with different configurations.

In Table 5.12, we compare a MicroBlaze core with a PolyBlaze core. In this table,

Table 5.9 Post place and route resource usage for different data cache configurations with 4kB direct-mapped instruction cache in a quad core PolyBlaze system.

D$ Rep. Policy	Size	Line Size	Registers	LUTs	BRAMs
Direct-Mapped (1-way LRU)	4 kB	4	21105	29563	81
		8	21354	28059	97
	8 kB	4	19896	24555	101
		8	22522	33110	97
	16 kB	4	19933	24567	101
		8	20285	23083	133
2-way LRU	4 kB	4	22150	30795	81
		8	23057	29227	97
	8 kB	4	24482	37080	81
		8	23565	33981	97
	16 kB	4	23998	28592	121
		8	-	-	-
4-way LRU	4 kB	4	24557	32409	81
		8	25968	32617	97
	8 kB	4	26585	37457	81
		8	26977	35432	97
	16 kB	4	-	-	-
		8	-	-	-

we have excluded modules such as L2 Arbiter and NPI interface even though they add to the total resource usage of PolyBlaze system. The reason is that since we only need one instance of these modules, we do not count them in comparing a MicroBlaze core with a PolyBlaze core. One import reason in the increased number of registers between PolyBlaze and MicroBlaze is the way we handle valid bits in caches (as explained in Section 3.1, we use register banks to store valid bits). Furthermore, the reason we use about 83.5% more LUTs in PolyBlaze is mostly because of coherency handling and all of multiplexers than need to be in the system for it. Also, it is worth mentioning that the amount of used resources can grow even higher if we change replacement policy of PolyBlaze (from direct-mapped).

Table 5.10 Post place and route resource usage for different instruction cache configurations with 4kB direct-mapped data cache in a quad core PolyBlaze system.

D$ Rep. Policy	Size	Line Size	Registers	LUTs	BRAMs
Direct-Mapped (1-way LRU)	4 kB	4	21105	29563	81
		8	21354	28059	97
	8 kB	4	22361	32970	81
		8	21985	29776	97
	16 kB	4	24829	39691	81
		8	23217	33121	97
2-way LRU	4 kB	4	22167	30895	97
		8	22013	29076	129
	8 kB	4	24441	35675	97
		8	23037	31376	129
	16 kB	4	28986	44678	97
		8	25299	35942	129
4-way LRU	4 kB	4	23528	32661	129
		8	-	-	-
	8 kB	4	26553	37391	129
		8	-	-	-
	16 kB	4	33227	48189	129
		8	-	-	-

Table 5.11 Post place and route resource usage for different modules in a dual core PolyBlaze system.

	A: 4kB DM			B: 16 kB 4-Way LRU		
Module	Registers	LUTs	BRAMs	Registers	LUTs	BRAMs
MicroBlaze 1	3673	5357	5	6708	10432	5
MicroBlaze 2	3673	5298	5	6711	10437	5
Data PBML Conditional Queue 1	23	42	0	23	42	0
Data PBML Conditional Queue 2	23	42	0	23	42	0
Data PBML Data Queue 1	23	158	0	23	158	0
Data PBML Data Queue 2	23	158	0	23	158	0
Data PBML Invalidation Queue 1	23	131	0	23	131	0
Data PBML Invalidation Queue 2	23	131	0	23	131	0
Data PBML Request Queue 1	23	256	0	23	256	0
Data PBML Request Queue 2	23	256	0	23	256	0
Instr. PBML Request Queue 1	23	152	0	23	151	0
Instr. PBML Request Queue 2	23	151	0	23	151	0
Instr. PBML Response Queue 1	23	158	0	23	158	0
Instr. PBML Response Queue 2	23	158	0	23	158	0
L1 Arbiter 1	2	43	0	2	43	0
L1 Arbiter 2	2	43	0	2	43	0
Data LMB Controller 1	2	6	0	2	6	0
Data LMB Controller 2	2	6	0	2	6	0
Instr. LMB Controller 1	2	2	0	2	2	0
Instr. LMB Controller 2	2	2	0	2	2	0
LMB BRAM 1	0	0	4	0	0	4
LMB BRAM 2	0	0	4	0	0	4
Memory Controller	3011	2037	17	3011	2039	17
L2 Arbiter	803	698	10	803	696	10
NPI	49	121	0	49	121	0
Total	12814	16576	45	18887	26793	45

Table 5.12 Increased resource usage (after place and route) for PolyBlaze compared to MicroBlaze

	Module	Registers	LUTs	BRAMs
MicroBlaze	MicroBlaze	2707	3076	5
	XCL	671	356	0
	Total	3378	3432	5
PolyBlaze	MicroBlaze	3673	5357	5
	PBML	138	897	0
	L1 Arbiter	2	43	0
	Total	3813	6297	5
Difference	Count	435	2865	0
	Percentage	12.9%	83.5%	0%

5.2 Timing Analysis

In this section, we briefly talk about operating frequency of a PolyBlaze system and compare it with that of a MicroBlaze system. We cannot present detailed information about critical paths due to copyright issues on MicroBlaze. Table 5.13 shows the maximum operating frequency of a few different systems. These numbers are coming from synthesis reports. In practice, we rarely get these numbers. For example, on our Virtex 5 boards, we can have MicroBlaze cores working at a maximum of 125 MHz[1]. In our implementation for PolyBlaze, we see a decrease from 152.207 MHz to 135.099 MHz. Some of this loss comes from coherency mechanism in data cache since it effects the critical path. Additionally, part of the loss comes from the different approaches we take in implementing different functionalities (e.g. cache hits, valid bits, etc.).

We see further decrement in the maximum operating frequency as the cache size grows. The same reasons as described earlier apply here. In our evaluations we have set our goal to 80 MHz. The lower number allowed design tools to faster reach their target frequency and reduced the build time.

Table 5.14 presents the maximum operating frequency of different modules in a dual core PolyBlaze system. As we can see, most of these modules have the potential to run at much faster speeds. For instance, L2 Arbiter which is a bottleneck in PolyBlaze systems can operate at 318.674 MHz (practically less than this, but still much more than 80 MHz).

Finally, it is worth mentioning that there is a lot of room for improvement here. Optimizing our implementation to use available logic in the FPGA better is one of many ways to improve the maximum operating frequency. Another approach could be modifying MicroBlaze's pipeline and add a sixth stage. Having an extra stage in

[1]Higher numbers might be still reachable, but 125 MHz is what design tools guarantee.

Table 5.13 Maximum operating frequencies from different systems.

Processor	I-Cache Config	D-Cache Config	Max Operating Frequency
MicroBlaze	4 kB direct-mapped	4 kB direct-mapped	152.207 MHz
MicroBlaze	4 kB direct-mapped	16 kB direct-mapped	152.207 MHz
PolyBlaze	4 kB direct-mapped	4 kB direct-mapped	135.099 MHz
PolyBlaze	4 kB direct-mapped	4 kB 4-Way LRU	135.099 MHz
PolyBlaze	4 kB direct-mapped	16 kB 4-Way LRU	128.700 MHz
PolyBlaze	16 kB 4-Way LRU	4 kB direct-mapped	128.700 MHz

Table 5.14 Maximum operating frequencies of different modules in a dual core PolyBlaze system.

Module	Max Operating Frequency
MicroBlaze	152.207 MHz
L1 Arbiter	942.507 MHz
L2 Arbiter	318.674 MHz
NPI	330.943 MHz
PBML	347.222 MHz
MPMC	259.134 MHz
Interrupt Controller	197.394 MHz
Clock Generator	1239.157 MHz
Debug Module	196.155 MHz
RS232 Uart	187.829 MHz
SysACE CompactFlash	408.497 MHz
Reset Module	406.009 MHz

the pipeline will allow us to break the critical path and improve maximum operating frequency. A third approach is to synthesize for a different FPGA chip. These approaches are some examples of how we can improve operating frequencies.

5.3 Latency

In this section we present the memory access latency for read operations and how that scales with the number of cores. In this test, ABACUS is connected to data cache on Core #0 and monitors the latency of all the incoming read operations. In

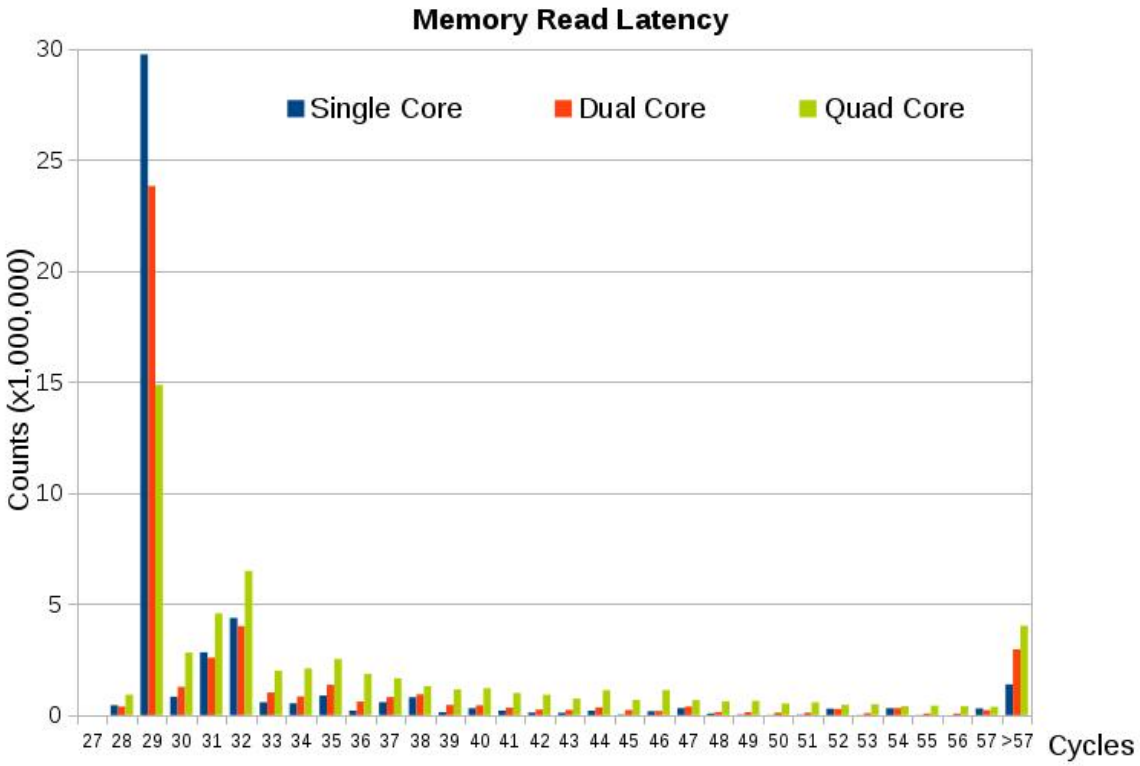

Figure 5.1 Memory Read Operation Latencies

order to capture the number of read latencies from 1 to 512, we have designed a latency unit in ABACUS with 512 total bins. For this test, we have used a quad core system with 4kB 4-Way LRU instruction and data caches. In three different stages of this experiment, we have booted the OS with one, two, or four cores. Then we have measured the latency of read operations from running *specrand* and binding it to the first processing core in the system. In measurements for dual and quad core systems, we have bound extra instances of *specrand* to the other active cores in the system while the main benchmark is being executed by the test core (Core #0).

Figure 5.1 shows the number of memory requests that took between 27 to 57 clock cycles. Anything higher than 57 cycles has been included in the rightmost column

(> 57). As you can see, for a single core system, most of memory read operations (if we do not have a hit in data cache) take about 29 to 32 cycles. As the number of active cores grows and with it the number of running benchmarks, we see that memory latency increases too. For instance, the number of memory read operations that take 29 cycles drops by about 50% in a quad core system compared to a single core system. At the same time, we see that the number of memory read operations that take between 30 to 35 cycles in a quad core system is more than twice of the same number in a single core system.

Another point worth noting is that we still see the bulk of memory read latencies for the quad core system to be around 29 to 35. If everything in the system was completely serialized, we expected the latencies to increase dramatically, but that is not the case here. The reason for this behavior is that the MPMC module can accept up to four outstanding read requests. Since processors often have one outstanding read requests we will not be able to see the real effect of increasing the number of active processing cores in the system. For that to happen, we need more than four processors.

Additionally, we can see that the ratio between the number of memory read operations that take more than 57 clock cycles and those that take 29 clock cycles, is very different. In a single core system, about 3% of memory read operations (excluding cache hits) take 58 clock cycles and more whereas in a quad core system, about 7% take that long.

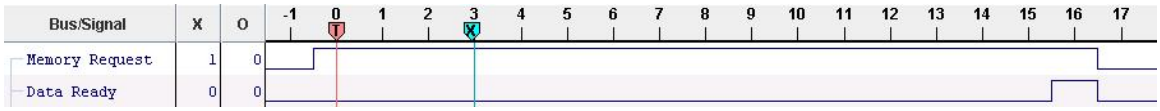

Figure 5.2 Waveform of a memory read miss operation for MicroBlaze

Figure 5.2 shows the waveform of a memory read operation for MicroBlaze processor

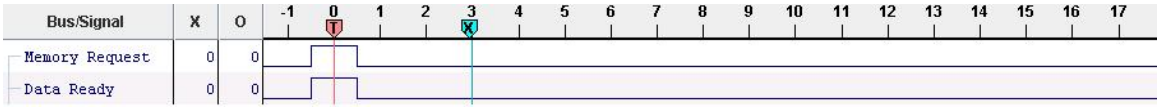

Figure 5.3 Waveform of a memory read hit operation for MicroBlaze

(read miss). If we have a hit, data would be ready in a single cycle as shown in Figure 5.3. As we can, see for a cache miss, we have a 17 cycle latency. Please note that this number is for a read operation without any interference from other modules at all. In other words, no one else is attempting to read from memory when this waveform was captured.

For PolyBlaze processor, the best case scenario for memory read latency on a cache miss is 27 clock cycles as shown in Figure 5.5. Compared to MicroBlaze, this is about 60% more clock cycles in a typical memory operation. To give you a better perspective as why we have about 10 more clock cycles in each memory read, we can look at the added delay in different modules that now sit between the processor and MPMC.

Figure 5.6 shows the latency of a memory read operation from the perspective of L1 Arbiter. As we can see in this figure, we have a 23 clock cycle latency. Since L1 Arbiter is working with the same frequency as processor here (80 MHz), we have a 4 clock cycle difference between L1 Arbiter and processor. These 4 clock cycles are spent in the asynchronous FIFOs between L1 Arbiter and data cache, i.e. PBML.

Furthermore, Figure 5.7 shows the latency of a memory read operation from the perspective of L2 Arbiter. Here the latency is about 21 clock cycles. Again, since L2 Arbiter is working with the same frequency as L1 Arbiter (80 MHz), we have a 2 clock cycle difference between them. Similarly, these 2 clock cycles are spent in the asynchronous FIFOs between L1 and L2 Arbiters, i.e. PBML.

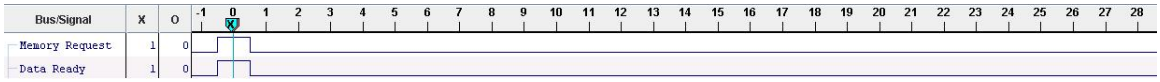

Figure 5.4 Waveform of a memory read hit operation for PolyBlaze

Finally, Figure 5.8 shows the latency of a memory read operation from the perspective of NPI port to MPMC. Here the latency is about 28 clock cycles. However, NPI is working twice as fast as L2 Arbiter (160 MHz). So in our comparisons, from processor's perspective, we have a 14 clock cycle latency. The difference between NPI and L2 Arbiter is about 7 clock cycles, but this time not all of that delay is because of the delay caused by the asynchronous FIFOs in PBML between L2 Arbiter and NPI. If we look at Figure 5.7 again, we see that in L2 Arbiter, there is a 2 clock cycle delay between receiving a request from L1 Arbiter and actually sending it to NPI. This delay is part of the logic of L2 Arbiter. We are not going into details about how this module works here. All of this being said, we still have a single cycle response time in the caches if we have a cache hit as shown in Figure 5.4.

An interesting fact here is that in our FPGA-based system, the average memory latency is about 20 to 30 clock cycles where as in real world systems, we see numbers in the range of multiple hundreds of clock cycles (based on the number of cache levels in a system). Therefore we need a way to model real world systems if want to do more accurate research. The simplest way to mimic the behavior of real world systems for us is to add delays in different stages of our system. For instance, assuming that we do not have a L2 cache present in the system, we can put delays in the NPI and send processor requests after a random amount of time (in the range of a few hundreds of clock cycles). If a L2 cache is present in the system, then we can put multiple delays before and after L2 cache.

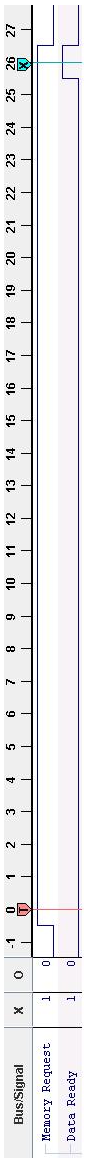

Figure 5.5 Waveform of a memory read miss operation for PolyBlaze (80 MHz)

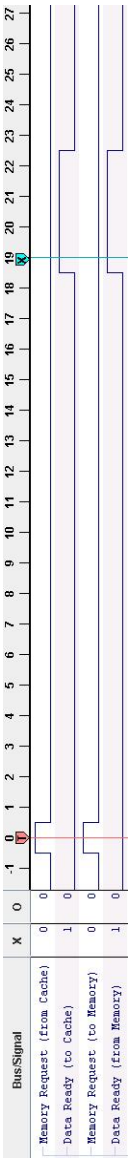

Figure 5.6 Waveform of a memory miss operation for L1 Arbiter in a PolyBlaze system (80 MHz)

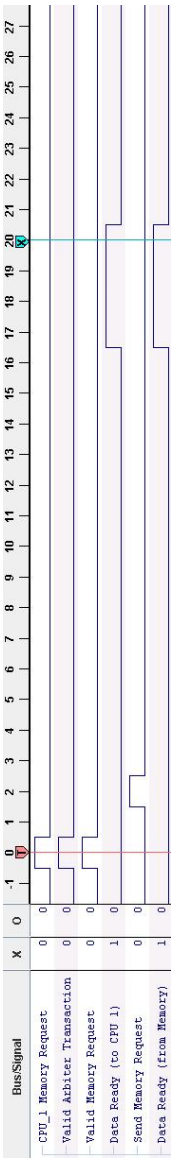

Figure 5.7 Waveform of a memory read operation for L2 Arbiter in a PolyBlaze system (80 MHz)

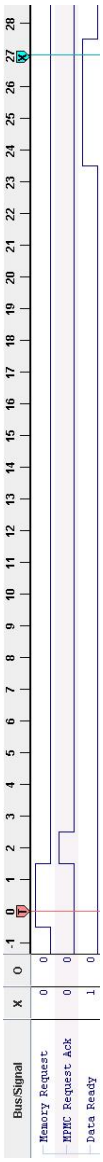

Figure 5.8 Waveform of a memory read operation for NPI in a PolyBlaze system (160 MHz)

5.4 Application Performance Study

In this section, we talk about the application performance on different cache configurations. For the purpose of meaningful analyses, we only look at data cache configurations[2]. In other words, we do not want to change too many variables at the same time; otherwise, we cannot make any meaningful statements about the difference in system behavior. All processing cores in the systems used in this study include a 16kB, 4-Way LRU instruction cache with 4 words per cache line. The data cache configurations vary in two different aspects: 1) number of ways in LRU replacement policy and 2) cache size.

This study is divided into two separate phases. In the first phase, we run each benchmark application with minimal interference from other processes [3] in order to get accurate data about that application's behavior. Moreover, we investigate each benchmark's performance on different data cache configurations and analyze its behavior.

Then, we take the results of the first phase and build a dual-core system with asymmetric data cache configurations to use in the second phase. We choose the data cache configurations based on configurations that work best for each pair of benchmark applications. Afterwards, we run two benchmark applications simultaneously on this dual-core system pinning each application to one of the cores. We also run the same pair of benchmark applications in reverse order meaning we pin each application to the core that does not have the preferred cache configuration. We expect the results from pinned benchmark applications to the core with their preferred cache

[2]It is possible to freeze data caches and look at different instruction cache configurations (or any other module in the system for that matter).

[3]Other major processes include system tasks and ABACUS profiling that are running in the background. There is no other benchmark process or any other user initiated application running on the system while the experiment is being run.

Table 5.15 Data cache read miss rate for bzip

	4 kB	8 kB	16 kB	32 kB
1-Way (Direct Mapped)	8.58	6.85	5.45	4.53
2-Way	7.06	5.73	4.67	3.89
4-Way	6.61	5.43	4.48	3.73

Table 5.16 Data cache read miss rate for libquantum

	4 kB	8 kB	16 kB	32 kB
1-Way (Direct Mapped)	18.44	17.83	17.57	17.07
2-Way	18.15	17.50	17.32	17.3007
4-Way	17.96	17.38	17.30	17.2951

configuration show an improvement over the other setup (with less desired cache configuration).

5.4.1 Phase 1: Single-Process Application Performance

In this section we talk about the application performance on different cache configurations. As mentioned before, for the purpose of a meaningful analysis, we only look at data cache configurations. All processing cores in the systems used in this study include a 16kB, 4-Way LRU instruction cache with 4 words per cache line. The data cache configurations vary in two different aspects: 1) number of ways in LRU replacement policy and 2) cache size. Tables 5.15 to 5.18 show the data cache miss rates of the bzip, libquantum, specrand, and h264ref benchmarks respectively. Similarly, Figures 5.9 to 5.12 illustrate the data cache miss rates of the benchmarks for easier comparison.

Table 5.17 Data cache read miss rate for specrand

	4 kB	8 kB	16 kB	32 kB
1-Way (Direct Mapped)	14.72	8.30	6.69	2.38
2-Way	12.61	6.57	1.76	0.87
4-Way	10.76	3.71	1.05	0.56

Table 5.18 Data cache read miss rate for h264ref

	4 kB	8 kB	16 kB	32 kB
1-Way (Direct Mapped)	7.79	5.94	3.09	2.18
2-Way	4.30	2.87	2.07	1.75
4-Way	4.14	2.60	1.92	1.52

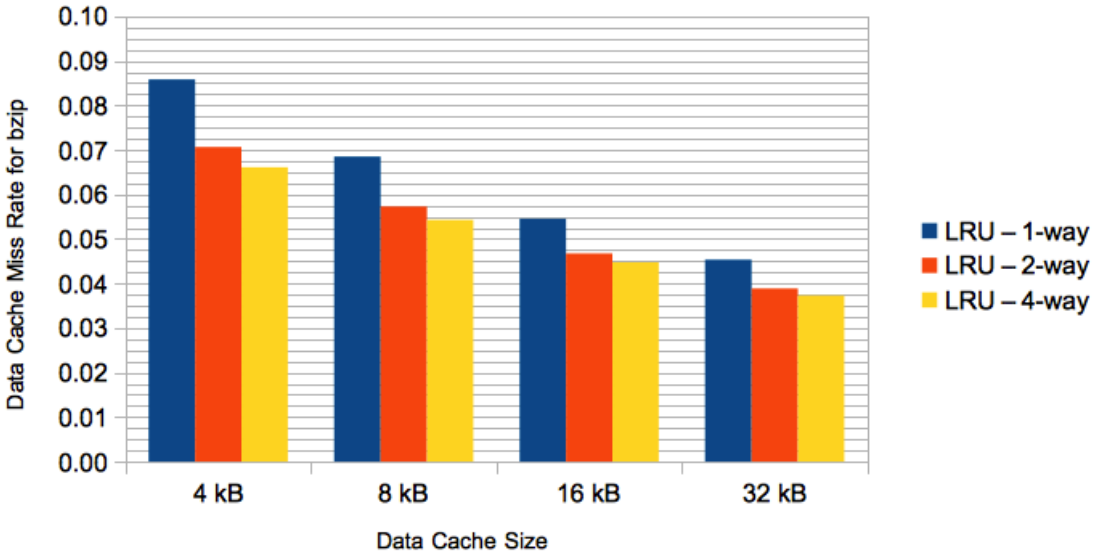

Figure 5.9 Data cache read miss rate for bzip

These results show that specrand and h264ref are more sensitive to replacement policy changes than bzip and libquantum. Also, we can see that overall, these benchmarks are more sensitive to cache size rather than the replacement policies. Additionally, we see a 14.16% cache miss rate improvement between the best case and worst case scenario in specrand while the same number for h264ref, bzip, and libquantum is 6.28%, 4.85%, and 1.37%, respectively.

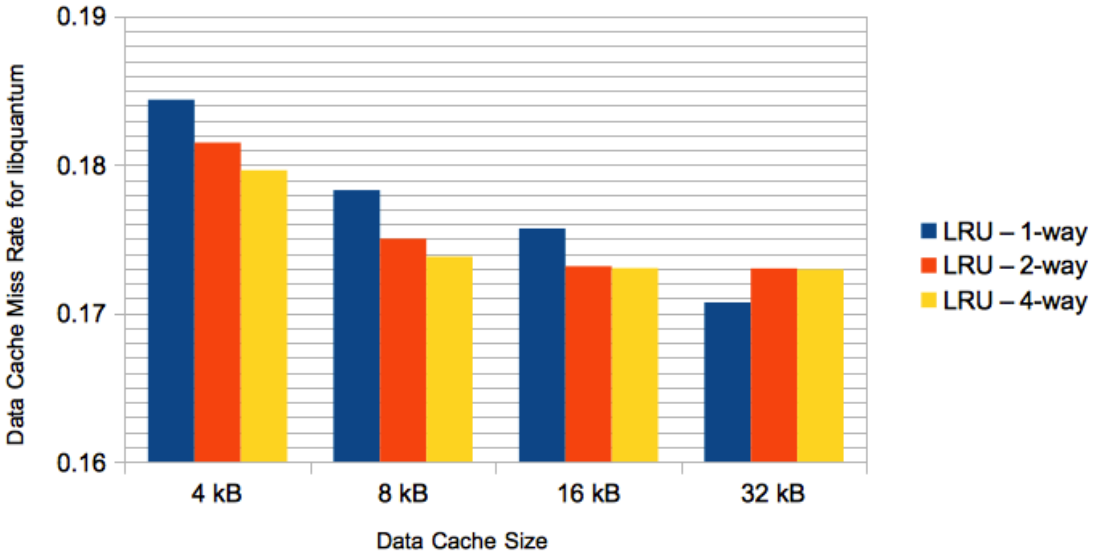

Figure 5.10 Data cache read miss rate for libquantum

5.4.2 Phase 2: Multi-Process Application Performance

In this section, we talk about the application performance on different cache configurations. For the purpose of meaningful analyses, we only look at data cache configurations. All processing cores in the systems used in this study include a 16kB, 4-Way LRU instruction cache with 4 words per cache line. The data cache configurations vary in two different aspects: 1) number of ways in LRU replacement policy

Table 5.19 Number of Load/Store instructions and their ratio in each benchmark application during its execution time

Benchmark	Load Instr.	Store Instr.	Memory Instr.	Load/Store Ratio	I-Cache Read Miss Rate
h264ref	17.22%	6.39%	23.61%	2.70	1.33%
bzip	14.55%	6.82%	21.37%	2.13	0.46%
libquantum	6.50%	3.40%	9.89%	1.91	0.51%
specrand	6.18%	3.89%	10.07%	1.59	5.50%

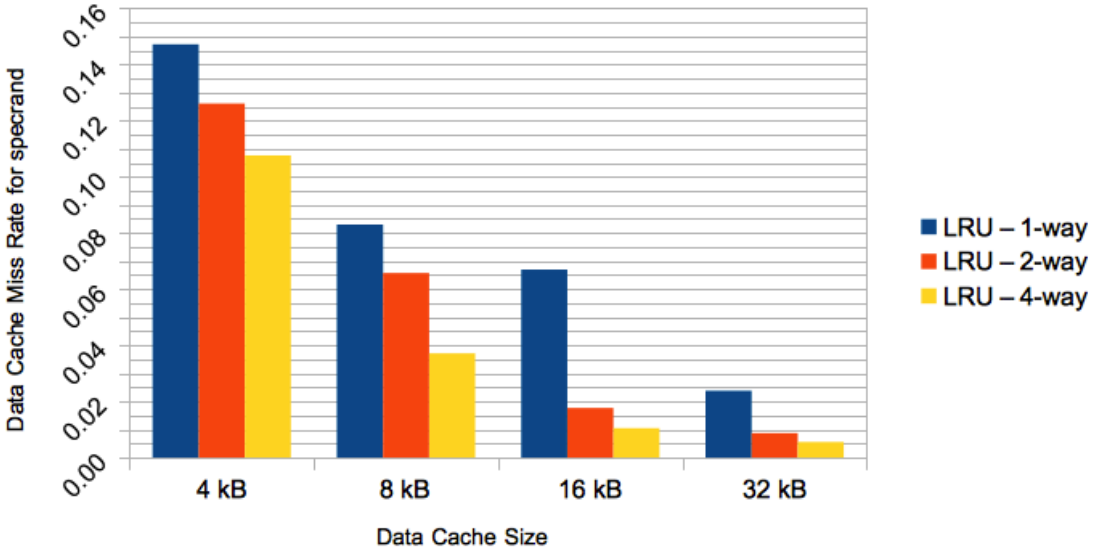

Figure 5.11 Data cache read miss rate for specrand

and 2) cache size.

As mentioned before, based on the results of the phase 1, we pick two cache configurations such that each works for a pair of our benchmark applications better than the other and vice versa. Then we build a dual-core system using each cache configurations for one of the data caches present in the system. According to our data from the first phase, we have picked an 8kB Direct-Mapped data cache for core A and an 4kB 4-way LRU data cache for core B. As you can see from Figures 5.9 to 5.12, libquantum and specrand prefer configuration A over B whereas h264ref and bzip prefer configuration B over A. For easier comparison, the numbers for these two configuration are also present in Table 5.20.

The pair of benchmarks used in phase 2 are h264ref and bzip for configuration A (i.e. 4kB 4-way LRU cache) and libquantum and specrand for configuration B (i.e. 8kB

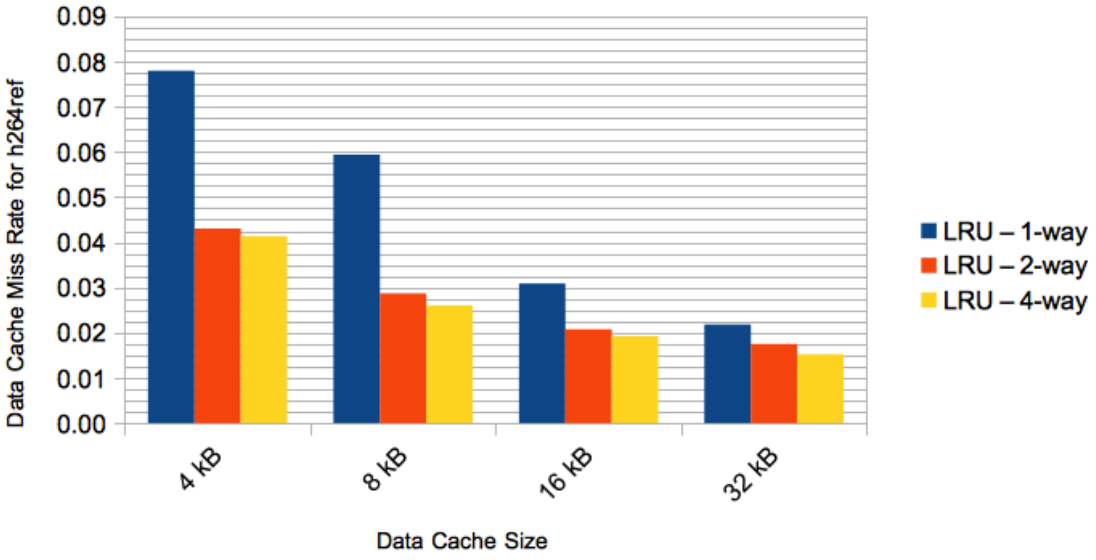

Figure 5.12 Data cache read miss rate for h264ref

Table 5.20 Data cache read miss rate of the four benchmark applications for the two selected cache configurations. Optimal assignments are shown in bold.

	4kB 4-way LRU (A)	8kB Direct-Mapped (B)
h264ref	**4.14**	5.94
bzip	**6.61**	6.85
libquantum	17.96	**17.83**
specrand	10.76	**8.30**

Direct-Mapped cache). In phase 2, we run two-process experiments, pinning one of the benchmark selected from mentioned pairs to one of the available two cores in the system. During the experiments, while profiling each process, the shorter processes were kept operating in an infinite loop until the longer process that was being profiled was finished. This would ensure that the measurements for the longer process are valid throughout its entire life time.

In different experiments, we pin the benchmark applications to both cores. A core

Table 5.21 Data cache read miss rate of the two-process experiments with optimal assignments

	h264ref	bzip	libquantum	specrand
h264ref	-	-	4.19 (4.14)	4.77 (4.14)
bzip	-	-	6.69 (6.61)	7.37 (6.61)
libquantum	17.80 (17.83)	17.94 (17.83)	-	-
specrand	9.76 (8.30)	8.08 (8.30)	-	-

Table 5.22 Data cache read miss rate of the two-process experiments with suboptimal assignments

	h264ref	bzip	libquantum	specrand
h264ref	-	-	4.25 (5.94)	6.30 (5.94)
bzip	-	-	6.84 (6.85)	7.39 (6.85)
libquantum	17.86 (17.96)	17.67 (17.96)	-	-
specrand	10.39 (10.76)	10.45 (10.76)	-	-

assignment in an experiment is an optimal assignment if the benchmark applications are pinned to the core with their preferred cache configurations. If the benchmark applications are pinned to the other core, which has a data cache with the less preferable configuration, we call that a suboptimal assignment.

What we expect to see in optimal situations is that data cache read miss rates in optimal assignments is lower than suboptimal assignments. Also, we expect to see that data cache read miss rates in two-process experiments are higher than single-process experiments.

Table 5.21 shows the data cache read miss rate of the two-process experiments in the optimal assignments with each benchmark application pinned to the core with its preferred cache configuration. The numbers in parentheses are data cache read miss rates for the single-process experiments from stage 1 on each cache configuration.

Table 5.22 is similar to Table 5.21, but it show the data cache read miss rate of the two-process experiments in the suboptimal assignments with each benchmark application

Table 5.23 Run time difference in seconds between optimal and suboptimal assignments in presence of different interfering applications

	h264ref	bzip	libquantum	specrand
h264ref	-	-	255	2588
bzip	-	-	13	7
libquantum	1	25	-	-
specrand	554	561	-	-

pinned to the core with the less desirable cache configuration. Again, the numbers in parentheses are data cache read miss rates for the single-process experiments from stage 1 on each cache configuration.

Besides a decrease in data cache miss rate between optimal and suboptimal assignments, we expect to see that application run times decrease too. Application data cache miss rate only takes the conflicts between applications into account. However, application run time takes the increased memory operation latency into account as well. Table 5.23 shows the difference between run times of our selected benchmarks in presence of different interfering applications. For instance, h264ref runs 255 seconds faster in an optimal assignment when libquantum is interfering on the other core.

Furthermore, Table 5.24, presents the percentage of increased run time for different assignments in presence of various interfering applications. The base line for these numbers are run times from the same assignments without interference. As we can see, any interference will increase the run time. However, even in presence of interfering applications, the run time of the benchmark application in optimal assignments are lower than suboptimal assignments. The only anomaly in the table is the negative numbers for specrand which probably comes from a measurement error in the baseline numbers. Despite this error, the overall behavior is still correct.

All that being said, our expectation was that we get better performance in opti-

Table 5.24 Percentage of increased run time for different assignments in presence of various interfering applications

Benchmark	Optimal Assignment	Suboptimal Assignment
h264ref	1.74%	6.42%
bzip	0.75%	2.23%
libquantum	0.78%	1.82%
specrand	-1.41%	-0.75%

mal assignments over suboptimal assignments and anything in between (if possible). Looking at two different measures, data cache miss rate and application run time, we see that optimal assignments yield better results. Therefore, if we gather these measurements on-line, we can exploit asymmetry in a system and benefit from it.

6 Conclusion and Future Work

Asymmetric architectures in the modern computing systems provide many opportunities. Efficient system utilization and better over-all performance are just two examples of such opportunities. However, there are many challenges in the way to reach these goals. For instance, system research requires better insight into the behavior of a multi-core system in order to be able to take advantage of asymmetric configuration of that system.

6.1 Conclusions

In this thesis, we have presented a asymmetric cache design based on MicroBlaze processor. As part of the extensions on PolyBlaze platform, we have outlined and implemented coherent L1 data and instruction caches along with L1 Arbiters. The arbitration modules help with system scalability by providing separate layers of multiplexers between instruction and data caches in a single processing unit and others. Additionally, the L1 Arbiter modules provide uniform interfaces to the L2 Arbiter in the system which help with better design and scalability of L2 Arbiter. Furthermore, the coherency protocol in the caches is required for the multi-core system to behave correctly in the presence of an OS and generic user applications.

The configurability of PolyBlaze framework allows us to have asymmetric systems. The asymmetry in the system, when exploited properly, shortens the run-time of applications by cutting on the number of read and write misses in L1 caches. Measuring different system metrics by hooking up the ABACUS hardware profiler to

different caches and Arbiters, gives better insight into application behavior and how OSs can allocate system resources for each application in order to have better over-all performance.

6.2 Future Work

As part of extending the PolyBlaze framework, there is a lot of opportunities for different asymmetric features in the system. Providing different prefetchers with different algorithms is an example of these features. Furthermore, separate work is being done on L2 cache design for PolyBlaze. Additionally, custom hardware accelerators can be connected to PolyBlaze processors and also be part of the consistent memory. Finally, multi-pumping techniques can improve the processor performance of the PolyBlaze processors.

Bibliography

[1] Xilinx Inc., *MicroBlaze Processor Reference Guide.* [Online]. Available: {http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/mb_ref_guide.pdf}

[2] E. Matthews, L. Shannon, and A. Fedorova, "Polyblaze: From one to many bringing the microblaze into the multicore era with linux smp support," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 224–230.

[3] "Standard Performance Evaluation Corporation, CPU2006 Benchmark Suite." [Online]. Available: {http://www.spec.org/cpu2006}

[4] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *Solid-State Circuits Society Newsletter, IEEE*, vol. 11, no. 5, pp. 33–35, Sept 2006.

[5] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[6] M. E. J. K. W. O. B. Ogden, *Introduction to the New Mainframe: z/OS Basics.* IBM Redbooks, 2012.

[7] I. Englander, *The Architecture of Computer Hardware and System Software: An Information Technology Approach.* Wiley, 2009.

[8] P. Stenström, T. Joe, and A. Gupta, "Comparative performance evaluation of cache-coherent numa and coma architectures," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 80–91. [Online]. Available: http://doi.acm.org/10.1145/139669.139705

[9] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for numa-aware contention management on multicore systems," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002182

[10] H. E.-R. M. Abd-El-Barr, *Fundamentals of Computer Organization and Architecture*. Wiley-Interscience, 2005.

[11] J. Handy, *The cache memory book*. San Diego: Academic Press, 1998.

[12] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," *Micro, IEEE*, vol. 25, no. 2, pp. 21 – 29, march-april 2005.

[13] *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3B: System Programming Guide, Part 2.* . [Online]. Available: {www.intel.com/Assets/PDF/manual/253669.pdf}

[14] *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. [Online]. Available: {support.amd.com/us/Processor_TechDocs/31116.pdf}

[15] P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, June 1990.

[16] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ser. ISCA '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 280–298. [Online]. Available: http://dl.acm.org.proxy.lib.sfu.ca/citation.cfm?id=52400.52432

[17] M. L. O. K. Keun Sup Shim, Myong Hyon Cho and S. Devadas, "Library cache coherence," Massachusett Institute of Technology, Tech. Rep., May 2011.

[18] J. H. David Pattersoon, *The Computer Organization and Design*. Morgan Kaufmann, 2012.

[19] (2014) The xilinx website. [Online]. Available: http://www.xilinx.com/

[20] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proc. of the 40th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2007, pp. 249–261.

[21] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of fpga-based soft processors," in *2005 Int'l Conf. on Compilers, architectures and synthesis for embedded systems*, 2005, pp. 202–212.

[22] P. H. Wang, J. D. Collins, C. T. Weaver, B. Kuttanna, S. Salamian, G. N. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang, "Intel® atom processor core made fpga-synthesizable," in *The ACM/SIGDA Int'l Symp. on FPGAs*, 2009, pp. 209–218.

[23] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel nehalem processor core made fpga synthesizable," in *The ACM/SIGDA Int'l Symp. on FPGAs*, 2010, pp. 3–12.

[24] Xilinx Inc., *PicoBlaze Processor Reference Guide.* [Online]. Available: {www.xilinx.com/support/documentation/ip_documentation/ug129.pdf}

[25] Altera Inc., *(2011, May) The NIOS Soft CPU Family.* [Online]. Available: {http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf}

[26] "RAMP - Research Accelerator for Multiple Processors." [Online]. Available: {ramp.eecs.berkeley.edu/}

[27] Daniel Burke, John Wawrzynek, Krste Asanović, Alex Krasnov, Andrew Schultz, Greg Gibeling, Pierre-Yves Droz, "Ramp blue: Implementation of a multicore 1000 processor fpga system," in *Reconfigurable Systems Summer Institute*, Urbana, IL, 2008.

[28] Chuck Thacker, MSR Silicon Valley, "Beehive: A manycore computer for FPGAs (v6)." [Online]. Available: {http://research.microsoft.com/en-us/um/people/birrell/beehive/BeehiveV6.pdf}

[29] J. Agron and D. Andrews, "Building heterogeneous reconfigurable systems with a hardware microkernel," in *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '09. New York, NY, USA: ACM, 2009, pp. 393–402. [Online]. Available: http://doi.acm.org.proxy.lib.sfu.ca/10.1145/1629435.1629489

[30] P. Huerta, J. Castillo, C. Sanchez, and J. Martinez, "Operating system for symmetric multiprocessors on fpga," in *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, Dec 2008, pp. 157–162.

[31] S. Asaad, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, T. Takken, and J. Tierno, "A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 153–162. [Online]. Available: http://doi.acm.org.proxy.lib.sfu.ca/10.1145/2145694.2145720

[32] "OpenSPARC FPGA | Field-Programmable Gate Array | OpenSPARC." [Online]. Available: {www.opensparc.net/fpga/index.html}

[33] *GRLIB IP Core User's Manual*. [Online]. Available: {www.gaisler.com/products/grlib/grip.pdf}

[34] "About — PetaLogix." [Online]. Available: {http://www.petalogix.com/about}

[35] E. Matthews, "A hardware platform for profiling system level interactions in multiprocessor systems," Master's thesis, School of Engineering Science, Simon Fraser University, 2012.

[36] M. V. P. Marwedel, *Advanced Memory Optomization Techniques for Low-Power Embedded Processors*. Springer, 2007.

[37] F. Raam, R. Agarwal, K. Malik, H. Landman, H. Tago, T. Teruyama, T. Sakamoto, T. Yoshida, S. Yoshioka, Y. Fujimoto, T. Kobayashi, T. Hiroi, M. Oka, A. Ohba, M. Suzuoki, T. Yutaka, and Y. Yamamoto, "A high bandwidth superscalar microprocessor for multimedia applications," in *Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International*, Feb 1999, pp. 258–259.

[38] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang, "Fpga implementation of a configurable cache/scratchpad memory with virtualized user-level rdma capability," in *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09. International Symposium on*, July 2009, pp. 149–156.

[39] ARM LTD, *big.LITTLE Technology: The Future of Mobile*. [Online]. Available: {http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf}

[40] Xilinx Inc., *ChipScope Reference Guide*. [Online]. Available: {http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug750.pdf}

[41] ——, *LogiCORE IP Multi-Port Memory Controller (v6.06.a)*. [Online]. Available: {http://www.xilinx.com/support/documentation/ip_documentation/mpmc/v6_06_a/mpmc.pdf}