Coherent Shared Memories for FPGAs

by

David Woods

A thesis submitted in conformity with the requirements for the degree of Master of Applied Sciences
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2009 by David Woods
Abstract

Coherent Shared Memories for FPGAs

David Woods
Master of Applied Sciences
Graduate Department of Electrical and Computer Engineering
University of Toronto
2009

To build a shared-memory programming model for FPGAs, a fast and highly parallel method of accessing the shared-memory is required. This thesis presents a first look at how to implement a coherent caching system in an FPGA. The coherent caching system consists of multiple distributed caches that implement the write-once coherence protocol, allowing efficient access to system memory while simplifying the user programming model. Several test applications are used to verify functionality, and assess performance of the current system. Results show that with a processor-based system, some applications could benefit from improvements to the coherence system, but for many applications, the current system is sufficient. However, the current coherent caching system is not sufficient for most hardware core based systems, because the faster memory accesses quickly saturate shared system resources. As well, the performance of distributed-memory systems currently surpasses that of the coherent caching system. Performance results are promising, and given the potential for improvements, future work on this system is warranted.
Dedication

I dedicate this to my parents for their support throughout my education. And most of all, I’d like to dedicate this to Laura for her help, understanding, and patience throughout my Masters Degree.
Acknowledgements

First, I would like to thank my supervisor Professor Paul Chow for leadership and guidance over the past years. Your help has been invaluable. I also would like to thank, Manuel and Arun for their expert advice. Finally, I'd like to thank all my other group members: Danny, Keith, Dan, Vince, Kam, Alex, Daniel, and Emanuel.
4 Test Applications
  4.1 Mandelbrot Area Estimation ........................................ 35
  4.2 Jacobi Heat Transfer .................................................. 37
  4.3 Matrix Multiplication .................................................. 38
  4.4 LU Reduction ............................................................ 40

5 Test Hardware and Platforms ............................................. 45
  5.1 Test Hardware ............................................................ 45
  5.2 Multiple Processors on a Single Shared Bus ....................... 46
  5.3 Multiple Processors with Coherent Memories ....................... 46
  5.4 Hardware Directly Connected to Memory ............................ 47
  5.5 Multiple Hardware with Coherent Memories ....................... 49

6 Results ................................................................. 51
  6.1 Processors on a Single Shared Bus ................................ 52
  6.2 Cache Line Size and Cache Size Variation ......................... 54
  6.3 Processors with Coherent Memories ................................ 57
  6.4 Hardware with Coherent Memories ................................ 59
  6.5 Comparison with Analytic Distributed Memory Model ............ 60
    6.5.1 Jacobi Analytic Model .......................................... 62
    6.5.2 Matrix Multiplication Analytic Model ......................... 64
    6.5.3 Analysis of Analytic Models ................................... 67

7 Conclusion .............................................................. 70
  7.1 Summary ............................................................... 70
  7.2 Conclusion ............................................................ 71
  7.3 Future Work .......................................................... 72
    7.3.1 Improvements to Performance .................................. 72
    7.3.2 Improvements to Functionality ................................. 74
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparison of memory access times</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Description of layouts tested for Jacobi application.</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Description of layouts tested for Matrix Multiplication application.</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Description of layouts tested for LU Reduction application.</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparison between execution time by a single processor and a single hardware core.</td>
<td>59</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Example of a Distributed Memory System .................................. 6
2.2 Example of a shared-memory System ................................. 8
2.3 Example of the Coherence Problem ................................. 9
2.4 Write-Once State Diagram ........................................ 11
2.5 Conceptualization of Sequential Consistency ................. 13
3.1 Overview of the Designed System ..................................... 20
3.2 User Input and Output of the Local Cache .................... 22
3.3 Read and Write User Timing of the Local Cache ............. 23
3.4 Diagram of the internal structure of the Local Cache ........ 24
3.5 Example of a direct-mapped cache read .......................... 26
3.6 Diagram of the Central Hub ........................................ 29
4.1 Diagram of potential data layouts ................................. 36
4.2 Data Layout for Jacobi Application .............................. 37
4.3 Data Layout for Matrix Multiplication Application ........ 39
4.4 Main Loop of Matrix Multiplication Application .......... 41
4.5 Data Layout for LU Reduction Application ................ 42
4.6 Main Loop of LU Reduction Application ..................... 42
5.1 Diagram of multiple processors on a single shared bus platform .......................... 46
5.2 Diagram of multiple processors with coherent memory platform .......................... 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Diagram of single hardware core directly connected to memory</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Diagram of multiple hardware with coherent memory platform</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>Speedup of Processors on a Single Shared PLB bus</td>
<td>53</td>
</tr>
<tr>
<td>6.2</td>
<td>Speedup of different CLS</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>Speedup of different Cache Sizes</td>
<td>56</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparison of Processors with Coherent Caches VS. Processor on a shared PLB bus</td>
<td>58</td>
</tr>
<tr>
<td>6.5</td>
<td>Speedup of hardware cores connected to coherent caches</td>
<td>61</td>
</tr>
<tr>
<td>6.6</td>
<td>Example of Distributed Jacobi Algorithm</td>
<td>63</td>
</tr>
<tr>
<td>6.7</td>
<td>Example of Distributed Matrix Multiplication Algorithm</td>
<td>66</td>
</tr>
<tr>
<td>6.8</td>
<td>Speedup of modeled distributed-memory system</td>
<td>69</td>
</tr>
</tbody>
</table>
List of Acronyms

µB  Xilinx MicroBlaze Soft Processor
ACP  Xilinx Advanced Computing Platform
BRAM  Xilinx Block Ram Memory
CAM  Content Addressable Memory
CLS  Cache Line Size
CPU  Central Processing Unit
DDR  Double Data Rate Memory
FIFO  First In First Out Memory
FPGA  Field Programable Gate Array
FSB  Front Side Bus
FSL  Fast Simplex Link
FSM  Finite State Machine
HW  Hardware
IF  Interface
LMB  Local Memory Bus
LUT  Look Up Table

MEM  Memory

MESI  Coherence protocol who’s states are Modified Exclusive Shared Invalid

MIMD  Multiple Input Multiple Data

MOESI  Coherence protocol who’s states are Modified Owned Exclusive Shared Invalid

MPI  Message Passing Interface

NOC  Network On Chip

NUMA  Non Uniform Memory Access

P2P  Point to Point

PLB  Processor Local Bus

Proc  Processor

SC  Sequential Consistency

SW  Software

UART  Universal Asynchronous Receiver/Transmitter
Chapter 1

Introduction

1.1 Motivation

Traditionally, because of the distributed nature of memory in an FPGA, and because of the lower level of abstraction, high performance computing application in FPGAs have typically used a distributed-memory model [1] [2] [3] [4]. Unfortunately, these types of machines are generally unpopular amongst software designers because all shared data must be explicitly exchanged between processing units. By contrast shared-memory machines are much easier to program. Like a distributed-memory system, most shared-memory systems have some memory connected to each processing unit in the form of a cache, however this cache is small with the majority of data stored in a large shared memory space. Most importantly though, all data transfers between caches, as well as to and from memory, are all automatic with no explicit programming required.

In recent years heterogenous processing systems where modern CPUs are connected to FPGAs via high speed busses have become more common. An example is, the Xilinx Accelerated Computing Platform (ACP) that allows multiple Virtex 5 FPGAs to be connected to Multiple Intel CPU’s via the Intel Front Side Bus (FSB) [1]. Due to their tight coupling of FPGAs with main system memory, and processors, these systems have
introduced the possibility of expanding the thread-based programming model to incorporate FPGAs. Several groups have already begun work on computing systems known as hardware thread based systems. In a hardware thread based system, software running on a standard commercial CPU can spawn a thread that will be executed in hardware on an FPGA, and access data from the shared main system memory [5][6]. In the process of creating this thesis, we are also exploring the concept of embedded hardware thread based systems. In an embedded hardware thread based system, threads are spawned and executed by hardware cores in an FPGA, without the need of an off-chip processor.

To enable an effective thread-based programming model, access to off chip memory is crucial. Often these memories suffer from the problems of high latency, limited bandwidth, and complicated interfaces. To solve these problems a caching mechanism with a simple interface is required. A single large cache for all processing units within the design would be easiest to implement, however, contention for this cache would quickly become an issue. As such, a design where multiple coherent caches distributed throughout the design is required. This thesis presents a first look at how to implement a coherent caching system in an FPGA.

1.2 Contributions

This thesis proposes a coherent caching system for use in a shared-memory multiprocessing system. The contributions are as follows:

1. The first implementation of a coherent caching system for FPGA-based hardware and software threads;

2. A set of custom designed software and hardware test applications, which were used to evaluate the feasibility and measure the performance of this approach;

3. A performance demonstration and comparison of the designed system with other
methods of parallelization in an FPGA.

1.3 Overview

The remainder of the thesis is organized as follows. Chapter 2 provides a background on shared-memory and distributed-memory machines, as well as caching, and the problems of cache coherence and consistency. Chapter 3 details the design and organization of the coherent caching system. Chapter 4 describes the applications used to test the system, and how they were generated in both hardware and software. Chapter 5 describes the FPGA hardware, and designs used to perform the various tests required. Chapter 6 examines the results of all tests run on the system comparing different methods of creating a shared-memory system, different hardware and software implementations, as well as comparison with distributed-memory systems. Finally, Chapter 7 provides a summary, conclusion, and description of future work to improve the system.
Chapter 2

Background

This chapter provides an overview of some background knowledge that will be important in understanding the remainder of this thesis. Section 2.1 compares shared-memory and distributed-machines, and describes the composition of each. Section 2.2 discusses cache coherence and snooping caching protocols. Section 2.3 discusses cache consistency. Finally, Section 2.4 examines some previous work related to this thesis.

2.1 Distributed-Memory and Shared-Memory Machines

In terms of high performance computing machines, the most common type of system today is the MIMD or Multiple Instruction Multiple Data machine. MIMD machines can be further sub-divided based on their memory architecture into two groups: Shared-Memory and Distributed-Memory systems.

In a distributed-memory machine, the system is divided into multiple sub-processing nodes, each of which have their own memory space that is separate and not directly accessible by any other processing node. The processing nodes are then connected by some form of interconnect network to communicate data with one another. In the sys-
tem, there is one central node that controls the system and the overall processing. To begin computation, the central node divides the initialization data, and explicitly distributes the data to all the processing nodes in the system so that each contains a small portion of the overall data in the system. The processing nodes can then perform the required computations. During the computation, if there are any data dependencies between data on different processing nodes, the data must be explicitly transferred between processing nodes via an interconnect network. Finally at the end, all processing nodes must send their results back to the central node. This process can be seen in Figure 2.1. The advantage of a distributed-memory system is that depending on the design of the interconnect, there is limited, or no dependency on any shared central resources. As such, a distributed-memory system is very scalable. However, the disadvantages of a distributed-memory system are that memory access to non-local data incurs a large latency, so some algorithms that frequently access data stored remotely will perform poorly. Another major disadvantage of a distributed-memory system is that all shared data must be explicitly transferred to other processing nodes. This makes programming much more difficult for the programmer.

The other type of MIMD system is the shared-memory machine. In these systems, there are several processing nodes. However, none of these nodes have a dedicated memory of their own. Instead all processing nodes and system memory are interconnected in a single large memory address space. An example of a shared memory system can be seen in Figure 2.2. In it, processing begins by a central node loading any initialization data into the shared system memory. Processing nodes can then retrieve data from memory and begin processing. During computation, if there are any data dependencies between processing nodes, data is automatically shared with other nodes as well as system memory. When processing completes, the central node can access the completed data from the system memory. The advantage of shared-memory systems is that they are much easier to program. The programmer does not need to explicitly transfer data within
Chapter 2. Background

(a) Distributed Memory System

(b) Data Distribution

(c) Data Sharing

(d) Data Reduction

Figure 2.1: Example of a Distributed Memory System. $\mu$B (MicroBlaze) signifies that this node is a processor, and $\mu$B/HW signifies that these nodes can either be a processor or a custom hardware core. In (a), we see the distributed-memory system. In this system there is one central control node, and three processing nodes. Each node has their own memory directly connected to them. When processing begins, initialization data is sent to each node, which can be seen in (b). While computing, shared data is explicitly shared by the processing cores as is seen in (c). Finally, when complete, results are returned to the central processor seen in (d)
the system, instead it is automatically shared by the interconnect network to the main memory. The disadvantage however of a shared-memory system is that because they rely on a shared centralized resources, contention can become an issue and scalability is not as good as that of a distributed-memory system.

2.2 Coherence

When multiple caches are used to access a single shared-memory, an important problem arises: coherence. In a memory system, users generally expect that any read of a particular value should return the most recently written value, regardless of where the write took place. In a system with multiple caches, this does not always happen. Take for instance the example in Figure 2.3. In Figure (a) we see a system with two processors (Proc 1 and Proc 2), each with caches on a shared bus accessing a single memory. In Figure (b) processor 1 reads from memory location A and stores the data in its cache. Afterwards, in Figure (c) processor 2 reads the same memory location A, and the data is also stored in its cache. Next, in Figure (d) processor 1 writes a new value to A that is updated in its own cache and main memory; however, because coherence is not being taken into account, the data in processor 2’s cache is still old. In Figure (e) when processor 2 tries to read memory location A the processor gets the wrong value. This is the essence of the coherence problem. [7]

To solve this problem, the various caches must somehow communicate with other caches to ensure that all data within the cache is correct. This communication between caches is called a coherence protocol. In Figures (f) and (g) we see how a coherence protocol could have prevented the incorrect read. In Figure (f), when processor 1 writes to memory, the coherence protocol also invalidates the cache line with memory location A in processor 2’s cache. Then in Figure 2.3g, when processor 2 tries to read memory location A, the processor must go to main memory and retrieve the correct value. [7]
Figure 2.2: Example of a shared-memory System. \( \mu B \) (MicroBlaze) signifies that this node is a processor, and \( \mu B/HW \) signifies that these nodes can either be a processor or a custom hardware core. The shared-memory system seen in (a) has four main components: a central processor for controlling the system, multiple processing nodes, a large memory space (labeled SHARED MEM), and a shared network for accessing memory. When processing begins, the central processor loads initialization data into the large shared-memory as can be seen in Figure (b). Processing nodes retrieve data from memory as seen in Figure (c). Any shared data is automatically shared with other processing nodes and main memory as is seen in Figure (d). Finally, when processing is complete the main processor retrieves data from memory as in Figure (e).
Figure 2.3: Example of the Coherence Problem
There are many different cache coherence protocols, each with their own benefits and draw backs. The two main categories of coherence protocols are: directory-based coherence protocols, and snooping coherence protocols. In a directory-based protocol, all coherence data is stored in a central or distributed directory. When a cache needs to access data it must make a request to the directory that will determine any coherence issues and perform the appropriate actions to ensure coherence of all caches. The advantage of a directory-based coherence protocol is that it is very scalable. However, the disadvantage is that memory access have a higher latency and in general distributed memory systems are more complex to implement. Traditionally, in snooping-based protocols, all caches sit on a single shared bus that they monitor, or snoop, for any memory requests by other caches. If they notice any actions on the bus that may cause a coherence issue, they take action within their own cache memory or on the bus by interrupting actions taken by other caches. Currently, a single shared bus is often not used, with other methods of broadcasting memory requests being more common. The advantages of a snooping coherence protocol is that it is much easier to implement, however, the disadvantage is that they are not as scalable as a directory-based caching system[7].

The first coherence protocol published is Write-Once, described by Goodman in 1983 [8]. This simple protocol has four states. The Invalid state means that the data in the cache line is no longer valid. The Valid state means the data in the cache line is valid and consistent with main memory. The Reserved state implies that the data is still consistent with main memory, however, no other cache has this data. Lastly the Dirty state means that the data in the cache line has been modified and is no longer consistent with main memory. The state transitions for various cache operations initiated by a processor request or by a snooped bus request can be seen in Figure 2.4 [9]. Other more modern and efficient protocols also exist; for instance, MESI and MOESI. These protocols are what are commonly used by processors today [10][11], however, their states and transitions are more complicated, and as such require more, and more complicated
Chapter 2. Background

11

(a) Processor Initiated

(b) Bus Initiated

Figure 2.4: Write-Once State Diagram. In (a) the name before the / represents a request made to the cache by the processor, and the name after is the resulting action taken on the bus. Possible processor requests are Rd for read, Wr for Write, and Ev for an eviction caused by a cache miss. Possible bus actions are Rd for read from memory, Wr for write to memory, and Flush that writes back the dirty cache line to memory. In (b) the name before the / represents an event snooped from the bus, and the name after is the resulting action taken on the bus. Categories of types of bus requests are the same as in (a). [9]

hardware. In this work it was decided to start with the simpler write-once protocol.

2.3 Consistency

Memory consistency has to do with the order in which data reads and writes are seen by caches in a system. Coherence defines that data written to a shared memory location will eventually be visible by all other caches, but it makes no guarantee as to when other caches will be able to see it; this is defined by consistency. The most simple consistency model to understand is Sequential Consistency (SC)[7]. Officially, a system
is defined as being sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and that the operations of each individual processor occurs in this sequence in the order specified by the program. This is a very complex definition, so a much easier way to think of sequential consistency can be seen in Figure 2.5. In this example the system acts as if all processors share a single connection to main memory. Each processor is allowed to access memory one at a time and all outstanding memory transactions are allowed to complete before access is given to another processor. This is generally how a programmer would expect their programs to run [7]. The set of sufficient conditions for SC are:

- All memory issues must occur in program order;
- After a write is issued, the issuing process must wait until the write has completed and is visible by all caches;
- Read must wait for the write who’s data is being read to be visible by all caches.

Due to these restrictions it is very difficult to take advantage of all possible optimizations. For example, out of order processor execution is extremely difficult to implement with SC. To improve performance other forms of consistency are used. These consistency models relax ordering by varying amounts, for example, processor consistency where the write after read consistency is not guaranteed, or weak consistency, where no ordering is guaranteed except for that dealing with synchronization. No matter the consistency model used, however particularly with a weakened consistency model, synchronization must be used to ensure proper operation of a program.

2.4 Related Work

Until now because of the distributed nature of memory on an FPGA, most computation on FPGAs has been done using a distributed-memory model. Some recent examples are
Figure 2.5: Conceptualization of Sequential Consistency. To understand sequential consistency it can be easier to think of each processing core having access to main memory one at a time, with some mechanism to fairly determine who gets access to memory at any one time. When one processor gets access, its memory transaction is allowed to fully complete before any other processor gets access to memory.[7]
shown here.

In Patel [3] and Saldana [12] the TMD-MPI system is presented. TMD-MPI is a distributed-memory system designed to run entirely within an FPGA and allows hard processors, soft processors, and hardware cores to communicate using a subset of the MPI standard. Being that MPI is a common standard for distributed-memory systems, it facilitates easier transition for existing parallel algorithms from a CPU to an FPGA environment. All communication within and between FPGAs is done using a customizable network on chip (NOC).

In Baxter [4] the Maxwell system is presented. This is a 64 node CPU and FPGA cluster. Each CPU is used to send control information to the FPGAs, and are connected to one another via an ethernet network. The FPGAs are connected via RocketIO links that allow for high-speed nearest neighbor connections. For communication with other FPGAs that are not neighbors, communication must pass across the ethernet network slowing this communication significantly. Programming for the Maxwell is done using a set of libraries called the Parallel Toolkit[13].

Finally in Williams [14] another system that uses MPI as its communication standard is presented. The network for communication is comprised of direct FSL (Fast Simplex Link)[15] connections between MicroBlaze soft processors[16].

Compared to distributed-memory systems in an FPGA, much less research has looked at shared-memory systems in an FPGA. One method is to use a single shared cache that all processors or hardware cores use to access main memory. Some examples of systems like this include the following: in Budiu a pipelined network is used to access a single cache[17]; in Garcia a reconfigurable fabric communicates with a hard processor by a shared L1 and/or L2 cache [18]; and in Mishra a system called Tartan uses a shared L1 cache to allow a processor and reconfigurable fabric shared access to memory[19]. As was noted by Budiu , one of the major limitations of these systems is contention for the shared cache, which they tried to reduce by using compilation techniques[20].
Another approach to caching is to distribute the cache throughout the design itself. This reduces contention for the single shared resource. In Andrews they describe the caching system they use for their shared-memory system [5]. This system more resembles a NUMA (Non Uniform Memory Access) system in that data is distributed throughout the FPGA with a portion of it associated with each processing unit. The system then allows fast access to data in memory directly associated with the processing unit, however, much slower access to data that is associated with other processors. This implementation is still more work than a true shared-memory system as data must be appropriately partitioned or performance will be very poor, however, it is less work than a distributed-memory system where data must also be explicitly shared amongst processing units.

In Putnam, the CHiMPS system is presented [6] [1]. In this system a “many-cache memory model” is used, where a single cache is used for each shared variable. These caches are distributed throughout the system and are all connected to a single shared-memory. With many caches such as this, the problem of coherence arises. However rather than solving the problem through a cache coherence scheme, the problem is avoided all together by allowing only a single copy of a variable to be cached. As well, the caches used are very elaborate, taking advantage of many different advanced techniques such as banking that allows multiple read ports.

In terms of coherent caching within an FPGA, even less work has been done. One example is in terms of caching for soft processors. In Huang, a system for coherent caching in an Altera Nios Processor and Avalon bus architecture is presented [21]. This system uses the caches built into the Nios Processor combined with an external controller which sends interrupts to the processor when an invalidation is needed. This presents two main problems. First, as noted by the authors, the interrupt service routine which process the interrupts from the external controller is inefficient and quickly becomes a bottleneck in the system. Second, because this system uses caches built into the Nios processor, it is not possible to implement this type of system with a hardware core which
does not have its own cache.
Chapter 3

Hardware Design

This chapter describes the design of the hardware for a coherent caching system on FPGAs. Section 3.1 describes the design goals considered when designing the system. Section 3.2 provides an overview of the system as a whole. Section 3.3 describes the specific use of the Write Once protocol over FSL links. Section 3.4 describes the design of the Local Cache module. Section 3.5 describes the design of the Central Hub module. Section 3.6 examines the resulting access times of various different cache operations. Finally, Section 3.7 examines the resource utilization both in terms of hardware resources, as well as timing.

3.1 Design Goals

As stated in Section 1.1, the overall goal of the project is to build a shared-memory machine. However, rather than spending significant time analyzing the problem, it was decided to create a working system to explore the problem in detail. Often in simulation and modeling, critical issues might be overlooked that a real hardware implementation cannot. In FPGAs, if designed properly, modification and improvements to a design can be made during development to improve performance and fix issues that may otherwise not have been noticed until a design has been fully implemented.
As described in Section 1.1 the memory access system will be composed of many distributed caches, that will access a single shared memory, and remain coherent with one another. To supply data to both soft processors such as a Xilinx MicroBlaze (µB) or a user designed hardware core, the cache should be fast. This means reads and writes should occur in a single clock cycle (single cycle) in the case of a cache hit, much like most L1 caches for processors. To simplify hardware accesses, the interface should be simple, preferably with a commonly used existing interface. As such it was decided to use a modified Xilinx Block Ram (BRAM) interface that would be the same as a standard BRAM interface, however, with the addition of a BUSY signal. This can be seen in figure 3.2. Hardware systems generally have many soft processors or hardware cores, and to support this the system should be as scalable as possible. Next, the physical resource utilization should be small, to leave as much room for the implemented design as possible. The system should run at a high enough clock rate to not inhibit the hardware cores using the system. As well the system should be able to support multiple clock domains, allowing the various distributed caches to operate at different clock frequencies.

Originally MESI selected as the cache coherence protocol for this system. This is because it is used by the Intel Front Side Bus (FSB)[10]. It was hoped this would allow the designed system to be more compatible if integrated with an Intel CPU based system. However, MESI is a more complex protocol, and because the details of the FSB interface were not available it was decided to use a simpler coherence protocol for the first implementation. Instead write-once was selected. Write-once is similar to the MESI protocol in that it has for states with similar meaning: Dirty is equivalent to Modified, Reserved is equivalent to Exclusive, Valid is equivalent to Shared, and Invalid is the same in both protocols. The difference however, is in the types of transitions possible. For instance, in write-once when in the Invalid state and a read is performed, then the only possible transition is to Valid. However, with MESI when in the Invalid state and a read is performed, depending on the response from other caches the next state could be either
As for a consistency model, it was decided to use SC. SC was selected for two reasons. First, SC is the most intuitive consistency model, that follows with the design goal of keeping the caches easy for a programmer to use. Second, a relaxed consistency model requires synchronization to ensure that shared data is correct. As of yet there is no synchronization method built into the coherence system, so a more relaxed consistency model is not currently possible.

### 3.2 System Overview

As can be seen in Figure 3.1 the coherent caching system designed has three main components: Local Caches, the Central Hub, and a series of FSL connections labeled Coherence Connect.

The Local Cache is the module where the actual cache data is stored. The Local Cache makes requests for data, and receives snooped requests from the Central Hub via the Coherence Connect connections.

The Central Hub receives data requests from the Local Caches, forwards these requests as snoops to other caches, and retrieves the requested data from memory via the Processor Local Bus (PLB) master port. There is also a PLB slave port that is bridged to the PLB master. As well, snoop commands are sent to the various connected caches ensuring they are coherent with any PLB slave requests.

Coherence Connect is the connection between the Local Caches and the Central Hub. It consists of four 32-bit Fast Simplex Links (FSLs) grouped together. An FSL is a point to point FIFO based communication channel, that allows for high-speed, low-latency unidirectional communication. Pairs of FSL can be used to allow bidirectional communication. One bidirectional pair of FSLs, called E (expected), is used by the Local Cache to send and receive requests from the Central Hub. The other bidirectional pair,
Figure 3.1: Overview of the Designed System

called U (unexpected), is used by the Local Cache to receive snooped requests from the Central Hub, and to send their responses back. FSLs are used because they can be run asynchronously, allowing the Local Cache and Central Hub to be in different clock domains.

3.3 Write-Once over FSL

The state diagram for the write-once protocol can be seen in Figure 2.4. The write-once protocol is designed to be used in a bus architecture, however, because a star architecture using FSLs as connections was used, a custom protocol had to be used. The key factor that allows a 32-bit FSL to be used to send the required coherence data is that the addresses used are all byte addressed, and all transactions are done in terms of words. This leaves the bottom two bits of any address unused. The caching system uses these bits to encode the request type. At the moment, two bits are sufficient to encode all request types, however in the future if more request types are required, the FSL control bit can also be used to allow up to eight different request types. The current three request
types are: Update, Read, and Write. An update request occurs when a modified cache line is evicted from a cache. The address and data of the cache line are sent back to the Central Hub, that forwards them to main memory. A read occurs when a cache request misses and requires data from main memory. In this situation a read request with the address of the required cache line is sent to the Central Hub. The Central Hub checks for coherence with other caches, retrieves the data from main memory, and forwards the data back to the requesting cache via an FSL. A write occurs when the cache is written to, and the cache misses or hits with the cache in a valid state. In this case, a write request followed by the data written is sent to the Central Hub. The Central Hub writes the data to memory, and invalidates the cache line in other caches in the system. Originally no acknowledgement was required for a write request, however, this violates the second requirement for sequential consistency as stated in Section 2.3. As such when the write has processed correctly, an acknowledgement is sent back to the requesting cache. If there is a conflict, in that multiple caches request a write to the same cache line at the same time, one is selected by an arbiter and processed correctly, the others are sent a fail acknowledgement forcing them to re-request the write.

3.4 Local Cache

The Local Cache is the module where the Cache Memory is stored. The Local Cache also maintains the state of all caches lines, and uses snooped commands from the Central Hub to ensure the cache is coherent with all other caches in the system. The cache is parameterized allowing for variable cache line size from two words to the total number of words in the cache, and cache sizes from 128 words to a maximum of however many BRAMs are free in the device being used. The top-level user interface can be seen in Figure 3.2. The inputs to the design are the same, and have the same timing as a standard BRAM interface. The output is also the same as a BRAM except for the BUSY signal.
Figure 3.2: User Inputs and Outputs of the Local Cache. On the left are the inputs to the cache, and on the right side are the outputs.

The BUSY signal indicates that the cache is processing a cache miss or a snooped request and that no data will be written, nor is the data at the output valid.

An example of the read and write timing can be seen in Figure 3.3. In Figure 3.3(a), first a read to address A occurs. With a standard BRAM the result would be expected on the next clock cycles after the address and enable is applied. However, in this case a cache miss occurs, and the BUSY signal goes high. Initially the data at the output (Data_OUT) is incorrect as is indicated by the grey area. When the data is valid and the Local Cache is finished processing, the BUSY signal will go low and the data can be accessed on the following rising clock edge. Afterwards, three reads to addresses A, B, and C occur. These reads hit, and each access occurs in a single cycle as with a standard BRAM memory. In Figure 3.3(b), first a write of data D(A) occurs to address A that misses. Normally, when data is written to a BRAM, the BRAM is ready in the next cycle for another operation. In this case, however, the BUSY signal goes high for several cycles indicating that the Local Cache is busy processing the write miss, and is not ready. After some time the BUSY signal goes low again signaling that the Local Cache is ready for another operation. Several cycles later three more writes occur to memory locations A, B, and C. In each of these cases, the cache hits, the BUSY signal never goes high, and the writes can occur in a single cycle one after the other.

The internals of the Local Cache can be seen in Figure 3.4. The first main component within the cache is the Cache Memory. This is the actual location where data is stored.
Figure 3.3: Read and Write User Timing of the Local Cache. In (a) a read misses causing the busy signal to go high, afterwards, three reads hit. In (b) a write misses causing the busy signal to go high, afterwards, three writes hit.
All BRAM inputs and outputs are attached to one port of the dual-ported Cache Memory. The other port is used to load data into the Cache Memory without affecting the timing of the BRAM interface. The second component is the Cache Table. This is a small dual port memory that stores the state of all cache lines in the cache. One port of the Cache Table is read only and reads the tag and state of the cache line of all requests as soon as they come into the cache. The other port is a read and write port and is used to update the Cache Table, as well as to check the cache line state for snooped requests. The final component of the cache is the FSM. The FSM is the main logic in the design and is responsible for: checking for a cache hit or miss, communicating with the Central Hub and main memory via the four FSLs, loading the appropriate data into the Cache Memory and Cache Table, handling all snoops from the Central Hub, and asserts the BUSY signal when the Cache Memory is not ready.

3.4.1 Caching Implementation

The caching scheme selected to be used was direct-mapped caching. More complex caching schemes with higher levels of associativity were considered, however, generally they require Content Addressable Memories (CAMs) and/or multiple cycles to process each cache transaction. A CAM memory is a memory that when given a data element
will give the address of the data element within the memory. Unfortunately, CAMs are inefficient to build in an FPGA, as they use large amounts of resources. As well, a system that takes multiple cycles for each cache transaction would not be appropriate, as a design goal was to have single cycle accesses on read and write hits. As such a direct-mapped scheme was selected. The advantages of a direct-mapped cache are that they are relatively simple to implement, and have fast access times requiring only a single cycle to process cache requests. The disadvantages of using direct-mapped caching is that the miss rate will be higher than other caching schemes.

In this implementation of a direct-mapped cache there are two main structures: the Cache Memory, and Cache Table. The Cache Memory, is a relatively large memory that stores the actual data in the cache. The Cache Table, is a much smaller memory that stores a tag value for each cache line. This tag is taken from the upper most bits of the address of the data stored in the Cache Memory and is used to identify which portion of memory the cache line refers to. The reason for the separation of these two structures is that it allows the smaller cache table to be better mapped to FPGA logic, saving memory usage. As well the separation of the two structures makes the overall design of the Local Cache much easier.

An example of a cache read can be seen in Figure 3.5. When a data read is received, the address is divided into three different portions. One goes to the Cache Memory, another goes to the Cache Table, and another is registered. In the next clock cycle the data retrieved from the memories is ready. The tag from the Cache Table is compared against the top bits of the address that were registered in the previous cycle. If they match, then there was a hit, and the data from the Cache Memory can be used. If they don’t match, then the cache missed and the cache must stall to retrieve data from main memory, and update the Cache Table.

With a direct-mapped cache, it is easy to perform a read on every clock cycle, however, to perform a write on every clock cycle more complex processing is needed. This is because
Figure 3.5: Example of a direct-mapped cache read. In this example we are using a 32 bit byte addressed address space, a cache size of 2048 words, and cache line size of 16 words.
a write to memory takes one cycle and checking the Cache Table also takes one cycle. If these tasks happen in parallel then data could incorrectly be written to Cache Memory and corrupt existing data in Cache Memory. If these tasks happen sequentially, then more than one cycle is required to perform a write. One option is to pipeline the input and allow the reading of the Cache Table and writing to Cache Memory to happen sequentially. Unfortunately this poses many problems especially when checking for conflicts where a line is written in one cycle and read from in the following cycle.

Another alternative is speculative writing. Speculative writing takes advantage of the fact that when data is written to a Xilinx BRAM, on the following cycle the data at the output of the BRAM is the previous contents of the memory location written. This is called read first mode [22]. When a write to the cache occurs the new data is written to the Cache Memory automatically without any knowledge as to whether the correct cache line is in the cache. In the next cycle, the contents of the Cache Table can be examined. If there was a hit, then the cache can continue without any changes. However, if there was a miss, then the old data from the Cache Memory is still accessible due to read first mode. The old data is registered, and later restored to its original location in the Cache Memory. This fixes any potential corruption that may have occurred when the new data was incorrectly written to the Cache Memory. Once the appropriate cache line is loaded into the memory, the new data is re-written to memory, and the cache can proceed as normal. The advantage of this system is that there is no potential for conflicts. However, the disadvantage is that in the case of a write miss, an extra cycle might be required to restore the old data to the Cache Memory. In the current system speculative writing was used to implement single cycle reads.
3.5 Central Hub

The Central Hub is responsible for handling all requests from the Local Caches, accessing main memory, and forwarding snoop messages to other Local Caches. The majority of the central hub is a large FSM with controls all data transfers. It is parameterizable for between one and eight caches, although with further work more should be possible. It is parameterizable for different cache line sizes, however, the cache line size should be the same as that of all the Local Caches. A diagram of the Central Hub can be seen in Figure 3.6. There are three main interfaces to the Central Hub. First, are the FSL connections to the Local Caches. Each connection to a cache is composed of four FSL connections that are used to make and receive memory requests as well as snooped requests from other caches. Next, the PLB master is used to access memory whether it be a BRAM on the PLB bus, or an off chip memory controller on the PLB bus. The PLB slave interface is bridged to the PLB master, however, while being bridged, the Central Hub sends snoops to all caches to ensure all caches are coherent with these requests. Finally, the Central Hub contains a large FSM which controls all data transfers and coherence requests.

Frequently multiple caches will make requests of the Central Hub at the same time. In these situations there must be a fair mechanism to determine which request should be processed first. As well, update requests should take priority over other request types. As such, a mixed round robin, fix priority arbiter was created. The reason a round robin arbiter was used, was to ensure that deadlocks or starvation occurred. The basic design for both the round robin component of the arbiter, as well as the fixed priority arbiter was adapted from [23]. Next a method for combining the arbiters together to provide the mixed round robin priority arbitration required was adapted from [24].
3.6 Access Time

In the case of a cache hit, both reads and writes occur in a single cycle. In the case of a miss though, depending on the type of memory request and state of the cache line, the memory access time varies. As well, access time varies depending on the cache line size (CLS), and the latency of the memory and bus being accessed. In the case of a read miss the access time in cycles is given by the equation

$$\sim 18 + CLS + (CLS\text{mod}16) \times (memory\_latency)$$

(3.1)

where $\sim 18$ represents the number of cycles required by the Local Cache and Central Hub to perform all of their required operations. For the Local Cache this includes checking the Cache Table, send the request to the Central Hub, and updating the Cache Table when the read completes. For the Central Hub this includes receiving the request from the Local Cache, arbitration, sending and receiving snoop requests to other Local Caches, and returning data from memory. $CLS$ is the cache line size in words, and represents the time required to send the data. $(memory\_latency)$ represents the time required for bus arbitration and memory access. Finally, $(CLS\text{mod}16)$ represents the fact that for
Table 3.1: Comparison of memory access times for BRAM and DDR memory on a PLB bus. All times were experimentally determined and are in cycles.

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>DDR</td>
<td>19</td>
<td>8</td>
</tr>
</tbody>
</table>

Throughout this thesis BRAM memory will be compared with off chip double data rate (DDR) memory. This is because DDR memory has a much higher latency than BRAM memory, effecting the memory access time values of equations 3.1 and 3.2. This intern has an effect on the performance of the overall system. A set of experimentally determined access times can be seen in Table 3.1.
3.7 Resource Utilization

The resource utilization of the system varies depending on the various parameters for Cache size, cache line size, and number of Local Caches used. In this section the values of 2048 words in each cache, a cache line size of 16 words, and eight Local Caches will be used.

Each Local Cache has as a total Look Up Table (LUT) utilization of 613 LUTs. Considering the smallest Virtex 5 FPGA has 30,720 LUTs total, this represents only 2.0%, and 0.9% of the 69120 LUTs in the Virtex 5 LX110T FPGA that was used to test the system. As well, three BRAMs were used, two of which were used for the Cache Memory, and one of which was used for the Cache Table. The Cache Table is much smaller than the Cache Memory, and likely will not use most of the BRAM that it was assigned. If additional BRAM resources are required, the Cache Table can be implemented in distributed memory, that makes use of the FPGA’s LUTs as a small memory.

The Central Hub uses a total of 2375 LUTs. This represents 7.7% of the smallest Virtex 5 device, and 3.4% of the Virtex 5 LX110T FPGA used. This number is much larger than that of the Local Cache, however because only one is needed for eight Local Caches, it is still overall a very small portion of the entire system. No BRAMs are required for the Central Hub.

Due to limitations of supporting hardware, all future tests in this thesis were run at 100 MHz and in the Virtex 2 and Virtex 5 FPGAs used, this frequency was achieved. In the Virtex 5 FPGA, the system was also tested to 150 MHz and all components met timing. The system was also tested at speeds above 200 MHz, and though the Local Caches were able to meet this speed, the Central Hub was unable. The failing path in the Central Hub relates to a timing path connecting the output of one FSL and the input of another both of which have a large delay. As an alternative, the FSL connections between the Local Caches and the Central Hub can operate asynchronously, so it is possible to
run the Local Caches at a higher clock speed than the Central Hub.
Chapter 4

Test Applications

To test correctness, and measure the performance of the coherent caching system, a series of test applications were necessary. Several properties were looked at when selecting the applications. First, the applications should be parallelizable in a shared-memory architecture. Next, the applications need to be relatively simple, as there is no operating system and only limited libraries available. Also, because currently there is only simple barriers to act as a means of synchronization, the applications should have minimal synchronization requirements. Finally, because we are testing the memory system, the applications should have a variety of memory access patterns and frequency. As such the applications selected are:

- Mandelbrot Area Estimation, that has low memory utilizations, and is described in Section 4.1.

- Jacobi Heat Transfer Simulation, that has moderate memory utilization, and is described in Section 4.2.

- Matrix Multiplication, that has high memory utilization, and is described in Section 4.3.

- LU Reduction, that also has high memory utilization, and is described in Section
4.4.

All these applications were implemented using a 32-bit fixed-point format, with a varying number of fractional bits depending on the requirements of the application. They were first developed and tested on a single processor before being parallelized to multiple processors. Parallelization occurs by a single central processor sending an FSL message on each iteration to every other processing unit in the system. The message contains a pointer to the data, the range of the data that the processing unit should process, and any other constants required. When the processing unit is complete it sends a message back to the central processor with the results. Unlike a shared-memory system with standard CPUs, there is no operating system to spawn threads and point to memory locations, nor is there method of synchronization. As such, the FSL messages between the central processor and the processing units serve as a method of spawning threads, indicating where data is stored, and act as barrier synchronizing the system.

When successfully tested with multiple processors, a hardware core was designed for each application. The hardware core operates using an FSM that computes the data in the same order as the processor, albeit much faster.

One important factor affecting the performance of a program executed with caches is the way data is laid out in memory. Memory in computing systems is inherently linear, so when a two or higher dimensional array needs to be placed in memory, it must be laid out linearly in some logical manner. There are many different ways that data can be laid out, and depending on the memory access pattern of the application some layouts are better and some are worse. If data is laid out incorrectly, cache thrashing occurs, where data is unnecessarily evicted to main memory and soon after brought back into the cache. If the layout is just right, then a read of one variable will cause soon to be used variables to also be loaded into the cache. To determine the ideal data layout, multiple layouts were experimentally attempted. The layouts tested consist of combinations of: row major order as can be seen Figure 4.1(a), column major order as can be seen in Figure 4.1(b),
and if multiple matrices are of the same dimensions then the data elements or even entire rows/columns of different matrices can be interleaved as is seen in Figure 4.1(c). In these tests, data accessed both BRAM and the slower DDR memory. When the layout was incorrect, cache thrashing hid the difference in latency in accessing BRAM and DDR memory and the execution time was more or less the same for both memories. When the layout was correct, the difference between the BRAM and DDR memories became apparent. Once the best layout for each application was determined, it was used for all further tests in this thesis.

4.1 Mandelbrot Area Estimation

Mandelbrot area estimation is the low memory utilization application, and is used to estimate the area of the Mandelbrot set. The Mandelbrot set is the set of numbers in the complex plane that when used as the constant $c$, causes the sequence $z_{n+1} = z_n^2 + c$ to converge. This set of numbers forms a fractal, and as such the exact area is not known and can only be estimated. This application calculates the area using a Monte Carlo simulation. In this simulation the shared memory is used to distribute the set of random variables used for the Monte Carlo simulation. Each processing unit accesses the shared memory to retrieve their respective random variables; when all monte carlo pathes have been explored the result is returned to the central processor. Each processing unit use this memory for reading their own unique set of values, and as there is no sharing of data values between caches. This application was adapted from the c_mandel.c example application in the OpenMP Source Repository [25]. Since the data layout in this application is inherently linear in nature, no tests for data layout were required. The hardware core generated for this application was successfully run at 100 MHz and the resource utilization was 3423 of 69120 or 5.0% of LUTs and 4 of 64 or 6.3% of the DSP48Es modules in the FPGA used.
Figure 4.1: Diagram of potential data layouts. In (a) data is laid out in a row major order. In (b) data is laid out in a column major order. In (c) both matrices are in row major order and all data elements are interleaved with one another.
Figure 4.2: Data Layout for Jacobi Application. Accesses were to both BRAM and DDR memory. The layouts are explained in Table. From this data it was determined the best layout is 4.

4.2 Jacobi Heat Transfer

The Jacobi heat transfer application is the medium memory utilization application and simulates the heat transfer equation using the Jacobi method. Jacobi is an example of a loosely synchronous style of computation, where the program alternates between periods of local computation and synchronous global communication [26]. There is moderate amount of data sharing in this application. In this algorithm some data sharing occurs and is associated with the sharing of bordering rows by adjacent processors. This application is based on a version used by Saldaña [2]. However, it was originally written for a distributed-memory MPI system and as such needed to be converted to a shared-memory implementation.

The algorithm uses two, two-dimensional arrays that are laid out linearly in memory. The tested layouts are described in Table 4.1. The data layout test results can be seen in Figure 4.2. Of the six layouts tested, layouts 3 and 4 performed best. Of these two, layout 4 performed slightly better and as such was used for all examples. In this layout both matrices are in row major order, and data elements of both matrices are interleaved
<table>
<thead>
<tr>
<th>Layout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Matrices in row major order, stored sequentially</td>
</tr>
<tr>
<td>2</td>
<td>Matrices in column major order, stored sequentially</td>
</tr>
<tr>
<td>3</td>
<td>Matrices in row major order, and rows of both matrices are interleaved</td>
</tr>
<tr>
<td>4</td>
<td>Matrices in row major order, and data elements of both matrices are interleaved</td>
</tr>
<tr>
<td>5</td>
<td>Matrices in column major order, and columns of both matrices are interleaved</td>
</tr>
<tr>
<td>6</td>
<td>Matrices in column major order, and data elements of both matrices are interleaved</td>
</tr>
</tbody>
</table>

Table 4.1: Description of layouts tested for Jacobi application.

The algorithm works by iteratively computing new values for matrix elements. The new values of a matrix element are computed by taking the average of the values surrounding it in the previous iteration. The algorithm proceeds until it is determined that the matrix has reached a steady state, at which point the algorithm finishes. When parallelizing, the rows of the matrix are divided amongst the available processing units, each being responsible for processing a set of rows. At the end of every iteration each processing unit returns the sum of squares of the matrix that is used to determine if the matrices have reached steady state. The hardware implementation of this algorithm was successfully run at 100 MHz and had a resource utilization of 2780 of 69120 or 4.0% of LUTs and 4 of 64 or 6.3% of the DSP48Es modules in the FPGA used.

### 4.3 Matrix Multiplication

This is one of the high memory utilization applications, and is a very common task performed in high performance computation. The application was originally adapted from the c_mm.c example application in the OpenMP Source Repository [25]. The method
Chapter 4. Test Applications

Figure 4.3: Data Layout for Matrix Multiplication Application. Accesses were to both BRAM and DDR memory. From this data it was determined the best layout is 4.

used to perform the multiplication is very basic, and there are better algorithms. However, they are more complex and as of yet have not been tested and as such it is not known if it is possible to implement in this system. In this application, the multiplication of two dense matrices A, and B are performed, with the result stored in a third matrix C. All three of these matrices are stored linearly in memory, in either row major or column major order. The algorithm itself has no sharing of data, however there was some false sharing that is inherent with this implementation of the algorithm.

The results of the data layout tests can be seen in Figure 4.3. The tested layouts are described in Table 4.2. Of the eight layouts tested, layouts 2 and 4 performed best. Of these, layout 4 was selected to be used. In this layout, the A matrix is stored in row major order, and the B and C matrices are in column major order. All matrices are stored one after the other as these matrices cannot be interleaved due to their different dimensions.

The main loop in the application can be seen in Figure 4.4. The outer loop is parallelized, with each processing unit being assigned a portion of the outer loop iterations. The hardware implementation of the Matrix Multiplication was successfully run at 100
Table 4.2: Description of layouts tested for Matrix Multiplication application.

<table>
<thead>
<tr>
<th>Layout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Matrix A, B, and C in row major order</td>
</tr>
<tr>
<td>2</td>
<td>Matrix A, and B in row major order, matrix C in column major order</td>
</tr>
<tr>
<td>3</td>
<td>Matrix A, and C in row major order, matrix B in column major order</td>
</tr>
<tr>
<td>4</td>
<td>Matrix A in row major order, matrix B, and C in column major order</td>
</tr>
<tr>
<td>5</td>
<td>Matrix B, and C in row major order, matrix A in column major order</td>
</tr>
<tr>
<td>6</td>
<td>Matrix B in row major order, matrix A, and C in column major order</td>
</tr>
<tr>
<td>7</td>
<td>Matrix C in row major order, matrix A, and B in column major order</td>
</tr>
<tr>
<td>8</td>
<td>Matrix A, B, and C in column major order</td>
</tr>
</tbody>
</table>

MHz and had a resource utilization of 4296 of 69120 or 6.2% of LUTs and none of the DSP48Es modules as all multipliers were converted to LUTs to improve timing.

### 4.4 LU Reduction

LU Reduction is another high memory utilization test application. The LU Reduction algorithm, also known as LU decomposition or LU factorization, is a popular benchmark for testing systems [27][28]. In this algorithm an input matrix A is decomposed into a lower and upper triangular matrix, L and U. When L and U are multiplied together the result is the matrix A. Once computed, L and U can be used to quickly solve many things, such as: the determinant of the A matrix, the inverse of the A matrix, or a system of equations described by the A matrix [29]. The version of LU Reduction used was adapted from the c.lu.c example application in the OpenMP Source Repository [25]. The application computes the LU Reduction based on a modification of Gaussian elimination, called Doolittle’s method [30][31]. In the implementation there are two
for (i=0; i<NRA; i++) {
    for (j=0; j<NCB; j++) {
        sum = 0;
        for (k=0; k<NCA; k++)
            sum += a(i,k) * b(k,j);
        c(i,j) = sum;
    }
}

Figure 4.4: Main Loop of Matrix Multiplication Application

matrices M and L. The M matrix holds both the input matrix A, as well as the upper half of the resultant matrix U from the LU algorithm. L is the resultant matrix L from the LU algorithm, minus the identity matrix. The algorithm itself has no sharing of data, however there was some false sharing that is inherent with this implementation of the algorithm.

The results of the data layout tests can be seen in Figure 4.5. The tested layouts are described in Table 4.3. Of the six layouts tested, 3 and 4 performed best. Of these, layout 4 was selected to be used. In this layout, like Jacobi, both the M and L matrices are stored in row major order, with data elements of both matrices interleaved with one another.

The main loop of the application can be seen in the Figure 4.6. The loop parallelized is the i loop, each processing unit being given a portion to compute.

Unfortunately, this application was not successfully implemented in hardware. The maximum clock speed achieved was only \( \sim 75 \) MHz and not the 100 MHz required; this is due to the divide operation in the algorithm that could not be computed faster. Since the hardware core was unable to meet timing, and because other components of the test
Figure 4.5: Data Layout for LU Reduction Application. Accesses were to both BRAM and DDR memory. From this data it was determined the best layout is 4.

<table>
<thead>
<tr>
<th>Layout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M and L matrices in row major order, stored sequentially</td>
</tr>
<tr>
<td>2</td>
<td>M and L matrices in column major order, stored sequentially</td>
</tr>
<tr>
<td>3</td>
<td>M and L matrices in row major order, and rows of both matrices are interleaved</td>
</tr>
<tr>
<td>4</td>
<td>M and L matrices in row major order, and data elements of both matrices are interleaved</td>
</tr>
<tr>
<td>5</td>
<td>M and L matrices in column major order, and columns of both matrices are interleaved</td>
</tr>
<tr>
<td>6</td>
<td>M and L matrices in column major order, and data elements of both matrices are interleaved</td>
</tr>
</tbody>
</table>

Table 4.3: Description of layouts tested for LU Reduction application.
for(k=0; k<size-1; k++) {
    for (i=k+1; i<size; i++) {
        L(i,k) = (M(i,k)<<BITS) / M(k,k);
        for (j=k+1; j<size; j++){
            M(i,j) = M(i,j) - ((L(i,k)*M(k,j))>>BITS);
        }
    }
}

Figure 4.6: Main Loop of LU Reduction Application

system used are not able to run at a slower clock speed than 100 MHz, this hardware core will not be used as one of the test applications in Section 6.4. Only the software version of this application is used in the relevant tests, the hardware version will not be used.
Chapter 5

Test Hardware and Platforms

To test all the various properties of the coherent caching system and to provide a comparison, several different platforms were created. In Section 5.1 the hardware used to build and test the platforms is described. In Section 5.2 a platform with multiple processors on a single shared bus is described. In Section 5.3 a platform with multiple processors using coherence shared caches is described. In Section 5.4 a platform with a hardware core directly connected to memory is described. Finally, in Section 5.5 a platform with multiple hardware cores connected to coherent caches is described.

5.1 Test Hardware

Though the coherent caching system was tested with older Xilinx Virtex 2 FPGAs, the results reported here were performed using a Xilinx XUPV5-LX110T board. The FPGA in the XUPV5-LX110T is an XC5VLX110T, that contains 69120 6-input LUTs, and 5328 KB of block ram [32]. On the XUPV5-LX110T board are various other components such as, power supply, serial Universal Asynchronous Receiver/Transmitter (UART) for I/O, and 256 MB of DDR2 memory which are used to assist and verify the system in the FPGA [33]. This board was used to perform all tests in the remainder of this thesis.
5.2 Multiple Processors on a Single Shared Bus

This platform is designed to demonstrate how using processors on a shared bus can be used to make a simple shared-memory system. A diagram of the platform can be seen in Figure 5.1. In this platform nine processors share a PLB bus, connected to both 64KB of BRAM memory and 256MB of DDR memory. One of the processors is the Central Processor, and will initiate all computation using FSLs connected to all other FPGAs. The other processors receive these initiations via FSL and begin processing, accessing the shared memory using the shared PLB bus. Each processor has their own private data and instruction memories, and as such the only memory accesses on the PLB bus will be to access the shared data in BRAM or DDR memory.

5.3 Multiple Processors with Coherent Memories

This platform is designed to demonstrate the potential of processors in a shared-memory system when connected to coherent caches. A diagram of this platform can be seen
in Figure 5.2. In this platform, like that described in Section 5.2, nine processors are used; one central processor to initiate processing, and eight compute processors that perform the computations. The difference between this platform and the one described in Section 5.2 is that rather than accessing shared memory through the shared PLB bus, each processor is connected to the coherent caching system that in turn accesses memory. The memories accessible are a 64KB BRAM memory, or 256MB DDR memory.

In this system, the MicroBlaze processors are connected to the Local Caches via a PLB bus and a modified PLB to BRAM interface. The modifications to the PLB to BRAM interface allows the PLB bus to stall while the Local Cache is busy processing a cache miss or a snooped request. Unfortunately, this extra bus and interface adds several cycles of access latency slowing the processor unnecessarily. Ideally, the caches would have been directly connected to the processors using an interface with single cycle latency. The only other interface available on the MicroBlaze for high speed access to memory is the Local Memory Bus (LMB) interface[16]. Unfortunately, this interface could not be stalled when the Local Cache’s BUSY signal was high. Alternatively, in an attempt to minimize the added latency of the extra bus and interface, the PLB bus was put in a P2P (Point to Point) architecture mode. This mode is allowed when only two devices (one master and one slave) are using the PLB bus. When in P2P mode, the PLB bus does not need to perform several steps such as arbitration and, as such, the latency of the bus is reduced, resulting in slight performance improvements.

5.4 Hardware Directly Connected to Memory

The purpose of this platform is to demonstrate the performance of a single custom hardware core without any slowdown as a result of the cache coherence system. A diagram of this platform can be seen in Figure 5.3. In this platform there is one Central Processor that initiates the computation, and one custom hardware core that performs the com-
Figure 5.2: Diagram of multiple processors with coherent memory platform. In this platform multiple MicroBlaze processors are connected to memory via the coherent caching system. $\mu$B represents a MicroBlaze Soft Processor.
Computation. The shared data is stored in a dual ported 64 KB BRAM memory. One port is connected to the PLB bus to allow initialization data to be loaded into memory. The other port is connected to the hardware core to perform the computation. The BUSY input to the hardware core, that would normally be controlled by the coherent cache is tied to low. This indicates to the hardware core that the standard BRAM used will always be ready to read or write data, and will never be busy. The Central processor initiates computation in the hardware core by issuing a command over the FSL, and when computation is complete the hardware core sends a response back via the FSL. This type of design can also be used to verify the correctness of a hardware core. Finally, the limitation of this platform is that only BRAM memory can be used, this is because the hardware cores were designed to use the BRAM interface of the Local Cache. As such the problem size is limited to the size of the BRAM.

![Diagram](image.png)

Figure 5.3: Diagram of single hardware core directly connected to memory. This is used to demonstrate the potential of single hardware cores without any slowdown due to caching or coherence. $\mu$B represents a MicroBlaze Soft Processor and HW represents a custom Hardware core.

5.5 Multiple Hardware with Coherent Memories

This platform is used to demonstrate the scalability of hardware cores using a coherent caching system. This platform is similar to that described in Section 5.3, however, in this case the compute processors have been replaced with custom hardware cores. Each of
these hardware cores are identical and are replications of one another. In this platform, the central processor still initiates the computation via FSL, however, the custom cores perform the processing, rather than the MicroBlaze soft processor. This means that memory accesses to the coherent caches will be more frequent, and have less latency because they are directly connected to the cache with no PLB bus to add latency. After computation is complete the hardware core signals the central processor via FSL. In this platform the memories are still a 64 KB BRAM and 256 MB DDR memory.

Figure 5.4: Diagram of multiple hardware cores with coherent memory platform. This platform is similar to that in Figure 5.2 however the processors have been removed and now hardware cores are directly connected to the caches and in turn main memory.
Chapter 6

Results

In this section the performance results of the coherent caching system will be described and analyzed. The reason for these tests is not to show how fast the particular applications can be accelerated, but rather to show the performance of the coherent caching system, as well as to show possible situations in which this system might be useful, to compare this system against other methods of parallelization, and to identify areas that require improvements. An important metric in determining the performance of a system is it’s efficiency. In this section, efficiency will be measured as a percentage of the ideal linear speedup of a system.

Section 6.1 examines a test where multiple processors share a single shared PLB bus. Section 6.2 examines the ability the vary the cache line size and cache size. Section 6.3 examines the use of processors with coherent caches. Section 6.4 shows the performance hardware cores connected to the coherent caching system. Finally, in Section 6.5 an analytical model for a distributed-memory system will be introduced, and compared against the results achieved using the coherent caching system.
6.1 Processors on a Single Shared Bus

In this section, the results for the platform described in Section 5.2 are presented. In this system computation is performed by eight processors that share a single PLB bus with no caching. All four applications described in Chapter 4 were tested.

In Figure 6.1(a) the results of the test applications when accessing BRAM memory are presented. In this chart it can be seen that when accessing BRAM, the shared PLB bus is sufficient for both the Mandelbrot and Jacobi test applications. Their speedup remains near ideal for all numbers of processors. The small variation in the Mandelbrot test application is due to variations in the workload computed by each processing unit. The shared PLB bus connected to BRAM memory is not sufficient for the Matrix Multiplication, and LU Reduction applications. Both applications were not able to speedup beyond two to three processors. This is because their higher memory utilization quickly saturated the limited bandwidth of the PLB bus.

In Figure 6.1b the results of the test applications when accessing DDR memory are presented. As with the BRAM memory, Matrix Multiplication and LU Reduction saturate the PLB bus. However, due to the added latency of the DDR memory the bus is saturated after only one processor. The added latency also effects the Jacobi test application, now saturating the bus bandwidth after approximately five processors. Mandelbrot with its low memory utilization shows little to no change with the increased latency of the DDR memory.

In summary, it is possible to create a shared-memory platform using a shared PLB bus, however, except in the case of the lowest memory utilization, performance will be poor. As well, if a hardware core were designed to used a shared PLB bus it would likely saturate the bus even faster due to the increased frequency of memory accesses.
Figure 6.1: Speedup of Processors on a Single Shared PLB bus. All speedups are with respect to one processor accessing BRAM memory. (a) shows the results of when BRAM memory is accessed, and (b) shows results for when the much slower DDR memory was accessed. In both, there is no caching.
6.2 Cache Line Size and Cache Size Variation

In this section the ability to modify cache line sizes and cache sizes, is examined and how this effects the performance of the various applications. The purpose of this section is not to demonstrate tuning of caches, this work has already been done in the past [34][35][36]. Rather it is meant to show that not only is it possible to tune the caches but also that it can be done iteratively, using the actual system, and not a simulation. These tests were performed using the platform described in Section 5.3.

First, the cache line size (CLS) was varied from between 8 and 2048 bytes. The speedup is measured with respect to the execution time of a cache line size of 64 bytes. The results can be seen in Figure 6.2. On average the best CLS is 64 bytes. The reason for this is that the PLB bus has a maximum burst size of 16 words, and with the 32 bit bus used, this corresponds to 64 bytes. So with a CLS of less than 64 bytes the burst capabilities of the PLB bus are not fully used. With a CLS beyond 64 bytes no added performance is achieved from the PLB burst capabilities, and increasing contention and false sharing of cache lines unnecessarily slows the system. If a different bus were used, this CLS value may have been different. For the remainder of this thesis a CLS of 64 bytes will be used. As well it should be noted that different applications react differently to varying cache line sizes. For Jacobi and Mandelbrot a less optimal cache line size causes only a slight decrease in performance. By contrast, the high memory utilization test applications, Matrix Multiplication and LU Reduction, varied significantly with a less effective CLS.

Next, the cache size was varied between 128 and 4096 words. The results can be seen in Figure 6.3. In the case of Mandelbrot, the cache size did not effect performance at all. For the other applications, the larger the cache the better the performance of the system. As such, as much memory as possible should be devoted to the caches. For the remainder of this thesis a cache size of 4096 words will be used.
Figure 6.2: Speedup of different CLS for different applications, and BRAM and DDR memory access. All speedups are with respect to a CLS of 64 Bytes accessing BRAM memory.
Figure 6.3: Speedup of different Cache Sizes for different applications, and BRAM and DDR memory access. All speedups are with respect to a Cache Size of 4096 Words accessing BRAM memory.
6.3 Processors with Coherent Memories

In this section the platform described in Section 5.3, where eight processors are connected to the coherent caching system, is tested using all test applications described in Chapter 4. The results can be seen in Figure 6.4, where they are also compared with the results from Section 6.1. First, the results for Mandelbrot can be seen in Figure 6.4(a). For this application, performance was already close to ideal even using a shared bus, as such no significant performance improvement was seen with the caches. Next, the results for Jacobi can be seen in 6.4(b). For this application, slight improvements are seen with BRAM access, even achieving greater than ideal speedup in some cases. This is due to the P2P PLB bus architecture used, which reduces the BRAM access time as compared to the standard PLB bus architecture. In the case of DDR memory access, the performance increase is significant, achieving speedups near that of BRAM memory access. This is because the caching was able to hide the memory latency, and reduce memory accesses. This in turn reduced contention for the shared resources. For the high memory utilization test of Matrix Multiplication and LU Reduction, the results can be seen in Figures 6.4(c) and 6.4(d) respectively. In these two, caching was able to significantly improve performance, both in the case of BRAM and DDR memory access. Again the caching was able to hide most of the access latency, and reduce contention for shared resources. However, performance still begins to degrade after four to five processors are used. The efficiency for eight processors when connected to DDR memory was only 85% and 65% for Matrix Multiplication and LU Reduction respectively. This indicated that though contention for shared resources is reduced it has not been eliminated completely. In this case the contention is for the shared Central Hub. In general, a processor based system could benefit from improvements to the coherent caching system. However, for many applications, the existing system is sufficient.
Figure 6.4: Comparison of Processors with Coherent Caches VS. Processor on a shared PLB bus. Speedups are measured with respect to a single processor on a shared PLB bus. Memory accessed is either BRAM or DDR memory.
<table>
<thead>
<tr>
<th>Application</th>
<th>SW Exec Time</th>
<th>HW Exec Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandelbrot</td>
<td>8.34 s</td>
<td>0.0330 s</td>
<td>252 x</td>
</tr>
<tr>
<td>Jacobi</td>
<td>56.7 s</td>
<td>1.649 s</td>
<td>34.4 x</td>
</tr>
<tr>
<td>Matrix Mult</td>
<td>0.00938 s</td>
<td>0.001314 s</td>
<td>7.13 x</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison between execution time by a single processor and a single hardware core.

### 6.4 Hardware with Coherent Memories

In this test, we examine the performance of several hardware cores connected to coherent memories. The applications tested were: Mandelbrot, Jacobi, and Matrix Multiplication. As was discussed in Section 4.4, the LU Reduction application will not be used for these tests. Though the purpose of this section is not to show the speedup of the applications themselves; it is important to see how much faster the hardware core is compared to a processor. This is because the faster hardware core, will in turn access memory faster, causing a higher utilization of the coherence system. The speedup of the hardware core is comparing a single processor on a shared PLB bus accessing BRAM memory, as described in Section 5.2, against a single hardware core directly connected to a BRAM memory, as described in Section 5.4. As can be seen in Table 6.1 speedups range from between 7.13 and 252.

Next, multiple hardware cores were connected to the coherent caching system. The results can be seen in Figure 6.5. Despite the fact that Mandelbrot is accessing memory over 250-fold faster than with a processor, it still shows near linear speedup for up to 8 processors achieving a maximum speedup of $\sim$7-fold when accessing either BRAM or DDR memory. This indicates the system is sufficient for low memory utilization applications. With Jacobi and Matrix Multiplication, however, performance gains begin to level off when only two hardware cores are used, and after six cores, performance remains the
same or even decreases slightly. The efficiency of eight hardware cores accessing BRAM memory was 45% and 40% for Jacobi and Matrix Multiplication respectively. The efficiency of DDR memory access was only 40% and 30% respectively. Simulations of the system indicated that the likely cause of this slowdown was contention for the Central Hub. It was seen that the Central Hub continuously had outstanding requests waiting to be processed. As such caches were forced to stall until their request was processed by the Central Hub. This indicates that for hardware cores with even moderate memory utilization, the current cache coherence system is not sufficient to accelerate execution beyond only a few hardware cores. Possible solutions to this problem will be discussed in Section 7.3.

6.5 Comparison with Analytic Distributed Memory Model

Another more common method of computing is using a distributed-memory machine. These systems, as described in Section 2.1, are more efficient than a shared-memory machine, as they don’t rely on a single shared resource. The purpose of these tests is to compare how well a distributed-memory model works compared to a shared-memory model. Due to the complexity of building a distributed-memory system, and because distributed-memory systems are more deterministic than a shared-memory system, it was instead decided to create an analytic model of a distributed-memory system. The analytic model is designed to be conservative so that it does not underestimate the performance of a real distributed-memory system. Using software with the coherent caching system, a near linear speedup was achieved for most applications. As such, the results of a distributed-memory system using software will likely be similar with little improvement. It was instead decided to use the hardware cores, where the coherent caching system had performed poorly. It is assumed that the distributed-memory system uses the same
Figure 6.5: Speedup of hardware cores connected to coherent caches. Speedups are with respect to a single hardware core directly connected to BRAM memory. (a) shows the results of when BRAM memory is accessed, and (b) shows results for when the much slower DDR memory was accessed.
hardware cores as those used by the coherent caching system, however, at specific points in execution data is transferred between hardware cores. It is also assumed that there is a limited amount of memory in the system. And so it was decided to keep only a single copy of data in memory at any time, and to share this data with neighboring hardware cores when required. To create the analytic model the execution time was calculated using the measured execution time of a single hardware core directly connected to BRAM memory as described in Section 5.4, and estimates of the time required to distribute data between the hardware cores. When determining the efficiency of a distributed-memory system, a critical parameter is the latency of the communication network between hardware cores. In consultation with the authors of [12], it was determined that a conservative latency for a fully connected network within an FPGA is four cycles. However, this value can vary depending on the network topology, congestion on the network, size of the network, and the speed of memory interfaces, etc. As such, more latency values will be explored to provide a spectrum of possible execution times.

The two applications modeled will be Jacobi, and Matrix Multiplication. Mandelbrot was not analyzed because it achieved near linear speedup using a shared-memory model and results using a distributed-memory model would likely be similar with few differences. LU Reduction was not investigated because it was not properly synthesized in hardware.

6.5.1 Jacobi Analytic Model

The Jacobi application is straightforward to parallelize in a distributed-memory manner. As can be seen in Figure 6.6, the matrices are divided by rows, equally amongst the available hardware cores with one row of overlap on both sides. During each iteration the computation occurs on the data stored locally. Between iterations the top and bottom overlapping rows are shared with neighboring hardware cores.

The equation used to determine the execution time can be divided into three parts. The first part is the initial distribution of data by the initial processor and the final
Figure 6.6: Example of Distributed Jacobi Algorithm. In this example data is distributed amongst three processing nodes. Each computes an iteration of the algorithm. Afterwards, they exchange shared rows of data with neighboring processing nodes, and iterate again[2].
returning of the data by the hardware cores. The equation to calculate this is

\[ \text{init\_and\_final} = ((\text{latency} + x_{\text{dim}} \times y_{\text{dim}}/\text{num\_proc} + 2 \times x_{\text{dim}}) \times \text{num\_proc}) \times 2 \] (6.1)

where \text{latency} is the latency of a burst data transfer, \(x_{\text{dim}}\) is the size of the x dimension of the matrix, \(y_{\text{dim}}\) is the size of the y dimension of the matrix and \text{num\_proc} is the number of processing nodes in the system. The next part of the equation is the time to perform the calculations. To calculate this the following equation is used

\[ \text{calc\_time} = \frac{\text{total\_calc\_time}}{\text{num\_proc}} \] (6.2)

where \text{total\_calc\_time} is the total time to perform all calculations measured in cycles. This value was experimentally computed based on a single hardware core directly connected to a BRAM memory as described in Section 5.4, and who’s performance can be seen in Table 6.1. The reason for using this value is that it provides the most accurate accounting of all calculations that need to be performed. The final part of the equation is the time required to exchange data with other processing units as well as to send interim results back to the main processor. The equation for this is

\[ \text{share\_data} = (4 \times (\text{latency} + x_{\text{dim}}) + \text{num\_proc} \times \text{latency}) \times \text{num\_itr} \] (6.3)

where \text{num\_itr} is the total number of iterations performed by the application. For the given input matrix 4165 iterations were required. Combining the above equations we get the final execution time in a distributed-memory system.

\[ \text{exec\_time} = \text{init\_and\_final} + \text{calc\_time} + \text{share\_data} \] (6.4)

### 6.5.2 Matrix Multiplication Analytic Model

The distributed-memory parallelization of the Matrix Multiplication application is more difficult than that of the Jacobi application. Ideally, one would divide the rows of matrix A amongst the processing units and send an entire copy of B to each processing unit.
The computation can be very quick as each processing unit would have all data required
to compute a portion of the matrix, and no communication between neighboring nodes
would be required. However, this would use a large amount of memory, as the entirety
of the B matrix would need to be replicated multiple times. An alternative is to divide
both the rows of matrix A and the columns of B amongst the processing units. The
processing units can then compute all values of $C$ that they have sufficient inputs for.
Once complete the columns of B are exchanged with neighboring processing units, and
further values of $C$ can be computed. This continues until all values of $C$ are computed.
This method can be seen in Figure 6.7 [37]. The process of transmitting the B matrix
must also be taken into account. The most efficient way to transfer the entire portion
of the B matrix is using a single burst. However, this would require a large buffer, and
doubling of the memory required to store the B matrix. An alternative is to send only
one data element at a time. This would require a single element buffer. However, this
would be very inefficient. As a compromise, a buffer of 16 elements was selected to be
used for the analytic model.

Like with Jacobi the calculation can be divided into three parts. The first is the time
required to distribute matrix A and B to the processing units and the time required to
retrieve the resulting matrix $C$ from the processing units. This is computed using the
equation

$$init\_and\_final = (latency \times num\_proc + dim \times dim) \times 3 \quad (6.5)$$

where $latency$ is the latency of a burst transmission, $num\_proc$ is the number of processing
units and $dim$ is the dimension of the matrices. The next part of the equation is the
time required to perform the calculations. This can be calculated with

$$calc\_time = total\_calc\_time/num\_proc \quad (6.6)$$

where $total\_calc\_time$ is the total time in cycles required to perform all calculations.
For this we will use the computed execution time with a single hardware core directly
Figure 6.7: Example of Distributed Matrix Multiplication Algorithm. In this example data is distributed amongst three processing nodes. Each computes a portion of the output matrix C that they have sufficient inputs A and B to calculate. They then swap sections of B with another processing unit, and compute a new section of C. This continues until all of C has been computed.
connected to a BRAM memory. The hardware platform used to acquire this value is described in Section 5.4 and the execution time can be seen in Table 6.1. The final portion of the equation is the time required to share the B matrix between processing units. This is computed using the equation

\[
\text{share\_data} = ((\text{latency} + 16) \times \text{dim}/16 \times \text{dim}/\text{num\_proc}) \times (\text{num\_proc} - 1)
\]

(6.7)

where 16 represents the buffer size of 16 selected to be used. The total execution time can be calculated by combining the above equations

\[
\text{exec\_time} = \text{init\_and\_final} + \text{calc\_time} + \text{share\_data}
\]

(6.8)

6.5.3 Analysis of Analytic Models

Using Equations 6.4 and 6.8 as described in Sections 6.5.1 and 6.5.2 respectively, the execution time was computed for one to eight processing units, and with communication latencies of four to 100 cycles. Speedup was then computed as compared to a single hardware core directly connected to a BRAM as described in Section 5.4. The resulting speedups can be seen in Figure 6.8, where they are also compared with the speedups of the processing units connected to coherent memories as described in Section 6.4. With the coherent caching system, it was seen that beyond two processing units, performance dropped significantly due to contention for the shared Central Hub. By contrast, with the distributed-memory model, because there is no shared central resource, performance was able to scale to eight processing units and beyond. With future improvements to the coherent caching system that will be discussed in Section 7.3, it is expected that the coherent caching system will perform better, similar to the performance of distributed-memory system with high interconnect latency values. However, it is unlikely the performance of a coherent caching system will ever surpass a distributed-memory system that has a low latency interconnect network. Given these results, it is easy to see why hardware designers would tend to develop distributed-memory systems. However, if their systems
are not accessible to programmers, few applications will be written to use them. As such, the difference between the distributed and shared-memory models can be defined as the penalty for a simplified programming model.
Figure 6.8: Speedup of modeled distributed-memory system as compared to the implemented shared-memory system. Speedup is measured with respect to a single hardware core directly connected to BRAM memory.
Chapter 7

Conclusion

In Section 7.1 a summary of the thesis is provided. In Section 7.2, concluding remarks are presented. Finally, in Section 7.3 future work, and potential improvements to the coherent caching system are presented.

7.1 Summary

The goal of this work was to develop the memory access system for a shared-memory processing system. This memory access system would consist of multiple coherent caches connected to either on-chip or off-chip memory. This coherent caching architecture was designed to be scalable, have fast access times, and have an easy interface for both software and hardware. Another key requirement was to develop a working hardware system first and then study the working system to understand the deficiencies that can be improved. In Chapter 2, background information on shared-memory programming and caching was provided. In Chapter 3, the hardware designed was described. First, the implementation of the write-once protocol was described. Next the hardware designed for the Local Caches and the Central Hub were described. Finally, the resource utilization of all hardware components were described. In Chapter 4, a series of test applications used to test the coherence system were described. The low memory uti-
lization test was Mandelbrot area estimation, the medium memory utilization test was Jacobi heat transfer, and the high memory utilization tests were Matrix Multiplication and LU Reduction. In Chapter 5, the various hardware systems designed to test the cache coherence system, as well the systems designed to compare against the coherent caching system were described. Finally in Chapter 6, the results of all tests performed were presented. It was seen that traditional methods of shared-memory parallelization were not able to cope with the significant load placed on them by multiple processors, particularly when connected to high latency memories, such as an off-chip DDR memory. However, when using a coherent caching system, the bus latency and contention is reduced, allowing multi-processor performance to scale much better. Though performance when connected to processors can be improved with changes to the coherence system, for many applications the current system is sufficient for processor-based applications. When connected to hardware cores, the coherent caching system did not perform as well as was hoped. Contention for the shared Central Hub caused performance to level off after only two hardware cores. Further improvements to the coherence system will be required to fully exploit a system using hardware cores. Finally, an analytic model for a distributed-memory system was presented. When compared with the coherent caching system, performance of the distributed-memory system was significantly better. This difference in performance between a shared and distributed-memory model is defined as the penalty for a simplified programming model.

7.2 Conclusion

In conclusion, it has been shown that for a processor-based system the coherent caching system, performance improvements over a shared bus architecture are significant. Efficiencies of between 97% and 65% were achieved with processors, and trends would indicate that efficiency would remain high for most applications. Though changes to
the coherent caching system would improve performance of some applications, for most applications the performance of the current system is sufficient. In a hardware accelerator system, with the exception of the lowest memory utilizing applications, congestion for the shared Central Hub saturates performance after only a few hardware cores are used. The Jacobi and Matrix Multiplication test applications saw efficiencies of just 40% and 30% respectively. To take advantage of a hardware-based system, changes to the coherent caching system are required. The performance of the coherent caching system is much worse than the competitive distributed-memory systems that do not see such slowdowns due to their lack of shared resources. It is hoped however, that with future improvements, the congestion at the Central Hub can be reduced to bring performance closer to that of the distributed-memory system. The difference between the performance of a distributed-memory system and that of a shared-memory system is defined as the penalty for a simplified programming model. Based on these results, it is felt that future work on the coherent caching system as well as a shared-memory programming model within an FPGA is warranted.

7.3 Future Work

In the past, much work has been done with regards to caching and cache coherence. This thesis has just scratched the surface of what is possible, and as such many improvements can be considered. There are two categories of improvements: those that improve performance and those that increase functionality. All improvements are in the order of which should be implemented first.

7.3.1 Improvements to Performance

Cache to cache transfers At the moment, if there is a read miss, data is retrieved from main memory, regardless of whether the data is in another cache. When accessing
slower memories it might be faster if data was retrieved from another cache rather than main memory. This feature is available in other coherence protocols such as the Illinois coherence protocol[38].

**More advanced caching mechanism** At the moment the caching mechanism is direct-mapped caching. This is fast and easily implemented although the miss rate is higher than other caching mechanisms. A 2-way or 4-way set associative cache might improve the hit rate and performance. However, it is important to be mindful of the added complexity.

**Different coherence protocol** At the moment a write-once coherence protocol is used. A different coherence protocol might be better, for instance as Mesi or MOESI[10][11].

**Broadcast filtering** In a shared bus caching architecture one method of reducing broadcasts is to keep a partial copy of the Cache Table at the Central Hub. If from the contents of the partial table it can be determined that a broadcast can be avoided then the caches can be left to perform other operations. An example of a system like this is RegionScout by Moshovos [39].

**Tighter integration with MicroBlaze** As discussed in Section 5.3, for a MicroBlaze processor to access a Local Cache, it must access it through a PLB bus and PLB to BRAM interface. This bus and interface adds several cycles of extra latency to each memory access. As well, there is already a caching mechanism built into the MicroBlaze processor, called Xilinx CacheLink[16]. If the coherent caching system could be integrated with the MicroBlaze cache system, it could allow for faster cache access and potentially a reduction in redundant logic in the coherent caching system, or the MicroBlaze processor.

**Changing the consistency model** This improvement does not in itself increase performance, but allows more improvements to be made that could increase perfor-
mance. The consistency model currently used is sequential consistency. As discussed in Chapter 3, sequential consistency was selected as the consistency model because of its intuitive nature, and the lack of synchronization in the system. If however, this model were relaxed, other performance improvements would be possible. For instance, processing multiple coherence requests at once, or out of order operations. Before this can be implemented though, it is essential to have a more complete synchronization mechanism.

**Out-of-order operations** The addition of out-of-order operations would allow caches to handle cache hits while an outstanding cache miss is being processed. Using the current BRAM interface it might be possible to perform out of order writes, where multiple writes can be buffered and committed out of order. However, out-of-order reads are not yet possible with the current BRAM interface. To perform out of order reads the interface to the cache would need to be changed substantially. As well, the consistency model must also support out of order operations. When using sequential consistency significant logic is required to ensure consistency.

**Banking** One way to increase performance is to increase the parallelism of the caches using banking as was used in [1]. In banking, the Cache Memory of a cache is duplicated one or more times to provide multiple read ports. In the case of a cache hit on all ports this would allow multiple addresses to be read from the cache at the same time, and in turn reduce memory access time for operations with multiple inputs from memory. When dealing with coherent caches however, potential ramifications to both coherency and consistency must be taken into account.

### 7.3.2 Improvements to Functionality

**Synchronization and atomic operations** At the moment, there is no integrated manner to perform synchronization. In the test systems, FSLs to the central processor
acted as a barrier in the program operation, however, this functionality should be performed within the cache coherence system. This is also an important feature to have before implementing a relaxed consistency model. One possible method to support synchronization is to implement an atomic test and set operation\textsuperscript{[40]}. In this type of operation a cache requests a write to a location in memory. If the current value of the location is zero, then the write proceeds, otherwise no change is made to the memory location. This happens atomically, and would allow for the creation of a lock as the basis of a synchronization mechanism.

**Integration with the FSB** The long term goal of this system is to make a shared-memory system that is integrated with the Intel FSB for the purposes of creating a hardware thread-based programming model. The current cache coherence system is the first step towards achieving this goal and it will be left as future work to achieve the final hardware thread system.

**Hierarchical organization of the Central Hub** At the moment, the Central Hub is a flat structure and only supports up to eight caches. Theoretically it is possible to increase the number of connected caches without resorting to a hierarchical design, however, eventually timing in the Central Hub will become an issue. In a hierarchical design the Central Hub would need to be modified to communicate with another Central Hub via FSLs rather than to a PLB bus, with the exception for the root Central Hub that would still communicate over PLB. This would allow the connection of many more caches without timing issues. As well, when combined with some of the performance improvements described above such as, filtering, cache to cache transfers, or a relaxed consistency model, contention for the Central Hub could be reduced improving system performance.

**Statistics gathering mechanism** Currently, there is no way to determine the exact hit/miss rate, or how often a cache line is evicted due to cache contention. A set of
statistics gathering mechanisms would be helpful in this regard, and could also be used to determine how changes to the coherence system effects cache performance.
Appendix A

Detailed Protocol Specifications

The protocol used to communicate between the Local Cache and Central Hub had to be custom designed as it was to work over two bi-directional pairs of 32-bit point-to-point FSL links. One pair of the links called E FSL (Expected) is used by the Local Caches to send request and receive responses from the Central hub. The other pair called U FSL (Unexpected) is used by the Central Hub to send and receive snooped requests to the Local Caches. The key to this protocol is the fact that all addresses are word addressed leaving the two bottom bits of any address free to encode the type of request. This minimizes the number of messages that need to be sent between the Local Cache and Central Hub.

On E FSL there are three types of requests: Update, Read, and Write.

Update

- Update occurs when a cache line is evicted from the cache and it was previously in the M state.
- Data is sent back to memory.
- The cache sends a request on E FSL with the base address of the cache line with $\{10\}$ in the bottom two bits.
• Afterwards it sequentially sends the data from the cache line on E FSL.
• Because an update is difficult to interrupt, once started the update takes priority over all other requests at the Central Hub.
• If the Central Hub receives an update request while waiting on a read or write, it will process the update before continuing with the read or write.
• The Central Hub then sends the data received to the address specified in main memory via the PLB Master.
• No acknowledgement is required.

Read

• Occurs when the cache line is invalid or not in the cache and new data needs to be loaded from main memory.
• The Local Cache places base address of cache line with \{00\} as the bottom two bits on E FSL.
• It then waits for data to arrive on E FSL.
• When the Central Hub receives a read request it sends a copy of the read request to all other caches on U FSL.
• When the other caches receive this request they check their Cache Table for the cache line.
• They update the state of their Cache Tables appropriately.
• Finally they respond with the original state of the cache line on U FSL.
• If the cache line was M then then they also place the contents of the cache line on U FSL.
• When the Central Hub receives all the responses from all the caches it has two options.
– If none of the caches respond with the M state then data is retrieved from memory and forwarded to the requesting cache.

– If a cache responded with M, then the data is retrieved from U FSL and forwarded to both main memory as well to the requesting cache.

• When the requesting cache receives the data it is stores in its Cache Memory and its Cache Table updated.

Write

• A write occurs when a cache line is written to when in the S state.

• The cache places the address of the word being written with the bottom two bits set to \{01\} onto E FSL.

• Afterwards it also places the data being written onto E FSL.

• The cache then waits for an acknowledgement.
  
  – Originally, no acknowledgement was required, unfortunately, not having one violates SC.
  
  – SC requires that the cache must wait until the write is globally visible before continuing.
  
  – Also because multiple writes can be waiting in the FSLs to be processed, the atomicity of writes can also be violated.

• If the acknowledgment is a PASS \{111\} then the write completed successfully and the cache can continue.

• If the acknowledgment is a FAIL \{011\} then the write failed because another cache what trying to write to the same cache line at the same time.
  
  – In this case, the Cache Table will have changed in the background.
  
  – The cache will have to re examine its Cache Table and start the process of requesting the write over again.
• When the Central Hub receives a write request it first sends data to main memory.
• It then sends a copy of the request on U FSL to all other caches so they can invalidate the data in their caches.
• The other caches receive the header on U FSL and invalidate the appropriate cache line in their Cache Table.
• The caches then send a response of \{111\} to signify that they have successfully invalidated the cache line in their Cache Table.
• When all the caches have signaled a successful invalidation, the Central Hub looks to see if any other caches are currently requesting a write to the same cache line as is currently being processed.
• If any are, the Central Hub removes the request from the E FSL and sends a FAIL acknowledgement to the cache that requested the conflicting write.
• Finally, the Central Hub sends a PASS acknowledgement to the original cache signaling the the write request is finished.

Two other possible situations are when a read or write is received on the PLB slave port of the Central Hub. In these situations the following occurs.

**PLB slave read**

• In this case the procedure for the Central Hub is the same as a standard READ with a few differences.
• First the read request on U FSL is sent to all caches.
• Second, the data to be returned (either from memory or another cache) is sent to the PLB Slave rather than an E FSL.

**PLB slave write**
• In this case a write request is sent over U FSL to invalidate the cache line in all the caches.

• When all the caches have responded, the data is sent to main memory.

• No response to the PLB slave is possible.
Appendix B

Local Cache FSM

In this section the state machine that controls the Local Cache is described. There are five states and 11 tasks that can be performed. Unless otherwise stated it is assumed that the BUSY signal is high indicating that processing is occurring.

**IDLE** BUSY signal is low indicating that cache is not processing. The Local Cache waits for a new request on the BRAM port, or for a snooped request on U FSL. When something is received on the BRAM port the state transitions to the COMPUTE_TASK state. If a snooped request is received the state transitions to READ_U_TABLE.

**COMPUTE_TASK** The contests of the Cache Table are examined. If there is a hit then the BUSY signal stays low, and the state transitions IDLE, or COMPUTE_TASK depending on whether another request is waiting to be processed. If there is a miss, a set of tasks are computed, and the state transitions to EXEC.

**READ_U_TABLE** Using the address from the snooped request the current state of the appropriate cache line is retrieved. The state then transitions to COMPUTE_U_TASK.

**COMPUTE_U_TASK** Depending on the state of the snooped cache line a set of tasks are computed and the state transitions to EXEC.
EXEC In this state the computed tasks are executed. The tasks are perform either sequentially or in parallel depending on the task to be performed. The possible tasks are:

ACK_REPLY Replies to a read request with the previous contents of the cache state on U FSL.

READ_REPLY If the cache line was M, the contents of the cache line is sent on U FSL.

WRITE_ACK Sends an \{111\} on U FSL to signal that the write has been received and processed.

U_UPDATE_TABLE Updates contents of the Cache Table based on a snooped request.

RESTORE_DATA Restores previous data to the memory in the case that a speculative write failed.

UPDATE_MEM Updates main memory when a cache line in the M state is evicted from the cache.

READ_FSL Sends a read request to the on E FSL. When data is returned it is saved to the Cache Memory.

WRITE_FSL Sends a write request to the on E FSL, and waits for a response from the Central Hub on E FSL. If the response is a FAIL then the state transitions to COMPUTE_TASK.

WRITE_BRAM Writes the data to the Cache Memory.

UPDATE_TABLE Updates Cache Table.

CLEAR_BUSY Holds the BUSY signal low for one cycle to indicate that processing is complete.
When all tasks are complete, the state transitions to IDLE, however, if another cache request or snooped request is waiting to be processed, the state transitions to either COMPUTE_TASK or COMPUTE_U_TASK instead.
Appendix C

Central Hub FSM

This section describes the FSM that controls the Central Hub. It is designed and constructed in a similar manner to that of the Local Cache however it has three states and 12 tasks.

**IDLE** The Central Hub is waiting for something to arrive on one of the E FSLs or from the PLB Slave. When something arrives the state transitions to COMPUTE.

**COMPUTE** A set of tasks is computed and the state transitions to EXECUTE.

**EXECUTE** The tasks which were previously computed are executed sequentially or in parallel depending on the tasks. When all tasks are complete the state transitions to IDLE. The possible tasks are:

**UPDATE_MEM** Occurs when an update request is received from a cache. Data from E FSL is written to memory at the address indicated in the request.

**READ_FSL** A read request is sent via U FSL to all caches except the one originally making the read request.

**READ_RECV** Waits for a response form all caches with regards to read.

**READ_MEM** If no cache responded with M in the READ_RECV task, then data is fetched from memory and sent to the original cache requesting the read. If
a cache did respond with M, then data is sent from the U FSL of the cache that responded with M, to main memory as well as the cache which requested the read.

**READ_P2P** Same a READ_MEM except data is sent to the PLB Slave rather than another cache.

**WRITE_P2P** Data written from to PLB Slave port is bridged to the PLB Master.

**WRITE_MEM** Data written from a cache is sent to main memory at the address in the request.

**WRITE_FSL** The write request is written to the U FSL of all the caches except the one that requested the write.

**WRITE_RECV** Waits to receive a response on U FSL from all caches the write was sent to. When a response is received from a cache, the caches E FSL is examined to see if there is a pending write request to the same cache line as was just written to. If so, FAIL_WRITE and CLEAR_FSL are added to the set of tasks.

**WRITE_ACK** Sends an acknowledgement of \{111\} back to the original requesting cache to indicate that the write competed successfully.

**FAIL_WRITE** Sends an acknowledgement of \{011\} to the cache which was identifies in WRITE_RECV as having a conflicting write.

**CLEAR_FSL** Clears the data associated with the write which was failed in FAIL_WRITE task.
Bibliography


conference on Architectural support for programming languages and operating systems, pages 163–174, New York, NY, USA, 2006. ACM.


