An Interconnection Network for a Cache Coherent System on FPGAs

by

Vincent Mirian

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

Copyright © 2010 by Vincent Mirian
Abstract

An Interconnection Network for a Cache Coherent System on FPGAs

Vincent Mirian

Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2010

Field-Programmable Gate Arrays (FPGAs) systems now comprise many processing elements that are processors running software and hardware engines used to accelerate specific functions. To make the programming of such a system simpler, it is easiest to think of a shared-memory environment, much like in current multi-core processor systems. This thesis introduces a novel, shared-memory, cache-coherent infrastructure for heterogeneous systems implemented on FPGAs that can then form the basis of a shared-memory programming model for heterogeneous systems. With simulation results, it is shown that the cache-coherent infrastructure outperforms the infrastructure of Woods [1] with a speedup of 1.10. The thesis explores the various configurations of the cache interconnection network and the benefit of the cache-to-cache cache line data transfer with its impact on main memory access. Finally, the thesis shows the cache-coherent infrastructure has very little overhead when using its cache coherence implementation.
Dedication

I would like to dedicate this thesis to my beloved parents, Fabian and Marie-Juliette Mirian, to my lovely sister, Stéphanie Mirian, for their support throughout my education and most importantly my godmother, Bernadette Caradant, for the inspiration to continue my education.
Acknowledgements

First and foremost, I would like to thank Professor Paul Chow for his invaluable guidance and advice throughout the years. Thank you for supporting this project and making it as fun and exciting as it was stimulating and enlightening. I am grateful for the opportunity to learn so much from you, not only from your extensive experience on technical matters, but on matters of professionalism and conduct as well. I am very grateful for the significant knowledge, guidance and patience provided by Paul Chow. Without Paul Chow, this project would not be possible!

Furthermore, I would like to thank everyone in the Chow group for their support and feedback over the years: Arun Patel, Manuel Saldaña, Jeff Goeders, Danny Gupta, Alireza Heidarbarghi, Andrew House, Professor Jiang Jiang, Alex Kaganov, Chris Madill, Daniel Le Ly, Daniel Nunes, Emanuel Ramalho, Keith Redmond, Kam Pui Tang, David Woods, Charles Lo, Rudy Willenberg, Taneem Ahmed, Jasmina Vasiljevic, Eric Lau and Xander Chin. It was great working with all of you and thank you for making this experience so enjoyable.
# Contents

1 Introduction 1
   1.1 Motivation .................................................. 1
   1.2 Contributions .............................................. 3
   1.3 Overview .................................................... 4

2 Background 5
   2.1 Shared-Memory Architecture ............................ 5
   2.2 Cache ....................................................... 7
   2.3 Cache Coherency Problem ............................... 9
   2.4 Cache Coherence Protocols ............................ 11
      2.4.1 MESI - The States ................................. 13
      2.4.2 MEI Cache Coherence Protocol .................. 13
      2.4.3 MESI Illinois Cache Coherence Protocol ........ 14
      2.4.4 Write-Once Cache Coherence Protocol .......... 14
      2.4.5 Cache Coherence Protocol With Cache-To-Cache Cache Line Data Transfer ............................................. 14
   2.5 Coherence-State Filtering ............................. 17
   2.6 Memory Consistency ...................................... 20
   2.7 Related Work .............................................. 21
3 A Cache Coherent Infrastructure For FPGAs (CCIFF) 25

3.1 General Overview ......................................................... 25
3.2 Cache Coherence Implementation ...................................... 26
  3.2.1 Cache Coherence Messages ........................................ 27
  3.2.2 Cache-to-Cache Cache Line Data Transfer ....................... 28
  3.2.3 Consistency and Coherency Correctness in CCIFF ............... 29
3.3 Components .............................................................. 30
  3.3.1 Cache ............................................................. 30
  3.3.2 Cache Interconnection Network .................................... 35
  3.3.3 The Memory Access Sequencer ................................... 40
  3.3.4 Modified Memory Bus .............................................. 43
  3.3.5 Cache To Cache Interconnection Network Bus ................... 44

4 Methodology 46

  4.1 The Simulation Framework ............................................ 46
  4.2 Modelling Heterogeneous Processing Elements ..................... 50
  4.3 Development of the CCIFF Model .................................... 51

5 Testing Platform 53

  5.1 Testing Platform ..................................................... 53
  5.2 Benchmarks .......................................................... 57
  5.3 Feasibility of FPGA Implementation ................................ 60
    5.3.1 Routing ....................................................... 60
    5.3.2 Logic and Memory Resources .................................. 62

6 Results 63

  6.1 Utilization Balance .................................................. 64
  6.2 Infrastructure Comparison .......................................... 64
  6.3 Cache-To-Cache Cache Line Data Transfer Benefit ................ 67
6.4 Reducing the *Memory Access Sequencer* Bottleneck ....... 71
6.5 *Cache Interconnection Network* Configuration Comparison ........ 77
6.6 Cache Coherence Overhead in CCIFF .......................... 79

7 Conclusion ........................................ 82
  7.1 Summary ........................................ 82
  7.2 Conclusion ..................................... 83
  7.3 Future Work .................................. 84

A Assertion List .................................. 87

B Directory Structure And File Description ................. 89

Bibliography ...................................... 97
List of Tables

2.1 State Transition Description ............................................. 12

5.1 The Cache Coherence Protocols Category Details ..................... 56
5.2 The Various Cache Coherence Protocols ................................. 57

6.1 Reduced Access Percentage For The Jacobi Benchmark ............... 73
6.2 Reduced Access Percentage For The Matrix Multiplication (MM) Benchmark 73
6.3 Reduced Access Percentage For The Dijkstra Benchmark ............. 74
6.4 Reduced Access Percentage For The Basicmath Benchmark .......... 74
6.5 Reduced Access Percentage For The SHA-2 Benchmark ............... 75
6.6 Reduced Access Percentage For The Stringsearch Benchmark ........ 75

A.1 A List Of Assertions ....................................................... 88

B.1 Numeric Value Corresponding to Benchmark ........................... 90
B.2 Numeric Value Corresponding to A Cache Coherence Protocol ....... 91
B.3 Filename Relating To Cache Coherence Protocols of Table 5.2 ....... 92
B.4 Filename Relating To Module For A Cache Component ............... 93
B.5 Testbench Subdirectory description ....................................... 96
List of Figures

2.2 The Direct Mapped Cache Structure ................................. 8
2.3 The Direct Mapped Cache Indexing Example ....................... 9
2.4 The Cache Coherency Problem Example [1] ....................... 10
2.5 The Cache Coherency Solution Example [1] ....................... 11
2.6 The MEI Cache Coherence Protocol State Diagram ............... 14
2.7 The MESI Illinois Cache Coherence Protocol State Diagram .... 15
2.8 The Write Once Cache Coherence Protocol State Diagram ....... 16
2.9 State Diagram of The MEI Cache Coherence Protocol With Cache-To-
    Cache Data Transfer ........................................ 17
2.10 State Diagram of The MESI Illinois Cache Coherence Protocol With Cache-
    To-Cache Data Transfer ...................................... 18
2.11 State Diagram of the Write Once Cache Coherence Protocol With Cache-
    To-Cache Data Transfer ...................................... 19
2.12 A Three-cache system a) Cache A asks the other caches in the system
    for the state of cache line entry X b) Cache B and Cache C responds
    with state M and state I respectively .......................... 20
2.13 Woods’ Cache-Coherent Infrastructure ............................ 24
3.1 The Proposed Cache-Coherent Infrastructure For FPGAs (CCIff) ... 26
3.2 Cache Coherence Message Format ................................. 28
3.3 The cache architecture used in CCIFF. ........................................ 31

3.4 The broadcast message from cache \( W \) viewed by cache \( X \) for cache line \( Y \).
The message requires that a previous message from cache \( Z \) on cache line \( Y \)
be serviced prior to itself. This message is placed on the top of Figures 3.4a
and 3.4b. a) values of the table prior to servicing the message; b) values
of the table once the message was serviced ................................. 37

3.5 The cache interconnection network with the least number of pending mes-
sages issuer. The I-CX-PY label represents Input signals from port \( Y \)
of cache ID \( X \) to the cache interconnection network. The O-CX-PY la-
bel represents Output signals to port \( Y \) of cache ID \( X \) from the cache
interconnection network. .......................................................... 39

3.6 The cache interconnection network with the fixed issuer. The I-CX-CY
label represents Input signals from the cache with ID \( X \) to the cache with
ID \( Y \). The O-CX-CY label represents Output signals from the cache with
ID \( X \) to the cache with ID \( Y \). .................................................... 40

3.7 a) The MPMC is busy with cache block \( S_Z \); b) cache block \( S_X \) request to
write/evict a cache line with address \( X \); c) cache block \( S_Y \) request to read
address \( X \); d) The MPMC will send incorrect data to cache block \( S_Y \); e)
The main memory and cache block \( S_X \) are coherent but not cache block
\( S_Y \) f) Legend ................................................................. 42

3.8 Cache To Memory Access Sequencer Message Packet Format .............. 43

4.1 a) Communication between two RTL modules; b) Communication between
two TLM modules ............................................................... 47

4.2 TLM Simulation Environment Model .......................................... 49

4.3 A CCIFF System with Heterogeneous Processing Elements ................. 51

4.4 Heterogeneous Processing Elements Trace Example .......................... 51
5.1 a) data elements is layout in row major order; b) data elements is layout in column major order; c) data elements is layout in row major order and interleaved .......................................................... 59

6.1 Work Distribution of the Basicmath, SHA-2, Stringsearch and Dijkstra Benchmarks .......................................................... 65

6.2 The Speedup of CCIFF over Woods Using the Write-Once Original Cache Coherence Protocol .................................................. 66

6.3 The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Jacobi Benchmark .......................... 68

6.4 The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Matrix Multiplication Benchmark .............. 68

6.5 The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Dijkstra Benchmark ...................... 69

6.6 The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Basicmath Benchmark ..................... 69

6.7 The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - SHA-2 Benchmark ......................... 70

6.8 The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Stringsearch Benchmark ................. 70

6.9 Topology Comparison Using the MEI C2C_CIN cache coherence protocol ............................................................................ 78

6.10 Topology Comparison Using the MEI C2C_MEM cache coherence protocol ................................................................. 78

6.11 The Cache Coherence Overhead in the CCIFF .................................................. 81
Chapter 1

Introduction

Field-Programmable Gate Arrays (FPGAs) are now capable of holding large multiprocessor systems that comprise heterogeneous processing elements. This presents new challenges as multiprocessors are migrated to the world of embedded computing. This thesis will introduce potential solutions addressing some of these challenges.

1.1 Motivation

Advancements in semiconductor manufacturing processes present more transistors in a single chip. In the realm of reconfigurable computing, these additional transistors are used to include more hard-block components as well as additional reconfigurable logic on the chip. The reconfigurable logic can be used as soft processors and custom hardware logic, and thus increase the processing power on the chip. For example, Saldaña et al. [3] configured a Xilinx Virtex 4 FPGA with 32 MicroBlaze processors [4]. Recently announced FPGAs, such as the Virtex 7, in comparison can theoretically hold 100’s of MicroBlaze processors. This trend will only continue if Moore’s Law continues to hold true.

The processing power available on FPGAs can be of various types. System designers use these various types of processing power to map tasks from applications best fit...
for their architecture and to exploit parallelism. This presents systems with heterogeneous processing elements in the realm of FPGAs, which produces new challenges in interconnecting the heterogeneous processing elements and programming such a system.

In software, programming models allow communication between processors abstracting the underlining architectural properties of the system. The programming models are designed in accordance to two generic memory models: the distributed-memory model and the shared-memory model [2]. The shared-memory model is an easier model for accessing memory since the processors share a single view of the data.

An important aspect of using shared-memory is the memory consistency model. The shared-memory programming model is built on the basis of a memory consistency model. Gharachorloo et al. [5] state that the advantage of a less strict memory consistency models, such as the weak consistency model, is increased performance. They also state that the disadvantage of a less strict memory consistency model is increased hardware complexity resulting in a more complex programming model. The reasoning behind this non-intuitive conclusion is that a less strict consistency model uses more hardware to provide additional functionality for proper execution.

Shared-memory systems can also contain caches. The primary uses of caches are to decrease main memory utilization and reduce the latency of processor memory requests. The caches need to be coherent for proper and valid execution. This requires cache coherency protocols that must communicate amongst the caches, which in turn allows processors in a multiprocessor system to communicate with each other as needed in parallel programming.

Extensive research evaluating trade-offs for achieving the best cache performance have been done in the world of application-specific integrated circuits (ASICs) [6]. An ASIC is an integrated circuit customized for a particular use. Once the ASIC is manufactured, the ASIC function is fixed. But FPGAs differ from ASICs with their limited resources, routing constraints and programmable architecture. For example, content-addressable
memories (CAMs) cannot be implemented efficiently and buses must use multiplexers because FPGAs do not have tri-state or open-collector drivers. This example underlines that the building block for both the FPGA and ASICs are different. Therefore, the research on ASICs is not directly applicable to FPGAs.

A cache-coherent shared-memory system for FPGAs becomes important especially with the growth of multiprocessors on FPGAs. However, they are not commonly used as they are difficult to build. Building cache-coherent shared-memory system for FPGAs requires the knowledge of caches, cache coherence and related topics which are individually a large research area on its own. Furthermore, the designing of any system absorbs the difficulty of mapping the system to the architecture and resources of an FPGA.

Currently in hardware, FPGA vendors do not provide a standard library of intellectual property (IP) that can be used as building blocks to construct shared-memory systems with caches. If a shared-memory system was available for FPGAs, it would ease programming of heterogeneous multiprocessors that are often implemented on FPGAs. This thesis will take the first step of programming heterogeneous multiprocessors in shared-memory systems by presenting a cache-coherent infrastructure for FPGAs (CCIff).

1.2 Contributions

The thesis proposes a cache-coherent infrastructure and evaluates the interconnecting network for a cache-coherent system for FPGAs. The primary contributions of this work are as follows:

- a cache-coherent infrastructure for heterogeneous processing elements on FPGAs
- a simulation framework to explore cache-coherent infrastructures for FPGAs
- a cache-coherent system based on messages passed between the caches
• an easily parametrizable simulator modelling the cache infrastructure using SystemC

• the first implementation of cache-to-cache cache line data transfer for FPGAs

• a cache structure independent of the cache coherence protocol

• a distributed coherence-state filtering mechanism

• a flush mechanism with the ability to flush the entire cache with a lookahead feature

• exploration of the interconnecting network for the MEI, MESI Illinois and write once protocols; with analysis of the interconnecting networks using these protocols with cache-to-cache cache line data transfer

1.3 Overview

The remainder of this thesis is organized as follows: Chapter 2 provides background material for understanding this thesis. Chapter 3 provides a description on the cache-coherent infrastructure for FPGAs (CCIFF) architecture, the cache coherence implementation of CCIFF and the major components of CCIFF. The simulation framework and its benefits to model the CCIFF are described in Chapter 4. Chapter 5 continues with detail on the testing platform used to generate results and a discussion on the feasibility of placing and routing the testing platform on a modern FPGA. Chapter 6 provides an analysis of the results. Finally, Chapter 7 provides a summary, conclusion, and description of future work for the system.
Chapter 2

Background

This chapter presents some background material on a shared-memory architecture, caches and the direct mapped structure and explains the cache coherency problem by providing an example. This chapter also defines the cache coherence states used by the cache coherence protocols of this thesis. Then, the cache coherence protocols used to generate the results of this thesis are described. The chapter continues with a definition of coherence-state filtering and an explanation of the memory consistency model used in this thesis. Finally, the chapter ends with a description of related work.

2.1 Shared-Memory Architecture

A shared-memory architecture consists of a shared-memory space. The shared-memory space is shared amongst the processing elements in the system. The underlining factor distinguishing a shared-memory architecture with other architectures is the shared network from the processing elements to the shared-memory space. Figure 2.1 shows an example of the architecture of a multiprocessor shared-memory system, often referred to as Uniform Memory Access (UMA). The example has four processors connected to one or more levels of cache. The highest cache level is connected to a shared medium that is also connected to the main memory and the I/O system.
Figure 2.1: Multiprocessor shared-memory Model [2]
2.2 Cache

Shared-memory systems use caches to decrease main memory usage and reduce processor memory access latency. In a shared-memory system, caches use memory units that are relatively smaller in size compared to the shared-memory space. There can be multiple levels of cache in a system. Typically the lowest level of cache is connected to the processor, and the highest level of cache is connected to the shared medium (Figure 2.1).

A cache can hold multiple cache line entries and can have various organisations. A simple and common organisation is the direct mapped cache structure. This thesis used this structure for the implementation of its caches because the infrastructure presented in this thesis is compared to the infrastructure of Woods [1] in Section 6.2.

In the direct mapped structure, the cache line entries consist of a cache line tag, a cache line state and the cache line data. A cache table stores the cache line tag and cache line state and a separate storage space contains the cache line data. These elements are shown on the right side of Figure 2.2.

In a direct mapped cache structure, the main memory is divided into segments where the segment size is equivalent to the cache line data size; and these main memory segments can be mapped to a single cache line entry in the cache. In the example of Figure 2.2, the main memory can be divided in 12 segments, the cache can store up to four cache line entries. The arrows show to which cache line entry the main memory segments can be mapped.

The cache line tag is the ID of the cache line entry and is used to distinguish the multiple main memory segments mapped to a single cache line entry from each other. The cache state dictates the state of the cache line entry and the state definition depends on the cache coherence protocol (discussed in Section 2.4). The cache line data contains a continuous set of words from the shared-memory space where its size must be a value from the set containing only power of twos.

The bits of the physical memory address are used to index the correct data from
Figure 2.2: The Direct Mapped Cache Structure
Chapter 2. Background

2.2 The Cache

Figure 2.3 shows an example of the bit partitioning of the physical memory address for a direct mapped cache structure with a size of 4096 words, cache line size of 16 words and a word length of 32 bits in a byte-accessible memory architecture. The lower two bits of the address correspond to the four bytes dictating the size of a single word in the shared memory space. The following four bits index the word in the cache line data. The following eight bits index the cache line entry. The remainder of the bits are used as the tag for the cache line.

![Address Showing Bit Position](image)

Figure 2.3: The Direct Mapped Cache Indexing Example

2.3 Cache Coherency Problem

An example of the coherency problem is depicted in Figure 2.4. The example shows two processors using a shared bus to communicate with the shared-memory space. The processors have their own cache and use a write-through policy but no cache coherence protocol. The shared-memory space originally contains the value 0 for memory address A as depicted in Figure 2.4a. In Figure 2.4b, Processor 1 (Proc 1) will read memory address A from the shared-memory space and place the content in its cache. Similarly Processor 2 (Proc 2) will do the same shown in Figure 2.4c. At this point, the contents for memory address A in the caches and the shared-memory space are identical. In Figure 2.4d, Proc 1 will write the value 1 in memory address A, which is propagated to the shared-memory space due to the write through policy. Proc 2 will now read memory
address A from its cache and read the old value of memory address A, which is incorrect as seen in Figure 2.4e.

![Figure 2.4: The Cache Coherency Problem Example [1]](image-url)
A potential solution is shown in Figure 2.5. The problem is present in Figure 2.4d where the caches become incoherent. A solution would be to broadcast the action of Proc 1 to the other caches in the system and invalidate the content of the caches corresponding to memory address A as depicted in Figure 2.5a. This will cause Proc 2 to instantiate a new read to the shared-memory space for the correct value as seen in Figure 2.5b.

![Figure 2.5: The Cache Coherency Solution Example [1]](image)

The shared medium between the caches play a big role for cache coherency. A shared medium is typically a bus such as the Front Side Bus (FSB) [7]. The benefit of the Front Side Bus is the ability of the nodes to snoop the bus lines and listen to transactions relevant to each node. There is no commercial block providing this functionality for shared-memory systems designed in FPGAs. This thesis leverages the existing FPGA architecture to design a shared communication medium for cache coherency on FPGAs.

### 2.4 Cache Coherence Protocols

As demonstrated in Section 2.3, the caches of a multiprocessor system need to be coherent for proper and valid execution. This requires cache coherency protocols that must communicate amongst the caches. Typically, the caches communicate with cache-coherent
messages holding information such as the state of a cache line entry.

There are two types of cache coherence protocols: the directory protocol and the snoopy protocol [2]. The directory protocol requires a table to maintain the sharing of cache line entries. The snoopy protocol requires the caches in the system to be aware of the actions the other caches take on memory addresses. This is typically done with caches broadcasting their actions to the other caches in the system. This thesis uses snoopy cache coherence protocols for reasons discussed in Section 5.1.

Table 2.1 defines the events triggering the state transitions for the cache coherence protocol state diagrams and the actions taken on state transitions. The state diagrams represent the state transition of a cache line entry. If the cache line entry is in the Modified state on a cache miss, it is initially evicted to the shared memory (not depicted in the state diagrams).

<table>
<thead>
<tr>
<th>Label</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Processor Read</td>
</tr>
<tr>
<td>B</td>
<td>Processor Read with cache line present in another cache in the system</td>
</tr>
<tr>
<td>C</td>
<td>Processor Read with cache line not present in another cache in the system</td>
</tr>
<tr>
<td>D</td>
<td>Processor Write</td>
</tr>
<tr>
<td>F</td>
<td>Bus Snoops Read</td>
</tr>
<tr>
<td>G</td>
<td>Bus Snoops Write</td>
</tr>
<tr>
<td>H</td>
<td>Cache To Cache Data Transfer</td>
</tr>
</tbody>
</table>

The remainder of this section defines the cache coherence states used by the cache coherence protocols of this thesis, and describes the cache coherence protocols used to generate the results of this thesis.
2.4.1 Mesi - The States

States are assigned to cache line entries. There are at most four states used by the cache coherence protocols in this thesis: Modified (M), Exclusive (E), Shared (S) and Invalid (I); the Modified, Exclusive and Shared states are also often referred to as the Dirty, Reserved and Valid states respectively.

The Modified state defines a cache line entry as unique and the cache line entry is the sole copy in all the caches in the system; furthermore, the cache line data content differs from that of the shared-memory space. When a cache line entry is in the Exclusive state, it means that there is only one copy of the cache line entry present in the caches of the system and that it contains the same memory content as the shared-memory space. Unlike the Exclusive state, a cache line entry in the Shared state means that there can be multiple copies of the cache line entry in the caches of the system and the cache line entry copies also have the same memory content as the shared-memory space. The Invalid state is assigned to cache line entries where the cache line tag and cache line data content should be ignored. The Mesi states are sufficient to test the infrastructure presented in this thesis for reasons discussed in Section 5.1.

2.4.2 Mei Cache Coherence Protocol

The MEI cache coherence protocol is a very restrictive cache coherence protocol. The protocol does not allow any sharing amongst the caches in the system. If this cache coherence protocol is used by an application where sharing amongst the caches is present, the cache line entries may be ping-ponged causing large amounts of traffic in the shared communication medium. The MEI states are also used by single processor systems with a write-back policy cache to maintain consistent data in the system. Figure 2.6 shows the state diagram of the MEI cache coherence protocol.
2.4.3 Mesi Illinois Cache Coherence Protocol

The Mesi Illinois cache coherence protocol is an extension of the MEI cache coherence protocol, where the Mesi Illinois allows shared cache line entries amongst the caches in the system. The Mesi Illinois Cache Coherence Protocol is also used by Intel [8]. Figure 2.7 shows the state diagram of the Mesi Illinois cache coherence protocol.

2.4.4 Write-Once Cache Coherence Protocol

The Write Once cache coherence protocol is the first cache coherence protocol published by Goodman [9]. The Write Once cache coherence protocol differs from the Mesi Illinois by executing a write-through action on a write to a newly allocated cache line entry. Figure 2.8 shows the state diagram of the Write Once cache coherence protocol.

2.4.5 Cache Coherence Protocol With Cache-To-Cache Cache Line Data Transfer

The cache-to-cache cache line data transfer allows a cache to send the cache line data of a cache line entry to another cache without the other cache requesting the cache line data from the shared-memory space. The action is triggered when a cache snoops a read or write of a cache line entry that resides in its cache and the state of the cache line is in the Modified or Exclusive state. The cache-to-cache cache line data transfer version of the
Figure 2.7: The MESI Illinois Cache Coherence Protocol State Diagram
Figure 2.8: The Write Once Cache Coherence Protocol State Diagram
MEI, MESI Illinois and Write Once cache coherence protocol state diagrams are depicted in Figures 2.9, 2.10 and 2.11 respectively. The annotation $X/Y$ on state transition in Figures 2.9, 2.10 and 2.11 refers to a state transition triggered on event $X$ and performing action $Y$.

Figure 2.9: State Diagram of The MEI Cache Coherence Protocol With Cache-To-Cache Data Transfer

### 2.5 Coherence-State Filtering

The infrastructure presented in this thesis requires some actions, such as cache-to-cache cache line data transfer, performed by the caches to know the cache line state of a cache line entry of all the caches in the system. Once the cache receives all the cache line states of a cache line entry from the caches in the system, the cache would need to filter cache coherence states to the state with highest priority. This is known as coherence-state filtering.

With the cache coherence protocols of Section 2.4, cache line entries corresponding to the same memory segment can be in at most a single state. In the coherence-state filtering mechanism used in this thesis the $M$, $E$ and $S$ states have equal priority but are higher priority than the $I$ state.

Figure 2.12 shows a three-cache system. In Figure 2.12a, Cache A asks the other caches in the system for their cache line state of a cache line entry $X$. Cache B responds
Figure 2.10: State Diagram of The MESI Illinois Cache Coherence Protocol With Cache-To-Cache Data Transfer
Figure 2.11: State Diagram of the Write Once Cache Coherence Protocol With Cache-To-Cache Data Transfer
with state \( M \) and \textbf{Cache C} responds with state \( I \), where the \( I \) state denotes that cache line entry \( X \) is not present in Cache C (Figure 2.12b). \textbf{Cache A} will filter these states, \( M \) and \( I \), and retrieve the state with highest priority. In this scenario, the state with highest priority is the \( M \) state.

![Diagram](image)

**Figure 2.12**: A Three-cache system a) \textbf{Cache A} asks the other caches in the system for the state of cache line entry \( X \) b) \textbf{Cache B} and \textbf{Cache C} responds with state \( M \) and state \( I \) respectively

### 2.6 Memory Consistency

The memory consistency model represents the order the data reads and the data writes are seen by the caches in the system. Coherence defines that the data written to a location of the shared-memory space will eventually be visible by all other caches, but it makes no guarantee as to when the other caches will be able to see it; this is defined by consistency. The memory consistency model used in this thesis is Sequential Consistency [10].

Sequential Consistency requires the result of any execution to be the same as if the operations of all the processors were executed in some sequential order. It also requires that the operations of each individual processor appearing in this sequence are in the order
specified by its program. The minimum set of requirements for Sequential Consistency 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;
- And reads must wait for the write who’s data is being read to be visible by all caches

The restrictive nature of the sequential consistency model presents difficulties for exploiting parallelism in designing a system using sequential consistency. But this consistency model is simple to understand and other consistency models require functionality outside the scope of this thesis.

2.7 Related Work

Significant work on caches, cache infrastructures and interconnection networks for caches exists in the ASIC realm for systems such as multi-core processors. Recent ASIC multi-core processors, such as the Single-Chip Cloud Computer (SCC) by Intel [11] and the Tile processors by Tilera [12], adopted novel techniques and implemented architectural changes to support cache coherency for a large number of cores.

The Single-Chip Cloud Computer (SCC) by Intel [13] does not have hardware cache coherency support amongst the cores. The SCC uses a Message Passing protocol to synchronize the caches. The Message Passing protocol requires routers interconnecting the cores and a routing scheme managed in software. The Message Passing protocol also requires a special instruction for cache consistency enforcing a single copy of the cache line to reside in at most a single cache on the chip. However, the SCC suffers
from software overhead for cache coherency and storage overhead with the large message buffers connecting the cores.

Tilera [12] uses the iMesh Technology and the Dynamic Distributed Cache technology for cache coherency. The iMesh technology uses reconfigurable routers managed by software to interconnect the cores. The Dynamic Distributed Cache technology allows cache line data transfer amongst the caches and permits the multiple L2 caches of all cores to be viewed as a L3 cache. The benefit of the Tilera design is found in its routing properties. The routing within a tile is directly connected to the processor pipeline allowing data transfer to occur as early as possible.

Heterogeneous multiprocessor systems with FPGAs have been developed with various architectures. The FPGA is generally used as a coprocessor alongside a CPU, as a peer processor to the CPU or as a standalone system with a self-contained multiprocessor. The typical use of heterogeneous multiprocessors systems is for performance increase via parallel programming. Parallel programs require communication amongst the processors. The programming model provides the inter-processor communication.

Andrews et al. [14] propose hthreads as a shared-memory programming model for an FPGA-based system. Andrews further expands the programming model with the features in Agron [15] to heterogeneous reconfigurable computing. Hthreads extends the existing Pthreads software model [16] to hardware. The hthreads programming model introduces a standardized hardware thread interface and hardware run-time OS-like microkernel to manage the threads in the system [17]. The hardware run-time OS-like microkernels use a single control center and require additional computation and communication latency for thread management. The hardware thread interface allows communication from heterogeneous processing elements to the rest of the system. In typical systems, a release operation on a synchronization variable requires the system to release the synchronization variable, select another thread and schedule the selected thread. These steps are performed in software, preventing the CPU from doing useful work. The underlying
benefit of \textit{hthreads} is that the steps for a release operation on a synchronization variable are done in hardware within the OS-like microkernel, running concurrently to the CPU \cite{14}. \textit{Hthreads} utilizes a Processor Local Bus \cite{18} as the shared medium to form the architecture found in Figure 2.1.

SHMEM \cite{19} is a hybrid message-passing/shared-memory programming model. SHMEM inherits properties from both the message-passing and shared-memory paradigms yielding SHMEM as a more complete and diverse programming model. It allows explicit communication and non-blocking, one-sided communication without involving explicit communication to remote processing elements. SHMEM+ \cite{20} is a programming model expanding some function calls of SHMEM to include reconfigurable devices and allows heterogeneous computing using CPUs and FPGAs.

Little work on cache coherency has been done in the FPGA world. Hung et al. \cite{21} introduce a cache coherency module allowing cache coherency amongst homogeneous soft processors. The cache coherency module snoops the memory transaction to shared memory and uses a directory to maintain coherency. This approach is known as a hybrid snooping protocol. The cache coherency module functions with the embedded caches of the soft processors and requires interrupt service routines to manipulate the caches. The system runs under a strict memory consistency where any read to a memory location returns the value stored by the most recent write operation to that location. Due to the memory consistency model and particularly the utilization of interrupts for cache manipulation, the system is slow.

The Tag Shared Variable Memory (TSVM) presented in Yamawaki et al. \cite{22} uses a separate communication network for cache coherency. TSVM benefits by combining communication and synchronization within the coherence scheme. This weak consistency memory model distinguishes between the synchronization variables and non-synchronization data types. The synchronization variables reside in the TSVM memory space as opposed to non-synchronization data types found in the shared-memory space.
This requires additional storage for the TSVM memory space and logic to identify the corresponding data type. The system also does not allow dynamic allocation of new TSVM data types nor does it have cache-to-cache data transfer.

Woods [1] presents a cache infrastructure for FPGAs functioning under the sequential consistency memory model. The caches allow one memory request per processing element at a time. Figure 2.13 shows Woods’ Cache-Coherent Infrastructure. The caches are connected to a Central Hub via full-duplex links. The Central Hub is the communication medium for both cache coherence messages and main memory access resulting in a large amount of network traffic. The coherence-state filtering function is also embedded in the Central Hub. All the functions packed in the Central Hub increases the logic found in the unit and in turn increases the difficulty for placing and routing a system using the Central Hub on an FPGA.

Figure 2.13: Woods’ Cache-Coherent Infrastructure
Chapter 3

A Cache Coherent Infrastructure For FPGAs (CCIFF)

This chapter will introduce the architecture of the Cache Coherent Infrastructure For FPGAs (CCIFF). Then, this chapter will detail the cache coherence implementation of the CCIFF. Finally, this chapter will further describe the major components of the CCIFF.

3.1 General Overview

To explore the architecture of a shared-memory system using caches in an FPGA, the Cache-Coherent Infrastructure For FPGAs (CCIFF) was developed. Figure 3.1 shows an example system using the Cache-Coherent Infrastructure For FPGAs (CCIFF) and underlines the major components of CCIFF: the modified memory bus, the cache, the cache interconnection network, the memory access sequencer and the full-duplex communication links. Each component executes functions upon requests generated by other components in the system using messages. The messages encapsulate the function call as well as its parameters. A component services a message if the component has sufficient resources to satisfy the request in the message.
The processing elements (PE) represent heterogeneous processing elements such as soft processors, hard processors and hardware cores. The processing elements use the modified memory bus to send memory requests to the caches ($\text{Cache blocks}$). The full-duplex communication links, essentially FIFOs, connect the caches to the memory access sequencer and to the cache interconnection network. Each cache connects to the cache interconnection network with up to seven ports. The memory access sequencer is connected to the memory controller, which in turn is connected to the shared-memory space, the off-chip DRAM.

### 3.2 Cache Coherence Implementation

One of the novelties of the CCIFF is its implementation of cache coherence using a unique set of cache coherence messages and a distributed coherence-state filtering mechanism. The coherency is established by sending messages from one cache to another as in the
SCC by Intel [11]. The cache coherence messages are sent over the cache interconnection network and the memory requests are sent from the caches to the memory access sequencer. This gives the CCIFF two distinct networks: one for the cache coherence messages to provide cache maintenance, as in Yamawaki et al. [22], and the other for memory requests via the memory access sequencer.

In Woods, the cache coherence messages and their snooped responses along with the memory requests are handled in the Central Hub. The CCIFF decouples the cache coherence messages and the memory requests to separate networks. These modifications will increase performance over the design of Woods by removing the contention at the Central Hub. Furthermore, the CCIFF contains a distributed coherence-state filtering mechanism. This feature reduces the logic of the Central Hub, analogous to the cache interconnection network, facilitating placing and routing of the CCIFF architecture on a FPGA and increases the scalability of the system.

A typical memory request procedure using the CCIFF would be as follows: a requesting cache sends a cache coherence message requesting the state of a cache line over the cache interconnection network. All caches except for the requesting cache would respond with the state of the cache line to the requesting cache over the cache interconnection network. Once the requesting cache receives all the response messages, it will proceed with the appropriate request.

### 3.2.1 Cache Coherence Messages

The caches maintain coherency using cache coherence messages. The cache coherence messages can be from one of two categories: the broadcast category where a message is sent to all other caches in the system, or the response category where the message is sent from a cache to a designated recipient. There are three message types in the broadcast category: the read shared, read exclusive and write invalidate types; and a single message type in the response category: the response type. The response message type
is used to respond to broadcast messages and to pass the cache line data from one cache to another.

Figure 3.2 shows the format of a cache coherence message. The \( \text{Msg} \) field corresponds to the message type. The \( \text{To Cache ID} \) and the \( \text{From Cache ID} \) field correspond to the cache ID of the receiving and sending caches respectively. For a broadcast message, \( \text{To Cache ID} \) and the \( \text{From Cache ID} \) field contain the same cache ID. The \( \text{Address/Data} \) contains the address for a broadcast message but can contain either an address or data for a \textit{response} message. The \( \text{State} \) field contains the state of a cache line and is only valid in a \textit{response} message. The \( \text{Priority Precedence} \) field contains the cache ID that must be serviced before the current message can be serviced. The \( \text{is_data} \) field is only used in the response messages, and distinguishes the data and the address in the \( \text{Address/Data} \) field.

<table>
<thead>
<tr>
<th>( \text{Msg} )</th>
<th>( \text{To Cache ID} )</th>
<th>( \text{From Cache ID} )</th>
<th>( \text{State} )</th>
<th>( \text{Address/Data} )</th>
<th>( \text{Priority Precedence} )</th>
<th>( \text{is_data} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.2: Cache Coherence Message Format

### 3.2.2 Cache-to-Cache Cache Line Data Transfer

There are two networks where data from one cache can be sent to the next, the \textit{cache interconnection network} and the interconnect found in the \textit{memory access sequencer}. As mentioned in Section 2.4.1, there are cache coherence states dictating that only one cache line entry is present in the system: the Modified and Exclusive states. A cache coherence protocol with cache-to-cache cache line data transfer will allow a \textit{cache} to send the cache line data only when the cache line entry is either in the Modified or the
Exclusive state. Implementing the cache-to-cache cache line data transfer feature allows the caches to act as shared memory with respect to the other caches in the system, similar to the Dynamic Distributed Cache technology from Tilera [12]. When cache line data is transferred using the cache interconnection network, the main memory utilization is reduced. Transferring cache line data over the cache interconnection network also occurs concurrently to main memory access, thus presenting parallelism and in turn potential performance gain.

### 3.2.3 Consistency and Coherency Correctness in CCIFF

This section will describe how the design and implementation of CCIFF satisfies the sequential consistency memory model and the cache coherence model. In a CCIFF system, the processing elements are allowed to issue one memory request to the cache at a time. The memory requests are issued in program order satisfying the first property of sequential consistency from Section 2.6: \textit{All memory issues must occur in program order.}

The remainder of the constraints described in Section 2.6 are satisfied because all cache coherence protocols developed for this thesis require \texttt{response} messages from all other caches in the system when a broadcast message is sent by a cache. Since the broadcast message denotes a read or write action, no read or write action is committed unless all the caches in the system (except the cache performing the read or write operation) return a \texttt{response} message. The receipt of a \texttt{response} message implies that the sender of the \texttt{response} message acknowledges the broadcast message representing the read or write operation.

The correctness of the cache coherence model in CCIFF is omitted from this Section since the correctness of the cache coherence model depends on the cache coherence protocol. The correctness of the cache coherence protocols will not be discussed as it is not in the scope of this thesis and the correctness of the cache coherence protocols have already been proven. However, implementation details of the cache coherence protocols
used in thesis will briefly be discussed.

Cache coherence correctness relies on a series of actions to occur: sequentially in a restricted time frame and/or in a single time frame. The implementation of the cache coherence protocols used in this thesis satisfies such requirements. Guaranteeing a series of actions to be performed sequentially in a restricted time frame is satisfied by giving the cache coherence protocols highest priority when involved in any form of arbitration requests. When a series of actions need to occur within a single time frame, the cache coherence protocols are implemented such that they will only service a cache coherence message if all resources required by the message are available.

3.3 Components

This section will discuss in further detail the major components in the CCIFF.

3.3.1 Cache

The cache component must service the memory request, reads and writes to a physical memory address, and the flush requests from the processing elements. The cache also provides coherence functionality. In the CCIFF, the cache organisation is direct mapped and each cache possesses a cache identifier (cache ID). Figure 3.3 shows the block diagram of the cache architecture.

When a cache acknowledges a memory request, the memory request is added to the transaction manager (Trxn Mgr). The Trxn Mgr module manages the state of the memory request currently being serviced by the cache. Although not depicted in Figure 3.3, the transaction manager has a connection to all the modules in the cache. The physical memory address of the memory request is mapped to a cache line entry. For a cache line entry, the cache line data is contained in the cache data module and the cache line tag and cache line state is held in the cache table.
Figure 3.3: The cache architecture used in CCIFF.
To determine whether a cache line entry corresponds to a memory request, the cache gets help from the Hit module. The Hit module dictates whether the current memory request is a cache hit, i.e. the cache line entry corresponding to the physical memory address requested by the processing element is in the cache, or a cache miss, i.e. the cache line entry corresponding to the physical memory address requested by the processing element is not in the cache. The memory content of a memory request resides in a cache line entry if the cache line state value is either M, E or S and the cache line tag matches the tag of the physical memory address of the memory request.

On a cache miss, the cache will need to know where the cache line entry resides in the DRAM or in other caches in the system. To acquire this knowledge the cache will send a cache coherence message to all other caches in the system via the cache interconnection network using the cache network interface module. All the other caches in the system will respond with a response message. The cache coherence response messages will be handled by the response message module.

Upon retrieval of all the response messages, the response message module, with the help of the cache coherence protocol implemented in the protocol module, will know where the cache line data resides. The cache line data can be stored in the DRAM or in other caches in the system. To retrieve the cache line data from DRAM, the response message module will send a message to receive the cache line data to the Arbiter module connected to the Cache Data Port B Cntrl module. Once the message passes arbitration and gets to the Cache Data Port B Cntrl module, Cache Data Port B Cntrl module will propagate this message through the interface module of the cache and memory access sequencer components, known as the MAS network interface module. The Cache Data Port B Cntrl module also implements the ability for the cache to: receive cache line data from the cache network interface module and the MAS network interface separately, send the read and the write operations to the MAS network interface, send cache line data to the cache network interface module and the MAS network interface module concurrently.
as well send cache line data to the cache network interface module and the MAS network interface separately.

Finally when the cache line data is retrieved, the protocol module will update the cache line tag and the cache line state of the cache line entry.

The cache allows processing elements to flush/evict a cache line entry to the DRAM. The cache can handle two flush operations. The first flush operation evicts a single cache line entry from the cache. When flushing a single cache line entry, the Trxn Mgr module will ask the cache table for the state of the cache line entry while resetting the state of the cache line entry. The cache table will output the answer to the flush data module. With the help of the protocol module, the flush data module will verify if the cache line state is unique to the cache. If the cache line is unique to the cache, i.e. the cache line state is in the Modified state, the flush data module will send a message to the Arbiter module connected to the Cache Data Port B Cntrl module. Once the message passes arbitration, the Cache Data Port B Cntrl module will evict the cache line entry.

The second flush operation is flushing the entire cache. When flushing an entire cache, the same procedure is used when flushing a single cache line entry except the flush counter module asks the cache table for the state of the cache line entry. The flush counter iteratively goes through all the cache line entries of the cache in ascending order. When evicting a cache line entry, the flush counter has the ability to continue asking the cache table for the state of a cache line entry for which the value of the last three iterations are preserved; this is known as the lookahead mechanism. If the Cache Data Port B Cntrl is busy performing a task, once the flush data sends a message to the Cache Data Port B Cntrl module to evict a cache line entry, then the states of the last three valid iterations are rolled back.

To reduce the contention for the cache table, the cache table is implemented using dual-ported BRAMs [23], where up to two reads (of different addresses) or one write can be serviced simultaneously. Furthermore, the cache table is replicated as many times as
there are ports connected to the *cache interconnection network* from a single cache. Of the two ports of the *cache table*, port A is a read/write port and port B is the read-only port. The inputs of port A are identical for all replicated *cache tables* but the inputs of port B correspond to a port connected to the *cache interconnection network*.

In the *cache* implementation, the result from the *Hit* module takes one cycle, which is the time required to determine if a cache line entry is present in the cache. It is only after the result arrives that the cache can alter the contents of a cache line entry. Therefore write operations take two cycles, one for the generating the result from the *Hit* module and the other for the write operation. However, the block RAMs (BRAMs) [23] have a one cycle latency on the read and the write operations. To preserve the BRAM latency for the write operation, the write speculate mechanism was adopted from Woods. The write speculate mechanism will always perform the write operation which is one cycle. On the same cycle as the write operation, the cache will validate the write operation with the *Hit* module and output the result of the validation on the next cycle. If the result of the validation of the write operation was a cache hit, the cache will accept the write and the performance of the write operation is not affected. If the result of the validation of the write operation was a cache miss, the cache will not accept the write and undo the write that was performed thus affecting performance of a write operation.

A novelty of the *cache* design is the cache coherence *protocol* module is independent of all the other modules in the *cache*. The cache coherence *protocol* utilizes and controls other modules in the *cache*. This allows the ability to easily change the cache coherence protocol yet have the remainder of the cache architecture untouched. This modularized design facilitates the creation of a library of cache coherence protocols that are independent of the cache architecture. The library can eventually be used as building blocks for a cache coherent shared memory system for FPGAs using caches.
3.3.2 Cache Interconnection Network

Functionality

The caches send cache coherence messages to each other via the cache interconnection network. Each cache has up to seven ports connected to the cache interconnection network. Upon receipt of cache coherence messages from the caches, the scheduler from the cache interconnection network selects the cache coherence message(s) where the resources needed by the message(s) are available. If contention occurs, the scheduler uses its priority scheme to allow precedence over the cache coherence messages selected. The scheduler’s priority scheme will order the cache coherence messages at the cache level using the round-robin priority arbitration scheme ordering the caches in descending order of priority, and then at the ports level using a fixed priority arbitration scheme for ordering the ports of a single cache. If various broadcast messages are chosen referring to the same cache line entry, the collision detector using the same priority scheme as the scheduler allows only one of these messages to get issued. This avoids any race condition and ensures consistency throughout the system.

Since the cache interconnection network arbitrates the cache coherence messages in any order, cache coherence messages arrive at the caches in any order. Servicing out of order cache coherence messages may violate consistency, therefore an ordering mechanism is developed. Response messages do not need to be ordered since each cache is servicing at most one memory request from the processing elements. Therefore, all response messages received by the cache are related to the memory request being serviced by the cache. However, if the message is a broadcast message, the message is ordered with the ordering mechanism. The ordering scheme is maintained via a square grid where the size of the sides are equal to the number of caches in the system. The grid has the same structure as a set-associative cache [2] to allow same-cycle lookup and permit multiple entries to map addresses to the same grid entry slot. Each grid entry will store the cache
line address and the cache ID of the last cache requesting the cache line. This ordering scheme is more suited for this system in comparison to the partial ordering global counter mechanism [24] because the grid provides cache line granularity and strict ordering of the broadcast messages. If necessary the cache coherence message is ordered and once that is complete, the cache coherence message is issued to a port using an issuing policy.

The ordering mechanism requires the caches to locally maintain the order of the cache coherence messages they have serviced. Each cache contains a table of size equivalent to the number of caches in the system and indexed by the cache ID. The table of cache A will store at entry B the cache line address of the last broadcast message received from cache B serviced by cache A. The example in Figure 3.4 depicts the contents of the table of a cache before and after servicing a broadcast message. Figure 3.4a shows cache X receiving a broadcast message from cache W for cache line Y where the message requires that a previous message from cache Z on cache line Y be serviced first. For the message to be serviced, the table of cache X requires to have the value of cache line Y in the table entry of cache Z as seen in Figure 3.4a. Once the message has been serviced, the value in the table entry of cache Z is invalidated and the value of cache line Y is placed in the table entry of cache W (Figure 3.4b).

The following example will explain the various functions of the cache interconnection network in a systematic manner. Assume a three processing element (PE) system with two ports connected to the cache interconnection network. PE 0 and PE 1 sends a read request for memory address X to their respectful caches and these caches suffer from a cache miss. The caches send a broadcast message to the cache interconnection network at the same time. The scheduler will allow both broadcast messages to be scheduled since there are sufficient resources available to service these messages. The collision detector will only allow one of the two broadcast messages to be issued because both of these messages are accessing the same memory location. Assume the arbitration scheme in the collision detector selects the broadcast message sent by the cache connected to PE 0. The
Figure 3.4: The broadcast message from cache W viewed by cache X for cache line Y. The message requires that a previous message from cache Z on cache line Y be serviced prior to itself. This message is placed on the top of Figures 3.4a and 3.4b. a) values of the table prior to servicing the message; b) values of the table once the message was serviced.

ordering grid in the cache interconnection network will add the memory address of the message, in this case X, with the cache ID, in this case 0, to correct slot in the grid. The cache ID added to the slot denotes the last cache ID that accessed the memory location of the slot. On the next cycle, the broadcast message from cache 1 will be serviced and a tag will be added to the message denoting that this message can only be serviced if the receiving cache has previously serviced a message for memory address X by cache 0. The slot mapping to memory address X will be updated to indicate the last cache ID requesting an action on memory location X is cache ID 1. Once cache ID 1 has serviced its memory request from PE 1, cache ID 1 signals the cache interconnection network and the entry mapping to memory address X in the grid will be reset.

In summary, the cache interconnection network will select cache coherence message(s) that sufficient resources are available to service it/them. The messages will be ordered using a predefined arbitration scheme. The collision detector will allow only one broadcast
message relating to a cache line entry to be issued. If a broadcast messages is issued, it is ordered using the ordering mechanism. Finally, the messages are issued using an issuing policy.

Routing Configuration

There are two routing configurations for the cache interconnection network: the fully-connected crossbar configuration and the fully connected point-to-point configuration. In both configurations, the output ports from a cache are connected to the inputs of the cache interconnection network and the output ports of the cache interconnection network are connected to the input ports of the cache.

In the fully-connected crossbar configuration, each input port of the cache interconnection network is connected to the output ports of the cache interconnection network that correspond to the other caches in the system. An example of a single input port connecting to each output port of the other caches of the system is shown in Figure 3.5. The complete configuration has these connections for each input port. The I-CX-PY label represents Input signals from port Y of cache ID X to the cache interconnection network. Similarly, the O-CX-PY label represents Output signals to port Y of cache ID X from the cache interconnection network.

For the fully-connected crossbar configuration when the number of ports connected from the cache to the cache interconnection network is greater than one, the issuing policy used is the least-number-of-pending-messages. This issuing policy will issue the messages to a free output port with the least number of messages in its full-duplex communication links. In this configuration, a message sent from any port can be received by any port of another cache. This topology resembles that of a port-level crossbar topology and hereafter will be referred to as the fully-connected crossbar.

In the case where the number of ports connecting the cache to the cache interconnection network is equivalent to the number of caches in the system minus one, there can
Figure 3.5: The cache interconnection network with the least number of pending messages issuer. The I-CX-PY label represents Input signals from port Y of cache ID X to the cache interconnection network. The O-CX-PY label represents Output signals to port Y of cache ID X from the cache interconnection network.

be two issuing policies. The first is the previously described least-number-of-pending-messages policy. The other is a fixed-port policy and effectively the second configuration of the cache interconnection network, the fully-connected point-to-point configuration. The fixed-port policy is simply the direct connection of a single input port of a cache to a single output port of a corresponding cache. In this configuration, there are sufficient ports to have a single direct connection from one cache to another cache in the system allowing all caches to communicate to another cache via a single port. This topology is similar to a cache-level crossbar and hereafter will be referred to as the point-to-point topology. The fully-connected crossbar topology is beneficial over the point-to-point topology as it provides more bandwidth from one cache to other caches.

An example using four caches with the fixed-port policy issuer is depicted in Figure 3.6. The I-CX-CY label represents Input signals from the cache with ID X to the cache with ID Y. Similarly, the O-CX-CY label represents Output signals from the
cache with ID X to the cache with ID Y. In this topology, the caches will send the cache coherence messages to the designated port corresponding to the receiving cache. In both configurations found in Figure 3.5 and Figure 3.6, the cache interconnection network acts as a central control system for the cache coherence messages.

Figure 3.6: The cache interconnection network with the fixed issuer. The I-CX-CY label represents Input signals from the cache with ID X to the cache with ID Y. The O-CX-CY label represents Output signals from the cache with ID X to the cache with ID Y.

3.3.3 The Memory Access Sequencer

Race conditions are prevented with the ordering of the cache coherent messages by the cache interconnection network. However, these cache coherent messages can be translated to shared-memory space requests. Therefore, it is necessary to preserve the order of requests to the shared-memory space.

The Xilinx Multi-Ported Memory Controller (MPMC) [25] provides a convenient mechanism for multiple devices to access a single memory. It supports various scheduling policies for arbitrating access to the memory and a queuing mechanism for each access.
port, but it does not offer first-come-first-serve access. The memory access sequencer is multi-ported and provides first-come-first-serve access to a memory controller.

Figure 3.7 shows a scenario where using the current MPMC can create incoherent data amongst the caches. The problem arises when the shared memory is the medium of communication amongst the caches for cache coherence protocols such as the MESI Illinois [8]. The MPMC is busy processing a request by cache $Z$ (Figure 3.7a). During this time, cache $Y$ will send a broadcast message for memory address $M(X)$ to all caches via the cache interconnection network. The cache line entry corresponding to memory address $M(X)$ is ‘dirty’ in cache $X$, and thus incoherent with main memory. Therefore, cache $X$ will attempt to evict the cache line entry needed by cache $Y$ to main memory (Figure 3.7b). However, the memory is busy servicing cache $Z$. Cache $Y$ will follow with a read request to main memory for the cache line (Figure 3.7c). The policy of the MPMC scheduler is set to service cache $Y$ before cache $X$. Thus when the MPMC is done servicing cache $Z$, the MPMC will service the read request of cache $Y$ (Figure 3.7d). Cache $Y$ will contain incorrect data. The MPMC finally schedules the write request, satisfying the evict operation, of cache $X$ (Figure 3.7e). The main memory and cache $X$ become coherent. However cache $Y$ is not coherent, thus breaking the cache coherency requirement.

The memory access sequencer solves this problem by keeping the request occurrences ordered. This is done by queueing the requests in order of arrival, therefore reversing the order of the actions depicted in Figure 3.7d and Figure 3.7e to maintain a fully coherent system. The queue dimensions are the number of caches connected to the memory access sequencer by the total number of memory requests (transactions) performable by the caches connected to the memory access sequencer. Each entry in the queue has a bit representing the presence of a main memory request from a cache, if any, and the bit is indexed by their cache ID. For example in an eight-cache system with a cache with cache ID 2 performing a main memory request, the bit vector 00000100 will be placed in
Figure 3.7: a) The MPMC is busy with cache block $\$Z$; b) cache block $\$X$ request to write/evict a cache line with address $X$; c) cache block $\$Y$ request to read address $X$; d) The MPMC will send incorrect data to cache block $\$Y$; e) The main memory and cache block $\$X$ are coherent but not cache block $\$Y$ f) Legend
the queue. In the scenario where more than one cache performs a main memory request at the same time, the necessary bits will be set and the memory access sequencer will service each memory request in order of ascending cache ID.

The memory access sequencer also contains the ability to perform cache-to-cache cache line data transfer. There are two forms of cache-to-cache cache line data transfer that can be performed in the memory access sequencer. The first utilizes the routing in the memory access sequencer to transfer the cache data to another cache as is done by the cache interconnection network. The second extends the latter by allowing a write to main memory while performing the cache-to-cache cache line data transfer.

Figure 3.8 shows the format of a message packet between the cache and the memory access sequencer. The Msg field corresponds to the packet message type. The Address/Data field can contain the the address or data value to be written to the main memory. The To Cache ID corresponds to the cache ID of the recipient of a cache-to-cache cache line data transfer operation using the memory access sequencer.

![Figure 3.8: Cache To Memory Access Sequencer Message Packet Format](image)

### 3.3.4 Modified Memory Bus

The modified memory bus is the bus connecting the processing elements to a cache and must be able to request functionality from the cache. Therefore, the modified memory bus encapsulates the same signals described in the data sheet of the Xilinx Local Memory Bus (LMB) [26] with the addition of a flush signal. To retain the read and write operations,
the flush signal must be set low. The flush signal dictates a flush action to the cache. To flush a single cache line entry, the flush signal and the enable signal are set high with the write enable signal low. The value on the address bus corresponds to the cache line entry to flush. The other values of the modified memory bus are ignored. To flush an entire cache with the lookahead mechanism, the flush signal, the enable signal and the write enable signals are set high. The other values of the modified memory bus are ignored.

3.3.5 Cache To Cache Interconnection Network Bus

The communication medium between the cache and the cache interconnection network are the full-duplex communication links of Figure 3.1. A port of a cache is connected to the cache interconnection network using a full-duplex communication links. In the fully-connected crossbar, the additional ports of a cache connected to the cache interconnection network provides more bandwidth from one cache to another. This is beneficial during cache-to-cache cache line data transfer. If one cache is sending a cache line data of size 16 to another cache over the cache interconnection network with no interference, it will take 16 cycles in the fully-connected crossbar configuration with one port. However, in the fully-connected crossbar configuration with four ports, this will only require four cycles.

The full-duplex communication links are analogous to that of a first-word-fall-through FIFO. The interface of the FIFO is similar to the FSL [27] by Xilinx with the exception of the next signal. The next signal defines a new action, the next action. The next action removes the first element of the FIFO and places it at the end of the FIFO. This only occurs if the read or write signal are not asserted. During the next action, which takes two cycles, the exist signal is low and the full signal is high providing the necessary information to the user to prevent a read or a write operation on the full-duplex communication links. During a next action, a read and a write action will not be serviced. The data signals contain the information of the message format of Figure 3.2. The minimum depth of the FIFO needed to prevent a deadlock in the system is the
number of memory requests (transactions) permissible by the caches in the system.
Chapter 4

Methodology

This chapter will introduce the requirements necessary to model the CCIFF. Then, the chapter will describe the framework used to implement the CCIFF and the benefits acquired from using the framework. The chapter will end with a quick discussion on the manner the framework was used to ensure correctness of the CCIFF system during development.

4.1 The Simulation Framework

Before a final CCIFF design can be implemented in hardware, there are several design parameters for the CCIFF that needed to be addressed. Therefore, a simulation framework has been used to model the CCIFF. The simulator will aid in exploring the properties and the architecture of the CCIFF given a series of benchmarks.

A CCIFF testing platform will be compared with the testing platform of Woods because both platforms are cache-coherent system using Sequential Consistency designed for FPGAs. Since the performance results of Woods are derived from execution on real hardware, cycle accurate simulation is the easiest way to make a comparison of the CCIFF with Woods. Furthermore, cycle accuracy will also help with understanding what is possible in an FPGA implementation.
Another important factor in the development of the CCIFF is that only functional descriptions are necessary, which are much easier to write than Hardware Description Languages (HDLs).

SystemC is a software library written in the C++ [28] language and provides a simulation framework for transactional level modelling (TLM). TLM can be viewed as an abstraction level above register transfer level (RTL). RTL abstracts structure, logic and timing where TLM abstracts communication. In an RTL environment, as depicted in Figure 4.1a, the RTL modules communicate via the pins connecting the modules. For the communication between two RTL modules in an RTL simulation, an event on any pin connecting the two RTL modules must be triggered. This can cause a large number of events for the RTL simulation kernel to handle, creating a slow simulation.

In contrast, Figure 4.1b shows how the communication between two modules in a TLM simulation is done using a function call. These function calls typically encapsulate many pins found in a RTL environment and only trigger one event. A large performance benefit is present in TLM simulations, typically on the order of 100x to 10000x [29].

![Diagram](image-url)

Figure 4.1: a) Communication between two RTL modules; b) Communication between two TLM modules
Chapter 4. Methodology

The simulation framework has to describe functionality to model the components in the CCIFF. Functionality is best captured in a behavioural model simply because it is the best match in terms of simulation performance. RTL models can communicate at the TLM level but because the mismatch of the simulation technology is so extreme, this is very counter productive and RTL becomes the bottleneck.

TLM benefits from taking simple abstract behavioural models and integrates them together in a concurrent simulation environment. The communication between the behavioural models is done by embedding the behavioural models in a communication wrapper. The communication wrapper deals with the communication, concurrency and synchronization in the simulation environment. Therefore whenever a behavioural model needs to communicate to another behavioural model in the system, it will encapsulate the communication into a transaction (a.k.a. message) and adds timing information (Figure 4.2).

The typical use case for TLM is virtual platforms for a complex system on chip (SoC). These systems are created by integrating many digital and/or analog hardware IP blocks together. The challenge is to obtain the models for the various IP blocks for them to work in a fast and efficient simulation environment. For the CCIFF simulation framework, all components have to be designed and implemented. Therefore, a great deal of time is saved by implementing the CCIFF with TLM and not RTL.

Since SystemC is written in the C++ language, it inherits properties from the C++ language. The C++ language makes it easy to modify parameters with simple variable changes while modelling hardware at a higher level of abstraction than that provided by HDL. Furthermore high level languages, such as C++, are designed to capture functionality in a behavioural model.

SystemC provides a simulation framework with TLM functionality using its own syntax, a distinct set of macros and a simulation kernel. The SystemC [30] simulation kernel, much like other simulation kernels like Modelsim [31], can provide cycle-accurate tim-
Figure 4.2: TLM Simulation Environment Model
ing information. The clock cycle information is simply represented by an event in the simulation environment. Therefore SystemC, in collaboration with the C++ semantics, provides us with a software framework facilitating system-level modelling, design and verification.

In addition, the SystemC simulation kernel supports the concept of delta cycles. A delta cycle is conceptually an infinitely small time duration used to quantify time in simulation. These infinitely small time duration virtually provide the time needed to execute functions triggered by events.

### 4.2 Modelling Heterogeneous Processing Elements

Heterogeneous processing elements can be integrated in the CCIFF model. Figure 4.3 shows an example of a CCIFF system with heterogeneous processing elements. However, it is difficult to find behavioural models of the various processing elements available for FPGAs. Even if behavioural models are available, a SystemC wrapper would need to be implemented to interface the processing elements to the cache components in the CCIFF model. This task would need to be repeated for every different processing element that a system designer wishes to integrate in a CCIFF system, since processing elements have different interface specifications. Furthermore, accurately modelling the execution time of each heterogeneous processing element is not required and will only add extra overhead to the simulator, as the CCIFF model only requires a limited set of knowledge on the operations performed by these processing elements.

Hence, a *generic processing* module is created that reads a trace file of memory operations. The operations include the conventional memory read and write operations as well as the flush operations provided by the cache. The computation time of a processing element is represented by another operation that tells the *generic processing* module to delay an amount of time representative of the time required to do processing between
the memory operations. An example of a trace file is found in Figure 4.4.

READ 0x0005000  //Read
READ 0x0005004
PROCESSING 2  //The PE does computation for 2 cycles
WRITE 0x0005400  //Write
FLUSH 0x0005400  //Flush cache line corresponding to address 0x0005400
FLUSH  //Flush entire cache

4.3 Development of the CCIFF Model

To ensure correctness of the CCIFF model, assertions defining the specification of the system was implemented. These assertions are rules that the system must satisfy and were executed on every clock cycle. If any of the assertions fail, the simulation would halt with a description of the failed assertion. A list of assertions are found in Table A.1.
In addition to the implementation of assertions, micro-benchmarks were implemented to test the individual features and individual components of CCIFF.

To minimize the simulation time, it is necessary to have a minimum set of events triggered in the simulation environment. This was accomplished by separating the tasks of a module in accordance to which module it communicates to and then encapsulating the output of the task into a single transaction/message. For example, the `protocol` module has various tasks to satisfy. Amongst them is to verify if a cache line state is unique in the system. Since this task and no other task from the `protocol` module communicates to the `flush data` module, the output of this task is encapsulated in a single transaction/message and sent to the `flush data` module. Furthermore, the implementation of this task in the `protocol` module is found in a single function.

The software model of CCIFF has several debug features implemented. It provides a verbose debug mode for delta cycle information and can generate a waveform file for functional debugging. Since the waveform file size data can be quite large when retaining a long duration of data, a software mechanism was developed to periodically record the data for a short time duration. Furthermore to verify progress of a simulation run, a run-time script with this functionality is available as described in Appendix B. In addition, Appendix B contains the directory structure and the file description for the simulation environment.
Chapter 5

Testing Platform

This chapter will introduce a testing platform using the CCIFF. Then, the chapter will continue with a description of the benchmarks and their criteria. Finally, the chapter will end with a discussion on implementing the testing platform on a modern FPGA.

5.1 Testing Platform

The testing platform of the CCIFF will be compared with that of Woods. For a fair comparison, the system clock frequency of 100 MHz was assumed in the CCIFF testing platform. This keeps the system clock frequency constant in both testing platforms. The cache architecture differs in the two platforms, but the same cache parameters are used in both testing platforms. The caches are direct mapped [2] with a cache line data size of 16 words and a cache size of 16KB. The memory latencies of the off-chip DRAM in both testing platforms are identical. The testing platform being simulated contains eight processing elements, each with a cache as in Woods; and the platform was evaluated with the different implementations of the cache interconnection network described in Chapter 3.

A requirement is to generate large amounts of traffic between the caches in the system. Large amounts of traffic will stress and test the worse case scenario for the cache...
interconnection network. Cache coherence protocols of the snoopy protocol type generate more traffic than that of the directory protocol type. However, the large amounts of traffic generated by the snoopy protocol type generally requires a tightly-coupled network topology which does not scale well with large systems. Snoopy cache coherence protocols will be used in this thesis because of its heavy traffic generation and the work of this thesis is compared to the infrastructure of Woods in Section 6.2 which uses a snoopy protocol. However, future research will investigate other cache coherence protocols since the snoopy protocol type does not scale well with large systems.

The cache coherence protocols using the MESI states are sufficient to generate heavy amounts of traffic. Not only are these states the most common states used in cache coherence protocols today but do not hold any underlining properties to reduces the traffic in the cache interconnection network. The Owned (O) and Forward (F) states hold any underlining properties reducing the traffic of the network interconnecting the caches by having a designated responder for any requests for the given cache line entry.

To evaluate the interconnection network of the CCIFF, a total of 10 variants of the MEI, the write-once [32] and the MESI Illinois [8] cache protocols were used. There are four categories classifying the variants of the cache coherence protocols.

The first category is the no cache-to-cache cache line data transfer (NO_C2C). Cache coherence protocols in the NO_C2C category use the shared memory to transfer the data amongst the caches and use the cache interconnection network to transfer only cache coherent messages.

The second category is the cache-to-cache cache line data transfer using the cache interconnection network (C2C_CIN). Cache coherence protocols in the C2C_CIN category use the cache interconnection network to transfer the cache line data and the cache coherent messages.

The third category is the cache-to-cache cache line data transfer using the routing network in the memory access sequencer (C2C_MEM). Cache coherence protocols in
the C2C_MEM category use the cache interconnection network solely for cache coherent messages and use the routing network in the memory access sequencer to transfer the cache line data. This allows any cache-to-cache cache line data transfer done through the memory access sequencer to be done concurrently with writes to the shared memory.

The final category is the mixed cache-to-cache cache line data transfer (C2C_MIXED). Cache coherence protocols in the C2C_MIXED category use the cache interconnection network to transfer the cache line data and the cache coherent messages, but also uses the memory access sequencer to transfer the cache line data. In the C2C_MIXED category, cache line entries that needed to be written to shared memory and transferred to another cache use the memory access sequencer and simple cache line data transfers use the cache interconnection network.

Table 5.1 shows a summary of the categories and Table 5.2 categorizes the 10 variants of the cache coherence protocols.
<table>
<thead>
<tr>
<th>Category</th>
<th>Allows Cache-To-Cache Cache Line Data Transfer</th>
<th>Cache Interconnection Network</th>
<th>Memory Access Sequencer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mechanism Used For Cache-To-Cache Cache Line Data Transfer</td>
<td>Mechanism Used For Cache-To-Cache Cache Line Data Transfer</td>
</tr>
<tr>
<td>NO_C2C</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>C2C_CIN</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>C2C_MEM</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>C2C_MIXED</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
### Table 5.2: The Various Cache Coherence Protocols

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI Illinois Original</td>
<td>NO_C2C</td>
</tr>
<tr>
<td>Write Once Original</td>
<td>NO_C2C</td>
</tr>
<tr>
<td>MEI C2C_CIN</td>
<td>C2C_CIN</td>
</tr>
<tr>
<td>MESI C2C_CIN</td>
<td>C2C_CIN</td>
</tr>
<tr>
<td>Write Once C2C_CIN</td>
<td>C2C_CIN</td>
</tr>
<tr>
<td>MEI C2C_MEM</td>
<td>C2C_MEM</td>
</tr>
<tr>
<td>MESI C2C_MEM</td>
<td>C2C_MEM</td>
</tr>
<tr>
<td>Write Once C2C_MEM</td>
<td>C2C_MEM</td>
</tr>
<tr>
<td>MESI C2C_MIXED</td>
<td>C2C_MIXED</td>
</tr>
<tr>
<td>Write Once C2C_MIXED</td>
<td>C2C_MIXED</td>
</tr>
</tbody>
</table>

The round robin scheduling scheme used in the *cache interconnection network* has eight slots, the number of slots are equivalent to the number of caches in the testing platform. Each slot has a duration of 20 clock cycles. The value of 20 chosen for the slot duration allows 25% more time required to transfer 16 words from one cache to another through the *cache interconnection network*. This increases the probability of the complete contents of cache line data of a cache line entry to be transferred in a single slot.

#### 5.2 Benchmarks

There are two sets of benchmarks used in this thesis: all three benchmarks used in Woods [1] and a threaded version of the MI-Bench [33]. The benchmarks from Woods [1] are the Mandelbrot area estimation, the Jacobi heat transfer and matrix multiplication representing low, medium and high memory utilization respectively. The Mandelbrot
benchmark has no shared data amongst the caches and the Jacobi benchmark has a very minimal amount of shared data amongst the caches. The matrix multiplication benchmark shares a large amount of data amongst the caches.

The benchmarks of Woods were executed with their corresponding memory layout and input data as described in Woods. The input data from the Mandelbrot is organized linearly in main memory. The Jacobi benchmark has two matrices to represent the previous iteration and the current iteration. The elements of these matrices are interleaved with each other and the matrices are in row major format in main memory. This layout is seen in Figure 5.1c. For a matrix multiplication A*B=C, the data of the multiplicand A in main memory is organized in row major as seen in Figure 5.1a; and the data of multiplicand B as well as the result matrix C are organized in main memory in column major as seen in Figure 5.1b.

The Jacobi algorithm terminates when the input data arrives at a stable state. With the input data selected, it requires 4165 iterations to arrive at a stable state. For proper execution of the Jacobi benchmark, a module was added to the testing platform to control the processing elements. The module provides a barrier function at the end of each iteration to synchronize the processing elements.

The matrices for the Jacobi are stored consecutively in main memory; the same applies to the matrices of the matrix multiplication benchmarks. The work of the Mandelbrot and the Jacobi benchmarks are divided equally amongst all the processing elements. The partitioning of the matrices for the matrix multiplication benchmark is that of the distributed matrix multiplication algorithm detailed in Woods. The benchmark trace files were generated using a model of the processing elements used in Woods.

There were four benchmarks used from the MI-Bench suite, one from each category. The benchmarks are the basicmath, Dijkstra, stringsearch and sha corresponding to the automation and industry control, network, office and security categories respectively.

The basicmath benchmark calculates the square root of the integer values from 0 to
Chapter 5. Testing Platform

Figure 5.1: a) data elements is layout in row major order; b) data elements is layout in column major order; c) data elements is layout in row major order and interleaved 32K. The workload was divided equally amongst the eight processing elements. In this benchmark, there is no explicit sharing of data amongst the processing elements.

The Dijkstra benchmark calculates the shortest path from each node to the rest of the nodes in the system. The input size is a system with 16 nodes. In this benchmark, there is a large amount of shared data amongst the caches.

The stringsearch benchmark searches for a string using the Pratt-Boyer-Moore method [34]. The search region is divided equally amongst the processing elements. The search string is the only data shared amongst the processing elements.

The sha benchmark will calculate the Secure Hash Algorithm version 2 (SHA-2) [35] on a set of 16 files. The file sizes are all on the order of 16 KB. Each processing element in the testing platform will compute the SHA-2 for two files. There is no explicit sharing amongst the processing elements.

The input sizes of the selected MI-bench benchmarks is equivalent or greater than the cache size described in Section 5.1. These input sizes will attempt to use the entire cache size except for the Dijkstra benchmark with the input of a system with 16 nodes.
The selected benchmarks from the MI-Bench suite were compiled under the MIPS [36] architecture. The trace files are created by running actual benchmarks on the gxemul [37] simulator with the R3000 MIPS processor. The simulator has been modified to output the memory operations into a file, as well as an estimate of the processing time between the memory operations. The estimate is done by counting instructions and assuming one cycle per instruction. This file is further processed to gather the traces for each parallel thread and outputting a trace file for each thread that will run on the simulator. The trace file only contains the threaded kernels of the benchmark. Each processing element in the testing platform runs a trace file corresponding to a thread.

Since the gxemul simulator was not executed with thread support, there is no private memory space for the threads. Thus, the threads use the same memory space for the local variables of the kernel functions. This causes a problem when a processing element performs two consecutive actions on the same cache line entry and out-of-order cache coherent messages get derived for the cache line entry. To solve this problem, the ordering mechanism of the cache interconnection network was expanded to allow a barrier function on the memory space corresponding to the local variables of the kernel functions. This ensures correctness during the execution of the selected MI-bench benchmarks.

5.3 Feasibility of FPGA Implementation

This section shows that the testing platform can be implemented on an FPGA.

5.3.1 Routing

This subsection argues that the routing of the testing platform described is feasible within an FPGA. First, the argument will identify topologies in the testing platform and map them to topologies from Saldaña et al. [3] and Woods. Finally, the results from Saldaña et al. and Woods will be used to show the routing feasibility of the testing platform on
modern FPGAs.

The full-duplex communication links connecting the caches to the memory access sequencer consist of 70 bits, 35 in each direction. The topology interconnecting the eight cache blocks with the memory access sequencer maps to the star topology. The results with 16 nodes from Saldaña et al. are used to evaluate the routing of the star topology. The reason is that the total number of bits in the FSL with 16 nodes used by Saldaña et al. is greater than the total number of bits in the full-duplex communication links from the testing platform, which provides a fair comparison. The results from Saldaña et al. show that the routing is successful with a maximum frequency of 150 MHz, which is well above the estimated 100 MHz assumed in the testing platform.

The full-duplex communication links connecting the cache ports to the cache interconnection network are 98 bits, 49 in each direction. The testing platform with the cache interconnection network configured as the point-to-point topology maps to the fully-connected point-to-point topology from in Saldaña et al. Once again using the results from 16 nodes, Saldaña et al. shows the routing of the fully-connected topology is successful and has a maximum frequency of 128 MHz. Thus, the testing platform can be routed on a modern FPGA with a clock frequency of at least 100MHz.

The cache interconnection network configured as a fully-connected crossbar with one port can be mapped to a topology found in Woods. The topology found in Woods refers to the connection between the caches and the Central Hub. In this case, the Central Hub is comparable to the cache interconnection network and the 16 ports connected to the Central Hub are equivalent to at least the total number of bits connecting the cache interconnection network to a cache with a single port. Since Woods was able to place and route his platform on an FPGA, the testing platform with a single port per cache connected to the cache interconnection network can be routed on an FPGA.

The fully-connected crossbar requires more routing resources than the point-to-point topology. The additional routing resource will render the routing difficult on the FPGA.
5.3.2 Logic and Memory Resources

This subsection will show that modern FPGAs contain sufficient logic and memory resources for the testing platform. First, the components of the CCIFF will be mapped to components in Woods. Second, the resource utilization of the mapped components will quantify the resources needed by the testing platform.

The amount of LUT and register usage is small in both the CCIFF and Woods so the usage of these resources can be neglected. The limiting resource in the CCIFF is the BRAMs. For the fully-connected crossbar with one port, each cache uses a total of 3 BRAMs which is the same value as in Woods. However to satisfy the bandwidth needed by adding additional port(s), additional BRAMs are needed. The additional BRAMs will retain the cache line states and their tags. For eight caches with seven ports, the additional BRAMs needed is equivalent to 48 times the BRAMs storing the cache line states and the tags of a single cache in the system of Woods (six additional BRAMs for each eight caches). To satisfy the additional storage requirements, the additional storage can be implemented with the BRAMs or distributed memory. Luckily, with the results shown in Chapter 6 the additional bandwidth does not benefit the testing platform. Thus, the additional ports are not needed. The number of BRAMs required for the testing platform using one port is easily achievable.
Chapter 6

Results

In this chapter, the number of memory requests serviced by the caches for the MI-Bench benchmarks is shown to be uniformly distributed amongst all the caches of the testing platform. The chapter also compares the infrastructure of the CCIFF with that of Woods. The chapter continues by showing the benefit of the cache-to-cache cache line data transfer in the testing platform. The benefit of the cache-to-cache cache line data transfer is further shown by quantifying the reduced amount of accesses to the memory access sequencer. Then, the chapter analyses the different configurations for the cache interconnection network. Finally, the chapter shows the overhead when using cache coherence in the CCIFF testing platform.

In the following, the speedup of $A$ over $B$ is defined as $\frac{\text{execution time of } B}{\text{execution time of } A}$. Figures 6.2 to 6.11 have the speedup on the $y$ axis and the different Cache Interconnection Network configurations on the $x$ axis. On the $x$ axis, the values 1 through 7 refer to the number of ports connecting a cache to the Cache Interconnection Network when configured as the fully-connected crossbar. The ‘pt to pt’ label refers to the Cache Interconnection Network configured with the point-to-point topology.
6.1 Utilization Balance

The utilization of a cache is defined as the number of memory requests serviced by the cache. The utilization of the caches in the Jacobi, Mandelbrot and the Matrix Multiplication benchmarks are equivalent to that of Woods, which is required for the experiments in Section 6.2 to provide a fair comparison between the two infrastructures. However for the MI-bench benchmarks, the workload was only partitioned equivalently amongst the processing elements, but this does not necessarily imply the utilization of the caches will be equivalent. Since the caches run concurrently and provide the work to the memory access sequencer and the cache interconnection network, balancing the cache utilization in turn will stress the use of the memory access sequencer and the cache interconnection network for a worst case scenario. The remainder of this section will demonstrate the MI-bench benchmarks used in this thesis provide a uniform distribution of the utilization amongst the caches.

In Figure 6.1, the x-axis has the cache ID in the testing platform and the y-axis contains the number of memory accesses. The uniform utilization distribution amongst the caches is shown with a fairly straight horizontal line.

6.2 Infrastructure Comparison

This section will compare the CCIFF infrastructure with that of Woods. The Jacobi, Mandelbrot and Matrix Multiplication benchmarks where executed on the CCIFF testing platform with the Write-Once Original cache coherence protocol. The speedup of the CCIFF testing platform over Woods is shown in Figure 6.2.

All three benchmarks exhibit a speedup with the CCIFF testing platform over that of Woods. The average speedup is: 1.09 for the Jacobi benchmark, 1.03 for the Mandelbrot benchmark and 1.10 for Matrix Multiplication benchmark. The speedup is primarily a result of the separate networks for the transmission of memory requests and cache
coherence messages as well as the distributed state-filtering mechanism in the CCIFF. The CCIFF outperforms the infrastructure of Woods, presenting CCIFF as a better cache-coherent infrastructure for shared-memory systems on FPGAs.

On a cache miss in the Woods infrastructure, the cache will send a message to the Central Hub. The Central Hub will broadcast this message to the other caches. The caches will respond to the broadcast message. The Central Hub will respond to the requesting cache with the state result from its state-filtering mechanism, request the cache line data entry from DRAM and send it to the requesting cache. Until the Central Hub finishes servicing the cache miss, no other memory request can be serviced by any other cache during this time. This design of the Central Hub limits the Woods’ infrastructure to a total of a one memory request to be handled by all the caches in the system at a time.

In the CCIFF, multiple caches can transmit cache coherence messages at a time. This allows overlapping of messages sent to the memory access sequencer and transmission of
the cache coherence messages. This overlap is effectively the reason for the speedup depicted in Figure 6.2.

On another note, an interesting trend from Figure 6.2 is the fairly constant speedup of the benchmarks in all the cache interconnection network configurations. The benchmarks only use any additional bandwidth provided by the cache interconnection network to reduce the latency of the transmission of the cache coherence messages. Since the communication medium amongst the caches is the DRAM, the *memory access sequencer* becomes the bottleneck. The cache-to-cache cache line data transfer feature was implemented to attempt to reduce the *memory access sequencer* bottleneck.
6.3 Cache-To-Cache Cache Line Data Transfer Benefit

In this section the benefit of the cache-to-cache cache line data transfer will be evaluated. All the benchmarks except the Mandelbrot benchmark were executed on the CCIFF testing platform with all the variations of the MESI Illinois and Write Once cache coherence protocols. The Mandelbrot benchmark does not have any shared data amongst the processing elements. Therefore, its performance will remain unchanged when the benchmark is executed with cache coherence protocols with the cache-to-cache cache line data transfer feature implemented. The speedups of MESI Illinois C2C_CIN, MESI Illinois C2C_MEM and MESI Illinois C2C_MIXED over MESI Illinois Original as the baseline were computed. Similarly, the speedups of Write Once C2C_CIN, Write Once C2C_MEM and Write Once C2C_MIXED over Write Once Original as the baseline were also computed. In all cases the baselines do not use cache-to-cache cache line data transfer. Figures 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8 show the speedup using cache-to-cache cache line data transfer cache coherence protocols for the Jacobi, matrix multiplication, Dijkstra, Basicmath, SHA-2 and Stringsearch benchmarks respectively.

Figures 6.3 through 6.8 clearly shows that cache-to-cache cache line data transfer increases the performance of the CCIFF. Although the matrix multiplication has a large amount of shared data amongst the processing elements, the speedups shown in Figure 6.4 with the cache-to-cache cache line data transfer cache coherence protocols are not proportional to the amount of shared data amongst the caches. The lack of speedup is due to a limitation of the cache-to-cache cache line data transfer implementation in the cache component.

The limitation is described as follows: during a cache-to-cache cache line data transfer between two caches, the caches are unable to satisfy memory requests from the processing elements, which in turn stalls the processing elements progress. This limitation exist
Figure 6.3: The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Jacobi Benchmark

Figure 6.4: The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Matrix Multiplication Benchmark
Figure 6.5: The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Dijkstra Benchmark

Figure 6.6: The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Basicmath Benchmark
Figure 6.7: The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - SHA-2 Benchmark

Figure 6.8: The Speedup Using Cache-To-Cache Cache Line Data Transfer Cache Coherence Protocol - Stringsearch Benchmark
because Port A of the cache data of Figure 3.3 is inactive when the Cache Data Port B Cntrl is performing a task. Additional research would be required to allow memory requests to be serviced by processing elements during a cache-to-cache cache line data transfer between two caches.

On average the cache coherence protocols in the C2C_MEM and C2C_MIXED categories perform better than the cache coherence protocols in the C2C_CIN category, implying the cache interconnection network is unable to satisfy the routing requirements for all the cache coherence messages generated by the cache coherence protocols in the C2C_CIN category. However with the matrix multiplication and the SHA-2 benchmarks, the C2C_MEM category cache coherence protocols perform noticeably slower than the cache coherence protocols in C2C_MIXED category. This observation suggests the use of both the routing networks in the memory access sequencer and the cache interconnection network should be leveraged to implement the cache-to-cache cache line data transfer feature for these cache coherence protocols. Using two separate network distributes the routing resources inside the CCIFF model. This design fits well in architectures designed for FPGAs as the routing resources are generally distributed uniformly on FPGAs.

6.4 Reducing the Memory Access Sequencer Bottleneck

As stated in Section 6.2, the bottleneck of the CCIFF testing platform is the access to the memory access sequencer. The CCIFF allows cache-to-cache cache line data transfer from one cache to another using cache coherence messages. The CCIFF design also allows overlapping of the transmission of these cache coherence messages with memory access sequencer access. Therefore, the memory access sequencer utilization should be reduced with cache coherence protocols using the embedded cache-to-cache cache line data transfer feature as they inherit work done by the memory access sequencer. In a
testing platform where the cache coherence protocols use the embedded cache-to-cache cache line data transfer feature, the cache data storage is viewed as a shared-memory to all other caches in the system as in the Dynamic Distributed Cache technology of Tilera [12].

In the following Section, the reduced access of $A$ over $B$ is defined as $\frac{\text{number of access to memory access sequencer of } B}{\text{number of access to memory access sequencer of } A}$. An access to the memory access sequencer can be a read operation, a write operation or an operation using the routing in the memory access sequencer to transfer a cache line data from one cache to another.

This section shows that using the cache-to-cache cache line data transfer feature reduces the amount of access to the memory access sequencer. To prove this deduction, the reduced access with the same combination of the experiments executed in Section 6.3 are calculated. Larger values of the reduced access are preferable since they signify reduced access to the memory access sequencer in turn reducing the bottleneck at the memory access sequencer. The results are shown in Tables 6.1 through 6.6 for the Jacobi, matrix multiplication, Dijkstra, Basicmath, SHA-2 and Stringsearch benchmarks respectively. The first row of the tables contain the various cache interconnection network configurations with the same naming convention as the figures of this chapter.
### Table 6.1: Reduced Access Percentage For The Jacobi Benchmark

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>pt-to-pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi - MESI Illinois Original</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MESI Illinois C2C_CIN</td>
<td>1.97</td>
<td>1.95</td>
<td>1.96</td>
<td>1.95</td>
<td>1.95</td>
<td>1.95</td>
<td>1.95</td>
<td></td>
</tr>
<tr>
<td>MESI Illinois C2C_MEM</td>
<td>1.97</td>
<td>1.97</td>
<td>1.97</td>
<td>1.96</td>
<td>1.96</td>
<td>1.97</td>
<td>1.98</td>
<td></td>
</tr>
<tr>
<td>MESI Illinois C2C_MIXED</td>
<td>1.97</td>
<td>1.97</td>
<td>1.97</td>
<td>1.96</td>
<td>1.96</td>
<td>1.96</td>
<td>1.97</td>
<td></td>
</tr>
</tbody>
</table>

(a) Reduced Access Percentage For The Jacobi Benchmark With the MESI Original Cache Coherence Protocol

### Table 6.2: Reduced Access Percentage For The Matrix Multiplication (MM) Benchmark

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>pt-to-pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM* - MESI Illinois Original</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MESI Illinois C2C_CIN</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td></td>
</tr>
<tr>
<td>MESI Illinois C2C_MEM</td>
<td>1.23</td>
<td>1.23</td>
<td>1.23</td>
<td>1.23</td>
<td>1.23</td>
<td>1.23</td>
<td>1.23</td>
<td></td>
</tr>
<tr>
<td>MESI Illinois C2C_MIXED</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td>1.78</td>
<td></td>
</tr>
</tbody>
</table>

(a) Reduced Access Percentage For The Matrix Multiplication Benchmark With the MESI Original Cache Coherence Protocol

(b) Reduced Access Percentage For The Jacobi Benchmark With the Write Once Original Cache Coherence Protocol

(b) Reduced Access Percentage For The Matrix Multiplication Benchmark With the Write Once Original Cache Coherence Protocol
### Chapter 6. Results

#### Table 6.3: Reduced Access Percentage For The Dijkstra Benchmark

<table>
<thead>
<tr>
<th></th>
<th>Basicmath - MESI Original</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>pt-to-pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesi Illinois C2C_CIN</td>
<td>2.85</td>
<td>3.05</td>
<td>2.89</td>
<td>2.76</td>
<td>2.89</td>
<td>2.87</td>
<td>2.8</td>
<td>3.22</td>
<td></td>
</tr>
<tr>
<td>Mesi Illinois C2C_MEM</td>
<td>1.58</td>
<td>1.71</td>
<td>1.71</td>
<td>1.65</td>
<td>1.66</td>
<td>1.67</td>
<td>1.69</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mesi Illinois C2C_MIXED</td>
<td>2.68</td>
<td>2.88</td>
<td>2.69</td>
<td>2.58</td>
<td>2.69</td>
<td>2.67</td>
<td>2.67</td>
<td>3.09</td>
<td></td>
</tr>
</tbody>
</table>

### Table 6.4: Reduced Access Percentage For The Basicmath Benchmark

<table>
<thead>
<tr>
<th></th>
<th>Basicmath - Write Once Original</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>pt-to-pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesi Illinois C2C_CIN</td>
<td>1.84</td>
<td>2.02</td>
<td>1.96</td>
<td>1.91</td>
<td>1.92</td>
<td>1.97</td>
<td>1.93</td>
<td>1.77</td>
<td></td>
</tr>
<tr>
<td>Mesi Illinois C2C_MEM</td>
<td>1.36</td>
<td>1.46</td>
<td>1.38</td>
<td>1.33</td>
<td>1.34</td>
<td>1.34</td>
<td>1.35</td>
<td>1.22</td>
<td></td>
</tr>
<tr>
<td>Mesi Illinois C2C_MIXED</td>
<td>1.80</td>
<td>1.97</td>
<td>1.93</td>
<td>1.89</td>
<td>1.88</td>
<td>1.95</td>
<td>1.91</td>
<td>1.76</td>
<td></td>
</tr>
</tbody>
</table>

(a) Reduced Access Percentage For The Basicmath Benchmark With the Mesi Original Cache Coherence Protocol

(b) Reduced Access Percentage For The Basicmath Benchmark With the Write Once Original Cache Coherence Protocol

(a) Reduced Access Percentage For The Dijkstra Benchmark With the Mesi Original Cache Coherence Protocol

(b) Reduced Access Percentage For The Dijkstra Benchmark With the Write Once Original Cache Coherence Protocol
Table 6.5: Reduced Access Percentage For The SHA-2 Benchmark

<table>
<thead>
<tr>
<th>SHA-2 - MESI Illinois Original</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>pt-to-pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI Illinois C2C_CIN</td>
<td>1.62</td>
<td>1.62</td>
<td>1.64</td>
<td>1.64</td>
<td>1.65</td>
<td>1.65</td>
<td>1.65</td>
<td>1.65</td>
</tr>
<tr>
<td>MESI Illinois C2C_MEM</td>
<td>1.42</td>
<td>1.40</td>
<td>1.41</td>
<td>1.41</td>
<td>1.42</td>
<td>1.42</td>
<td>1.41</td>
<td>1.41</td>
</tr>
<tr>
<td>MESI Illinois C2C_MIXED</td>
<td>1.60</td>
<td>1.60</td>
<td>1.61</td>
<td>1.62</td>
<td>1.63</td>
<td>1.62</td>
<td>1.62</td>
<td>1.62</td>
</tr>
</tbody>
</table>

(a) Reduced Access Percentage For The SHA-2 Benchmark With the MESI Original Cache Coherence Protocol

Table 6.6: Reduced Access Percentage For The Stringsearch Benchmark

<table>
<thead>
<tr>
<th>Stringsearch - MESI Illinois Original</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>pt-to-pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI Illinois C2C_CIN</td>
<td>1.20</td>
<td>1.19</td>
<td>1.21</td>
<td>1.19</td>
<td>1.21</td>
<td>1.20</td>
<td>1.27</td>
<td></td>
</tr>
<tr>
<td>MESI Illinois C2C_MEM</td>
<td>1.18</td>
<td>1.18</td>
<td>1.19</td>
<td>1.17</td>
<td>1.20</td>
<td>1.19</td>
<td>1.20</td>
<td>1.23</td>
</tr>
<tr>
<td>MESI Illinois C2C_MIXED</td>
<td>1.20</td>
<td>1.20</td>
<td>1.21</td>
<td>1.21</td>
<td>1.20</td>
<td>1.21</td>
<td>1.21</td>
<td>1.26</td>
</tr>
</tbody>
</table>

(a) Reduced Access Percentage For The Stringsearch Benchmark With the MESI Original Cache Coherence Protocol

Table 6.5: Reduced Access Percentage For The SHA-2 Benchmark

<table>
<thead>
<tr>
<th>SHA-2 - Write Once Original</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>pt-to-pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Once C2C_CIN</td>
<td>1.39</td>
<td>1.4</td>
<td>1.41</td>
<td>1.42</td>
<td>1.42</td>
<td>1.42</td>
<td>1.42</td>
<td>1.4</td>
</tr>
<tr>
<td>Write Once C2C_MEM</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
</tr>
<tr>
<td>Write Once C2C_MIXED</td>
<td>1.39</td>
<td>1.4</td>
<td>1.41</td>
<td>1.41</td>
<td>1.41</td>
<td>1.41</td>
<td>1.42</td>
<td>1.39</td>
</tr>
</tbody>
</table>

(b) Reduced Access Percentage For The SHA-2 Benchmark With the Write Once Original Cache Coherence Protocol

Table 6.6: Reduced Access Percentage For The Stringsearch Benchmark

<table>
<thead>
<tr>
<th>Stringsearch - Write Once Original</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>pt-to-pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Once C2C_CIN</td>
<td>1.20</td>
<td>1.20</td>
<td>1.19</td>
<td>1.19</td>
<td>1.20</td>
<td>1.20</td>
<td>1.19</td>
<td>1.25</td>
</tr>
<tr>
<td>Write Once C2C_MEM</td>
<td>1.03</td>
<td>1.02</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
</tr>
<tr>
<td>Write Once C2C_MIXED</td>
<td>1.19</td>
<td>1.20</td>
<td>1.20</td>
<td>1.19</td>
<td>1.18</td>
<td>1.19</td>
<td>1.20</td>
<td>1.24</td>
</tr>
</tbody>
</table>

(b) Reduced Access Percentage For The Stringsearch Benchmark With the Write Once Original Cache Coherence Protocol
All the reduced access values in Tables 6.1 through 6.6 are greater or equal to 1. This shows a reduction in the access to the memory access sequencer when using cache coherence protocols with cache-to-cache cache line data transfer. The cache coherence protocols in the C2C_MEM category do not reduce the accesses to the memory access sequencer as significantly as the cache coherence protocols in the C2C_MIXED and C2C_CIN categories. The maximum difference between the reduced access of the cache coherence protocols in the C2C_MEM category and those from the C2C_MIXED and C2C_CIN categories is 0.87 with the Basicmath benchmark, shown in Table 6.4 with the cache interconnection network configured as the fully-connected crossbar with four ports using the MESI Illinois C2C_CIN cache coherence protocol and the cache interconnection network configured as the fully-connected crossbar with three ports using the MESI Illinois C2C_MEM cache coherence protocol. The lack of performance from the cache coherence protocols in the C2C_MEM category is because the routing in the memory access sequencer is used to transfer cache line data from cache line entries with the cache line state in the Exclusive state. The cache line entries in the Exclusive state do not need to be written to the DRAM. This simply adds more work for the memory access sequencer that can be absorbed by the cache interconnection network since this work does not use the memory access sequencer for its primarily use.

The cache coherence protocols from the C2C_MIXED and C2C_CIN categories do not use the memory access sequencer for simply routing cache line data from one cache to another. They use this functionality in conjunction with the conventional write operation. And because of this, the network traffic caused by cache-to-cache cache line data transfer is balance within the two networks. On average the reduced access for the cache coherence protocols in the C2C_MIXED and C2C_CIN categories are similar. These results further affirm the benefit of using two networks to implement cache-to-cache cache line data transfer on FPGAs, as well as the need for a memory access sequencer supporting cache-to-cache cache line data transfer coupled with the write operation for an FPGA.
Chapter 6. Results

6.5 Cache Interconnection Network Configuration Comparison

In this section, the various configurations of the cache interconnection network will be analysed. To complete this task, the testing platform was executed with the MEI C2C,CIN and the MEI C2C,MEM cache coherence protocols. The results from these protocols are the most interesting as they stress the cache interconnection network and the network in the memory access sequencer with shared cache line entries being ping-ponged amongst the caches. The results from the Mandelbrot benchmark are omitted once again, as it does not contain any shared data. The Basicmath benchmark is omitted because the benchmark frequently accesses the local variables of the kernel functions, which may trigger the barrier function implemented in the cache interconnection network. Triggering the barrier function implemented in the cache interconnection network causes all the caches to stall until the last cache accessing a cache line entry containing a local variable has finished servicing its memory request. Triggering the barrier function too frequently in turn causes the caches to stall frequently affecting the performance of the system. The speedups of the various cache interconnection network configuration over the cache interconnection network fully-connected crossbar configuration with one port are shown in Figures 6.9 and 6.10.

Figures 6.9 and 6.10 show that as the number of ports increases in the fully-connected configuration, the performance increases by a maximum of 0.09 as seen with the Dijkstra benchmark. The additional ports in the fully-connected crossbar configuration increases the bandwidth to each individual cache from each cache in the system. This is fairly intuitive.

The fully-connected crossbar configuration with seven ports is very costly in hardware and would not scale well in the realm of FPGAs where routing resources are limited. When comparing the point-to-point configuration to that of the fully-connected crossbar
Figure 6.9: Topology Comparison Using the MEI C2C_CIN cache coherence protocol

Figure 6.10: Topology Comparison Using the MEI C2C_MEM cache coherence protocol
configuration with seven ports, on average pt-to-pt configuration is 0.03 more than the fully-connected crossbar configuration with seven ports. This difference is very small and shows the two configuration to be equivalent. Therefore, the larger amount of routing resources needed to implement the fully-connected crossbar configuration with seven ports is not needed.

The abnormality in Figure 6.9 with the matrix multiplication benchmark is once again a result of the limitation of the cache-to-cache cache line data transfer implementation in the cache component. The matrix multiplication contains a large amount of shared data. The large amount of shared data with these protocols causes cache-to-cache cache line data transfer; and during the time the caches are performing a cache-to-cache cache line data transfer, the caches are unable to service memory requests from the processing elements.

With the data from Figures 6.9 and 6.10, the point-to-point configuration outperforms the fully-connected crossbar configurations. The maximum speedup achieved with the point-to-point configuration is just over 1.26 with the Dijkstra benchmark. Compared to the baseline, the point-to-point configuration requires additional FPGA routing and BRAM resources. Taking into consideration the amount of resources needed for the point-to-point configuration, the speedup of the point-to-point configuration relative to the baseline is not substantially large. The baseline configuration, the fully-connected crossbar configuration with one port, is concluded to be the better configuration for the CCIFF because of its minimal use of FPGA routing and BRAM resources needed for its implementation.

### 6.6 Cache Coherence Overhead in CCIFF

This section will measure the cache coherence overhead found in the CCIFF testing platform. To measure the cache coherence overhead, two experiments using the testing
platform were executed with the Mandelbrot benchmark. The Mandelbrot benchmark is used since there is no shared data amongst the caches, therefore cache coherence is not required for proper execution of this benchmark. The Mandelbrot was executed with a cache coherence protocol from Table 5.2. The selection of the cache coherence protocol does not matter since results from the execution of the Mandelbrot with a cache coherence protocol will be the same regardless. Nonetheless, the MESI Illinois Original cache coherence protocol was selected for this experiment. The Mandelbrot benchmark was then executed on the testing platform without a cache coherence protocol. To mimic a system without a cache coherence protocol in the CCIFF, the protocol module from the cache component was replaced with a special module. The special module did not transmit cache coherence messages on a cache miss but would directly access the DRAM for the cache line entry. The speedup of the Mandelbrot benchmark with a cache coherence protocol over that of the Mandelbrot benchmark without a cache coherence protocol was calculated for all cache interconnection network configurations and is depicted in Figure 6.11.

Figure 6.11 shows very low cache coherence overhead because of the benefit of having the overlap of accesses to the memory access sequencer and transmission of the cache coherence messages. The maximum overhead speedup is 1.0026 with the fully-connected crossbar configuration with one port, thus the overhead consists of 0.0026 from the execution time of Mandelbrot with a cache coherence protocol. The cache coherence overhead decreases as the number of ports increases with the fully-connected crossbar configuration. The minimal overhead is found with the the fully-connected crossbar configuration with seven ports and the point-to-point configuration with a speedup of 1.0016, which shows an overhead of 0.0016 from the execution time of Mandelbrot with a cache coherence protocol. Therefore, the CCIFF presents very little cache coherence overhead for a cache-coherent shared-memory system on FPGAs.
Figure 6.11: The Cache Coherence Overhead in the CCIFF
Chapter 7

Conclusion

In Section 7.1 a summary of the thesis is provided. In Section 7.2 concluding remarks are presented. Finally Section 7.3 presents future work and potential improvements to the CCIFF.

7.1 Summary

The interconnection network for multicore systems remains a hot topic and presents a whole new set of questions in the realm of FPGAs. The purpose of the thesis was to reduce the difficulty of programming heterogeneous multi-processors in shared-memory systems by presenting the CCIFF. The CCIFF is a cache-coherent system based on messages passed between the caches where the caches implement the first cache-to-cache cache line data transfer feature for FPGAs and a distributed coherence-state filtering mechanism not found in any other system. The modularized design of the cache component allows a cache structure independent of the cache coherence protocol and contains a flush mechanism with the ability to flush the entire cache with a lookahead feature.

Chapter 2 provides background material on shared-memory architectures, cache structures, cache coherence states and the cache coherence protocols used in this thesis. Chapter 3 details the cache coherence implementation of the CCIFF and further describes the
major components of the CCIFF and their functionality. Chapter 4 introduced the SystemC simulation framework as the simulation framework to model the CCIFF. The SystemC simulation framework allows an easily parametrizable simulator modelling the cache infrastructure and permits the exploration of the cache-coherent infrastructures for FPGAs. Chapter 5 describes the testing platform and benchmarks used in this thesis and justifies the management of the testing platform on modern FPGAs. Finally in Chapter 6, results showing: a uniform utilization distribution amongst the caches for the MI-Bench benchmarks, a comparison of the CCIFF with the infrastructure of Woods, the benefit of the cache-to-cache cache line data transfer and its impact on the number of accesses to the memory access sequencer, a comparison of the various cache interconnection network configurations and the cache coherence overhead in the CCIFF were presented.

7.2 Conclusion

In conclusion, the CCIFF was introduced as a new cache-coherent infrastructure for shared-memory systems on FPGAs that outperforms the infrastructure of Woods. The CCIFF possesses many new features not implemented in previous cache-coherent infrastructures for FPGAs. Amongst these features is the cache-to-cache cache line data transfer. The cache-to-cache cache line data transfer increases the performance of the CCIFF and reduces the number of accesses to the memory access sequencer. However, through experiments a design limitation of the the cache-to-cache cache line data transfer implementation was discovered. To remove the design limitation, modification to the cache component design is required to allow memory requests to be serviced by processing elements during a cache-to-cache cache line data transfer between two caches.

Experiments also show using two separate networks to perform the cache-to-cache cache line data transfer is highly beneficial. The two networks in the CCIFF providing routing for the cache-to-cache cache line data transfer feature are the memory access
sequencer and the cache interconnection network. This implies that a memory access sequencer with the ability to route cache line data from one cache to another is required in a shared-memory system for FPGAs. Furthermore, using two separate networks to perform the cache-to-cache cache line data transfer fits well with FPGA architectures since FPGA generally contain a uniform distribution of routing resources.

Cache coherence protocols implemented to meet the requirements of the C2C_MIXED category perform best in a CCIFF system. These cache coherence protocols reduce the amount of traffic in the two networks.

As for the cache interconnection network configuration, the fully-connected crossbar with one port is concluded to be the best fit configuration for FPGAs. This configuration provides reasonable performance in comparison to the other configurations and utilizes less routing and BRAM resources on the FPGA. Most importantly, the CCIFF does not exhibit a large overhead to provide cache coherence.

7.3 Future Work

A significant amount of research has been done with regards to caching and cache coherence. This thesis has contributed to this research area and in its journey presents several new interesting questions in this research area for FPGAs. This section will list these additional research topics that require further exploration.

First, it is necessary to modify the cache design to resolve the limitation of the cache-to-cache cache line data transfer. Resolving this limitation would allow memory request to be serviced by processing elements during a cache-to-cache cache line data transfer between two caches. Furthermore, exploring other cache organisations such as set-associativity in the CCIFF model may provide performance benefit. However, one should note the additional complexity and resource utilization that may not be suitable for FPGAs. Other cache coherence related properties can be surveyed in the CCIFF
model, some properties include: broadcast filtering and directory-protocols. These properties have the ability to reduce network traffic with the cost of additional complexity and resources.

This thesis presented a modularized design for the cache component. The thesis did not justify the management of the cache structure implementation on FPGAs. It would be worthy to determine the best internal cache structure suited for an FPGA.

The CCIFF supports a limited set of processing elements. Adding functionality to the CCIFF model would allow more universal processing elements to be supported by the CCIFF. To support even more heterogeneous processing elements, it would be worthy to add heterogeneous functionality to the cache structure. Adding a synchronization mechanism for handling synchronization variables (i.e. mutexes) and support multiple memory request for runahead processors and out-of-order processing elements would cover the majority of processing elements.

The CCIFF is currently modelled for a single system on a single FPGA. The Accelerating Computing Platform by Nallatech [38] constitutes of a system allowing FPGAs and CPUs to share the communication medium, the Front Side Bus (FSB), to access the shared-memory space. The CCIFF would need to implement off-chip communication to support multi-FPGA systems, and the CCIFF would require FSB support to unite cache coherence of the processing elements from the FPGA with the CPU.

Another interesting functionality to incorporate in the CCIFF model is thread support. This approach is well-suited for this type of research since caches are utilized to create more efficient shared-memory systems. These caches need to be coherent, thus requiring cache coherency communication. In a multi-processor shared-memory system, the processors may run multiple threads, which requires communication. Since both threads and cache need to communicate and reside in a shared-memory system, it would be efficacious to couple them in a single infrastructure. It is also worthy to note that all of these elements have not yet been explored together in FPGAs.
Finally, once a complete, stable and well researched infrastructure has been designed, the hardware implementation of the infrastructure would be required for practical use. Nonetheless, this is an initial step to create a standard library of IPs for shared memory models on FPGAs using caches.
Appendix A

Assertion List
Table A.1: A List Of Assertions

- There can be at most one memory request from a processing element to a cache
- There can be at most one broadcast message per cache in the system
- For a cache coherence protocol without the use of cache-to-cache cache line data transfer, the number of response messages in the system destined for a cache should not exceed the number of caches in the system
- For a cache coherence protocol with the use of cache-to-cache cache line data transfer, the number of response messages in the system destined for a cache should not exceed the number of caches in the system plus the number representing the cache line size
- For a cache coherence protocol without the use of cache-to-cache cache line data transfer, the number of response messages from a cache to a corresponding cache should not exceed one
- For a cache coherence protocol with the use of cache-to-cache cache line data transfer, the number of response messages from a cache to a corresponding cache should not exceed one plus the number representing the cache line size
- The memory access sequencer should not have more than one plus the number representing the cache line size message from a cache
- The order in which the caches satisfy a memory request must be in the order the ordering mechanism has placed the caches
- For each table local to the cache aiding with the ordering mechanism requires all the valid entries of a single table to be different
Appendix B

Directory Structure And File Description

All the following files and directories are relative to the root directory of the simulation project. The files and directory corresponding to testing and the assertions of the CCIFF model using the simulation environment are omitted in this description.

The run_sims.sh shell script when executed will sequentially run the simulation environments of all the benchmarks with all the cache coherence protocols and with all the configurations of the cache interconnection network. The run_sims.param.sh shell script when executed will sequentially run the all the benchmarks with all the configurations of the cache interconnection network with a specified cache coherence protocol. The cache coherence protocol is specified by the first parameter of the bash script. The shell script is executed by the following command on a linux kernel: 

```
./run_sims_param.sh $1
```

where $1 is a value from Table B.2.

The progress.sh shell script when executed displays the progress in percentage (%) of each processing elements during a simulation run. The shell script is executed by the following command on a linux kernel:

```
./progress.sh BENCHMARK CACHE_COHERENCE_PROTOCOL PORTS.
```
Table B.1 and B.2 and display the numeric values for the `BENCHMARK` and the `CACHE_COHERENCE_PROTOCOL` parameters respectively. Values 1-7 for the `PORTS` parameter correspond to the number of port connected by each cache to the cache interconnection network configured as the fully-connected crossbar, and the `PORTS` parameter with value 8 corresponds to the fully-connected configuration.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>0</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>1</td>
</tr>
<tr>
<td>MatrixMultiplication</td>
<td>2</td>
</tr>
<tr>
<td>Dijkstra where the input is a system with 16 nodes</td>
<td>3</td>
</tr>
<tr>
<td>basicmath</td>
<td>4</td>
</tr>
<tr>
<td>SHA-2</td>
<td>5</td>
</tr>
<tr>
<td>stringsearch</td>
<td>6</td>
</tr>
<tr>
<td>Dijkstra where the input is a system with 64 nodes</td>
<td>7</td>
</tr>
</tbody>
</table>

The `event.h` file in the `event` directory contains simulation events used by the components in the CCIFF model. The `memory.h` file in the `memory` directory represents the storage of a 128MB DRAM from Figure 3.1. The `sc_main.cpp` file in the `sim` directory describes the infrastructure of Figure 3.1.

The `memory_controller` directory contains the files modeling the memory access sequencer. The `MemoryControllerWeak.cpp` file contains the implementation of the memory access sequencer component and the `memory_controller_constants.h` file contains the constants used by the memory access sequencer component.

The `cache_protocol` directory contains the `cache_state_list.h`, listing all the sets of states, and the `cache_protocol_list`, listing all possible cache coherence protocols. The
Table B.2: Numeric Value Corresponding to A Cache Coherence Protocol

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI Original</td>
<td>0</td>
</tr>
<tr>
<td>Write Once Original</td>
<td>1</td>
</tr>
<tr>
<td>MEI C2C_CIN</td>
<td>2</td>
</tr>
<tr>
<td>MESI C2C_CIN</td>
<td>3</td>
</tr>
<tr>
<td>Write Once C2C_CIN</td>
<td>4</td>
</tr>
<tr>
<td>MEI C2C_MEM</td>
<td>5</td>
</tr>
<tr>
<td>MESI C2C_MEM</td>
<td>6</td>
</tr>
<tr>
<td>Write Once C2C_MEM</td>
<td>7</td>
</tr>
<tr>
<td>MESI C2C_MIXED</td>
<td>8</td>
</tr>
<tr>
<td>Write Once C2C_MIXED</td>
<td>9</td>
</tr>
</tbody>
</table>

cache_protocol.h header file also in the cache_protocol directory holds common parameters used by all cache coherence protocols. The cache_protocol directory also contains two subdirectories: state and protocol. The state subdirectory contains the MESI_STATE.h file defining the MESI states of Section 2.4.1. The protocol subdirectory contains the PROTOCOL_BYPASS.h file representing no cache coherence protocol, this file was used to generate the results of Section 6.6. Table B.3 relates the remainder of the files in the protocol subdirectory to the cache coherence protocols from Table 5.2.

The app_engine directory contains the files of the generic processing module. The app_engine.cpp file contains the implementation of the generic processing module and the app_engine_constants.h file contains the constants used by the generic processing module.

The cache directory contains the files of the cache component. The common_cache_attribute.h file contains parameters defining a cache such as the cache ID. The trxn.h file defines
Table B.3: Filename Relating To Cache Coherence Protocols of Table 5.2

<table>
<thead>
<tr>
<th>Filename</th>
<th>Protocol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESI_Illinois.h</td>
<td>MESI Original</td>
</tr>
<tr>
<td>MESI_WRITE_ONCE.h</td>
<td>Write Once Original</td>
</tr>
<tr>
<td>MESI_Illinois_NO_S_FORWARD.h</td>
<td>MEI C2C_CIN</td>
</tr>
<tr>
<td>MESI_Illinois_ME_FORWARD.h</td>
<td>MESI C2C_CIN</td>
</tr>
<tr>
<td>MESI_WRITE_ONCE_FORWARD.h</td>
<td>Write Once C2C_CIN</td>
</tr>
<tr>
<td>MESI_Illinois_NO_S_FORWARD_READ_SKIP.h</td>
<td>MEI C2C_MEM</td>
</tr>
<tr>
<td>MESI_Illinois_ME_FORWARD_READ_SKIP.h</td>
<td>MESI C2C_MEM</td>
</tr>
<tr>
<td>MESI_WRITE_ONCE_FORWARD_READ_SKIP.h</td>
<td>Write Once C2C_MEM</td>
</tr>
<tr>
<td>MESI_Illinois_ME_FORWARD_READ_SKIP_E_CIN.h</td>
<td>MEI C2C_MIXED</td>
</tr>
<tr>
<td>MESI_WRITE_ONCE_FORWARD_READ_SKIP_E_CIN.h</td>
<td>Write Once C2C_MIXED</td>
</tr>
</tbody>
</table>

A class file corresponding to a memory request. The cache_constants.h file contains the constants used by the cache component. The cache.cpp file describes the cache architecture of Figure 3.3. Table B.4 relates the remainder of the files in the cache directory to the modules building the cache component.
Table B.4: Filename Relating To Module For A Cache Component

<table>
<thead>
<tr>
<th>Filename</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_table.h</td>
<td>The <em>cache table</em></td>
</tr>
<tr>
<td>table_arbiter.h</td>
<td>The Arbiter module connecting to portA of the <em>cache table</em> of Figure 3.3</td>
</tr>
<tr>
<td>cache_netif.h</td>
<td>The <em>cache network interface</em> module</td>
</tr>
<tr>
<td>c2c_engine.h</td>
<td>The <em>response message</em> module</td>
</tr>
<tr>
<td>cache_data_arbiter.h</td>
<td>The arbiter connecting to the <em>Cache Data Port B Cntrl</em> module of Figure 3.3</td>
</tr>
<tr>
<td>cache_hit.h</td>
<td>The <em>hit</em> module</td>
</tr>
<tr>
<td>cache_data.h</td>
<td>The <em>cache data</em> module and <em>Cache Data Port B Cntrl</em> module</td>
</tr>
<tr>
<td>trxnmgr.h</td>
<td>The <em>trxnmgr</em> module</td>
</tr>
</tbody>
</table>
The common\_constants directory contains the common\_constants.h file. The common\_constants.h file holds parameters needed by more than one component in the testing platform. The FSL\_fifo directory contains the files modeling the synchronous version of the FSL from Xilinx and its implementation is found in the FSL\_fifo.h file. The FSL\_fifo\_reorder directory contains the files modeling the full-duplex communication links component and its implementation is found in the FSL\_fifo\_special.h file.

The fwft\_fifo directory contains the files defining the port, interface and structure of a first-word-fall-through FIFO used by the FSL\_fifo and FSL\_fifo\_reorder components. The sc\_fwft\_fifo\_ports.h file contains the port definition, sc\_fwft\_fifo\_ifs.h file contains the interface definition, the sc\_fwft\_fifo.h defines the structure and the fwft\_fifo.h is the header file linking the port, interface and structure definitions.

The service\_flip\_flop directory contains the files for the implementation of a J-K flip flop [39]. The J-K flip flop was used for testing the CCIFF model. The service\_flip\_flop.h file contains the implementation of the J-K flip flop and the service\_flip\_flop\_constants.h file contains the constants used by the J-K flip flop module.

The network\_logic directory contains the files for the cache interconnection network component. The NetworkLogic.cpp file contains the implementation of the cache interconnection network component and the network\_logic\_constants.h file contains the constants used by the cache interconnection network component.

The msg directory contains the messages passed among the components of the CCIFF model. The cache\_mem\_msg.h file in the cache\_mem\_msg subdirectory contains the message class file for the message of Figure 3.8. The cache\_cache\_msg.h file in the cache\_cache\_msg subdirectory contains the message class file for the message of Figure 3.2. The app\_cache\_msg.h file in the app\_cache\_msg subdirectory contains the message class file for the messages between the generic processing module and the cache component.

The cache\_table\_protocol\_msg.h file in the cache\_table\_protocol\_msg subdirectory contains the message class file for the messages between the protocol module and the cache ta-
ble module, passing via its respectful arbiter of Figure 3.3. The cache_data_protocol_msg.h file in the cache_data_protocol_msg subdirectory contains the message class file for the messages between the protocol module and the Cache Data Port B Cntrl module, passing via its respectful arbiter of Figure 3.3.

The log directory contains a log of the progress done by a simulation run. The format of the directory is as follows:

log/BENCHMARK/CACHE_COHERENCE_PROTOCOL/PORT. Where BENCHMARK is a subdirectory string of Table B.5, CACHE_COHERENCE_PROTOCOL is the value representing a cache coherence protocol of Table B.2 and PORT is the value representing the configuration of the cache interconnection network. The values and descriptions of the PORT parameter are identical to those used for the PORT parameter for the progress.sh shell script.

The testbench directory contains the benchmarks detailed in Section 5.2. The subdirectories of the testbench directory classify the benchmarks of Section 5.2. Table B.5 relates the subdirectory of the testbench directory to the benchmarks. The threadX.txt files in subdirectories of testbench, where X is a value from 1 to 8, correspond to the trace files of the eight processing elements of the testing platforms.
## Appendix B. Directory Structure and File Description

### Table B.5: Testbench Subdirectory Description

<table>
<thead>
<tr>
<th>Subdirectory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>The Jacobi Benchmark</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>The Mandelbrot Benchmark</td>
</tr>
<tr>
<td>MatrixMultiplication</td>
<td>The Matrix Multiplication Benchmark</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>The Dijkstra Benchmark where the input is a system with 16 nodes</td>
</tr>
<tr>
<td>basicmath</td>
<td>The Dijkstra Benchmark with input</td>
</tr>
<tr>
<td>SHA-2</td>
<td>The SHA-2 Benchmark</td>
</tr>
<tr>
<td>stringsearch</td>
<td>The stringsearch Benchmark</td>
</tr>
<tr>
<td>Dijkstra64</td>
<td>The Dijkstra Benchmark where the input is a system with 64 nodes</td>
</tr>
</tbody>
</table>
Bibliography


[13] Intel Inc. The SCC Platform Overview - Revision 0.7.


[18] Xilinx Inc. Processor Local Bus (PLB) v4.6 (v1.04a), 2009.


[23] Xilinx Inc. Block RAM (BRAM) Block (v1.00a), 2009.


[26] Xilinx Inc. Local Memory Bus(LMB) V10 (v1.00a), 2009.
[27] Xilinx Inc. LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c), 2009.


