Design and Implementation of a Directory based Cache Coherence Protocol

Dimitris Tsaliagos

Thesis submitted in partial fulfillment of the requirements for the
Master of Science degree in Computer Science at the:

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossos Av. Heraklion, GR-71409, Greece

Thesis Supervisor: Prof. *Manolis G.H Katevenis*

Work performed at the:

Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
Computer Architecture and VLSI Systems (CARV) Laboratory
100 N. Plastira Av. Vassilika Vouton, Heraklion, GR-70013, Greece

April 2011

University of Crete
Computer Science Department

Design and Implementation of a Directory based Cache Coherence Protocol

Thesis submitted by

Dimitris Tsaliagos

in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Dimitris Tsaliagos

Committee approvals: ______________________
Manolis G.H. Katevenis
Professor, Thesis Supervisor

____
Dionisios N. Pnevmatikatos
Professor

____
Dimitrios S. Nikolopoulos
Associate Professor

Departmental approval: ______________________
Angelos Bilas
Associate Professor, Director of Graduate Studies

Heraklion, April 2010

Abstract

As the number of processors per chip increases, so does the need for efficient and high-speed communication support. This is necessary so that applications can exploit the numerous cores available in contemporary chip multiprocessors. Although explicit communication mechanisms such as RDMA can be used, implicit replication of data among the cores significantly simplifies the programming effort in large scale systems, by providing a simple and intuitive programming model.

This approach, however, introduces a problem known as cache coherence, where multiple copies of the data need to be kept consistent. An orthogonal solution to implicit migration of data, is to use directory based coherence protocols, which offer increased scalability by reducing the volume of messages exchanged as opposed to broadcast protocols. In this thesis a directory based cache coherence protocol is implemented in a four-core FPGA based prototype that was developed at the CARV (Computer Architecture and VLSI Systems) laboratory of FORTH (Foundation for Research and Technology – Hellas).

The protocol that is implemented can support up to 16 processors and extented an existing system, which provides local memory for cache and scratchpad use, RDMA and special hardware support for synchronization support [1].

Finally, our main finding is that the logic overhead for coherence, without accounting for directory memory, as opposed to a non-coherent is 4%. Preliminary evaluations of our protocol uses custom software micro-benchmarks, which emulate common synchronization operations (found in parallel applications), such as locks and barriers. Also matrix multiplication and producer-consumer test application were developed for evaluating the protocol. Results show that matrix multiplication scales on our coherence implementation achieving a speedup of 3.74 on 4 cores.

Contents

1 Introduction **1**

1.1 Cache Coherence Basics 2

1.1.1 The Coherence Problem 2

1.2 Implementation Schemes 4

1.3 Thesis Contributions 6

2 Background **9**

2.1 Directory-Based Coherence Protocols 9

2.2 Consistency Models 12

2.3 Directory Organizations 13

2.3.1 Flat Schemes 15

2.3.2 Hierarchical Schemes 15

2.3.3 Reducing Directory Memory Overhead 16

3 Design and Implementation **19**

3.1 Baseline System 20

3.2 Design 21

3.2.1 Hash Directory Organization 21

3.2.2 Protocol Design 24

3.2.3 L2 Cache Architecture 30

3.2.4 L2 Cache Controller Modifications 33

3.2.5 Directory Controller Design 38

3.3 Implementation 40

3.3.1 Protocol Packet Format 40

3.3.2 Directory Controller NoC Input Module 41

3.3.3 Directory Controller Hash Lookup Module 43

3.3.4 Directory Controller Action Lookup Module 48

3.3.5 Directory Controller NoC Output Module 50

4 Evaluation **53**
4.1 Target FPGA 53
4.2 Hardware Cost 53
4.2.1 Directory Controller Resources 54
4.2.2 L2 Cache Hardware Resources 56
4.3 Performance 57
4.3.1 Protocol Performance Metrics 57
4.3.2 Micro-benchmarks 59

5 Conclusions **67**
5.1 Future Work 68

List of Figures

1.1 Cache Coherence problem 3
1.2 Simple Directory Design 5

2.1 Basic MSI Protocol Cache FSM 10
2.2 Basic MSI Protocol Directory FSM 12
2.3 Directory Schemes 14
2.4 Sparse Directory Associativity Demands 18

3.1 System Block Diagram 20
3.2 Hash Directory Organization 23
3.3 Simple Protocol Transactions Example 26
3.4 Transient States and Protocol Races 28
3.5 L2 Cache Block Diagram 31
3.6 Atomic-Fetch-and Φ operation FSM 38
3.7 Directory Controller Block Diagram 39
3.8 Coherence Packet Format 41
3.9 Directory Network Input Module 42
3.10 Directory Hash Lookup 44
3.11 Directory Entry 45
3.12 SRAM Memory Clock Generation 46
3.13 NoC Output Datapath 51

4.1 Directory LUT's and FF's utilization 55
4.2 LUT's Comparison of Coherent and Baseline Design 56
4.3 Read Miss Timing Diagram 60

List of Tables

3.1 Protocol Messages Categorization 25
3.2 Coherence VC / Direction 30
3.3 L2C (Stable) States transition Table 34
3.4 L2C (Transient) States transition Table 35
3.5 Directory Protocol States 49

4.1 Virtex-5 LX110T 53
4.2 System Resource Utilization 54
4.3 Directory Hardware Resources breakdown 55
4.4 L2 Cache controller LUT's distribution 57
4.5 Coherence Transactions Latency 58
4.6 Shared Counter Slowdown 62
4.7 Atomic Fetch and Add Latency 62
4.8 Synchronization Primitives Latencies 63
4.9 Producer Consumer Latency 64
4.10 Matrix Multiplication Speedup 64

Acknowledgments

The work reported in this thesis has been conducted at the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology - Hellas (FORTH), and has been financially supported by a FORTH-ICS scholarship, including funding by the European Commission.

First of all I would like to thank my advisor Prof. Manolis G.H Katevenis as also Prof. Dionisis N. Pnevmatikatos and Prof. Dimitris S. Nikolopoulos, for their guidance and their support throughout this work. Their constructive remarks and the time they devoted to me constitute a significant amount of help. Furthermore, I would like to thank all my fellow students and/or co-workers at FORTH for their help and their support in all the good and bad times. Working in the same environment with you my friends (Manolis Marazakis, Mixalis Ligerakis, Spyros Lyberis, Stamatis Kavadias, Vasilis Papaefstathiou, Michalis Alvanos, Giannis Klonatos, and many othesr....) has been an honor and very pleasant. Also, I would like to thank especially George Nikiforos and George Kalokairinos for guiding and helping me throughout this work as the initial design of the system was designed by them. Finally, I would like to thank my family (Mixalis, Zoe, and Marialena) for their love and support they have offered me all these years. They have sacrificed everything in order to help me reach my goals. Without their help I would certainly have not made it to here. Last but not least, I would like to thank my close friends for encouraging me and supporting me all these years: Thanos Makatos, Zoe Sebepou, Artemis Papakonstantinou, Alexandros Kapravelos, Vicky Papavisileiou, Evi Dagalaki, Evi Galanou and Antigoni Konstantinou.

to my nephew Michalis

Chapter 1

Introduction

Having a large number of cores on a single chip, certainly alters the architectural decisions that have been considered efficient in traditional multiprocessors until now. Chip Multiprocessor (CMP) architectures are very suitable for throughput computing, where several independent programs run in different processing cores. Nevertheless, the need to make all the processing cores cooperate efficiently for a single computation, is of major importance for the scalability of these architectures. A primary component of these architectures, on which a great amount of attention is focused, is the communication architecture among the multiple processors, as well as between processors and off-chip main memory.

Cache coherence protocols are a key component, which provides a way for the programmer to write parallel applications that employ conventional ld/st instructions to shared addresses and allows implicit replication and migration of data among the caches of the processors comprising the system, to improve performance.

With an increasing number of cores the most suitable way of providing cache coherence is to implement a directory-based protocol, alternative protocols are utilize broadcasts to other caches in the system and thus are not as scalable as directory based protocols. Bus-based broadcast protocols, usually called snooping protocols as also directory broadcast protocols are not very power-efficient due to the large amount of messages they generate. To avoid broadcasts, directory based protocols require a system wide auxiliary structure to hold the state of the blocks cached or the corresponding state of a block in main memory. A significant part of the scalability of such architectures depends on the area and increased complexity that the cache coherence protocol adds, which are mostly accounted to the directory memory controllers as it will be discussed in Chapter 4.

There are two central aspects of directory implementation; the directory

organization and the set of messages types and message actions of the protocol that is discussed in Chapter 3. The former provides the basic properties of the abstract data structures used to store the directory information, which determine the amount of state required to store the sharing information. Furthermore, it may affect the latency of directory accesses, since some directory organizations require more complex logic to implement than others as it is will be discussed in Chapter 2. Message types and actions are dictated by the specific coherence protocol, and must be carefully designed to account for potential interactions with attributes of the chosen directory organization, as it will be described in Section 3.2.1.

In this thesis we design and implement a directory based cache coherence protocol, focusing on the directory state organization. An MSI cache coherence protocol is used to maintain the coherence property among L2 private caches in a prototype board that implements the SARC architecture [1]. A single node of the prototyped system is a single Xilinx XUPV5 board that consists of four microblaze soft-core processors, each with a private cache hierarchy, on a single Field Programmable Gate Array (FPGA) chip. The prototypes supports multi-board configurations, utilizing 3 SATA connections. The prototype architecture will be detailed in section 3.1. Memory resource limitation, in the baseline FPGA design, prevent an on-chip directory implementation. Thus, our design employs Off-chip Static Random Access Memory (SRAM), for directory memory, while the directory controller is implemented on the FPGA chip.

A primary goal of the coherence design in this work, is to reduce the average latency of directory access exploiting an address-based hashing to retrieve coherence information of cached blocks, that are kept in off-chip. SRAM.The directory organization we propose can support coherence up to 16 processors, which is supported in the baseline architecture by connecting multiple FPGA boards.

1.1 Cache Coherence Basics

This section presents some background information about the cache coherence property as well as the two basic protocol implementation schemes that enforce the cache coherence property.

1.1.1 The Coherence Problem

The use of private caches in a shared memory multiprocessor environment, introduces an inherent cache coherence problem. If more than one processors maintain locally cached copies of a memory data block , any modification to the

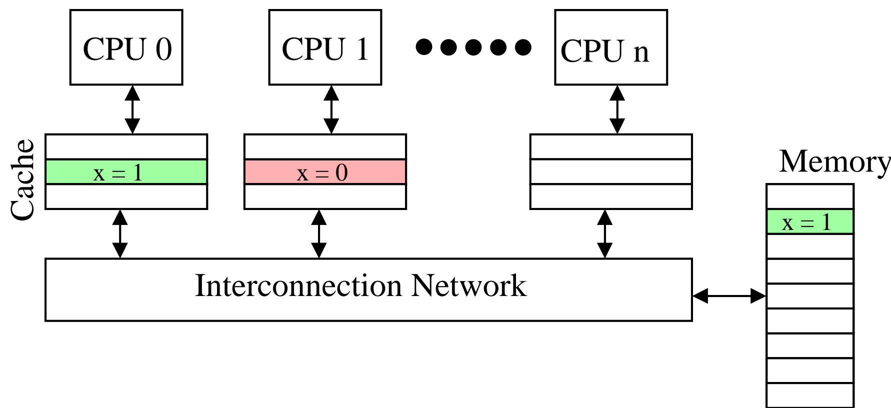


Figure 1.1: Cache Coherence problem

block's data is likely to lead to data incoherence. .

The cache coherence problem is illustrated in Figure 1.1. Assume that, initially memory contains the value 0 at the memory location for variable x, and that both CPU0 and CPU1 read x and cache it locally. So if subsequently CPU0 writes value 1 in x, data in CPU1 cache will become stale, since reading x from CPU1 will return the old value of x by accessing the copy locally cached.

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This definition, although intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs. The first aspect, called coherence, defines what values can be returned by a read. The second aspect, called consistency, determines when a written value will be returned by a read which is discussed in Chapter 2. Let's look at the strict definition of coherence first [7].

A memory system is coherent if

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

3. Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors

can never read the value of the location as 2 and then later read it as 1.

The first property simply preserves program order—we expect this property to be true even in uni-processors. The second property defines the notion of what it means to have a coherent view of memory: If a processor could continuously read an old data value, we would clearly say that memory was incoherent. The need for write serialization is more subtle, but equally important.

If appropriate coherence actions were invoked whenever a cache line was written, then the load issued from CPU1 would return the correct value either by invalidating all the cached copies of location x or by updating the values of all the cached copies of x with the latest value. Although some hybrid techniques have been proposed, most modern cache-coherent multiprocessors use the invalidation technique rather than the update technique since it is easier and scalable in terms of network traffic than update protocols.

1.2 Implementation Schemes

The previous section describes the cache coherence problem and introduces the coherence protocols as the agents that solves the coherence problem. There are two main implementation schemes of cache coherence protocols, bus-based protocols (snoopy) and directory-based protocols. In this section, we present the two dominant hardware schemes that are used to enforce the cache coherence property.

Bus Based Protocols (Snoopping)

Shared memory systems that are based on a shared broadcast medium follow the Snooping approach, no parts of memory are assigned to any processor. Assuming a single level of private caches, a processor that requests to access a memory block, which does not resides in its local cache, it sends a message to all the other caches and main memory. All the caches snoop the traffic on the interconnection network to identify a new message. If no cache has a copy of the requested block then the block is loaded from main memory. If, however, one or more caches maintain a valid copy, one of them sends the requested block back to the cache that requested it. Messages are used not only to facilitate data transferring. Every message is assigned a type, which has a specific meaning for the coherency protocol. Based on this type caches that receive such messages are becoming aware of the intention of the requesting processor.

Having this knowledge, caches are able to follow the steps imposed by the coherency protocol. This category of coherency protocols add a requirement to the interconnection network properties, which constitutes the basic property of

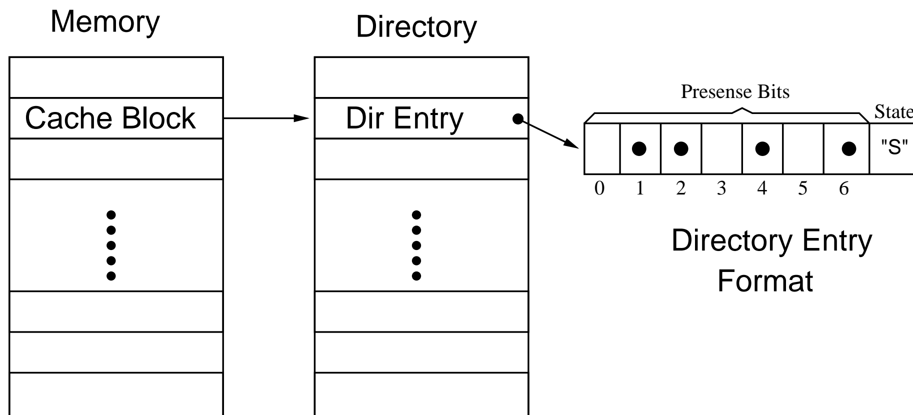


Figure 1.2: Simple Directory Design

the protocol. This requirement refers to the ability that must be offered to any cache to broadcast messages and also to snoop the bus activity. Otherwise, it is impossible for the distributed protocol to synchronize the requests of processors. In snooping protocols the bus act as the serialization point for coherence transactions.

Directory Protocols

The use of an arbitrary multi-stage interconnection network poses challenges to the implementation of cache-coherent shared memory. Although connecting the processing nodes on a scalable network topology, .i.e (Mesh, hybercube), yields to potentially more bandwidth efficient system, it also takes away the inherent broadcast capabilities of a shared bus that can be exploited to implement broadcast-based coherence. Instead, such systems are based on tracking which processor cache contain a memory line, to send the number of necessary messages, and avoid broadcasts. Sharing information is kept in an auxiliary data structure called a directory [6], illustrated in Figure 1.2.

Furthermore, directory information can be distributed to multiple directory engines to avoid the performance bottleneck of a single, monolithic directory. Each node or group of nodes is associated with a directory corresponding to the locations in that node's group local memory. As shown in Figure 1.2, an example of one node's group directory contents. The directory consists of a collection of directory entries, one for each memory block in the node's local memory. Because the processor caches interface to the system at a cache line granularity — that is, each processor cache miss or write back transfers a single cache line of data between memory and the cache — the size of the memory block tracked by each directory entry is usually one cache line. In its simplest form, a direc-

tory entry contains two fields: a state indication and a presence bit vector. In invalidation-based protocols the state indication specifies whether the memory line associated with the directory entry is held shared (i.e., read-only) in one or more caches or whether it is held exclusive (i.e., with read/write permission) in a single processor's cache. The presence bit vector indicates which processors are caching the memory line; if the memory line is held exclusive, only one presence bit may be set. The directory entry depicted in Figure 1.2 shows a case in which the corresponding memory line is held shared, indicated symbolically by the "S" in the state field, and is present in the caches of processors 1, 2, 4, and 6, indicated by the presence bit vector.

When a memory request arrives at a processing node, the controller of the node then retrieves the corresponding directory entry to determine what additional actions are required to service the request. For example, as shown in Figure 1.2, if processor 3 requested exclusive access to the memory line, the memory line first must be removed, or invalidated, from all processor caches currently holding it. In a distributed system, the controller of the node must consult the presence bit vector to determine that explicit invalidation messages need to be sent to processors 1, 2, 4, and 6. In a bus-based system, these invalidation's would be performed automatically when processor 3's exclusive request was issued on the bus.

This is a simplified case is just one example of the operation of a distributed cache coherence protocol. In practice, these protocols are complex, especially because so many race conditions can occur as a result of the lack of a shared bus to serialize all processors' memory requests. More details of distributed cache coherence protocol implementation can be found in [18].

1.3 Thesis Contributions

This thesis presents the design and implementation of a directory-based cache coherence protocol in an FPGA prototype. The main contribution of this thesis are the following:

- Working hardware implementation of an MSI directory-based cache coherence protocol empolying the necessary transient states in order to handle distributed controller cooperation and protocol races (Chapter 3).
- Design and implementation of a hash based directory state organization in off-chip SRAM that minimize the latency of directory accesses to conceptually associative directory state (Sections 3.2.1, 3.3.3).

The rest of the thesis is organized as follows. Background information is discussed in the rest of this chapter. Chapter 2 presents various directory organi-

zations that are commonly used and the problems that arise from each one. In Chapter 3 the cache coherence support on the SARC prototype is detailed. The evaluation of the proposed directory organization in terms of area and complexity, and the protocol performance is presented in Chapter 4. Finally, Chapter 5 concludes this study and future work directives are discussed.

Chapter 2

Background

2.1 Directory-Based Coherence Protocols

The basic states of a cache block in a directory-based protocol are exactly like those in a snooping protocol, and the states in the directory are also analogous to snooping protocols. Thus we present simple state diagrams that show the state transitions for an individual cache block and then examine the state diagram for the directory entry corresponding to each block in memory. These state transition diagrams do not represent all the details of a coherence protocol; they only show the basic MSI protocol, without showing the detailed implementation that depends on a number of details, such as the interconnection network ordering properties, the directory organization restrictions, and the buffering structures that are used.

In this section we present the basic protocol state diagrams. The issues involved in implementing these state transition diagrams are examined in Section 3.3. Figure 2.1 shows the protocol actions to which an individual cache responds. The notation that we use is the following: requests coming from the directory are showed in green and processor, requests that are sent to the directory are showed in blue, and processor events are showed in gray.

The state transitions for an individual cache are caused by read misses that generate GETS messages, write misses that generate GETX messages and from invalidation requests that generates write backs; these operations are all shown in Figure 2.1.

GETS and GETX requests require data value replies, and these events wait for replies before changing state. Knowing when invalidates complete is a separate problem and is handled separately. The operation of the state transition diagram for a cache block is as follows. Upon a load to a cache block that it is in shared or invalid state, a GETS request is sent to the directory. For stores to a

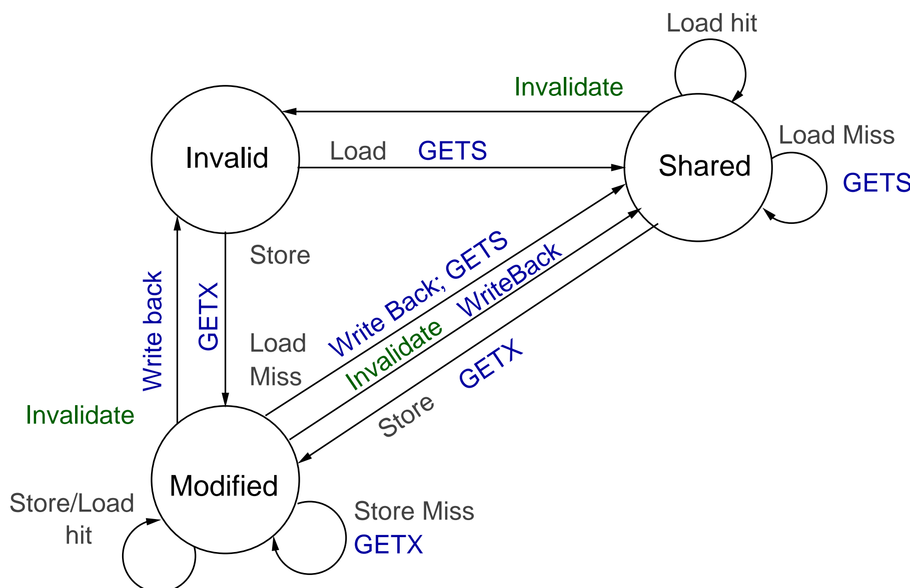


Figure 2.1: Basic MSI Protocol Cache FSM

cache block that hit in the cache, a GETX request is sent instead. Furthermore, the directory send data replies and invalidation requests respectively according to the directory side of the protocol. Furthermore, write back messages are generated from the cache in response to an invalidation request, or due to cache eviction as it is shown in the state diagram below.

In a directory-based protocol, the directory implements the other half of the coherence protocol. A message sent to a directory causes two different types of actions: updating the directory state and sending additional messages to satisfy the request. The states in the directory represent the three standard states for a block; unlike in a snoopy scheme, however, the directory state indicates the state of all the cached copies of a memory block, rather than for a single cache block. The memory block may be uncached by any node (invalid), cached in multiple nodes and readable (shared), or cached exclusively and writable in exactly one node (modified). In addition to the state of each block, the directory must track the set of processors that have a copy of a block; we use a set called Sharers to perform this function.

Directory requests need to update the set Sharers and also read the set to perform invalidation's. Figure 2.2 shows the actions taken at the directory in response to messages received. The directory receives three different requests: GETS, GETX and write back. Our simplified protocol assumes that some actions are atomic, such as requesting a value and sending it to another node; a realistic implementation cannot use this assumption.

To understand these directory operations, let's examine the requests received and actions taken state by state. When a block is in the invalid state, the copy in memory is the current value, so the only possible requests for that block are:

- *GETS* : The requesting processor is sent the requested data from memory, and the requestor is made the only sharing node. The state of the block is made shared.
- *GETX* : The requesting processor is sent the value and becomes the sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state, the memory value is up to date, so the same two requests can occur:

- *GETS* : The requesting processor is sent the requested data from memory, and the requesting processor is added to the sharing set.
- *GETX* : The requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made modified.

When the block is in the modified state, the current value of the block is held in the cache of the processor identified by the set Sharers (the owner), so there are three possible directory requests:

- *GETS*: The owner processor is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).
- *Write Back* : The owner processor is replacing the block and therefore must write it back. This write back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the Sharers set is empty.
- *GETX* : The block has a new owner. A message is sent to the old owner, causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

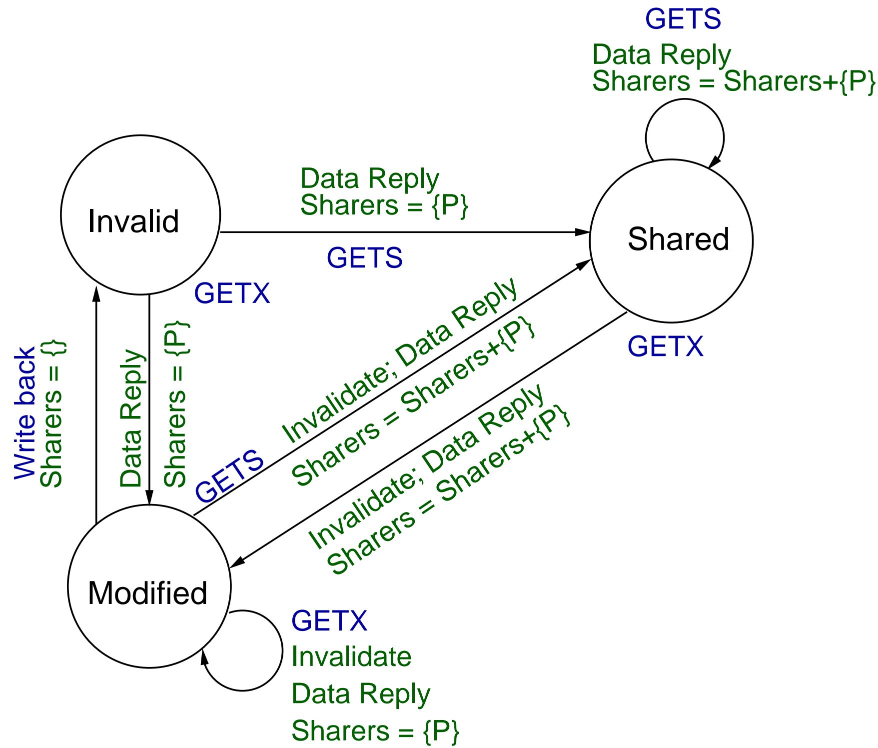


Figure 2.2: Basic MSI Protocol Directory FSM

This state transition diagram in Figure 2.2 is a simplification. In the case of a directory, as well as a snooping scheme implemented with a network other than a bus, our protocols will need to deal with non-atomic memory transactions and implementations issues. Section 3.2.2explores these issues in depth.

2.2 Consistency Models

A Consistency model [5] is a contract between the software and the memory system. It says that if the software agrees to obey certain rules, the memory promises to work correctly according to the programmer's expectations. Unfortunately but expectably there is a trade-off between the restrictions the consistency model poses on the programmer and the performance of such a model when utilized in a distributed shared memory system. The most intuitive is the Sequential Consistency model.

Sequential consistency is a slightly weaker memory model than strict consistency. Lamport [2], who first defined it, posted the following definition: A system is sequentially consistent if:

"The result of any execution is the same as if the operations of all processes

were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".

This fairly complicated definition states that any interleaving of programs operations is acceptable behavior, but all processors must see the same sequence of memory references. A memory system in which one processor sees one interleaving and another processor sees a different one is not sequentially consistent.

A better solution in terms of performance, would be to let a processor finish its critical section and then make sure that the final results were visible everywhere, not worrying whether all intermediate results had also been propagated to all memories in order, or even at all. Such a consistency model is the weak consistency model. The properties of the weak consistency models are the following:

- Accesses to synchronization variables are sequentially consistent.
- No access to a synchronization variable should be issued until all previous processor data accesses (writes) have completed.
- No data access (read or write) should be issued until all previous processor accesses to synchronization variables have been performed.

By doing a synchronization before reading shared data, a processor can be sure of getting the most recent values. It puts a greater burden on the programmer, but the potential gain is better performance. This model is most useful when isolated accesses to shared variables are rare, with most coming in clusters; otherwise there might be a lot of accesses to synchronization variables per shared memory locations that could cause some undesired overhead. Writes done by a single node are received by all other processors in the order in which they were issued, but writes from different nodes may be seen in a different order.

2.3 Directory Organizations

Directory-based cache coherence protocols have been used for long in shared memory multiprocessors. These protocols introduce directory memory overhead due to the need of keeping the sharing status of a memory block in a directory structure. In the past, this structure would provide an entry for every block of main memory and, because of its size, was kept in DRAM.

The directory information represent memory overhead as it adds state information either for each cached or also for each non-cached memory block in the system, depending on the directory organization.

However, this overhead could become very high depending on both the sharing code and the number of cores that comprise the multiprocessor system, and

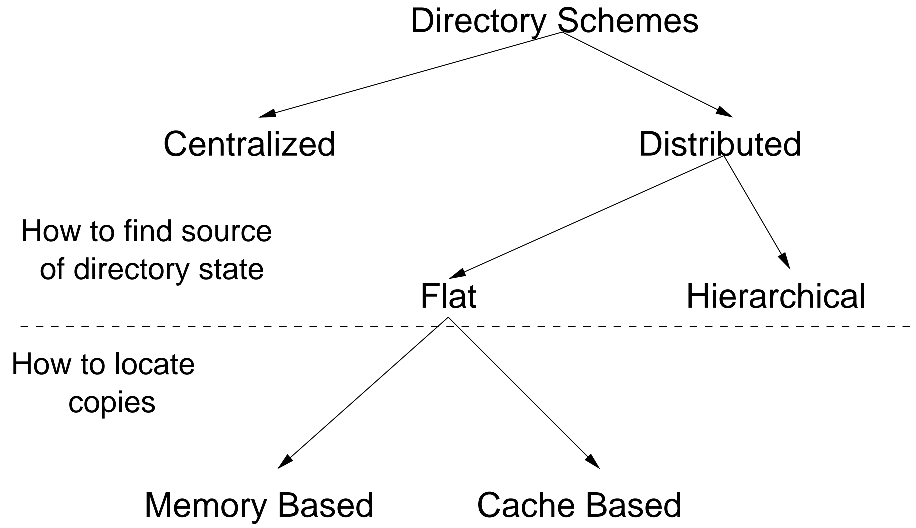


Figure 2.3: Directory Schemes

even be in large systems prohibitive. In this section, we study a directory organization for CMPs that addresses the problem discussed above. Then it reviews the main alternatives for storing the directory information and offers a proposal to optimize look-up time for the directory organization used in this work.

Moreover, the straightforward way of tracking sharers of a block is by using a full-map sharing code where each bit represents a core in the system, which is set when that cache holds a copy of the block. The size of this directory structure scales with the number of cores (P) in the system. In particular, the order of its size is (P $\times$ M), where M is the number of memory entries and P is the number of cores in the system. For the purposes of this discussion on directory state organizations, we assume a single level of caches, since this is sufficient. Thus, the number of caches and the number of cores is assumed the same.

Based if the memory is distributed among the nodes the two categories of directories schemes are the the centralized and the distributed schemes, where the memory is distributed and multiple directories are responsible for a portion of the address space. As shown in Figure 2.3 , the two alternatives for finding the source of the directory information for a block are known as flat directory schemes and hierarchical schemes [18]. The taxonomy that is showed, also divides Flat schemes into two categories based on the way they use in order to locate the copies of the memory blocks. In the following sections we analyze the distributed schemes categories.

2.3.1 Flat Schemes

Flat schemes are more popular than hierarchical, and they can be classified into two categories: memory-based schemes and cache-based schemes. Memory-based schemes store the directory information about all main memory blocks, or only cached copies, at the home node of each block. The conventional architecture of Figure 2.3 which uses the full-map sharing code, is memory based. Examples of memory based system are the Stanford FLASH/DASH and SGI Origin systems [10, 22, 11]. In cache-based schemes (also known as chained directory schemes), such as the IEEE Standard Scalable Coherent Interface (SCI) [16], the information about cached copies is not all contained at the home but is distributed among the copies themselves. The home node contains only a pointer to the first sharer in a distributed double linked-list organization with forward and backward pointers. The locations of the copies are therefore determined by traversing the list via network transactions.

The most important advantage of cache-based directory schemes is their ability to significantly reduce directory memory overhead, since the number of forward and backward pointers is proportional to the number of cache entries, which is much smaller than the number of memory entries. Several improvements have been proposed for chained directory protocols [17] and commercial multiprocessors have been designed according to these schemes, such as Sequent NUMA-Q, which has been designed for commercial workloads, and Convex Exemplar [14] multiprocessors, destined to scientific computing. Nevertheless, these schemes increase the latency of coherence transactions as well as overload the coherence controllers and lead to complex protocols implementations [18]. In addition, they need more cache states and extra bits for forward and backward pointers, which implies changing processor caches. These factors make more popular memory-based schemes than cache-based ones.

The problem of the directory memory overhead in memory-based schemes is usually managed from two separate points of view: reducing directory width and reducing directory height. The width of the directory structure is given by the directory entries and it mainly depends on the number of bits used by the sharing code. The height of the directory structure is given by the number of entries that comprise the directory. In the following subsection we discuss the two alternatives that try to reduce the directory memory overhead.

2.3.2 Hierarchical Schemes

Hierarchical memory schemes treat the processing cores as the leaves of a logical tree, with main memory distributed along with the processing nodes. Every block is assigned to a home node (leaf) in which it is allocated, but this does not

mean that the directory information is maintained or rooted there. The internal nodes of the tree are not processing cores and only hold directory information. Each such directory node keeps track of all memory blocks that are being cached or recorded by its sub-trees and it uses a presence vector per block to tell which of its sub-trees have copies of the block and a bit to tell whether one of them has it dirty. It also records information about local memory blocks that are being cached by processing nodes outside its sub-tree. This information is used then to decide when requests originating within the sub-tree should be propagated further up the hierarchy. In general, the advantages of hierarchical schemes are tightly related to the amount of locality shown by memory accesses, as the delay is high if all the buses/levels that need to be traversed to serve a high percentage of the memory accesses.

The main drawback of such schemes is the latency problem, because the number of network transactions sent up and down the hierarchy to satisfy a request tends to be larger than in a flat memory-based scheme. Even though these transactions may be more localized in the network, each one is a network transaction that also requires either looking up or modifying the directory at its (intermediate) destination node. This increased endpoint overhead at the nodes along the critical path tends to prevail any reduction in the total number of network hops traversed and hence network delay, especially given characteristics of modern networks.

2.3.3 Reducing Directory Memory Overhead

A way to reduce the size of directory entries is to use compressed sharing codes instead of a full-map code. These sharing codes compress the full coherence information in order to represent it using fewer bits than a full-map, at the cost of multicasting group of invalidation's.

Compression introduces a loss of precision, i.e., when the coherence information is reconstructed, sharers that do not cache the block can appear. For example, coarse vector [15], which was employed in the SGI Origin 2000 [11] multiprocessor, is based on using each bit of the sharing code for a group of K processors. A bit is set if at least one of the processors in the group holds the memory block. Another compressed sharing code is tristate [13], also called superset scheme, which stores a word of d digits where each digit takes one of three values: 0, 1 and both, denoting all the sharers whose identifiers agree for both values of both. Gray-tristate [9] improves tristate in some cases using Gray code to number the nodes.

Other authors propose to reduce the size of directory entries by having a limited number of pointers per entry, which are chosen for covering the common

case [19, 20]. The differences between these proposals are found in how the overflow situations are handled, i.e., when the number of sharers of the block exceeds the number of available pointers. The two main alternatives are to broadcast invalidation messages or to eliminate one of the existing copies. Examples of these proposals were implemented on FLASH [10] and Alewife [4].

Other proposals try to reduce directory height, i.e., the total number of directory entries that are available. A way to achieve this reduction can be by combining several entries into a single one (directory entry combining)[21]. An alternative way is to organize the directory structure as a cache, called a sparse directory [15, 12], or even include this information in the tags of shared caches, thus reducing the height of the directory down to the height of these caches. This proposal requires a shared directory or last level cache and is based on the observation that only a small fraction of the memory blocks can be stored in caches at a particular moment of time.

The idea of having duplicate tags has also been used for distributed shared memory multiprocessors as, for example, in Everest [8]. In Everest, the directory structure, called complete and concise remote (CCR) directory, keeps tag and state information of the memory blocks belonging to the local home that can be cached in remote nodes. The CCR directory contains the same amount of memory as a sparse directory and keeps the same information as a full-map directory. However, the number of entries in the CCR directory grows linearly with the number and size of cores and caches in the system.

Unfortunately, these techniques introduce directory misses, i.e., the directory information for a memory block missing in cache may not be found. This situation can be managed by broadcasting invalidation messages to all processors, which can impact coherence traffic and applications' performance. In general, all the described techniques, except sparse directories, result in extra coherence messages being sent, or in increased cache miss rates, reducing the directory memory overhead at the expense of performance and/or power, as a consequence of an increased network traffic.

The reduction in directory height, provided by sparse directory organization or the Everest tag duplication schemes, comes at the cost of an associative directory look-up for blocks with the same index, cached at private parts of the hierarchy above the shared directory or cache. Consequently, increasing the associativity of each private cache or the number of private parts in the cache hierarchy with more cores, lead to increased complexity of directory access look-up. Figure 2.4 show an example of such an organization, and depicts the high-associativity demands of these architectures. Assuming four private, 4-way associative, L2 caches in a system, a sparse directory must keep state for groups of blocks with the same index in all caches throughtout the system. This, results

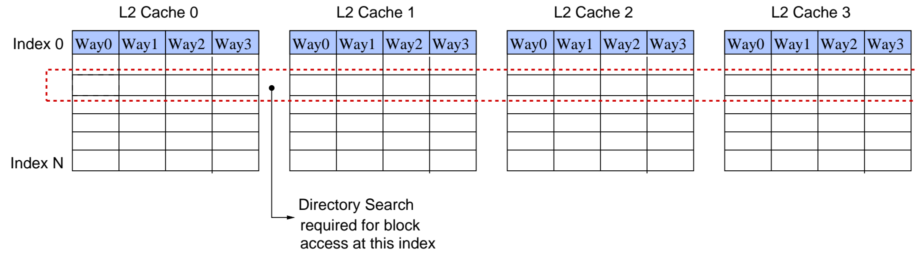


Figure 2.4: Sparse Directory Associativity Demands

to a structure with a degree of associativity equal to the aggregate associativity of all the system caches. For the specific example, an associativity degree of 16, thus the directory structure, needs to support this associativity which makes the implementation challenging as the number of cores and caches increases. With a tiled 16-core CMP, a duplicate tag store will contain the tags for all 128 possible caching locations (simply by mirroring system caches tags). If each tile implements 8-way associative L2 caches, then the aggregate associativity of all tiles is 128. For fast directory look-up and to check the tags in order to locate the sharing vector, a large power-hungry 128-way content addressable memory (CAM) may be required.

Based on this observation, as well as the memory resource limitations in the prototype that we use, in the next chapter we propose a directory organization that removes these high associativity demands using a hashing structure for the directory.

Chapter 3

Design and Implementation

In this chapter we describe:

1. The baseline system design on top of which we add the coherence protocol (Section 3.1)
2. The design of a hash table structure for the directory that is directly implementable in hardware (Section 3.2.1).
3. The design of the directory controller and the integration of coherence logic throughout the system (Section 3.2.5).
4. The design of the full protocol transitions including the transient states, to manage distributed controller interactions and possible races, as well as handle the requirements imposed by directory organization as a hash table (Subsection 3.2.2 and Subsection 3.2.1).
5. The implementation of 2, 3 and 4 which includes an off-chip SRAM controller and the management of the clock domains that are required across the design and the memory clock generation circuitry that the SRAM needs to operate correctly (Section 3.3.3.1).

Specifically, Section 3.1 presents the baseline system and the limitations that arise from the pre-existing FPGA design, as it concerns the implementation of the coherence protocol. Section 3.2.1 introduce the hash directory organization that we use in this work and discuss the motivation behind the specific organization. Furthermore, the architecture of the directory controller as also the implementation of the coherence protocol are presented in sections 3.2.5 and respectively. Finally we present the implementation details for the above in section 3.3.

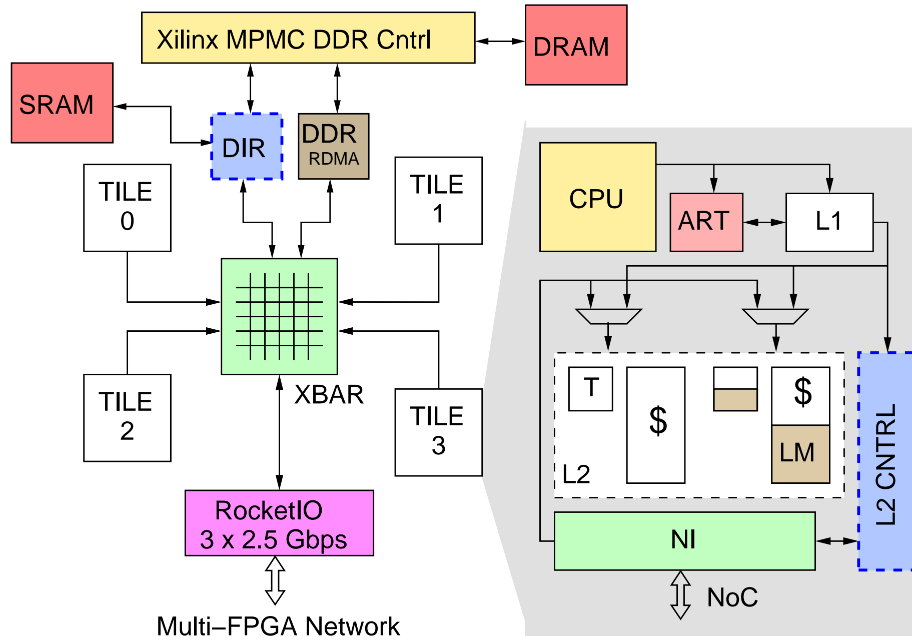


Figure 3.1: System Block Diagram

3.1 Baseline System

The baseline hardware prototype that we use in this work, is implemented in a Xilinx Virtex-5 FPGA using four MicroBlaze soft-cores as processors. The processors are 32-bit, in-order, and have a traditional 5-stage pipeline that also supports single-precision floating point operations. Each processor tile has a private data cache hierarchy, with a configurable Level 2 cache/scratchpad memory tightly-coupled with an advanced Network Interface (NI).

The prototype is equipped with a 256MB DDR2 SDRAM which is used as main memory and is shared between tiles and we use the Multi-Port Multi-Channel Xilinx DDR controller to communicate withe the off-chip DRAM. Communication between tiles and the on-chip DRAM memory controller is achieved through a 64-bit, bufferless, 7-port crossbar switch that features three priorities. An additional switch port can be used to provide multi-FPGA connectivity through multiple external high-speed serial links (RocketIO), and thus this modular design can be expanded with multiple boards in order to build larger scale systems. The block diagram along with the major components is illustrated in Figure 2.3.3.

Every tile of the prototype implements a private data L1 cache and a private, configurable, data L2 cache/scratchpad. These are smaller than one would expect in a CMP, due to limited FPGA resources. The L1 cache of each tile

is 16KB, direct-mapped, with 32-byte cache-lines. Also L1 caches are write-through, with 256-bit wide (one cache line) re-fills, and a single cycle hit latency, and follow "no-allocate" policy on store misses. Each L2 cache is 64 KB, 4-way set-associative, write-back, with 32-byte lines. Furthermore, L2 cache supports multiple hits under a single miss in order to minimize processor idle time. Also, the L2 controller serves write-backs and fills on misses, using the advanced transfer primitives of the tightly-coupled NI as described in [1].

On-chip memory resource limitations on the FPGA prototype, lead us to use the off-chip Zero-Bus Turn-around (ZBT) SRAM that exist in the FPGA board. Consequently, the directory state resides on this off-chip SRAM, while the directory controller logic is implemented in the FPGA chip, and great consideration is given to the directory organization, in order to reduce the number of the off-chip accesses that are needed to find directory information, by exploiting the large size of the specific SRAM, that is organized as 256Kx36 bit. Finally, the changes to the L2 cache controller and the directory controller core are depicted in Figure 2.3.3 with blue dashed lines. Furthermore. the DDR interface with RDMA capabilities that existed on the baseline system is assigned to a different port of the Xilinx MPMC controller in order to be usable for software that need this functionality.

3.2 Design

In this section we discuss the design of the core components of the coherence protocol, the cache organization and a directory organization based on hashing. Moreover, we discuss the necessary additions and modifications to the pre-existing prototype that are needed, in order to implement the directory cache coherence protocol.

3.2.1 Hash Directory Organization

In this section we show a directory organization based on hashing that can scale up to a certain number of cores depending on the system parameters. Furthermore, the described directory organization minimize the directory look-up latency, as also avoids the high-associativity demands of sparse directories.

The described directory organization keeps precise information about all blocks stored in private caches, i.e., directory misses only take place when the block is not stored in any private cache and, therefore, no extra coherence actions are needed as consequence of directory misses.

3.2.1.1 Hashing

One way to reduce the number of accesses, for finding a directory entry, is to apply a hash function on the address bits of a cache block. Consequently, the hash function will distribute the directory information in the directory memory.

By doing so, the directory access latency will decrease and the directory organization will inherit the hash performance benefits, which is the O(1) average access cost.

However, because of potential collisions produced from the hash function, and in order to correctly identify misses in our sparse directory design, we need to occasionally remove a directory entry from the hash table of the directory. Whenever a collision is detected then the associated directory entry that is currently in the directory must be evicted, and the corresponding cache block must be flushed from all the caches.

Hashing techniques, use linked-lists for handling collisions, therefore in order to remove an item from the hash, we must iterate the list until we find the corresponding entry and then we must appropriately fix the previous and next pointer of the removed entry. A Linked-list implementation of the hash requires additional space for holding the next and previous pointers, that prohibit us using such an architecture,a s it would require additional accesses to the memory that holds the next pointers. Furthermore, the complexity in terms of logic for the linked lists manipulation also becomes a bottleneck, as we would have to keep a free list that would contain the available directory entries, whenever we wanted to allocate a new one.

Implementing linked-lists is prohibited in our design due to implementation constraints that arise from the width of the available SRAM chip of the targeted development board, as discussed in Section 3.1, as it is narrow – for holding a next pointer and the sharers bit vector, as well as the required tag used for handling the collisions that may occur. An alternative to this, is to combine multiple words for a single directory entry, in order to have available space for holding the necessary information that comprise a directory entry, and the additional book-keeping information needed for handling the insertion and the removal operations to the hash structure. Instead of this approach, that would exhibit increased latency due to the need of reading multiple words each time a directory entry is either searched, or removed, or inserted in the hash, we use a similar approach that is discussed in next subsection.

3.2.1.2 Address Hashing using Buckets and Slices

We propose a slightly different approach than traditional hashing, grouping all the cache lines of a specific index, and then hashing some bits of the tag portion, of the cache line address, as Figure 3.2 depicts. The main idea, is to logically divide the SRAM into large contiguous areas called "Buckets", which correspond to a cache index. Every bucket, has now a size equal to the SRAM/Index Size of a cache. Consequently, each bucket is under-utilized , i.e., Figure 3.2, shows an SRAM that can store (256k directory entries), in a system

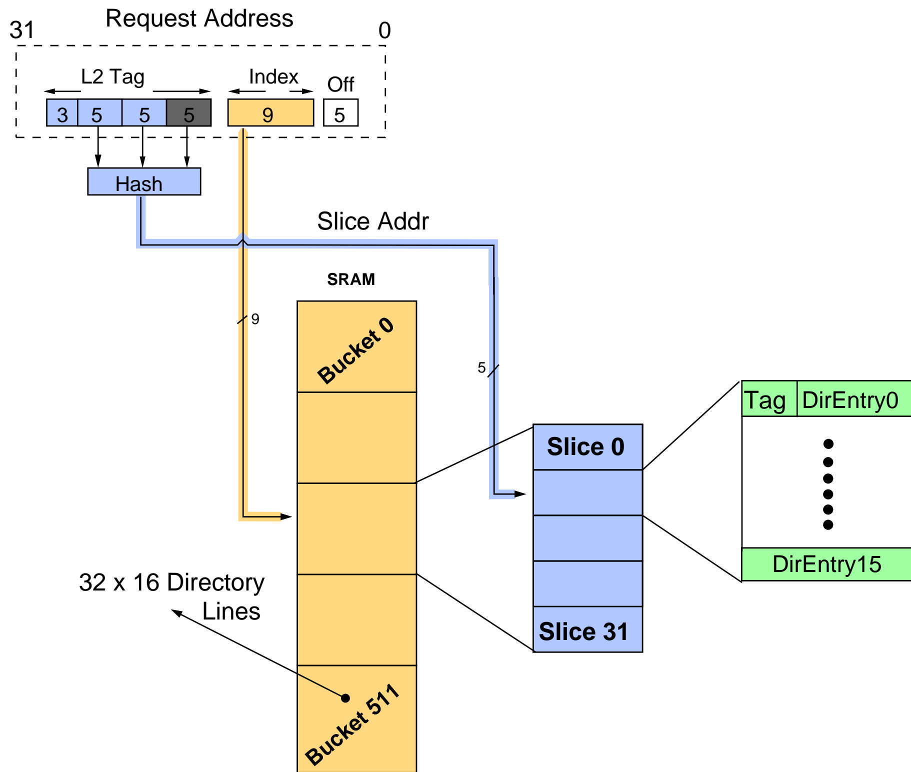


Figure 3.2: Hash Directory Organization

with four 64KB, (4-way) private caches. Thus the utilization of each bucket is 16/512.

Based on this observation, we expand this scheme further, by hashing a portion of the tag address bits of cache line, to map it inside a bucket. This results to a second level of partitioning, stems from the use of the hash function inside a bucket, and divide a bucket into a number of "Slices" according to the size of the hash value. As shown in Figure 3.2, we divide a bucket to 32 slices, each one containing 16 directory entries; as the aggregate associativity of the caches. With this scheme we are able to distribute the directory entries among the 32 available slices that map to a specific bucket.

Due to the fixed size of each slice, that can withstand the aggregate number of cache lines for a specific index of the cache (i.e 4 caches with 4-way associativity degree, would result to a slice capable of holding 16 directory entries), directory have to be notified upon eviction of a cache line in order the slice to not overflow. Therefore, each time a cache line that is associated with a directory entry is evicted from all the caches, and only then, the directory hash must remove the associated directory entry, otherwise, a new directory entry which would hash

in the same slice may not have a free directory entry. In case of a write back message the directory entry which is associated is removed implicitly from the directory controller.

The main advantage of this approach, except from the hashing, is the simplicity and the performance of the remove operation. For every directory entry, we store a directory tag along with the directory entry fields that are commonly used for directories; state and sharing-bit vector. In case of a collision in a slice, the directory tag is compared against the tag from the request address, and when found, the corresponding directory entry is removed from the slice, and the last entry inside the slice is swapped with the removed one. By doing so, the maximum number of accesses to the SRAM is equal to the maximum number of directory entries that a Slice can hold plus the read-modify-write operation that is needed for writing back the last directory entry.

In Subsection 3.3.3 we discuss in more detail the internals of the hashed directory access as well as the area requirements and the limitations for the required control logic.

3.2.2 Protocol Design

The protocol chosen for our design is the three-state MSI write-back invalidation protocol, as described in Chapter 2. The protocol uses three states to encode the state of a cache block that resides in a processor cache.

The main difference from the basic MSI protocol, as described in the previous chapter is that due to the organization of the directory, the directory protocol, has to handle now replacement events that will cause directory entry deletions. This is necessary, because the hash directory organization as described, dictates the use of eviction notification messages, in order to work correctly, and always find space to allocate a new directory entry.

The need of replacement messages in order to notify the directory for evictions, as also the addition of all the transient states at both, L2 protocol FSM and to the directory protocol FSM, complicates the design of the protocol, in order to handle all the corner cases and protocol races that may occur. Examples of such scenarios are discussed, along with an example of two protocol transactions that show the basic functionality of the MSI protocol.

3.2.2.1 Protocol Messages

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directory for the purpose of handling misses and maintaining coherence. These messages basically fall into to two categories. The first category category refers to the requests and the second to the responses. The messages type and the virtual channel (VC) used for each type, and their source and destination is shown in

<table>
<tr><th>Type</th><th>VC / Direction</th></tr>
<tr><td>GetS</td><td rowspan="2">Low</td></tr>
<tr><td>GetX</td></tr>
<tr><td>INV</td><td>Medium</td></tr>
<tr><td>REPL</td><td rowspan="2">High_Cache2Dir</td></tr>
<tr><td>INV-ACK</td></tr>
<tr><td>FILL</td><td>High_Dir2Cache</td></tr>
</table>

Table 3.1: Protocol Messages Categorization

Table 3.1.

As described in Chapter 2, caches can generate, depending on the state of the cache line and the processor request, GetS and GetX messages. The semantics of these request are similar for the coherence protocol, and typically, these messages request read or write access write for a memory block. Furthermore, replacement messages use explicitly eviction messages for shared lines and belong to the request class too. Replacement messages (Repl) are essential in our implementation due to the directory organization which necessitates the notification when an eviction occurs at a cache, as described. Except from these requests, invalidation messages, that are generated from the directory, belong to the request class, but they have an opposite direction, and are destined to caches.

Response class, includes the invalidation acknowledgment (INV-ACK) message that refers to the successfully reception of an invalidation message from the directory, and it is generated from the cache controller, and it is used to inform the directory either about the completion of the invalidation at a cache. Moreover, write back (WBACK) and data responses (FILL) messages, are responses and they contain data instead of control information in contrast with the other messages. Specifically, write-back messages are destined to the directory and they are generated when a cache evicts a cache line in modified state, or the processor issue a store to a shared line, or when a cache receives an invalidation.

3.2.2.2 Simple Protocol Transactions

Figure 3.3, shows three typical coherence transactions, a read miss in a cache line that is currently in modified state in another cache, a write miss to a shared cache line, and an eviction of a shared line. For the read miss, we assume that one cache has write access to the cache line initially, thus the directory and the cache that holds the line is in modified state. When another cache issue a load to the same address, then it issues a GetS message to notify that wants the cache line for read access. In the protocol we design the directory issues and invalidation request to the owner of the block and waits the write-back from the

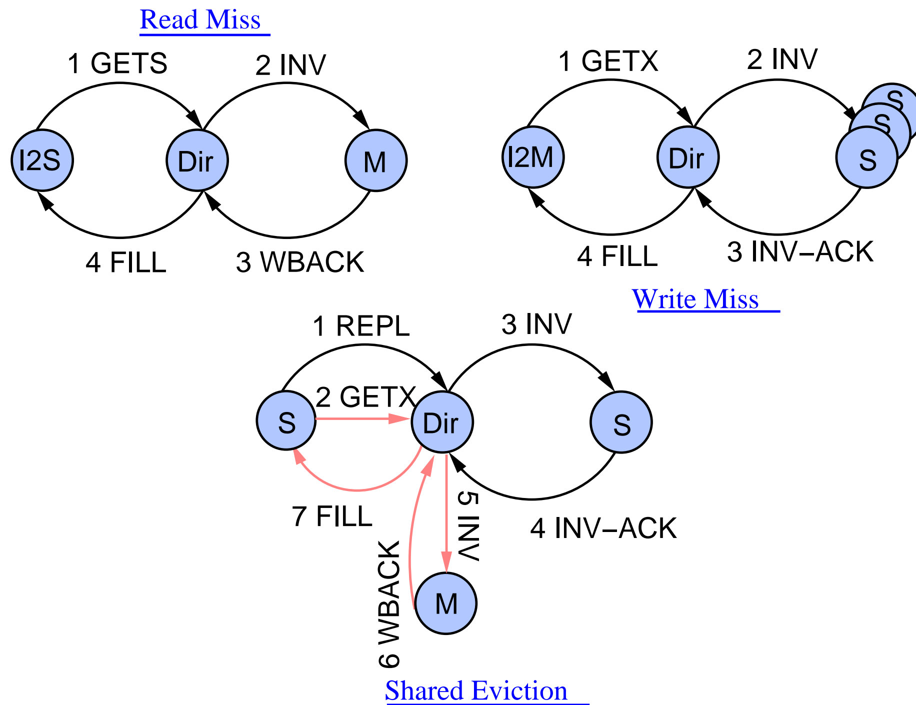


Figure 3.3: Simple Protocol Transactions Example

invalidate cache until it responds with the fill request to the initial requesting cache. Another alternative, would be to forward the data from the cache that had write access to the block directly to the requesting cache, instead of using the directory to sent the response. Although, with this scheme the latency of the read miss is reduced, the complexity of the protocol increases as, the directory would not have exact information about the completion of the transaction, in case the forwarded response get lost.

Similar, the write miss scenario to a shared line, produces invalidation's to the current holders of the block. As a consequence, the caches that receive the invalidation message sent from the directory, respond with an acknowledgment message to notify the directory for the invalidation. The response then with the data is sent to the requesting cache only when all the invalidation's are gathered to the directory. With this approach directory ensure that no other caches except the initial cache has the block, which will violate the coherence property.

Finally, the shared eviction example is more complicated than the others. Assuming a block is shared among 4 caches, and a cache decides to replace the specific line with another one that is cached in a different processor in modified state, the cache on the left issue a replace message (Repl). Subsequently direc-

tory determines the sharers of the block and issue invalidation's. Also, because the fact replace a shared line with a line that needs write access, it issues back to back a GetX message for the new address. Upon successful reception of the invalidation acknowledgments the directory process the GetX request, otherwise due to the limitation imposed from the hash directory organization, the associated entry of the GetX request may not find available space at the directory slice that is mapped.

Furthermore, in case a replacement message is generated in between the reception of the acknowledgments and the invalidation to the owner of the new block, the replace messages are ignored from the directory, because the directory entry has already been deleted.

3.2.2.3 Transient States and Protocol Races

Figure 2.3.3 show an example of three different scenarios, that need transient states, and make use of the blocking property of the directory we implement, as will be discussed also in subsection 3.2.5. The simplest scenarios, are those that concern the blocking property of the directory. Assuming that a block is cached in Shared state in two caches (seer the S2M example), if a GetX arrives at the directory due to a write to this block, then the directory sends invalidations as necessary and *blocks* all the subsequent incoming requests,(postpones the service of the Low VC), until it receives all invalidation acknowledgments and it replies to the GetX request. This blocking behavior guaranties that no request by the directory will be serviced for a block, while another is in progress (serialization of requests to the same block).

Our blocking directory implementation is very restrictive, since it supports only a single outstanding request at the directory. Allowing concurrent service of requests to independent blocks is possible, by replication of some components of our directory controller, or by implementing a macro-pipeline of the controller components. Concurrent service of independent requests would also require an associative structure for guarding against initiating service of requests with a previous outstanding request, which should also keep track of per block expected invalidation acknowledgments for Shared to Modified transitions. The M2M example shows the corresponding situation for an M2M transition, where the directory blocks a subsequent GetX by a cache until the modified block is written back and sent toward the requester whose GetX arrived first at the directory.

Similar to the S2M blocking example, in case of a change of write ownership, the directory blocks again waiting a write-back message from the current owner of the block.

On the other hand, using an associative structure to keep the outstanding coherence transactions as also serving the High VC, when it is near full, would

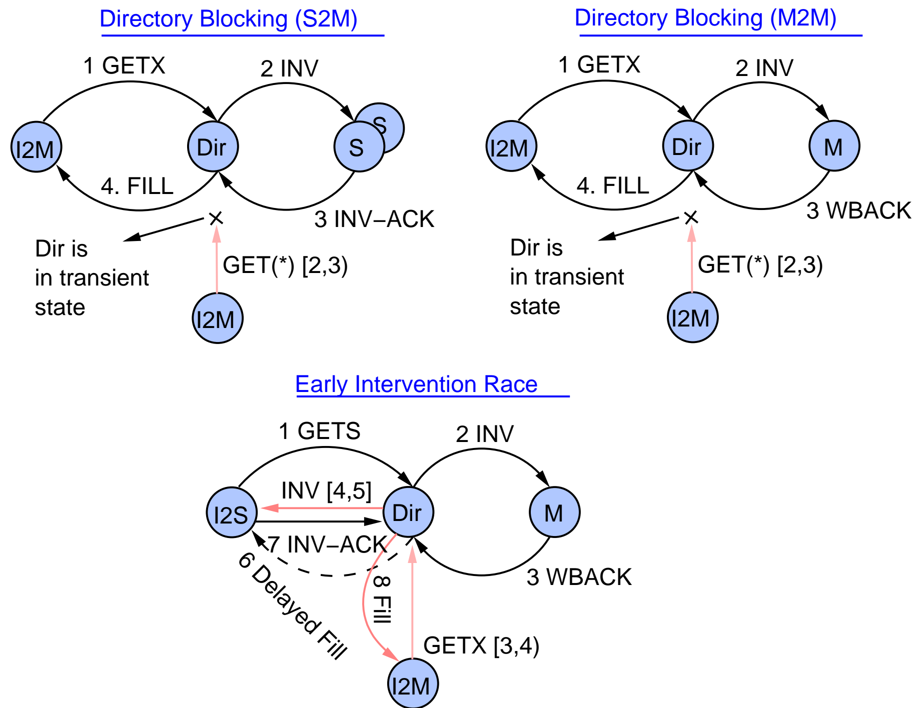


Figure 3.4: Transient States and Protocol Races

be beneficial, but with an additional cost in terms of logic and complexity.

Finally, the early intervention race [3] example exposes the need for transient states at the cache side of the protocol. For the specific example we assume that a cache has write access to a specific block and a GetS arrives at the directory followed by a GetX, both for the specific block. If in between the time that the invalidation produced from the former, another request (GetX) is serviced, then a spurious invalidation – destined to the cache that generate it initially– can be received without having the data. In such case, the data response actually GetS delayed. As a consequence the cache must be able to handle such cases and acknowledge to the invalidation and transition to transient case.

Likewise, a lot of corner cases exist due to the distributed nature of the protocol and due to the deadlock avoidance strategy. This makes directory protocols difficult to debug and implement. A more detailed representation of the transient states that are needed is presented in Subsection 3.2.4.

3.2.2.4 Deadlock Avoidance

Deadlock freedom is ensured when a series of n messages, generated by different controllers uses n independent networks (or virtual channels), and the protocol implementation guaranties that the last message in every such series

can always be serviced, while each of the n-1 previous messages in the series can be serviced if there is space in the independent network (or VC) where its response must be placed. To ensure that a controller can process the response for a message it sends for a coherence transaction, we use two techniques. The initiating cache controller, reserves space for the reply, so that is guarantied that the reply can be processed (i.e a coherence request is not initiated until such space can be reserved) . In addition, the directory is blocking the processing of requests, accepted in "lower-order" networks, until it has received, in "higher-order" networks, the responses to complete the processing of previous request. Note that this kind of blocking directory controller operation is only required on a per address basis, up to the number of outstanding transactions the directory can support. Our implementation only support a single outstanding transaction to avoid increased complexity required for larger number of concurrent requests.

In general, the number of logical channels that we need to ensure deadlock freedom in a protocol is equal to the maximum protocol sequence. In our case , the maximum sequence is four messages, required for transactions that need invalidations (e.g GetX-> Inv->WB-> Fill).

Sending messages using channels by ascending order prohibits the creation of cyclic dependencies. As an example a load/store from a processor will sent a GetS or GetX request to the directory in the lowest priority, because such requests could generate other requests, like invalidation's. In order to avoid deadlocks, the cache controller has to respond to a virtual channel different from the channel that receives a request, this dictates that the invalidation's that could be generated according to the protocol, must be acknowledged using a higher Virtual Channel (VC). Otherwise, messages could be blocked behind independent traffic and potentially form cycles that can lead to deadlock [3].

Furthermore, invalidation acknowledgments also generate data packets from the directory controller to a cache. Data packets are a final response and they are always consumed at the endpoints, which could be a cache or the directory

On the other hand, replace requests are generated due to evictions of shared or modified blocks. Consequently, GetS requests depend on replace requests and have to be consumed at the directory controller in order to make progress, and service the subsequent request for the new block.

The need of replacement messages stems from the directory organization, directory controller must be notified when a processor evicts a block so it will not generate unnecessary invalidation's to caches that don't hold the specific block. The assignment of the message types to VC's is shown in Table 3.1. Also, the number of VC's that either the cache controller or the directory use per direction are described in Table 3.2.

As stated before, when a replacement occurs at a cache and in, i.e, the line is

Module	Input Queue	Output Queue
Cache	Medium	Low
	High_Cache2Dir	High
Directory	Low	Med_Dir2Cache
	High_Cache2Dir	High_Dir2Cache

Table 3.2: Coherence VC / Direction

in Shared state, then two messages are en-queued back to back, a replacement message, followed by a GetS referring to the new block that will be fetched. For the current implementation, we assume an ordered network, because the directory must service always first the replacement message and remove the directory entry associated with the request address, before it processes the subsequent GetS or GetX request for the same cache line. Furthermore, because of the directory arbitration policy for the incoming messages, the replace message as it belongs to the High VC, it is ensured to be processed first, in order to remove the corresponding directory entry, and invalidate the caches that are caching the specific address.

3.2.3 L2 Cache Architecture

In this subsection we describe the basic architecture of the L2 cache that is modified in order to add the coherence extensions. The difference of caches from our baseline system conventional caches is that cache blocks can be used as normal directly addressable memory (scratchpad). In this way each cache block could be explicitly managed by the programmer, without any interference from a coherence protocol.

Moreover a scratchpad block could act as a network interface command buffer, where an RDMA transfer description is formerly or it could act as a queue or counter with some advanced features that are proposed in the SARC architecture. Thus, the L2 cache controller is modified in order to maintain the coherence property, by sharing the existing functionality for supporting these explicit communication mechanisms and the tightly-coupled network interface that is merged with the cache controller. The overall structure of the cache controller is depicted in Figure 3.5. Except from the main cache controller (L2 Cntrl), the network interface is merged with the cache design.

Tag and data arrays reside in separate on-chip sram blocks, implementing in a phased cache design where after the tag match, a data array is read, and the data are returned to the processor in case of a cache hit. Also the cache controller makes use of a single miss status holding register (MSHR) to support outstanding requests. The specific register is used to support multiple hits under a single miss. Concerning the state of a cache line, the state bits of all cache-

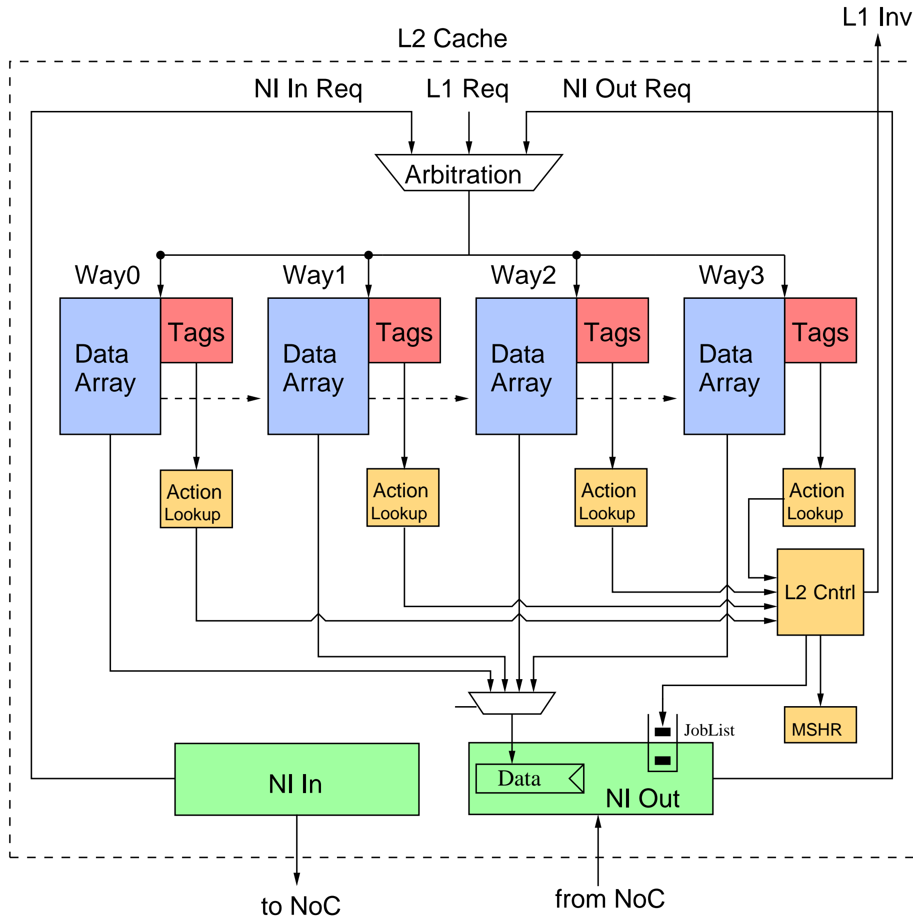


Figure 3.5: L2 Cache Block Diagram

lines of an index are kept together with the replacement policy information in a separate, single-ported memory block. Specifically, the replacement policy of the L2 cache is, pseudo-LRU among lines that are not scratchpad and are not in a transient state. Such a bookkeeping scheme, where control bits are packed together, allows fast cache-line replacement decisions.

As it is depicted in Figure 3.5, the incoming network Interface (NI In) and the outgoing network interface (NI Out), arbitrate for the tags of the L2 cache and also for the data array, in case of an incoming data message from NI In. Furthermore, the L2 control is responsible for the allocation of the Miss Status Holding Register (MSHR), as the protocol indicates. The Action Lookup modules, as shown in the above figure, calculate the proper actions in case of a hit, depending on the state of the cache line and the event generated by the network interface (NI In or NI Out). In general, action lookup module encompasses the cache side of the protocol.

The L2 Control logic, executes the actions that are generated through the action look-up modules, regarding the generation of network messages, as well as the invalidation the L1 cache, the manipulation of the MSHR and reading or writing the tag and data arrays. In the case of network messages that need to be sent from the L2 cache, a descriptor of the message is en-queued to the joblist FIFO, which contains a descriptor of the message. Data, in case of a write back message are read from the data arrays by NI Out, to an internal register and then are sent through the NoC.

The logic that we add to the L2 cache, for implementing the coherence protocol, in a large extent affects the L2 Cntrl and the action lookup modules. For the network interface (incoming and outgoing), we only add the support for sending and receiving the additional coherence messages. Furthermore, the functionality of the MSHR is enhanced along with other changes to the L2 Control. The additional states and actions that we add to the L2 control FSM are presented in subsection 3.2.4.

Actions that are generated from the L2 cache, are calculated in a per way basis, and thus the action lookup logic inside the L2 controller is replicated four times. This design choice that concerns the action lookup logic and has impact in the logic overhead of the cache controller as will be discussed in Chapter 4.

Next, we show the changes needed, in order to implement an atomic operation in order to be able to implement synchronization primitives such as locks.

3.2.3.1 Synchronization Primitives

Synchronization mechanisms are typically built with software routines that rely on hardware-supplied atomic operations. In a multiprocessor system the key ability required is to atomically read and modify a memory location. Therefore the hardware must support such an operation.

3.2.3.2 Consistency

In order to support a weak consistency model, memory barriers have been implemented exporting some internal signals of the L2 cache controller to the software allowing the processor to be notified when there is no pending transaction (write or read). Using the memory barrier, all the outstanding store and load misses of processors are completed.

By definition, due to the coherence protocol, the system is cache consistent. That is, for any specific address x, coherence protocol guarantee the serialization of the accesses to this location only and the global order of writes to x, as also the other properties of sequential consistency, as described in Section 2.2.

Furthermore, if the memory barriers are not used, the memory system is said to be processor consistent. Effectively there are no guarantees about the order in which different processors see writes, except that two or more writes

from a single source must arrive in order, as though they were in a pipeline. In this model all writes generated by different nodes are considered concurrent. Also for every memory location, x, there be global agreement about the order of writes to x. Writes to different locations need not be viewed in the same order by different processors.

3.2.3.3 Atomic-Fetch-and Φ operations

Typical operations that provide atomic semantics are the Test-and-Set, Fetch-and-Add or atomic exchange instructions. We design and implement an atomic-fetch-and-Φ instruction, as we describe in this subsection.

To be able to build any of these primitives we need special instructions for issuing an atomic operation. Because we don't have access to the instruction set architecture (ISA) of the microblaze processor, we use a memory mapped register inside the L2 cache, to implement the functionality needed to support atomicity between a pair of load and store instructions. Therefore, in order for the programmer to use such atomicity, the address which has to be atomically read and written, must be set into the L2 cache register, to mark the cache line that holds it as atomicity capable. After marking the cache line as atomicity capable, the subsequent load and store to this address are treated as atomic., along iwth register operations between them

Consequently, the subsequent load to the address that is marked as atomic capable, generates a GetX instead of a GetS in contrast to the case in the basic protocol. If an attempt is made to modify the memory location by another cache, before the store to the "locked" address occurs, then, an invalidation message will be received. In this case, the L2 cache controller buffers the invalidation, and when the store to the atomicity capable cache line is issued from the processor, the cache controller issues a write-back and invalidates the specific cache line. In particular, for deadlock avoidance only register to register commands can be inserted in between this pair of instructions (the load and the store to the atomicity capable line). Otherwise, a miss between the load and store instructions would not to be issued in the current implementation. The additional states is presented in Section 3.2.4.1.

3.2.4 L2 Cache Controller Modifications

In this subsection we describe the additional functionality that is added to the L2 Cache controller finite state machine (FSM). Furthermore we present the cache side of the protocol. In general, the cache controller issue requests to the outgoing NI whenever the protocol indicates, and receives request from the incoming network interface and the L1 cache.

Coherence message requests are issued to the NI Out joblist FIFO, as depicted in Figure 3.5, and wait to be scheduled. In case of outgoing requests

State \ Event	**Load**	**Store**	**Eviction**	**Inv**
Invalid	GetS MSHR_alloc TagWrite / **I2S**	GetX MSHR_alloc TagWrite / **I2S**	-	InvAck / **Invalid**
Shared	L1Fill DtRd Hit / **Shared**	Repl GetX MSHR_alloc DtWr L1Inv / **S2M**	REPL GetX | GetS MSHR_alloc TagWr L1Inv / **(I2M | I2S)**	InvAck TagWrite / **Invalid**
Modified	L1Fill DtRd Hit / **Modified**	DtWr Hit / **Modified**	WBack MSHR_alloc TagWrite DtWr / **M2IS | M2IM**	Wback MSHR_Alloc TagWrite / **M2I**

Table 3.3: L2C (Stable) States transition Table

waiting a response from the directory controller, the cache controller stores the appropriate transient state to the MSHR and to the tag. As discussed in Subsection 3.2.2.4, the cache incoming network interface maintains two queues, for the two different VC's, one for data responses and one for the invalidation messages. Similarly, the cache outgoing interface manages two VC's, one for write-back messages, invalidation acknowledgments, and another for replacement messages and cache requests (GetS, GetX). Thus, for sending a message to the directory, the cache controller en-queue descriptors to the NI Out joblist, that contains the VC, the address to which the message refers, and the system wide fill address (which includes the cache way in which the fill is expected and the node id). This adress is used from the directory controller in order to find the node id, as also to generate a response to the requesting cache.

In case of write-back requests or coherence misses, the MSHR structure is used. De-allocation of the MSHR occurs when a response is received that cause a cache block to transition to a stable state (M,S,I for cacheable requests). For that reason, the existing network interface was modified in order to support these coherence responses from the directory controller.

The cache-side of the protocol transitions and associated actions are detailed in Tables 3.3 and 3.4, where each entry contains an <action list / next state> tuple. When the current state of a block corresponds to the row of the entry and the next event corresponds to the column of the entry, then the specified action is performed and the state of the block is changed to the specified new state. If only a next state is listed, then no action is required. If no new state is listed, the state remains unchanged. Impossible cases are marked with "–", which means no action or state change is required.

State \ Event	Inv	Data	Unblock
I2S	InvAck TagWrite/ **ISD**	DtWr TagWrite MSHR_free L1Fill Hit / **Shared**	-
ISD	-(3)	TagWrite MSHR_free DtWr L1Inv Hit / **Invalid**	-
I2M	InvAck TagWrite / **IMD**	MSHR_free DtWr TagWrite Hit / **Modified**	-
M2I	- (1)	-	MSHR_free TagWrite / **Invalid**
S2M	InvAck TagWrite Hit / **I2M** (2)	TagWrite MSHR_free Hit / **Modified**	-
IMD	-(3)	WBack DtWr L1Inv TagWrite/ **M2I**	-
M2IM	GetX TagWrite MSHR_Alloc / **I2M**	-	GetX TagWrite MSHR_Alloc / **I2M**
M2IS	GetS TagWrite MSHR_Alloc / **I2S**	-	GetS TagWrite MSHR_Alloc / **I2S**

Table 3.4: L2C (Transient) States transition Table

(1) Issued Write-back, thus no need to acknowledge invalidation
(2) Directory received GetX before remove request
(3) Cannot receive another invalidation,directory is in transient state

The actions that the protocol indicates are mainly the allocation of the MSHR, the generation of various messages which is described in Subsection 3.2.2, and the actions that notify the L1 cache for an invalidation or a cache line fill. Moreover, the cache controller actions, for reading and writing the tag arrays and the data array are shown in the state transition tables below. Specifically, each time the processor issue a load instruction the state of the cache line is read and if it is Invalid then a get shared message is sent and the MSHR is allocated waiting for the response from the directory controller. If the

incoming network notifies the cache controller with an invalidation before the response with the data arrives the state of the cache line is in transient state (I2S) thus an invalidation acknowledge message has to be sent to the directory and the state must be updated to ISD in order to wait for the final response to arrive.

Although such a spurious invalidation request cannot be received from the on-chip network due to the blocking property of the directory and the ordering of the NoC, this occur, when the data response from the directory is delayed in the L2 network and the invalidation request generated from a get exclusive request from another cache reach the cache controller before the response. In that case the invalidation request is buffered in the MSHR. At the time a data response arrive, the data are returned to the processor and the cache line GetS invalidated and the MSHR is de-allocated. The same scenario occur for cache lines that are transitioning from Invalid to I2M when a store from a processor is issued.

Loads that find the cache line state in Shared or Modified state always hit in the L2 cache and the data is returned to the L1 cache. In case of a modified cache line stores also hit in the cache according to the MSI protocol. On the other hand stores to cache lines with Shared state need to issue an upgrade request, sending a get exclusive message to the directory and allocating again the MSHR as also writing the data to the corresponding field of the MSHR. It must be noted that L1 cache must be invalidated in order to the caches to be coherent.

The cache line transitions to the Modified state, only if a data response is received from the incoming Ni In. If again, an invalidation message is received before the response that gives the write access to the specific cache line, then it is acknowledged and the cache line is updated with IMD state waiting for the data to arrive and discarded, and then issue a write-back to the directory.

Invalidation requests finding a cache line in shared state always generate an acknowledge to the directory and invalidate the L1/L2 cached copy. Modified lines have to issue a write-back request and allocate the MSHR in order from the outgoing interface to read the data from the data array and sent them to the NoC. For this reason, a modified cache line transitions to the M2I state waiting for the outgoing interface to be granted from the arbiter and read the data that have to be sent. Upon completion from the network interface with the Unblock operation code (OpCode), the cache line state is updated with the Invalid state and the MSHR is de-allocated.

As it concerns replacements events, these are only allowed between cache lines that are not in transient state, thus for shared lines a replace message must be sent to the directory as described in section , that notifies that the

specific address is evicted from the specific L2 cache. Depending on the cause of the replacement due to a store miss or a load miss, the new state of the cache line is either I2S or I2M and a (GetX or GetS) message is sent back to back in order to fetch the new cache line and remove the old one from the directory, in case of receiving a Repl message from the last sharer, also in case of Modified state, if a write back is received, then again the corresponding directory entry is removed.

Similarly, Modified lines are treated similar to the shared lines with the exception that the copy of the data must be purged and sent to the directory. Consequently, the cache line state is updated to a transient state – either M2IM or M2IS – according to the type of miss.

In the same way of purging copies from modified state, when an invalidation message is received, an MSHR is allocated and when the outgoing network interface read the data; that have to be sent, from the data arrays, the MSHR is de-allocated and the cache line state transitions to Invalid.

3.2.4.1 Atomic-Fetch-and Φ Implementation

In order to implement the atomic operation as described in Subsection 3.2.3.1, we have to add a number of states to the protocol of the L2 cache. These states are used in order to identify atomic operations and treat them differently from the conventional load and store instructions that the processor issue. The portion of the L2 cache protocol that is used for the atomic operations is illustrated below with the state diagram in Figure 3.6. The states shown with red, are used in order to buffer the invalidation that came before the response withe the data; in case another cache issued and get the atomic variable.

States shown with green, are the initial states where, the GetX request is issued due to a load to the atomic capable cache line. Finally, the ML state is the actual state where the store to the atomic variable is allowed to be issued and take exclusive access. The basic events and actions that occur and must be executed are shown with Blue an green respectively.

Assuming, that an atomic operation is issued, a load on the atomic capable cache line is issued (LLd) which causes the cache controller to transition to the corresponding state according to the current state of the line – that states are S2L , I2L, and M2L. If an invalidation is received in between for this cache line, before the actual data are received from the incoming network interface of the cache controller then the cache transition to the S2LI, or I2LI, or MLI respectively. In this case, when the store to the atomic capable cache line occurs a write back message is issued, and the atomic register is cleared and the store is executed.

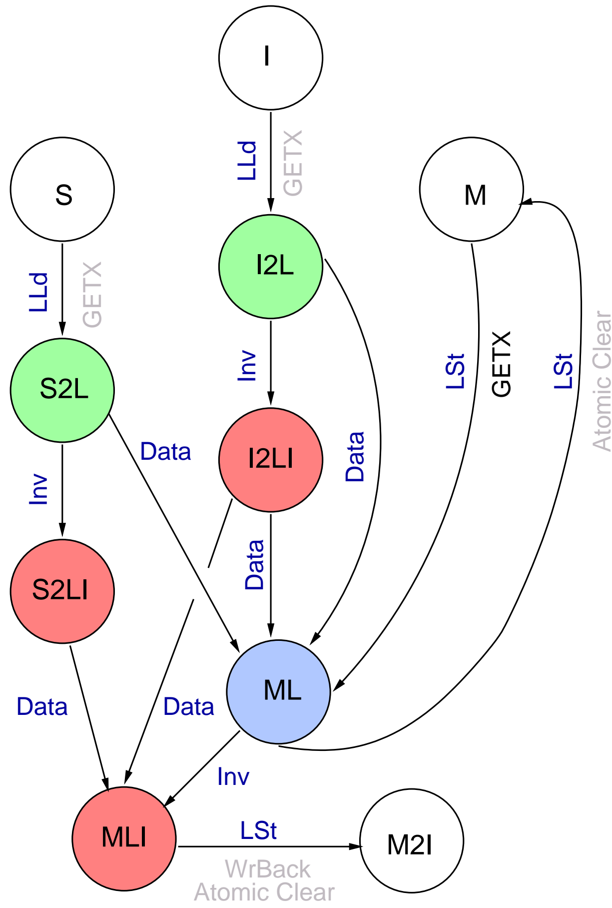


Figure 3.6: Atomic-Fetch-and Φ operation FSM

In case, the cache line is in ML state, that means that no other cache took the atomic variable so far, so upon a store to this cache line (LSt) it transitions to the Modified state. However, and invalidation message can be received in this window, where the cache line is in ML state, consequently it transitions to the MLI state in order to buffer the invalidation and execute the store, when it occurs. Furthermore the M2I state exists due to the write back message that have to be issued, similar to the Modified / Invalidation tuple in Table 3.4.

3.2.5 Directory Controller Design

In this subsection we present the architecture of the directory controller block. Directory controller is responsible for the reception and the transmission of the appropriate coherence messages from and to the L2 caches, according to the directory side of the protocol. Figure 2.3.3 presents the block diagram of the directory controller.

The core blocks that comprise the directory controller are the network input

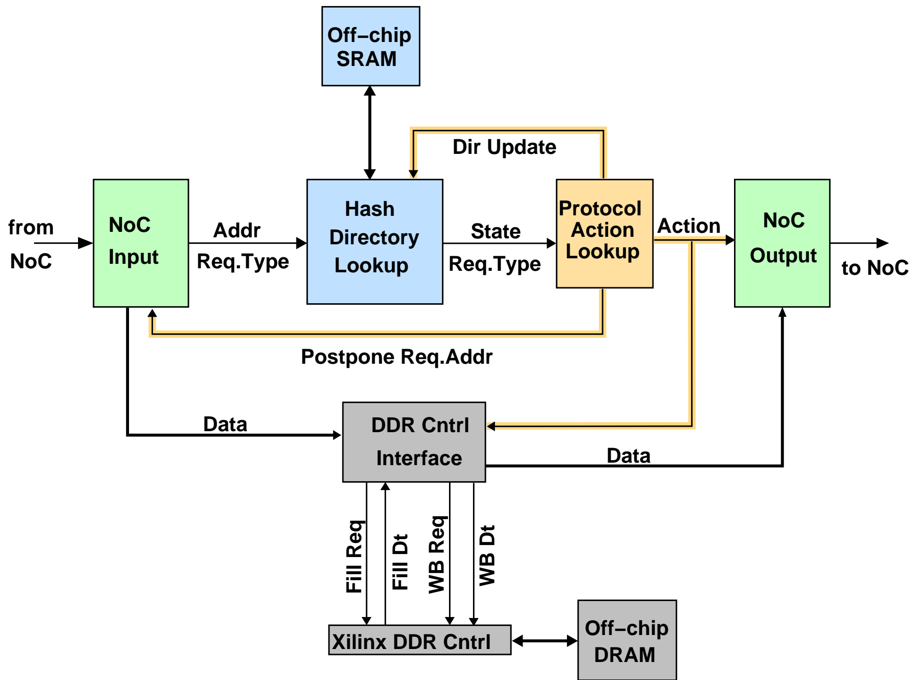


Figure 3.7: Directory Controller Block Diagram

and output modules, the hash directory lookup module, the DDR controller interface, and the protocol action look-up module. Figure 2.3.3 shows how the directory controller executes a protocol state transition coherence transaction in four or five stages. Transactions that require invalidation require multiple passes through the directory macro-pipeline depicted.

In the first stage, the NoC input module, upon the reception of a coherence message, passes the address and the request type of the message received to the next stage of the controller, where the hash directory lookup module retrieves the state associated with the specific address . The blocking property of the directory. This is done with a signal from the protocol action look-up module to the NoC input module as one of the actions taken for the specified state transition. In Subsection 3.3.2, we describe this functionality, and we present the internals of the NoC input module.

After the NoC input module stage, where the actual packet is read from the network, the associated directory entry is searched to the hash directory in the off-chip SRAM, as will be detailed in Subsection 3.2.1. Hash directory lookup logic and the sram controller it implements retrieves the state of the requesting address. Subsequently, the request moves to the protocol action loook-up module, where the appropriate coherence actions as imposed by the protocol, are fetched from a look-up table.

During the third stage, actions that must be executed from the controller are sent to all othe directory modules, as necessary. These actions represent the coherence protocol, and may be some of the following: read from main memory, write to main memory, invalidate current sharers, postpone a request, and update the state of an address. A more detailed description of the actions is presented and discussed in Subsection 3.3.4.

Furthermore, in order to read and write from the off-chip DRAM of the prototype board, a controller interface is implemented for the Xilinx DDR controller, as depicted in the Figure 2.3.3. This interface, can also bypass the DRAM and send the data from the NoC input directly to the NoC output. This functionality is further described in Section 3.3.

At the final stage of the execution, the NoC output module sends either the response with the data, using the DDR controller Interface, or issue invalidation messages as the protocol action look-up indicates.

3.3 Implementation

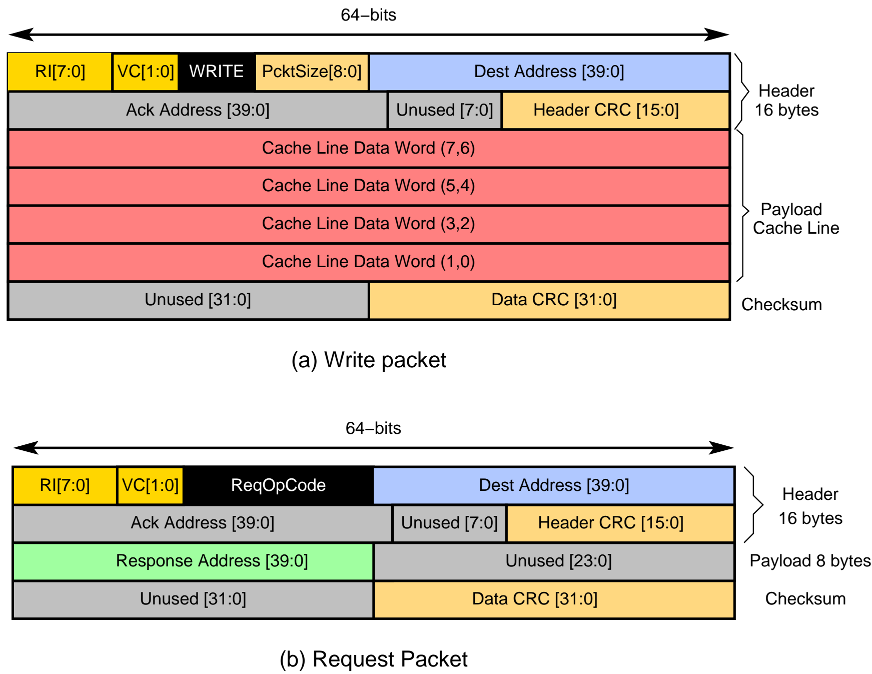
In this section we present the internals of the directory controller implementation. Furthermore,we show the actual implementation of the coherence protocol and we discuss some aspects of the performance of various operations of the protocol, such as the directory look-up latency, the remove operation cost for the hash-directory.

3.3.1 Protocol Packet Format

The packet format of the coherence messages is shown in Figure. Packets have an 128bit header containing the routing information and the virtual channel they belong, as also a header CRC, and an acknowledge field that is not used from the cache coherence protocol. Protocol packets that were described in Subsection 3.2.2.1 , are divided into two categories, requests and responses.

The first format as shown in Figure 3.8 , is the write packet format, that is used from the directory controller when a fill response must be sent back to a requesting cache or from the cache controller to the directory in case of a write-back. The second type of format represent the class of requests messages, either from the cache or the directory.

The only difference from other requests; generated from the caches, is that the response address field is ignored, because the address that need to be invalidated is contained in the destination address field of the packet. Also, cache requests use all the available fields of the request packet format. Another required field, is the response address, which is used from the directory controller to determine the requesting processor, as also the cache way in which the requested cache line exists;to generate the response back to the requesting cache.



(a) Write packet

(b) Request Packet

Figure 3.8: Coherence Packet Format

Acknowledgments to invalidation requests that are generated from the directory are sent with the request packet format to the directory and contain only a destination address and a response address as described before; in order to identify the processor id .

Finally, replacement packets are alike invalidation acknowledgments, and only differ in the ReqOpCode field and the VC they belong.

3.3.2 Directory Controller NoC Input Module

Directory controller receives packets through the incoming network interface that is depicted in Figure 3.9. The NocIn module has two clock domains. The first one is the cache-processor clock domain and the second is the directory clock domain that Action Lookup Module, SRAM Hash Lookup and DDR Interface belong. The NoC input module (NocIn) maintains two FIFO's for the two VC's (High_Cache2Dir and Low) that were discussed in Subsection 3.2.2 , and two set of registers (Incoming and Pending).

These two FIFO's are implemented inside a single BRAM block and are used for crossing the clock domain between the cache-processor clock domain which the NoC belongs and the directory clock domain. As it is shown in Figure 3.9, the packets are en-queued according to the VC that they belong to the corresponding FIFO, and then the contents of the header, and the write

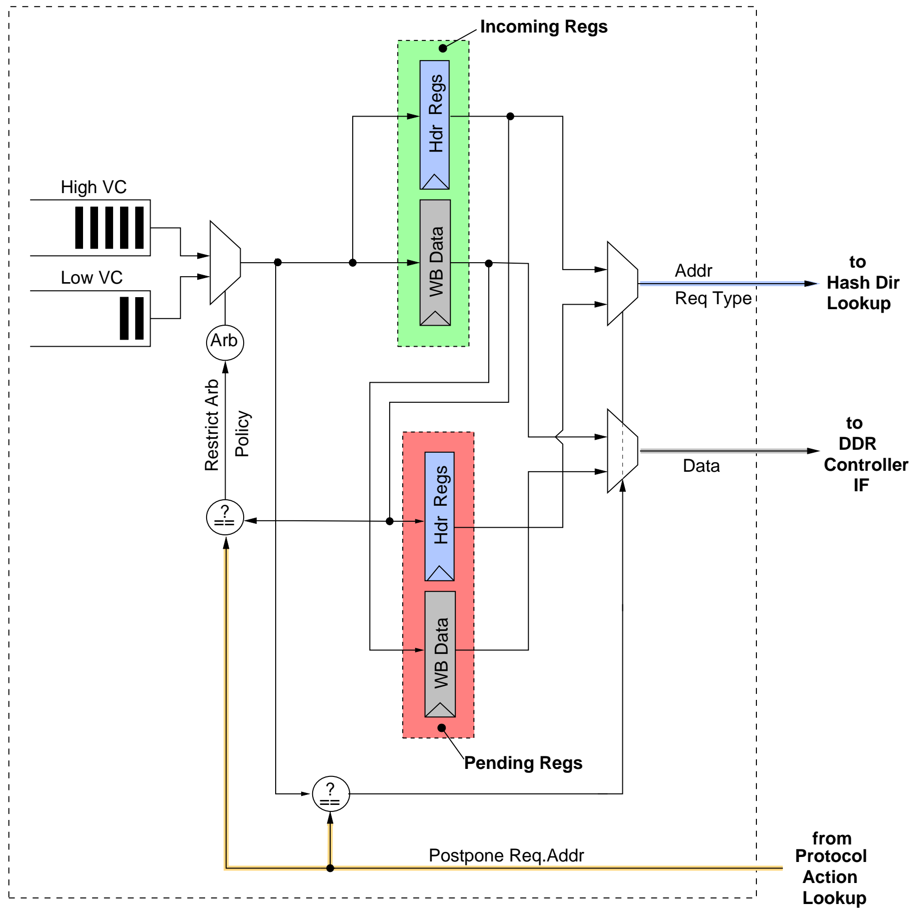


Figure 3.9: Directory Network Input Module

back data are stored to the Incoming registers, that hold the request type, the processor id, and the request address.

According to the protocol action look-up module decision, the arbitration policy is changed whenever an outstanding transaction exists. If the action lookup indicate to postpone a request address, , that occurs whenever we have an outstanding access on a specific address, the contents of the incoming register are saved to the pending register, and upon the reception that tag matches with the is postpone request address, then the request that is saved to the pending register is serviced.

Furthermore the NocIn module, forwards the data directly to the DDR interface that bypass the DDR access, when the pending register contains a GetX message and a write back message is received.

Moreover, the cost of the de-queue operation of the NoCIn module is 4 clock cycles (directory clock cycles), in case of request messages, replacement

messages, and invalidation acknowledgment, and 8 clock cycles for a write back message.

3.3.3 Directory Controller Hash Lookup Module

In this subsection we present the implementation of the directory hash lookup module, which is responsible for retrieving the state and the sharers bit vector for a memory block. As we described in Subsection 3.2.1, we use an off-chip SRAM in order to store the directory state that is needed. Therefore, a number of limitations exists as it concerns, the scalability of the directory design. That is the narrow width of the SRAM (32 bits + 4 parity) , the lack of associativity, as well as, the latency of the off-chip SRAM.

In order to be able to support a full-map bit vector with 16 processors, the size of the directory tag – which we use in order to find a directory entry – have to be minimized, so that the sharers bit vector, the state and the directory tag can be stored in a single SRAM word.

In Subsection 3.2.1, we described an abstract view of the hash organization, and use portion of the request address bits to index the SRAM array and then applied a hash function on the tag bits of the address in order to find the slice that this address corresponds to. For this implementation we use a simple XOR as the hash function, which allows us to reduce the directory size from 18 bits to 13 bits, based on the one to one property of the XOR function. Figure 3.10 shows how the hash value is calculated from the tag bits of the address. We divide the tag bits to three pendants (High, Medium, Low) and to an unused 3 bit quantity. In order to compute the hash value of the address, we simply xor these pendants and store only the medium and low pendants to a directory entry along with the three unused bits. Based on the xor properties, we are able to reconstruct the hight pendant when a directory entry is read by simply XORing again the directory tag bits – medium and low pendants only– with the hash value that it is computed from the requesting address.

In more detail, in order to find the corresponding directory entry of a request address, we index the SRAM in order to find the corresponding and we calculate the XOR in order to find the slice that resides. Because multiple address can have the same XOR, we use the directory tag in order to find the correct directory entry of an address. Once we have found the corresponding slice, the directory tag of the entry is read and the high pendant is reconstructed in order to compare it against the request address.

If the tag matches then the directory entry is said to be found, and the state fields as also the sharers bit vector is retrieved. Else, if the directory tag does not match with the request address tag, then we search inside the slice linearly until we find a valid entry that tag match or find an invalid directory entry.

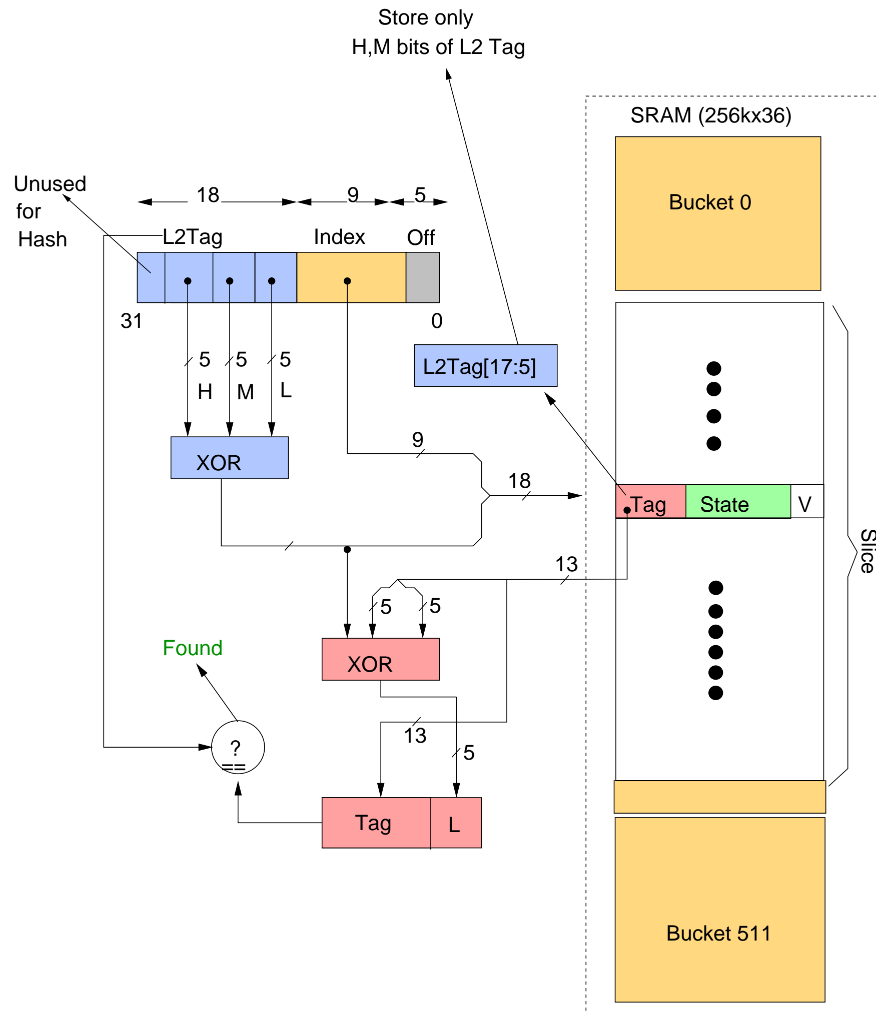


Figure 3.10: Directory Hash Lookup

In case of finding an invalid directory entry, we allocate a new directory entry for this address, by setting the valid bit of the directory entry and storing the directory tag as depicted in Figure 3.2.1.

For the deletion of directory entries, whenever that is necessary – due to cache evictions – the same steps for finding the entry are taken, with the exception that after we find the stale directory entry that needs to be removed, the search algorithm continues until it finds the last valid entry. Upon finding it, the contents of the stale directory entry are swapped with the contents of the last valid entry in order to rearrange the slice and directory entries to reside in consecutive locations inside the slice.

The format of the directory entry is shown in Figure 3.11. We use a full-map

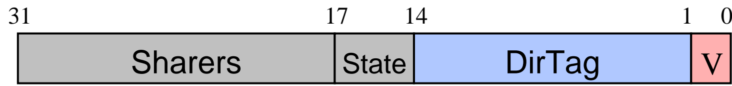


Figure 3.11: Directory Entry

bit vector of 16 bits to hold the sharers of memory block, 4 bits for the state of the memory block that is used from the protocol action look-up module and 13 bits for the directory tag, as well as a valid bit for marking directory entries.

Due to the source synchronous design of the directory hash look-up module that needs to interact with the off-chip SRAM of the board, in the next subsection we present the additional circuitry that is needed in order to generate the clock for the SRAM. It must be noted that the directory hash lookup module is clocked at 125 MHz and belongs to the same clock domain as the rest components of the directory controller, except from the NoC In/Out modules that are crossing clock domains, as the NoC operates at 62.5MHz.

3.3.3.1 Memory Clock Generation

Inside the FPGA, clocks are distributed using dedicated clock trees, which ensure that the clock signals reach every flip-flop relatively simultaneously. If the clock inputs of the ZBT memory is driven by the output of the FPGA, then the clock signal at the memory will be delayed by the sum of the propagation delay through the FPGA output pins and the propagation delay of the PCB trace.

To correct this skew at the memory devices, we need to drive the ZBT clock inputs with a phase-shifted version of the clock, so that the rising clock edge reaches the memory devices at the same time that it reaches all the registers in the FPGA. To generate this phase-shifted clock, a delay-locked loop (DLL) is used. DLL's are fundamentally analog components. There is no way to infer a DLL using Verilog code, so they must be instantiated. The Xilinx library component containing a DLL is the digital clock manager, or DCM.

The following is a high-level and incomplete description of the operation of DCM's. Essentially, a DCM takes a reference clock input signal on its ClkIn port, and outputs a delayed copy of that clock on its ClkOut output port. The ClkOut output is generally used to drive a clock distribution tree (a BUFG primitive in the Xilinx library). One output of the clock distribution tree should be used to drive the feedback input (ClkFb) of the DCM. The delay between the ClkIn and ClkOut ports on the DCM is automatically adjusted by a feedback loop until the ClkIn and ClkFb inputs are in phase. Once the phase difference between ClkIn and ClkFb has been minimized, the DCM is said to be "locked", and outputs of the clock distribution tree should be exactly in phase with the

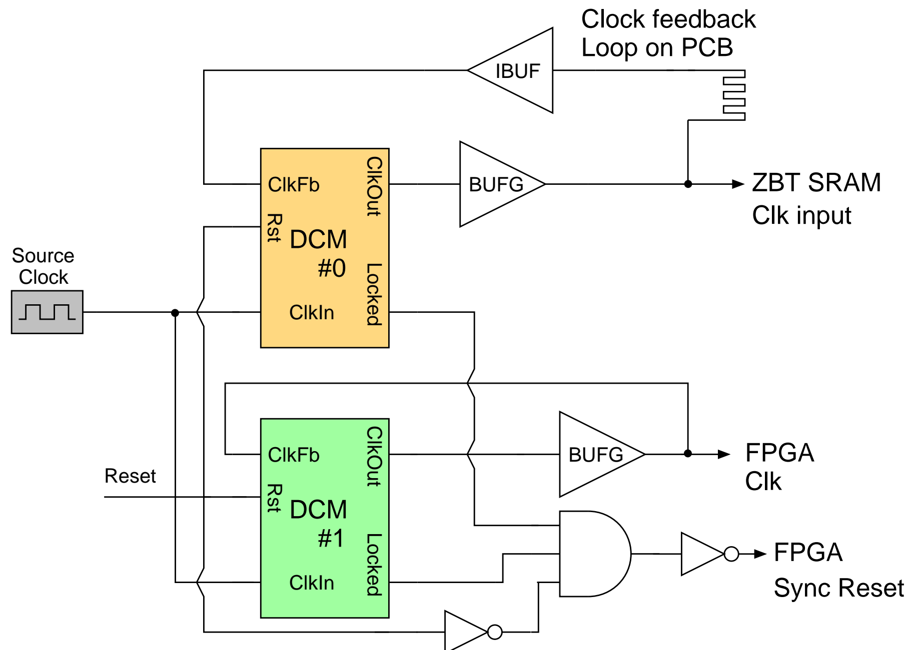


Figure 3.12: SRAM Memory Clock Generation

input clock signal. Effectively, the propagation delay through the clock distribution tree has been canceled, because the total propagation delay from the ClkIn pin of the DCM to the output of the clock distribution tree is exactly one clock period. Figure 3.12 shows how two Dc Ms can be used to ensure that the ZBT memory is clocked at exactly the same time as the FPGA flip-flops.

In the circuitry above, the lower DCM is used to ensure that the FPGA clock signal, which clocks all of the FPGA flip-flops, is in phase with the reference clock (Source Clock), in this example). The upper DCM is used to generate the de-skewed clock for the external ZBT memory. The feedback loop for this DCM includes a trace on the XUPV5 PCB. The propagation delay from the output of the upper DCM back to its ClkFB input should be almost exactly the same as the propagation delay from the DCM output to the SRAM.

The reset is used to ensure the DCM's lock properly when the FPGA finishes its configuration process. During configuration, the FPGA's I/O pins are all held in tristate. The Dc Ms are released from reset a few clock cycles before the global tristate signal is released. The upper DCM therefore attempts to lock without a feedback input. It is possible for the DCM to get stuck in a funny state, and never properly lock. To prevent this, a shift register is used to trigger a reset of both Dc Ms shortly after the entire FPGA configuration process is complete and the I/O pins have been enabled.

The locked output of a DCM signals that the feedback loop on the DCM's internal DLL has stabilized. The reset signal in the circuit above should be used as an active-high reset signal for any logic driven by FPGA Clock.

Note that, in order to minimize routing delay on the clock signals, this code utilizes input clock buffers (IBUFG) for both the source clock and feedback input.

3.3.3.2 Directory look-up latency

For the directory controller hash look-up module a ZBT SRAM controller is implemented. Due to the 2-cycle latency of the SRAM, we pipeline the accesses to the SRAM. Therefore, assuming no collisions – that is a directory entry would be found at once, without searching further inside a slice – the latency of the look-up operation is 6 clock cycles, with each collision costing 1 additional clock cycle. For allocation of a new directory entry the latency is 4 clock cycles, because we can issue the writes without waiting the completion from the controller. In average, each slice is underutilized as expected, and due to the grouping of cache lines of the same together, the probability of a collision is very small, that results to the minimum latency of 6 clock cycles.

The main drawback of the proposed hashing scheme is that in a cache eviction, the directory line may have to be evicted from the corresponding slice. Furthermore the valid directory lines of the slice must be in consecutive locations because the directory search algorithm looks for the first invalid line to allocate a new entry. When deleting an existing directory entry the directory entry allocator must rearrange the entries in the slice in such way, that no gap will exist among directory entries.

For that reason, the remove operation, after it finds the directory entry to remove, it then searches for the last valid entry in the slice and swap it with the removed one, in order to rearrange the slice, and have the directory entries in consecutive addresses. This results to an additional overhead for the directory look-up operation, but not frequently. This overhead is equal to the number of directory entries that exist inside a slice at a given time plus an additional read-modify operation for swapping the state directory entry with the last valid entry.

The total number of accesses that are required for the worst case for a remove operation is 6 clock cycles for finding a valid entry (in the worst case scenario, the slice must be full, and the entry to be removed must be the first entry inside the slice). In such case, the remove operation must issue a read for all the remaining entries (15 in this example) and issue a write to the address of the removed entry with the data that are read from the last entry inside the slice.

3.3.3.3 Increasing the Number of Cores - Limitations

In Figure 3.10, the directory hash lookup module is presented for a 4-core system with four way associative L2 caches. In order to support more than four cores, and take advantage of the multi-board configuration of the prototype, we can have multiple directories interleaved with the most significant bits of the address, and reduce the size of the xor accordingly, in order the slice to have size – in terms of directory entries – equal to the aggregate associativity of the systems caches in the coherence domain. Although the implementation is targeted for a multi-board configuration (4 boards), only a four core system is tested. Furthermore, reducing the XOR width, increases the collisions, thus the latency of the look-up operation and limits the implementation due to the narrow width of the SRAM, as we need to increase the size of the directory tag.

3.3.4 Directory Controller Action Lookup Module

Apart from the protocol of the L2 caches, directory controller must execute the directory side of the protocol to respond to protocol messages that are exchanged between the directory and the caches. The protocol states, except from the Modified, Shared, Invalid include a number of transient states that are used to implement the blocking property of the directory controller, and handle various corner cases as described in Subsection 3.2.2.

In general, coherence transactions, are categorized to those that does not need a specific response from a cache, and are said to have finished when the associated state is one of the stable states of the protocol. The other category of coherence transactions include those that need a response from a cache in order to transition to a stable state.

As it concerns the protocol actions, these are based on the message type of a packet received, as also to the state and the sharing status of a specific address. Table 3.5 shows the protocol transition table and the actions it indicates, for each combination of message type and state.

Similar to the cache controller transition tables each entry contains an <action/next state> tuple. When the current state of a block corresponds to the row of the entry and the next event corresponds to the column of the entry, then the specified action is performed and the state of the block is changed to the specified new state. If only a next state is listed, then no action is required. If no new state is listed, the state remains unchanged. Impossible cases are marked with "–" means no action or state change is required.

The protocol action lookup inputs, except from the state, include the sharing status and the invalidation acknowledgments counter, in order to compute the actions and the new state. Specifically AckCount keeps the number of the acknowledgments that the controller has received from the time it has issued

State	**Data**	**GetS**	**GetX**	**Repl**	**InvAck**
Invalid	-	DtRd SharerAdd, Response / **Shared**	DtRd MarkOwner Response / **Modified**	-	
Shared	-	DtRd Response SharerAdd / **Shared**	Inv, S2M_Stall MarkOwner / **S2M**	*LastSharer* SharersClear / **Invalid** Inv SharersDel / **S2I**	-
Modified	DtWr / **Invalid**	Inv M2S_stall, MarkSharer / **M2S**	Inv M2M_stall MarkOwner / **M2M**		
S2M	- / **S2M**	- / **S2M**	- / **S2M**	*~AckCount* SharerDel AckCount– / **S2M** *AckCount* SharerDel SharerSwap DtRd Response / **Modified**	*~AckCount* SharerDel AckCount– / **S2M** *AckCount* SharerDel SharerSwap DtRd Response / **Modified**
M2S	PcktDtRd Response SharerSwap Response / **Shared**	- / **M2S**	- / **M2S**	-	
M2M	PcktDtRd Response SharerSwap Response / **Modified**	- / **M2M**	- / **M2M**		
S2I	- / **S2I**	*&AckCount* SharerAdd DtRd Response / **Sharer** *~AckCount* S2I_stall / **S2M**	*AckCount* SharerDel SharerSwap DtRd Response / **Modified** *~&AckCount* MarkSharer S2I_stall / **S2M**	*~AckCount* SharerDelAckCount– / **S2I** *Ack-Count*SharerDel/ **Invalid**	*~AckCount* SharerDel AckCount– / **S2I** *AckCount* SharerDel / **Invalid**

Table 3.5: Directory Protocol States

an invalidation request; to the current holders of a cache line. This counter is used in to two transient states, S2M and S2I, in order to finalize a coherence transaction and transition to a stable state, as their name implies.

Other actions shown in the transition table include the invalidation requests to the current sharers or to the current owner (Inv action), DtRd/DtWr action which is used to read or write the data from DRAM, as also, PcktDtRd action which bypass the DRAM and forwards the data that was sent to the directory controller due to an invalidation or a replacement of modified line. Subsequently, the response action is used to sent the data back to the requesting cache. Sharers manipulation actions also are used to describe the operations that must be done to the sharing bit vector.

In case of a cache line in Shared state, the possible actions concerning the sharing status is to add a sharer (SharerAdd action) or to delete an existing sharer (SharerDel action). The former, results from a GetS message, while the latter is due to an eviction event at a cache. If the requesting cache line has multiple sharers, invalidation's are issued and the acknowledge counter is set to the number of acknowledgments it awaits. When all the acknowledgments are gathered, the associated directory entry is removed. Otherwise if a GetS or a GetX for this address exists in t then protocol state is updated to S2M or Shared accordingly.

When in Modified state, if a GetS or GetX request is received, an invalidation request is issued to the owner to issue a write-back and the Share or NewOwner is marked, in order to respond with the data, when these arrive at the directory. In both cases when the write-back data arrive, the Sharer vector is swapped with the marked one and is written to its associated directory entry.

3.3.5 Directory Controller NoC Output Module

In this subsection we present the internal structure of the directory outgoing network interface. The NoC output is responsible for the packet generation of the responses that the action lookup module indicate. The NoC Output module (NoCOut) generates either invalidation packets according to the request mask that is retrieved from the Hash Lookup module or Fill packets to a requesting processor.

In more detail, the internal structure of the NoC Out module is depicted in Figure 3.13. According to the request type, which can be either Fill or Invalidate, the NoC Output module generates the packet headers either by reading the NocIn registers to calculate the response address, or by using also the priority enforcer to issue multiple invalidation packets to the current sharers of a cache block.

Invalidation packets use the medium VC in contrast with the Fill packets,

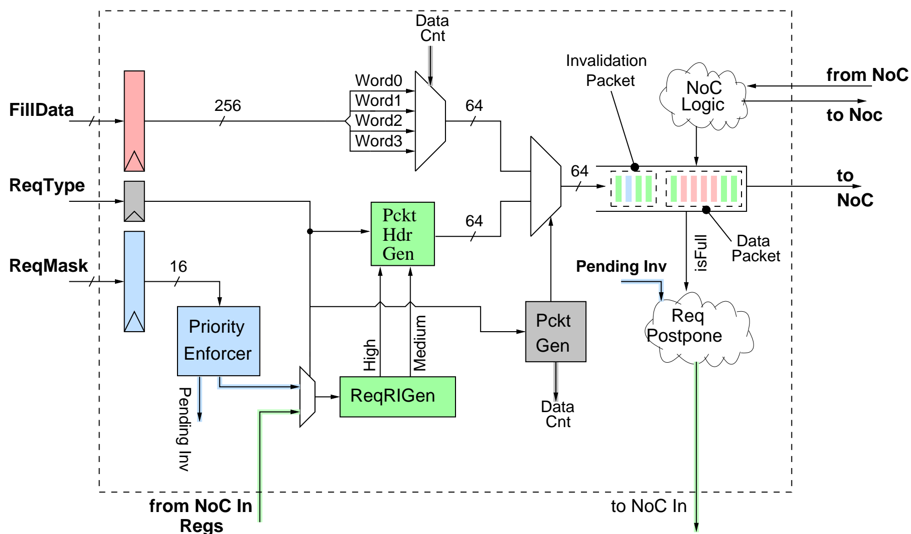


Figure 3.13: NoC Output Datapath

and until all the invalidation packets are en-queued to the outgoing FIFO of the NoC Out module, subsequent requests are blocked, in order to sink all the invalidation packets that are required for the completion of a coherence transaction. For that reason the NoC In module can postpone the service of requests for the incoming network. Moreover, similar to the NocIn module, the FIFO that resides in the Noc Out module is used also for crossing clock domains.

The latency for a fill packet is 7 clock cycles plus the additional latency that is due to the grant logic of the NoC which accounts 1 cycle assuming that the NocOut module is granted. Invalidation packets on the other hand have a latency of 4 clock cycles plus one clock cycle for calculating the next invalidation based on the request mask if multiple invalidation's have to be generated. Again the cost of the NoC grant must be accounted.

Chapter 4

Evaluation

In this chapter we discuss the performance of the protocol, and present the detailed hardware cost for the coherent system design compared to the baseline system, in terms of logic gates.

4.1 Target FPGA

Device	Virtex-5 Slices	Block RAM Blocks	CMTs	GTP
XC5VLX110T	17,280	148 (5.328Kb)	6	16

Table 4.1: Virtex-5 LX110T

The entire system is designed for and implemented on a Virtex-5 FPGA, embedded in a Xilinx University Program board (XUPV5). The size of the FPGA is 17K (in slices) and the speedgrade is -1. Each slice contains four 6-input look-up tables (LUT's) and four flip flops. Also XUPV5 has an on-board 9Mb ZBT synchronous SRAM. The memory is organized as 256K x 36 bits providing a 32-bit data bus, with support for four parity bits. The external SRAM memory is used to store the directory state, as described in the previous chapter.

The resources available for the specific FPGA on the Virtex-5 are shown in Table 4.1.

4.2 Hardware Cost

In this section, the logic costs for the cache coherent system is presented and discussed in contrast to the non-coherent system. The additional logic introduced by the coherence protocol is to a great extent due to the addition of the

Resources	Occupied	Available	%
FFs	30,172	69,120	43
LUTs	48,348	69,120	69
BRAM	135	148	91
IOBs	184	640	28
BUFGs	8	32	25
DCMs	2	12	16
PLL_ADV	1	6	16

(a) Coherent

Resources	Occupied	Available	%
FFs	29,798	69,120	43
LUTs	44,879	69,120	65
BRAM	132	148	90
IOBs	128	640	20
BUFGs	8	32	25
DCMs	0	12	0
PLL_ADV	1	6	16

(b) Non Coherent

Table 4.2: System Resource Utilization

directory protocol controller. Tables 4.2b and 4.2a present the FPGA resources used for the system with and without the coherent caches and the directory controller. The coherent design has 4% increase in terms of LUT's and a 3% in flip-flops. The directory controller increase the number of BRAM blocks used compared to the non-coherent design, since it utilizes three BRAM's for its network interfaces FIFOs and one BRAM block for the DDR interface.

Moreover, the additional LUT's, are due to the directory controller and the additional control logic that is added to the L2 cache controller for the coherence protocol. Also, the difference in the IOB count, refers to the ZBT SRAM controller that we implement for the Directory hash look-up module. Finally, the two extra DCM's are also used by the SRAM control, to de-skew the clock that is sent to the synchronous SRAM, with the one that clocks the flip-flops inside the FPGA, which we have described in subsection 3.3.3.1. In the following subsections, we detail the added hardware costs for the directory controller – excluding directory memory overhead – and the L2 cache controller.

4.2.1 Directory Controller Resources

The major overhead in logic, in compared to the coherent system, is due to the directory controller. In this subsection we analyze the directory controller internal structure. A detailed breakdown of the directory logic in terms of flip

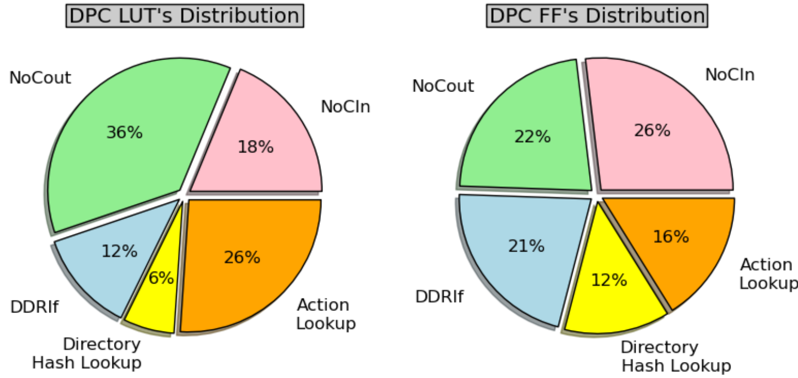


Figure 4.1: Directory LUT's and FF's utilization

Directory Controller	**LUTs**	**FFs**
Incoming Network	347	477
Outgoing Network	659	399
Directory Hash Lookup	128	238
DDR Interface	231	383
Action Lookup	475	277
	1840	**1774**

Table 4.3: Directory Hardware Resources breakdown

flops and LUT's is presented in Table 4.3. Also, the flip flop breakdown of the directory controller is shown in the table above, where the largest flip flop counts are in the Outgoing network interface and the DDR controller interface. Furthermore, the outgoing NI is the most complex block in the directory controller, because of the packet generation, as it has to generate invalidation messages as well as fill messages, and interact with the DRAM interface, and the protocol action look-up. The incoming NI accounts for 15% over the directory controller total LUT count, as shown in Figure 4.1, mostly utilized to implement the blocking property of the controller.

Moreover, a large portion of the LUT count is due to the glue logic of the directory controller, that connects the various modules together. In this count, the FSM's for handling and executing the various actions that are imposed from the action look-up module are taken into account. Also the sharers manipulation logic, that concern the deletion or addition of a sharer in the sharers bit vector of a directory entry are included, as well as, the pipeline registers for each stage of the controller.

Furthermore, the logic required for implementing the hash look-up module and the ZBT SRAM controller account 6% over the total LUT count of the

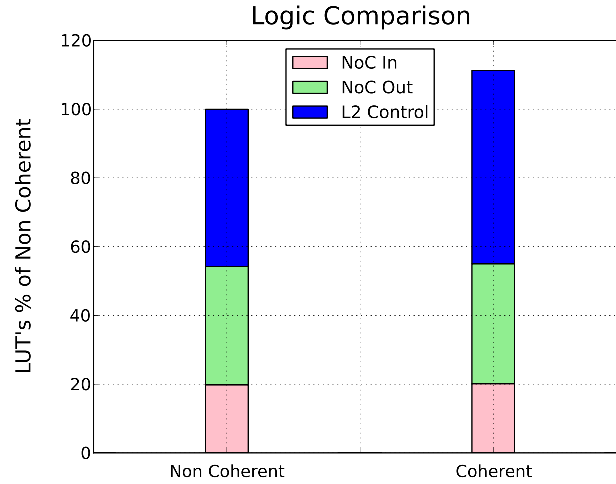


Figure 4.2: LUT's Comparison of Coherent and Baseline Design

directory controller, as shown in Figure 4.1.

Finally, the DDR controller interface module, includes the additional port that is added to the Xilinx MPMC DRAM controller (approx. 200 LUTs/port), in order to differentiate coherence requests from other types of accesses to the existing DRAM interface which has DMA capabilities.

4.2.2 L2 Cache Hardware Resources

The main increase in LUT's for the L2 cache is due to the additional logic introduced to the control logic of the L2 cache. Changes made to the incoming and outgoing network interface of the cache controller account only 1.3% and 1.25% of their total size respectively.

The major overhead in the L2 cache controller is due the additional states that are added for the coherence protocol, implemented in the Action Lookup blocks of the L2 cache (Figure3.5). The L2 controller overhead is blown to 23%, because the Action Look-up of the L2 cache is replicated in a per way basis, in order actions lookup for per way actions fo the protocol to proceed in parallel with tag matching. Consequently the added logic to the L2 controller is only one fourth of the measured one. The detailed area increase is shown in Table 4.2.2. Moreover, Figure 4.2.2 shows the percentage difference of the LUTs for the baseline system and the coherent system of the L2 cache.

Module	Non-Coherent	Coherent	Increase
NiIn	1149	1165	1.3 %
NiOut	1996	2021	1.25 %
L2Cntrl	2647	3259	23 %

Table 4.4: L2 Cache controller LUT's distribution

4.3 Performance

The next step of the evaluation procedure inspects the performance of coherent operations, and discusses the functionality of the various micro-benchmarks that we used in order to measure the performance of synchronization mechanisms and of a matrix multiplication test application. For the measurements that are shown below, we present the various latencies in clock cycles. Specifically, the two clock domains of the system are the processor clock domain (62.5MHz) and the directory clock domain (125MHz). The processor clock domains also includes the NoC that operates at the same frequency

4.3.1 Protocol Performance Metrics

An execution over an invalidation-based protocol has two important performance measures. The invalidation frequency (writes to shared blocks), and the number of invalidation mean size, which represents the number of invalidations needed for each exclusive request. Directory schemes are advantageous, compared to broadcast-based schemes, if the invalidation size is small and the invalidation frequency is significant. Data access patterns are important in understanding invalidation patterns and the latency of a coherence miss. The following access patterns are common and the rest of this subsection provides data to analyze the latency of these patterns.

- read-only: never written once they have been initialized; there are no invalidating writes, so data in this category is not an issue for directories;

- producer-consumer: the invalidation size is determined by how many consumers there have been each time the producer writes the value.

- migratory: data migrates from one processor to another, being written and usually read by each processor; ex: global sum, on which each processor adds its local sum;

- irregular read-write: irregular or unpredictable read and write access patterns. These usually lead to wide-ranging invalidation size distributions

Operation Type	Latency
Upgrade Request *(S2M)*	40
Dir Data Fwd *(M2M)*	30
Downgrade	30
Load/Store Miss (*I2S,I2M,S2S)*	64

Table 4.5: Coherence Transactions Latency

4.3.1.1 Directory Controller Latency

In this subsection we summarize the latency of specific operations which are presented in Table 4.5. The latencies are measured in directory clock cycles, and refer to the directory latency. Specifically, the measurements shown are calculated from the first en-queue of a coherence request to the directory controller until the first word of the response to be injected into the network. The following operation types that we present correspond to the following scenarios :

1. Upgrade Request: A cache line is in shared state with only one sharer, and a cache issues an upgrade request (GetX request to a Shared cache line).

2. Directory Data Forwarding (M2M): A cache line is in Modified state, and another cache wants to write the specific cache line. The measured time starts on data write back reception, until the time that the fill response is injected into the network.

3. Downgrade: A cache intents to read a cache line, that is in Modified state in another cache (GetS request to Modified cache line). The measured time is the same as in 2.

4. Load/Store Miss: The specific cache line does not reside in any other cache, thus, needs to be fetched from DRAM (GetS, GetX to Invalid cache line).

In directory data forwarding and downgrade cases, the data are forwarded from the NocIn module to the NocOut module of the directory controller, thus, the latency of these operations is less than that of the other types, since the data are provided along with the request (write back message). For a GetS or GetX request to a cache line in Invalid state, the latency is increased, as the data need to be fetched from the DRAM and the directory hash look-up module, has to allocate a new directory entry. In the next subsection we present a detailed

timing diagram for a read miss operation, where the various latencies of the directory controller and the cache controller are shown.

4.3.1.2 Read Miss Latency Example

In Figure 4.3, we present a timing diagram for a read miss operation, that details the time spent in each module of the system. Processor signals and cache signals are shown in red. Directory signals are shown in blue, except from the NoC interface signals of the directory controller that are shown in orange. The two clocks shown, correspond to the two clock domains, processor (top) and directory (middle).

Furthermore, the markers that are shown in red correspond to the processor and the cache controller events, that are (a) the load from the processor, (b) the start of the en-queue to the NiOut joblist of the cache controller, (c) the start of the de-queue operation from NiIn, and (d) the reply to the processor with the data. Markers shown in green, refer to directory controller events, that are (a) the end of the de-queue operation from the directory's NoCIn and the start of the en-queueue to the directory's NocOut module.

In more detail, we assume that a processor issues a load for a cache block (Proc Req marker), that does not reside in any other caches in the system. After the processor read request reaches L1 cache, in 2 clock cycles, a GetS message descriptor is en-queued to the joblist of the L2 cache (NI Out Enq marker). Later the GetS request is de-queued from the directory controller (DirIn Deq marker) after 16 cycles of the processor clock. Afterwards, the directory hash look-up module retrieves the state (in this case the latency of the look-up is 6 cycles of the directory clock), and then we issue a request to the DDR If (o_DDR_RdReq signal). When the last word of the data is received and the state of the cache line is updated from the directory hash look-up module (i_LookupHit signal), the reply with the data is en-queued to the NoCOut module of the directory controller (DirOut Deq marker).

Finally, the fill message, with the data is de-queued from the NiIn of the requesting cache (Deq marker) after 7 clock cycles of the processor clock. Then the data are returned to the processor (Proc Ready marker) after 9 clock cycles. The resulting miss latency for the read miss is shown in the top-left corner (which shows the delta's for the markers), and it is in this case 58 cycles of the processor clock.

4.3.2 Micro-benchmarks

In order to run the micro-benchmarks that will be described in this section ticket locks are implemented – using an atomic fetch and add operation – and

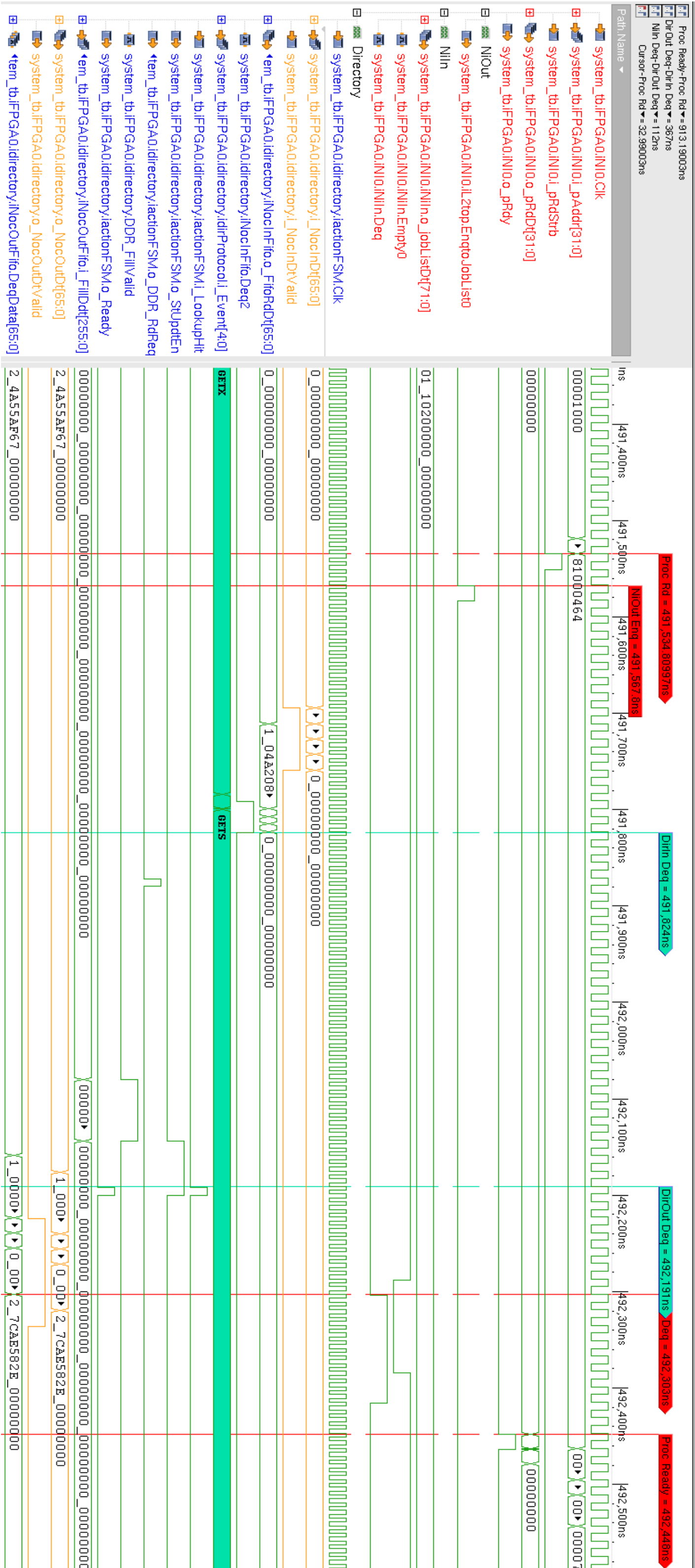


Figure 4.3: Read Miss Timing Diagram

a centralized sense reversal barrier , as well as a matrix multiplication, and a producer consumer benchmark.

Furthermore, we implement memory barriers, that ensure the proper order of writes (as it concerns the processor that initiates the writes), and are used in order to provide to the programmer a weak consistency model.

4.3.2.1 Shared Counter

The first program written for the system is the shared counter micro-benchmark. The four processors using a lock structure try to gain access to the shared variable that corresponds to the counter. Once the lock has been acquired by a processor the variable is increased by one and then the lock is released.

Once the lock is released, the other processors can acquire it to perform the same action. Using ticket locks, the system behavior results to an alternation of the four processors on having access to the shared variable.

In example, traffic generated by two processors when running this program corresponds to the transfer of the lock structure and the shared variable from one cache to another. The first action taken by the processor to get the lock, is a write access to its variable in the lock-structure. This corresponds to a GetX message to be sent to the directory controller, and the corresponding cache block to enter the cache. The same processor reads constantly the variable within the lock, that corresponds to the current holder of the lock, in order to be notified for its release. In this case, no traffic is generated, since all the read accesses hit in the cache.

The other processor that holds the lock, eventually release it, by receiving an invalidation message and issue a write back to the directory. The next time a processor tries to access the lock, a GetX message is generated again. At that time the processor that has generated the invalidation, has access to its critical section. Its actions are to read the shared counter (generation of a GetS message) and to write to it the new value (generation of an invalidation message). Finally, it releases the lock by clearing its value in the lock structure.

These set of actions is continuously repeated until the end of the program. In order to measure the performance of the program, its length of execution is measured in processor cycles. Such a program is rather meaningless as far as its functionality is concerned, however, it provides a good insight of the cost of the shared memory synchronization. Table 4.6 present the execution time of the shared counter for 10K increments.

#Processors	Execution Time
1	2927810
2	4280920
4	5016070

Table 4.6: Shared Counter Slowdown

CPU	1	2	3	4
Cycles	90	101	114	124

Table 4.7: Atomic Fetch and Add Latency

4.3.2.2 Atomic Fetch and Add

A basic operation that we need in order to implement various synchronization primitives such as locks and barriers is the atomic fetch and add operation that is implemented. We run a simple micro-benchmark in order to estimate the latency of the fetch-and-add operation using a simple ticket lock where each processor tries to get a ticket number using fetch-and-add on a shared variable that represents the lock structure.

The critical sections were empty, in order only study the performance of the atomic operation without accounting in the delay of possible coherent misses that may be produced from the shared variable inside the critical section.

In case of a contented lock, where four processors try to enter the critical section, the atomic operation latency is increased because other processors invalidate the specific cache line, in order to increment the atomic variable. In such case, the directory must receive invalidation acknowledgments from all the other processors, and respond with the data.

Table 4.7 summarizes the performance of the atomic operation, measured in processor clock cycles. For a single processor doing atomic operations the latency is 90 clock cycles for fetching the cache block, and after that, the fetch and add operations always hit in the L1 cache – because no other processor would generate an invalidation.

In case of two processors, invalidation's are produced, because each processors try to increment the variable, and according to the time that the invalidation arrives at the current holder of the cache line, the invalidation is buffered and afterwards, the cache controller responds with the data that has been written. Then the data have to be sent back to the requesting cache.

For four processors contending to increment the shared variable, the latency increases, as expected due to the number of processors that need to be invalidated.

Also it must be taken in account the latency of 4 clock cycles, due to the L2

CPU	1	2	3	4
Cycles	186	217	291	322

(a) Ticket Locks Latency

CPU	1	2	3	4
Cycles	101	173	184	237

(b) Centralized Sense Reversal Barriers Latency

Table 4.8: Synchronization Primitives Latencies

pipeline that have to be paid in order to initiate an atomic operation as described in Section 3.3 – which corresponds to the L2 memory mapped register that has to be set in order to initiate an atomic operation.

Tables 4.8a and 4.8b shows the corresponding latencies for contented ticket locks and for the centralized sense reversal barriers that are implemented. For the specific micro-benchmarks we run 10K iterations and measure the lock/unlock pair and barrier latency. As expected, when we increase the number of cores that content for the lock, the latency is increased due to the additional invalidation messages that will be generated because of the additional contender for the lock. This is similar to the barrier micro benchmark.

For implementing barriers, a fetch and decrement operation is needed in order to decrease the count, whenever a processor reach the barrier. The messages that are generated for the barrier is a GetS message for reading and polling on the sense variable in order to wait the other processors to enter the barrier, and wait until a GetX message is sent to the directory from a processor, and change the state of the sense variable from Shared to Modified. Then, again a GetS changes the state of the sense variable to Shared from Modified. This alternation of events happens until the count reach zero.

4.3.2.3 Producer Consumer

The second program implemented generates a producer-consumer communication traffic pattern between the four processors. The master processor is responsible for generating new data and placing them in a shared buffer, and it is responsible also for spawning the consumer threads.

The other processors consume these data by reading them from the buffer with a random think time and by consuming only a random portion of the generated data each time. The buffer lies in shared address space and is organized as a FIFO.

Every time the master processor generates new data, it appends them at the end of the queue updating atomically the tail pointer. The consumer processor

#Consumers	Execution Time (cc)	Slowdown
3	5322819	1.57
2	5804732	1.72
1	3376300	1

Table 4.9: Producer Consumer Latency

#Processors	Execution Time (cc)	Speedup
1	424950	1
2	216759	1.96
4	113586	3.74

Table 4.10: Matrix Multiplication Speedup

retrieves new data from the head of the queue. Head and tail pointers lie in subsequent cache lines, to avoid using a shared lock.

Each time the producer processor wants to add a new word to the queue, it first writes the data to the memory location pointed by the tail pointer and then increments the tail pointer.

On the other side, the consumer, which constantly reads the tail pointer, checks the availability of generated data, by comparing the head and tail pointers. If this amount of data is present in the shared buffer, then it will also be de-queued. If not, the consumer will start over. Head pointer is updated on each de-queue of a word. Table 4.9 shows the execution time of the application in processor clock cycles when run with a different number of consumers for 10k items.

The slowdown as compared to one to one communication via the producer and the consumer is due to the excessive synchronization between the head and tail pointers. A relative speedup against the two consumers still exists, but it is insignificant.

4.3.2.4 Matrix Multiplication

A simple matrix multiplication algorithm is developed in order to measure the speed-up gained when we increase the number of cores. As expected, it is found that the speed-up that is gained reaches 3.74% in contrast with the sequential version of the algorithm.

The lack of synchronization leads to such an improvement because in contrast with the shared counter micro-benchmarks that issue exclusive requests, a sharing pattern is produced avoiding the frequent invalidation's of the other caches in the system, as opposed to the shared counter micro-benchmark. Table 4.10 shows the execution times for a 32x32 (4 bytes / element) matrix multipli-

cation. The execution time is measured in processor cycles.

Chapter 5

Conclusions

In CMP architectures, the cache-coherence protocol is a key component since it can add requirements of area or power consumption to the overall system, and also increase the complexity and the verification effort, therefore, could restrict severely its area and power scalability. Although directory-based cache-coherence protocols are the best choice when designing shared memory many-core CMPs, the memory overhead introduced by the directory structure may not scale with the number of cores, when the coherence information is kept by using a full-map sharing code. Proposals are made for reducing the directory memory overhead, organizing it as a cache (sparse directories), that reduces the height of the directory at the cost of a high associativity degree, or by reducing the width of the directory using compressed sharing codes or coarse grain sharing vectors.

In this work, we demonstrate a sparse directory organization based on hashing, which is expected to be more scalable in terms of power than previous implementations that require a high associativity degree. The rule to achieve this scalability is to organize the directory as a hash table, group the directory entries that correspond to the same cache index together, and use hashing in order to find the corresponding entry of an address. Therefore, the hashing-based directory can support the high associativity demands of duplicate tag based or sparse directory organizations, which would require the aggregate associativity of the system's caches, while requiring less directory accesses in the average case.

This thesis demonstrates the design and implementation of a directory based coherence protocol and proposes a hash directory organization that can support up to 16 processors in our baseline system. We merge coherence with the explicit communication mechanisms of the baseline system, and find that the logic overhead is only 4% for a 4-core CMP. Moreover, we verify the correctness of

our protocol and evaluate its performance using simple micro-benchmarks, such as ticket locks and barriers, and a matrix multiplication test application.

5.1 Future Work

Some future objectives of this work is to :

- Implement a different version of the hash directory using different hash functions.
- Implement the non-blocking property of the directory controller.
- Test the coherence protocol using multiple FPGA boards.
- Evaluate the protocol using the SPLASH-2 benchmark suite.

[1] M. Katevenis, V. Papaefstathiou, S. Kavadias, D. Pnevmatikatos, F. Silla, and D. Nikolopoulos, "Explicit communication and synchronization in sarc," *IEEE Micro*, vol. 30, pp. 30–41, September 2010.

[2] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[3] M. Chaudhuri and M. Heinrich, "The impact of negative acknowledgments in shared memory scientific applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, pp. 134–150, February 2004.

[4] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The mit alewife machine: architecture and performance," *SIGARCH Comput. Archit. News*, vol. 23, pp. 2–13, May 1995.

[5] S. V. Adve, *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, Madison, WI, USA, 1993. UMI Order No. GAX94-07354.

[6] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. Comput.*, vol. 27, pp. 1112–1118, December 1978.

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

[8] A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph, "High-throughput coherence control and hardware messaging in everest," *IBM J. Res. Dev.*, vol. 45, pp. 229–243, March 2001.

[9] S. S. Mukherjee and M. D. Hill, "An evaluation of directory protocols for medium-scale shared-memory multiprocessors," in *Proceedings of the 8th international conference on Supercomputing*, ICS '94, (New York, NY, USA), pp. 64–74, ACM, 1994.

[10] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The stanford flash multiprocessor," in *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, (Los Alamitos, CA, USA), pp. 302–313, IEEE Computer Society Press, 1994.

[11] J. Laudon and D. Lenoski, "The sgi origin: a ccnuma highly scalable server," in *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, (New York, NY, USA), pp. 241–251, ACM, 1997.

[12] B. W. O'Krafka and A. R. Newton, "An empirical evaluation of two memory-efficient directory methods," *SIGARCH Comput. Archit. News*, vol. 18, pp. 138–147, May 1990.

[13] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proceedings of the 15th Annual International Symposium on Computer architecture*, ISCA '88, (Los Alamitos, CA, USA), pp. 280–298, IEEE Computer Society Press, 1988.

[14] R. Thekkath, A. P. Singh, J. P. Singh, S. John, and J. L. Hennessy, "An evaluation of a commercial cc-numa architecture: The convex exemplar spp1200," in *Proceedings of the 11th International Symposium on Parallel Processing*, IPPS '97, (Washington, DC, USA), pp. 8–17, IEEE Computer Society, 1997.

[15] A. Gupta, W. dietrich Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *In International Conference on Parallel Processing*, pp. 312–321, 1990.

[16] D. B. Gustavson, "The scalable coherent interface and related standards projects," *IEEE Micro*, vol. 12, pp. 10–22, January 1992.

[17] Y. Chang and L. N. Bhuyan, "An efficient hybrid cache coherence protocol for shared memory multiprocessors," *IEEE Trans. Computers*, vol. 48, pp. 352–360, 1999.

[18] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann, Aug. 1998.

[19] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Limitless directories: A scalable cache coherence scheme," *SIGPLAN Not.*, vol. 26, pp. 224–234, April 1991.

[20] R. Simoni and M. Horowitz, "Dynamic pointer allocation for scalable cache coherence directories," in *In International Symposium on Shared Memory Multiprocessing*, pp. 72–81, IPS Press, 1991.

[21] R. Simoni, "Cache coherence directories for scalable multiprocessors," tech. rep., 1992.

[22] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The stanford dash multiprocessor," *Computer*, vol. 25, pp. 63–79, March 1992.