A SCALABLE CACHE COHERENT MEMORY ARCHITECTURE FOR RECONFIGURABLE COMPUTING

Master of Science Thesis

Gizem YAĞAN

Eskişehir 2019

A SCALABLE CACHE COHERENT MEMORY ARCHITECTURE FOR RECONFIGURABLE COMPUTING

Gizem YAĞAN

MASTER OF SCIENCE THESIS

Electrical and Electronics Engineering Program

Supervisor: Assist. Prof. Dr. İsmail San

Eskişehir

Eskişehir Technical University

Institute of Graduate Programs

June 2019

FINAL APPROVAL FOR THESIS

This thesis titled “A Scalable Cache Coherent Memory Architecture for Reconfigurable Computing” has been prepared and submitted by Gizem YAĞAN in partial fullfillment of the requirements in “Eskişehir Technical University Directive on Graduate Education and Examination” for the Degree of Master of Science in Electrical and Electronics Engineering Department has been examined and approved on 25/06/2019.

Committee Members	**Title, Name and Surname**	**Signature**
Member (Supervisor)	: Assist. Prof. Dr. İsmail SAN	…….
Member	: Prof. Dr. Atakan DOĞAN	…….
Member	: Assist. Prof. Dr. Nihat ADAR	…….

…….....

Director of Institute of Graduate Programs

ÖZET

YENİDEN YAPILANDIRILABİLİR HESAPLAMA İÇİN ÖLÇEKLENEBİLİR ÖNBELLEK-TUTARLI BELLEK MİMARİSİ

Gizem YAĞAN
Elektrik-Elektronik Mühendisliği Anabilim Dalı
Eskişehir Teknik Üniversitesi, Lisansüstü Eğitim Enstitüsü, Haziran 2019
Danışman: Dr. Öğr. Üyesi İsmail SAN

Alanda programlanabilir kapı dizileri, tekrar programlanabilme ve uygulamaya özgü verimli donanım tasarlama imkânı sunduğu için yüksek performanslı hesaplamada büyük bir potansiyele sahiptir. Ancak, algoritmalara özel donanım mimarilerini, tasarım süreçleri zor olan düşük seviye programlama dilleri ile tanımlamak gerekmektedir. Yakın zamanda yapılan araştırmalar, yüksek-seviye programlama dilleri ile verimli donanım tasarımı yapmayı mümkün kılmıştır. Yüksek-seviye sentezleme (YSS) derleyicileri, yazılım programlarını otomatik olarak kaydedici-transfer seviyesi tasarıma dönüştürerek programlama kolaylığı sağlar. Bu derleyiciler verilen algoritma için verimli ve bağımsız veri yollarını ve sonlu durum makinelerini üretirken veriye ulaşımda tutarlı, verimli ve özel bir bellek mimarisine ihtiyaç duyar. Bu tezde, bir YSS derleyicisi için üretilen veri yollarını sürekli besleyecek, bekleme sürelerini kısaltacak ve verilerin tutarlı olmasını sağlayacak ölçeklenebilir önbellek-tutarlı bir bellek mimarisi önerilmiş ve Verilog dilinde gerçeklenmiştir. Dizin-tabanlı yazmada-güncelle protokolüne uyan bu bellek mimarisi, yeni bir tutarlılık protokolüne sahiptir. Derleyici tarafından belirlenen tutarlı önbelleklerin ve dizinlerin sayısı isteğe bağlıdır. Tutarlı önbellekler, farklı tutarlılık alanlarına ait olabilir ve dizin, tutarlılık trafiğini sadece aynı tutarlılık alanındaki önbellekler arasında yönetir. Derleyiciye entegre edilen protokolün, 51 temel referans uygulama için üretilen donanımlarda hatasız bir şekilde çalıştığı yazılım-donanım karşılaştırması ile doğrulandı. Bu testlerde, L2 önbelleklere bağlı olan 2 dizin yer alırken, gerçeklenen algoritmaya bağlı olarak değişen L1 tutarlı-önbelleklerin sayısı 2 ile 39 arasındadır. Modelin ölçeklenebilirliği ve performans potansiyeli gösterilmiştir.

Anahtar Sözcükler: Önbellek tutarlılığı, Dizin-tabanlı, Yazmada-güncelle, FPGA belleği

ABSTRACT

A SCALABLE CACHE COHERENT MEMORY ARCHITECTURE
FOR RECONFIGURABLE COMPUTING

Gizem YAĞAN
Electrical and Electronics Engineering Program
Eskişehir Technical University, Institute of Graduate Programs, June 2019
Supervisor: Assist. Prof. Dr. İsmail SAN

Field programmable gate arrays have significant potential for high performance computing since it provides reprogramming and application-specific efficient hardware design. However, application-specific hardware architectures are required to be defined by low level programming languages that have hard design processes. Recent researches allow efficient hardware design with high-level programming languages. High-level synthesis (HLS) compilers provide ease of programming by automatically converting software programs to register-transfer level design. These compilers require an efficient, coherent and special memory architecture on reaching data, while generating efficient and independent data paths, and finite state machines. In this thesis, a scalable cache coherent memory architecture that feeds the generated data paths constantly, shortens the latency time and ensures that the data is coherent, is proposed and implemented in Verilog language for an HLS compiler. This memory architecture following directory-based write-update protocol has a novel cache coherence protocol. Number of coherent caches and directories, specified by the compiler, are arbitrary. Coherent caches can belong to different coherence domains and the directory manages coherence traffic only between caches that are in same coherence domain. It is verified by software-hardware comparison that the protocol integrated to the compiler runs without error in hardware generated for 51 benchmarks. In these tests, there are 2 directories connected to L2 caches, while number of coherent L1 caches that varies depending on the implemented algorithm is in the range of 2 and 39. The scalability and performance potential of the model are demonstrated.

Keywords: Cache coherence, Directory-based, Write-update, FPGA memory

25/06/2019

STATEMENT OF COMPLIANCE WITH ETHICAL PRINCIPLES AND RULES

I hereby truthfully declare that this thesis is an original work prepared by me; that I have behaved in accordance with the scientific ethical principles and rules throughout the stages of preparation, data collection, analysis and presentation of my work; that I have cited the sources of all the data and information that could be obtained within the scope of this study, and included these sources in the references section; and that this study has been scanned for plagiarism with "scientific plagiarism detection program" used by Eskişehir Technical University, and that "it does not have any plagiarism" whatsoever. I also declare that, if a case contrary to my declaration is detected in my work at any time, I hereby express my consent to all the ethical and legal consequences that are involved.

.....

Gizem YAĞAN

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude towards my supervisor Assist. Prof. Dr. İsmail San for his support, patience and encouragement. His guidance always helped me during this thesis. I hope that we will continue to work together and to be part of other quality studies.

I would like to acknowledge Prof. Dr. Atakan Doğan for his support during this thesis. His contributions to this work are significant. I also would like to thank him and Assist. Prof. Dr. Nihat Adar for serving on my committee. Their advices will be guidance to me for rest of academic life.

I would like to thank all members of Erendiz Superbilgisayar Company and especially to Dr. Kemal Ebcioğlu for his valuable ideas that form the basis of this thesis.

I especially thank to my beloved husband Ali Can Yağan for his support during this thesis. I am grateful to him for his suggestions and proof reading. I also would like to thank my parents Fatma Gülmez and Orhan Gülmez, brother Okan Gülmez and grandparents Zeynep Gülmez and Mehmet Gülmez for their support throughout my life.

Gizem Yağan

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Page

Page

LIST OF TABLES

Page

ABBREVIATIONS

BRAM: Block RAM

CAM : Content-addressable Memory

CPU : Central Processing Unit

DRAM: Dynamic Random-access Memory

HLS : High-level Synthesis

FF : Flip Flop

FFT : Fast Fourier Transform

FIFO : First-in, First-out

FPGA : Field Programmable Gate Array

FSM : Finite State Machine

GCD : Greatest Common Divisor

L1 : Level-1

L2 : Level-2

LLM : Lower Level Memory

LUT : Look Up Table

PCIe : Peripheral Component Interconnect Express (PCI Express)

RAM : Random Access Memory

RTL : Register-transfer Level

1. INTRODUCTION

1.1. Motivation

Parallel computing is used in many areas to improve performance in cost effective way [1]. The memory is one of the bottlenecks behind achieving high performance due to the gap between processor and memory speed. Therefore, implementing efficient memory hierarchy is quite important. Memory latency is alleviated by using levels of caches. In the case of sharing a common memory by these caches, memory consistency and coherence problems arises. Appropriate memory consistency model and coherence protocol should be implemented to solve these problems.

Field Programmable Gate Arrays (FPGA's) are reconfigurable devices that can implement any digital circuit. These devices include bit-level processing elements whose numbers can reach a few million [2] and capacity of them increases year by year as number of transistors on a single chip increases [3]. FPGA's have significant performance potential for computing [4], however, implementing an algorithm manually on FPGA's is quite difficult. Fortunately, FPGA's based computing is simplified by High-Level Synthesis (HLS) compilers that automatically convert a program written in a programming language, to register-transfer level (RTL). Nevertheless, an efficient memory hierarchy is not supported by most of these compilers [5].

A special HLS compiler that generates an application specific supercomputer from a single threaded software program is proposed in [6]. In this thesis, a scalable cache coherent memory architecture is presented for this compiler.

Proposed memory architecture in this work is composed of coherent caches that follow directory-based protocol with write-update policy. All of caches that share a line update their own data with a new copy when one of them stores to this shared line in write-update policy. This policy has advantages on write-invalidate policy since the copies of the line are valid after store process. However, write-update policy does not preferred due to bandwidth overhead. In proposed method, directories always know correct set of owner caches, therefore, this overhead is alleviated. Furthermore, to our knowledge, the proposed architecture is the first example of directory-based write-update protocol.

In this proposed model, number of directories is arbitrary, in other words, a model with one central directory or distributed directories is possible. These directories can be distributed over L2 caches or memories. Each directory is responsible for managing coherence traffic only between caches in same coherence domain. In this protocol, exclusive, shared and invalid states are used to indicate status of a line. Number of caches and coherence domains are determined by the compiler. In this architecture, the compiler provides synchronization between the dependent memory operations.

1.2. Contributions

A coherent cache design is proposed in this thesis for the specific HLS compiler. The contributions are as follows:

1. The first coherence protocol that uses directory-based protocol with write-update;
2. A novel coherence protocol called as ESI protocol;
3. The first distributed directory scheme implemented on FPGA.

1.3. Thesis Organization

A background on the cache coherence problem and an overview of related works are presented in Chapter 2. Proposed coherence protocol is discussed in Chapter 3. Chapter 4 presents implementation details of the proposed hardware. Chapter 5 includes verification efforts, experimental results and discussion. Finally, conclusions and future works are given in Chapter 6.

2. BACKGROUND

2.1. Shared Memory Model

Parallel processing is necessary to achieve sustainable performance. Parallel computing machines are classified into four categories based on the type of parallelism, i.e., Single Instruction Stream Single Data Stream (SISD), Single Instruction Stream Multiple Data Streams (SIMD), Multiple Instruction Streams Single Data Stream (MISD) and Multiple Instruction Streams Multiple Data Streams (MIMD) [7]. MIMD machines, such as multithreading devices and multiprocessors, have capability of executing multiple instructions on multiple data. These machines are basically designed with shared or distributed memory model.

In shared memory model, all processors reach to the same address space and the performance of the model can be improved by caches [8]. Basic structure of the shared memory multiprocessors is demonstrated in Figure 2.1. All processors ($P_1, P_2, \ldots, P_N$) are connected to their private caches ($C_1, C_2, \ldots, C_N$). These caches communicate with each other and the memory via a bus. Sharing same address space provides advantages such as ease of programming. However, shared memory multiprocessors suffer from consistency and coherence problems.

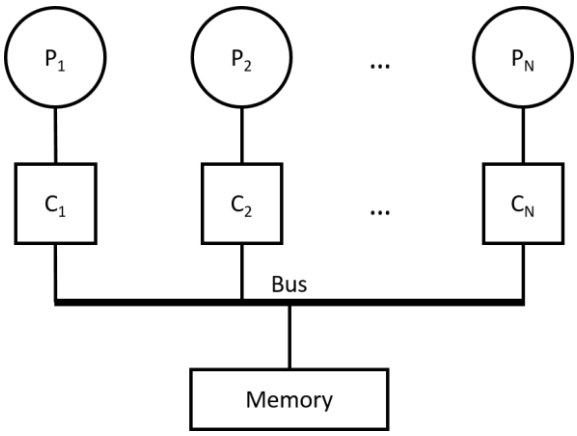

Figure 2.1. *A shared memory architecture*

2.2. Memory Coherence

Coherence, one of the two main problems of the shared memory multiprocessor, occurs when more than one cache have same block and at least one of them write(s) to this block. An example of coherence problem is demonstrated in Figure 2.2. The example model contains three private caches (C_1, C_2 and C_3) that are connected to the memory via a bus shown in Figure 2.2 (a). Initially, both C_1 and C_2 have the block X and the value of this block is A for both caches and the memory. However, C_3 does not have this block. Remind that caches are implemented with write-through or write-back policy. Caches that follow write-through policy immediately send new data to the lower level memory (LLM) when they receive a write request. However, write-back caches do not send written data to the LLM until this block is replaced. This example demonstrates the problem with both write-through and write-back private caches in Figure 2.2 (a) and Figure 2.2 (b), respectively.

In the design with write-through caches, P_1 sends a write request to C_1 for block X to change its value as B and then C_1 stores B. The cache immediately transfers this write request to the memory. Although the block X is updated by P_1, C_2 still has stale data (A). C_2 responds a read request coming from P_2 for the block X with stale data and the problem thus emerges. When C_3 receives a read request for the block X from P_3, the updated data B is obtained from the memory since this block is miss. Same requests are applied to the design with write-back caches in Figure 2.2 (c). In this model, when C_1 receives a write request for the block X from P_1, C_1 does not immediately transfer the write request to the memory. C_2 again sends stale copy of the block when it receives a read request for block X. When C_3 receives a read request for block X from P_3, this request is transferred to memory since this block is miss. However, the memory sends stale data to C_3 since it is not updated.

Several hardware and software protocols have been introduced in literature, to overcome coherence problems [9]. The proposed work only includes hardware-based solutions.

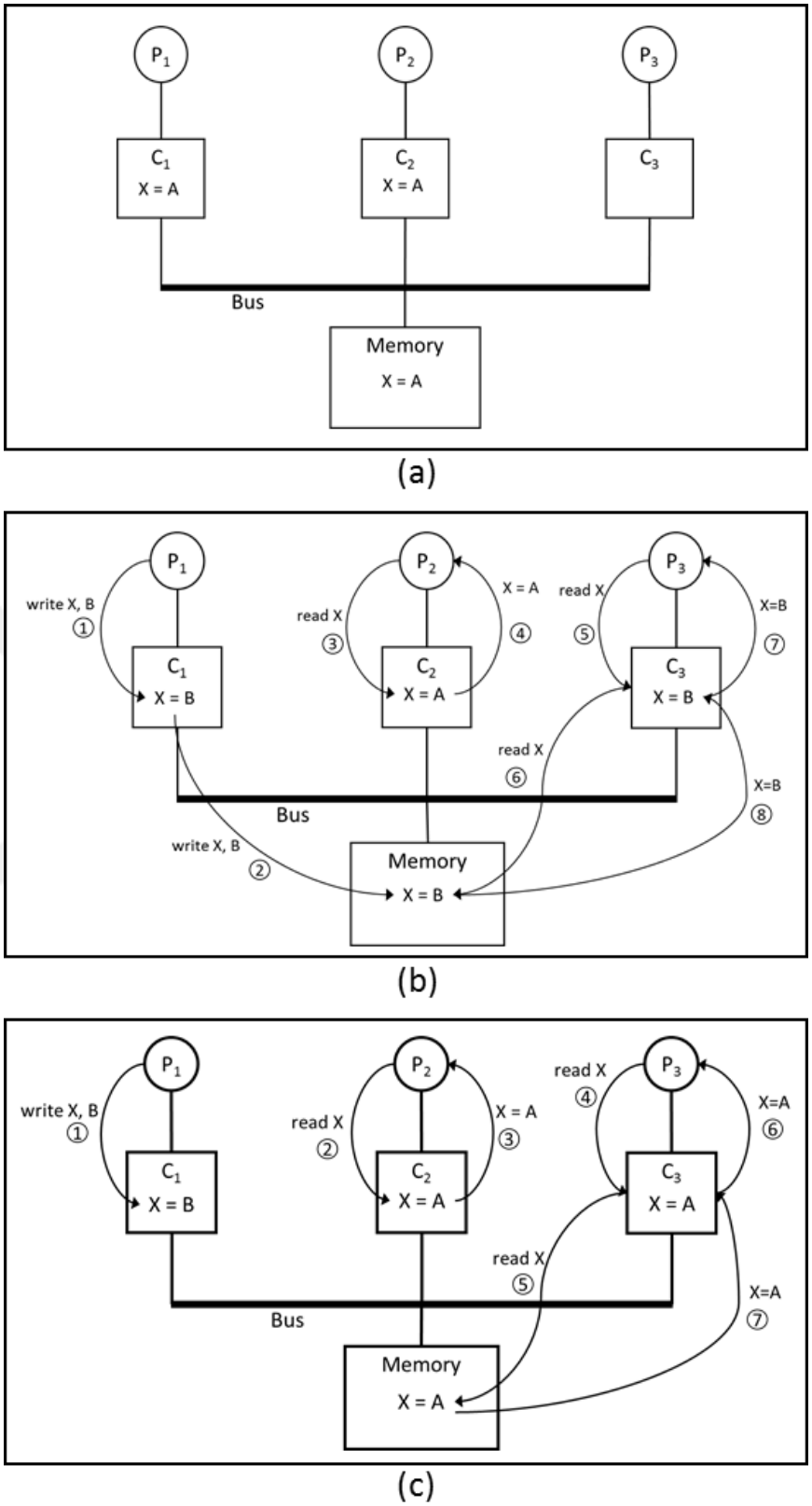

Figure 2.2. *An example of coherence problem (a) initial model, (b) problem is applied to write-though caches, (c) problem is applied to write-back caches*

2.2.1. Hardware-based coherence solutions

Coherent caches are implemented based on two protocols as snooping protocol and directory-based protocol. These protocols identify the hardware architecture of the coherence mechanism. They are matched with a proper write policy, such as write-update and write-invalidate. State bits are also added to coherence design to indicate the situation of cache blocks.

2.2.1.1. *Coherence protocols*

Physical organization of coherence mechanism is determined by coherence protocols. Basically, snooping and directory-based protocols are two classes of them.

In snooping coherence protocol, caches are connected to each other and the LLM via a bus. Each individual snoopy cache keeps status bits for each data blocks. Necessary coherence requests are broadcasted on the bus while the bus is simultaneously watched by all caches. Possible coherence actions can be read miss, update or invalidate. Read miss is broadcasted when the block requested by processor is not valid in related cache. The corresponding data is then obtained by copies sent from caches via the bus. Invalidate or update requests are broadcasted when a cache writes to the block shared with other caches. Other caches that have corresponding block respond these requests by updating or invalidating their own copies. Therefore, all caches keep their shared data coherent with others.

In directory-based protocol, coherence is provided by a hardware component called as directory. Private caches are connected to either one central directory or distributed directories by an interconnection network in this model. The information of exact locations of blocks is kept in directory entries. Caches send requests to directories if coherence transactions are required. The directory controls status of the data block and then sends necessary coherence requests to related caches.

In distributed directory schemes, the information of blocks' locations is distributed over directories while directories are also distributed over either the memory or caches. An example of distributed directory scheme is demonstrated in Figure 2.3 (a). In this model, directories (D_1, D_2, …, D_K) are distributed over memory modules (M_1, M_2, …, M_K). In central directory schemes, the information is stored in only one directory.

However, this situation causes contention since all caches try to access to the same directory. Figure 2.3 (b) shows an example of central directory scheme.

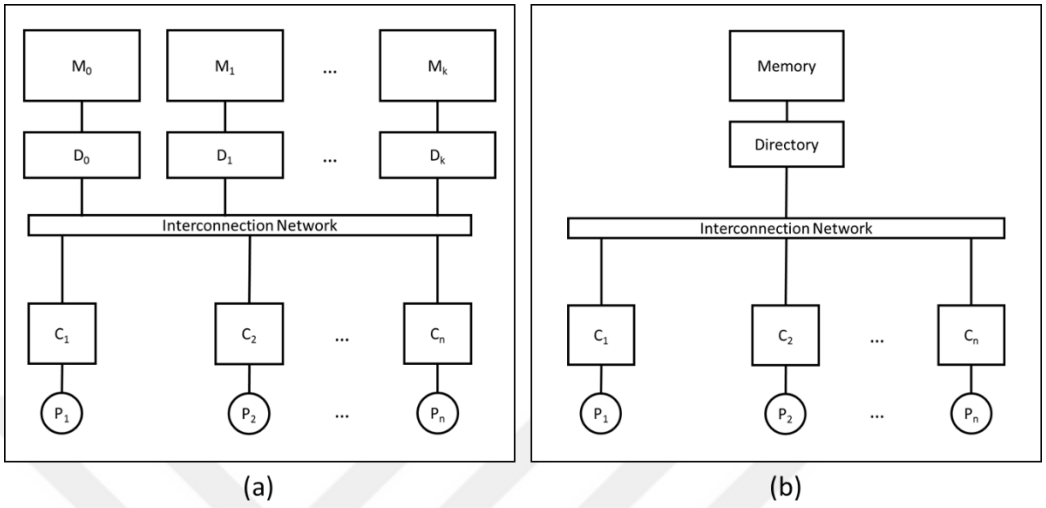

Figure 2.3. *Directory schemes (a) a distributed directory scheme, (b) a central directory scheme*

The directory-based protocols scale better than snooping protocols since bandwidth overhead caused by broadcasting increases with number of processors in snooping designs [10]. However, directory-based coherence schemes are harder to implement and require extra messages [10].

2.2.1.2. *Write policies*

As mentioned before, a cache sends either an invalidate request or an update request to provide coherence when this cache receives a write request to the block shared among caches. The update or the invalidate request is chosen according to the write policy, such as write-update or write-invalidate.

In write-invalidate policy, a cache responds a write request corresponding to the block shared between caches, by sending invalidate request to owner caches. Caches that receive an invalidate request naturally invalidates their own copies. The block is then miss when one of these caches tries to access its copy. Therefore, the new copy of this block is supplied by the cache that previously writes to the block.

In write-update policy, a cache that receives a write request for a block sends an update request. The new data is included in this update request. Caches that receive this

update request refreshes their copies with this new data. This time, one of these caches that update its copy can directly obtain this block since the block is hit and updated unlike write-invalidate policy.

Assume that write-invalidate and write-update policies are applied to the coherence problem previously defined in Figure 2.2. These policies are implemented with both directory-based and snooping models. Figure 2.4 demonstrates these models and initial conditions for both directory-based and snooping protocol. In the directory-based model, private caches following write-back policy are connected to a central directory via an interconnection network. Each directory entry contains presence bits since the directory is a full-map directory. Remind that C_1, C_2 and the memory initially have the block X and the value of this block is A. The directory entry corresponding to the block X is also shown in Figure 2.4 (a). Since three caches are connected to directory, this entry has three bits. The rightmost two bits of it are one, meaning that C_1 and C_2 have this block. However, remaining bit is zero, in other words, C_3 does not contain the block. In the snooping model, caches and the memory are connected to each other via a bus with same initial conditions shown in Figure 2.4 (b).

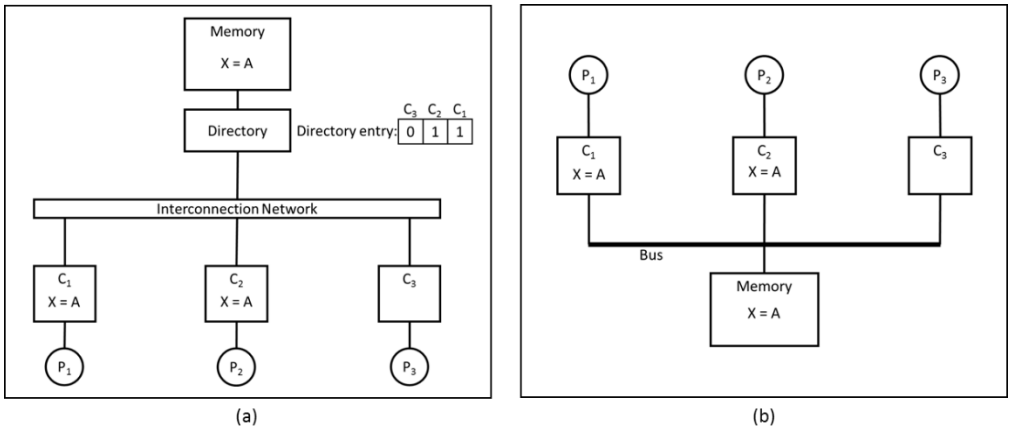

Figure 2.4. *Initial models for solution to the defined coherence problem (a) directory-based model, (b) snooping model*

Figure 2.5 demonstrates the directory-based solution to coherence problem with write-invalidate policy. The first request is write B sent by P_1 to C_1 as in the defined problem. As a response C_1 stores this new value and then sends an invalidate request for the block X to the directory in order to disable stale copies of the block given in Figure

2.5 (a). The directory controls presence bits of this block to determine owner caches. The invalidate request is then transmitted to C_2 by the directory since only C_2 has this block except for C_1. The directory changes the presence bit corresponding C_2 as zero to indicate that C_2 is not an owner cache anymore. When the invalidate request arrives to C_2, the copy of the block X is immediately disabled.

Remind that the problem emerges when P_2 sends a read request to C_2 for the block X. After receiving this request, C_2 transfers it to the directory since C_2 does not have a valid copy as demonstrated in Figure 2.5 (b). This situation is called as ping-pong effect. The directory transfers this read request to C_1 since C_1 is unique owner of this block. When the data is received from C_1, the directory sends it to C_2 shown in Figure 2.5 (c). Presence bits are also updated by the directory and C_2 becomes one of the owners again. Then, C_2 provides correct data to P_2 and therefore, the coherence problem is solved.

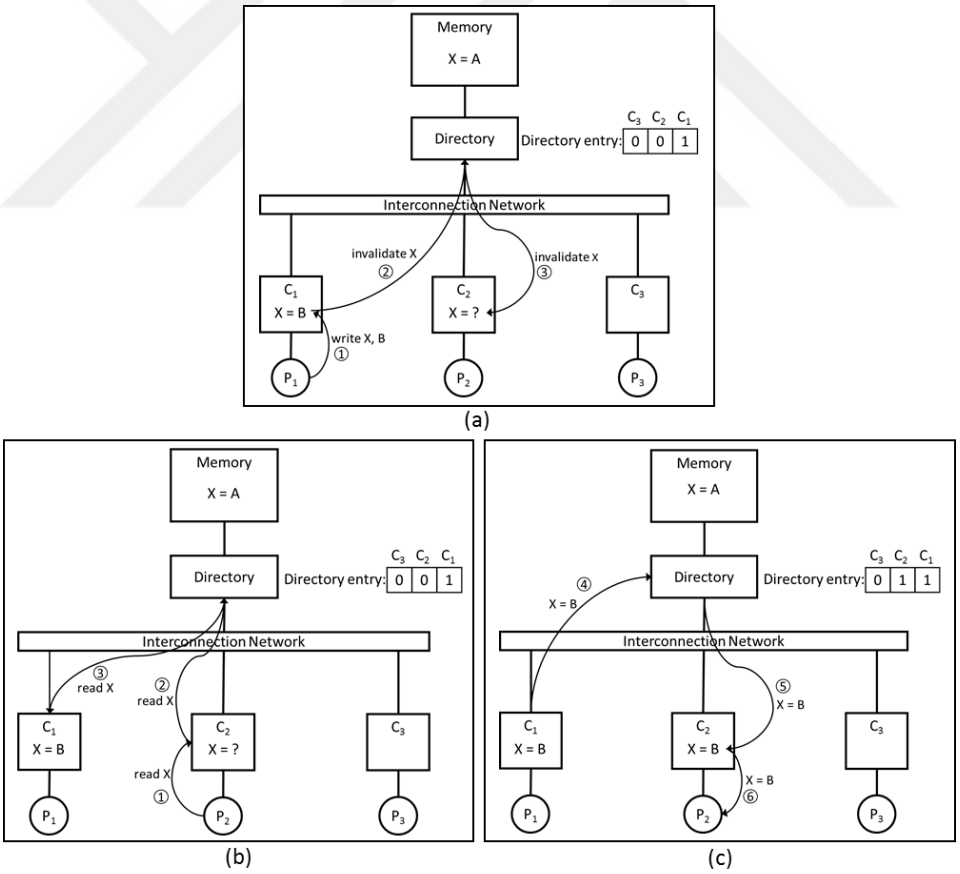

Figure 2.5. *Directory-based solution to the coherence problem with write-invalidate policy*

Figure 2.6 shows the directory-based solution to the problem with write-update policy. Firstly, C_1 receives write B request for the block X as demonstrated in Figure 2.6 (a). This time, C_1 sends an update request instead of an invalidate request to the directory. This update request contains new data B. Since C_2 is also owner of this block, the directory transfers the update request to it. The directory entry is not changed since caches that have the block are still same. C_2 changes its copy with B after receiving the update request. When C_2 receives a read request for the block X from P_2, C_2 directly supplies corresponding data to P_2 since it has the recent copy as explicitly shown in Figure 2.6 (b).

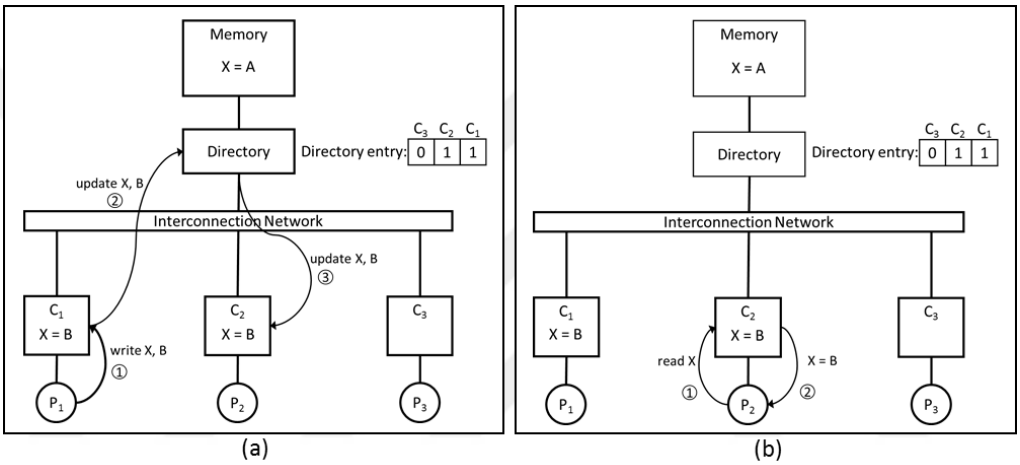

Figure 2.6. *Directory-based solution to the coherence problem with write-update policy*

The memory has stale data for both write-invalidate and write-update policy, since the private caches follow write-back policy as shown in Figure 2.5 and Figure 2.6. Note that this situation does not cause any problem even if a cache wants to read a block that is not cached. Figure 2.7 illustrates actions after C_3 receives a read request for the block X. C_3 sends this read request to the directory since it does not have this block. The directory checks for presence bits to find another owner. The directory chooses one of the caches that have the block (C_1 or C_2). For this example, the directory sends the read request to C_1 and the data is provided from it. These actions are same for both write-invalidate and write-update policies. Write-through private caches protect memory from having stale data since they flush new data after each write request. The defined coherence problem is also solved by using these caches as in write-back private caches.

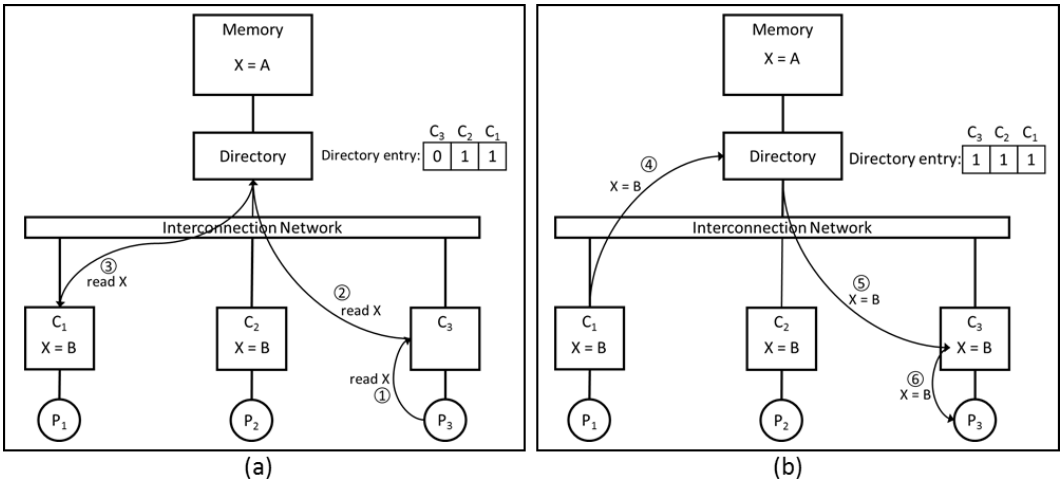

Figure 2.7. *Solution to the problem that occurs with write-back caches*

The snooping model defined in Figure 2.4 (b) also solves the coherence problem with write-invalidate or write-update policy. Moreover, without communicating with directory, caches directly broadcast coherence requests on the bus and then related caches snoops these requests. Figure 2.8 and Figure 2.9 demonstrate solutions to the defined coherence problem in snooping model with write-invalidate and write-update policies, respectively.

Write-update policy consumes more bandwidth as compared to write-invalidate policy since both address and data are sent to update other copies instead of sending address only as in write-invalidate policy [10]. However, write-invalidate policy suffers from ping-pong effect since recently written block is invalidated by caches [9].

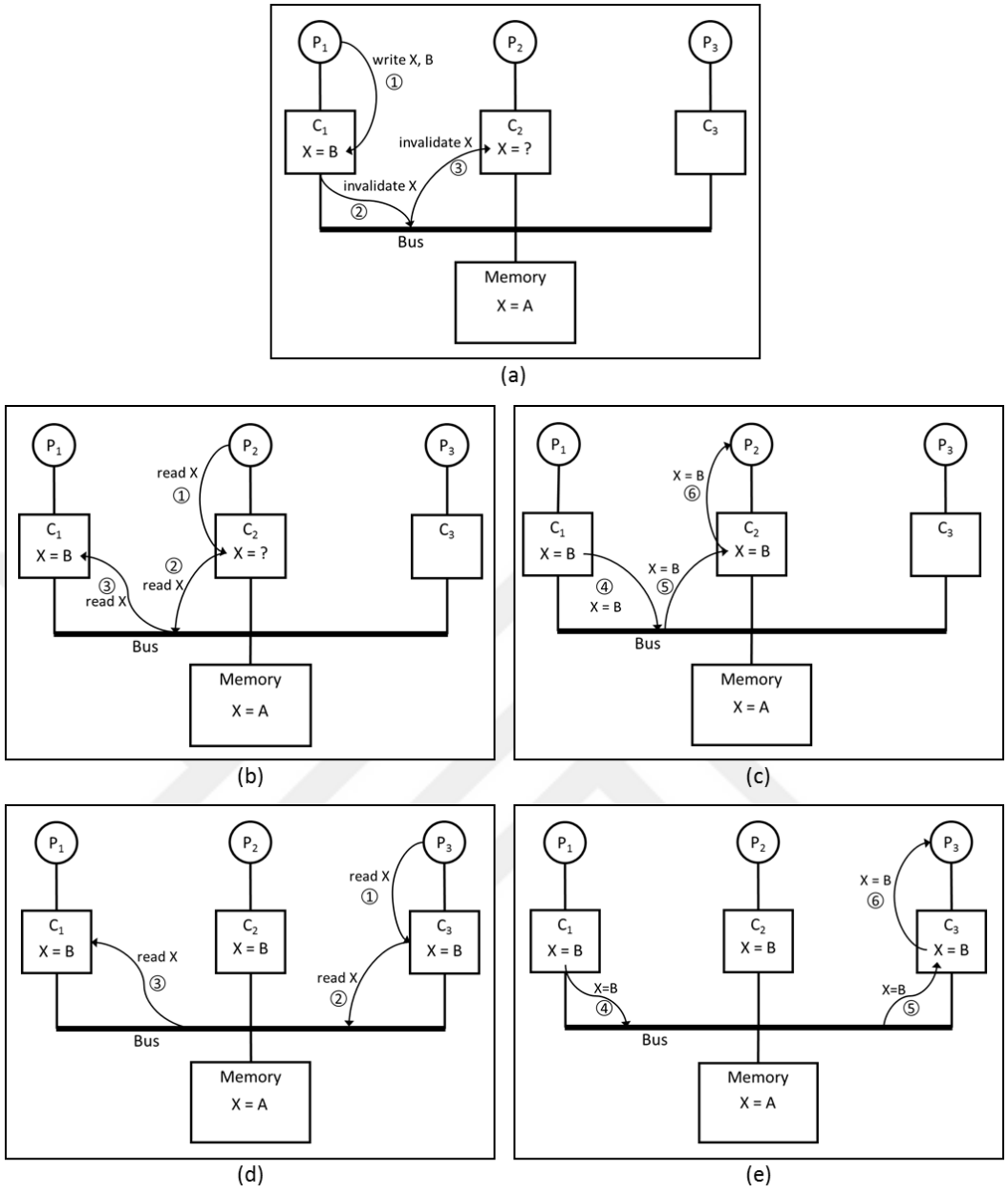

Figure 2.8. *Snooping solution to the coherence problem with write-invalidate policy*

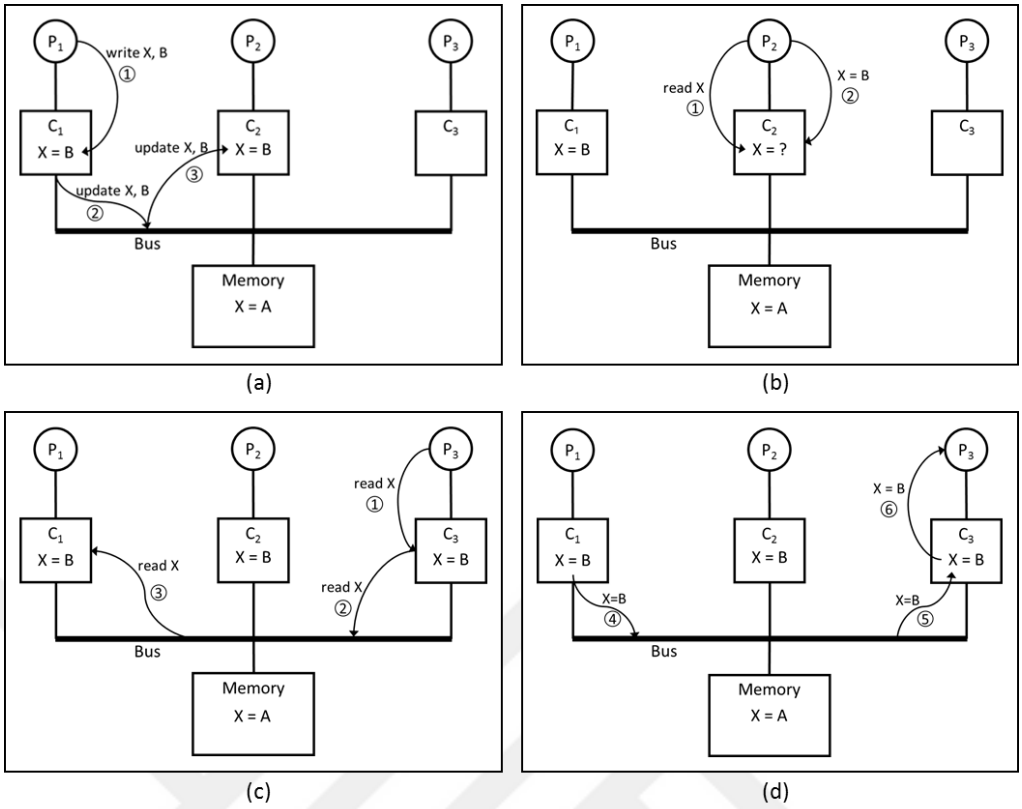

Figure 2.9. *Snooping solution to the coherence problem with write-update policy*

2.2.1.3. *States*

In a coherent cache design, each private cache requires to keep information called as state for each block. Caches obtain necessary properties such as validity, dirtiness, exclusivity and ownership of blocks from these states [10]. State machines transfer the states to each other as Modified (M), Owned (O), Exclusive (E), Shared (S), and Invalid (I). A block with I state means that either it is not stored inside related cache, or it has stale data. The blocks that are valid and owned by only corresponding cache have a state either E or M. In fact, clean blocks correspond to E while dirty ones are corresponding to M. The state is S if either the cache has sent its clean copy to other cache or it has received a copy from one of them. Furthermore, the state is O if the cache has sent its dirty copy to other caches and then it becomes responsible for writing this block to the LLM among sharers. Cache protocol can be implemented by using all these states, i.e., MOESI or just by selecting some of them, such as MSI, MESI, etc.

In a design with write-through private caches, only SI states are sufficient since write requests are directly transferred to the LLM [11]. A block is tagged as I if a cache does not have the block or the block is not valid in this cache, otherwise the state is S. In a design with write-back private caches, at least MSI states are necessary. M state is added to tag blocks that is written by processors.

Although MSI protocol solves coherence problems, adding exclusive state alleviates write overhead [11]. Invalidation process is not required when a cache receives a write request to an exclusive block since the cache knows that it is unique owner of the block. MESI, one of the most popular protocols, has been first proposed in [12]. Their coherence mechanism has been designed with snooping and write-invalidate protocols. Table 2.1 shows states, requests and change in states and also transactions according to these requests. Caches response two type of requests coming from processor or snooping on the bus, such as processor read and processor write or bus read and bus read-exclusive, respectively [13]. A bus read request is broadcasted on the bus when one of the caches wants to read corresponding block. In a bus read-exclusive request, invalidate request accompanies to the bus-read request. The cache only accepts processor's requests for blocks in I state. Processor read and processor write requests coming to the block in I are responded by broadcasting bus read and bus read-exclusive requests on the bus, respectively. The state of the block is changed as E or S according to sharing property of this block after response to bus-read request is received. Sharing property of blocks is determined by a special signal. Furthermore, the next state becomes S if sharing signal is asserted otherwise it becomes E. In the case of receiving a processor write request, the next state then becomes M for all initial states. However, the cache broadcasts bus read-exclusive, if the block is in I or S states. Any coherence transaction performed by the cache is not required for a processor request corresponding to a block in E or M since this cache is a unique owner. A block that has a state M, E or S maintains its state after a processor read request is received. Bus read and bus read-exclusive requests coming to the cache for a valid block are required to change the state as S and I, respectively. Moreover, corresponding data is flushed if initial state of the block is M or E. In the case of initial state with S, this data is flushed by only one of the caches among sharers.

Table 2.1. *The MESI protocol*

Current State	Request	Next State	Transaction
M	Processor read	M	-
	Processor write	M	-
	Bus read	S	Flush
	Bus read-exclusive	I	Flush
E	Processor read	E	-
	Processor write	M	-
	Bus read	S	Flush
	Bus read-exclusive	I	Flush
S	Processor read	S	-
	Processor write	M	Bus read-exclusive
	Bus read	S	Flush’
	Bus read-exclusive	I	Flush’
I	Processor read	E	Bus read ($\bar{S}$)
		S	Bus read (S)
	Processor write	M	Bus read-exclusive

Suppose that MESI protocol specified in Table 2.1 is applied to the coherence problem previously given in Figure 2.2. The solution to this problem is given Table 2.2. Remind that initially C_1, C_2 and the memory have the block X. Initial states for both caches’ blocks are S since more than one cache own this block. First request is sent by P_1 to write the block X and then C_1 broadcasts a bus read-exclusive request to invalidate the copy of C_2. C_2 flushes corresponding data and invalidates its copy after receiving this request. C_1 then stores the written data and changes the state of the block X as M. After this process, P_2 sends a read request for the block X and then C_2 broadcasts a bus read request on the bus. C_1 snoops this bus read request and supplies corresponding data to C_2 before C_1 and C_2 change their states to S. Finally, P_3 sends read request for the block X and C_3 broadcasts a bus-read request since this block is miss. The corresponding data is provided by either C_1 or C_2 and then C_2 receives this data in S state.

Table 2.2. *Solution to the defined coherence problem with the MESI protocol*

	States for block X				
Request	**C_1**	**C_2**	**C_3**	**Transaction**	**Data Supplied by**
-	S	S	-	-	-
P_1 writes to X	M	I	-	Bus read-exclusive	C_2
P_2 reads X	S	S	-	Bus read (S)	C_1
P_3 reads X	S	S	S	Bus read (S)	C_1 or C_2

States are a little bit different for write-update policy. Dragon protocol designed with snooping model is an example to write-update policy [14]. This protocol includes E, Shared-Clean (SC), Shared-Modified (SM) and M states. E and M states are same as explained before while SC and SM states are used instead of S state in this protocol. Caches that have clean copies of same block tag these blocks as SC while a cache that modifies the block tags this block as SM. Although more than one cache have same block in SC state, only one cache has the block in SM state and this cache is responsible for writing back the block to the LLM. Note that the protocol does not contain I state. Dragon protocol is demonstrated in Table 2.3 as explained in [13]. A cache receiving a request from its processor for the block that is miss, broadcasts a bus read request. If there is a cache that has this block in M or SM states, this cache is responsible for providing data. Otherwise, the data is supplied by the memory. In the case of receiving a processor read request that is miss, the cache has corresponding block as either E or SC states. The state is SC if at least one more cache has this block and it is E if only requester cache has this block. Caches detect sharing with a special bus line [15]. In the case of receiving a processor write request corresponding to the block that is miss, the related cache stores the new data in M or SM state. This state is decided according to sharing after response of the bus-read request is obtained. This state is M if the block is not shared. Otherwise, the state is SM so the cache broadcasts a bus update request. Bus update request is used for updating copies of sharer caches and the response of this request is called as Update. A cache broadcasts a bus update request when it receives a processor write request to the blocks SC or SM. Thereafter, states are changed to M if the cache realizes that the block is not shared anymore. State of a SM block is changed to SC when a cache snoops a bus update for this block. Therefore, the cache is no longer responsible for writing back the

block to the LLM. A cache does not receive bus update requests for blocks in E or M states since they are not shared. Next state is SC if a cache receives bus read request to a block with E state. In the case of receiving a bus-read request to the block with M, the state becomes SM. Other transactions are analogous to previous protocols.

Table 2.3. *The Dragon protocol*

Current State	**Request**	**Next State**	**Transaction**
	Processor read (Miss)	E	Bus read ($\bar{S}$)
		SC	Bus read (S)
	Processor write (Miss)	M	Bus read ($\bar{S}$)
		SM	Bus read (S) & Bus update
E	Processor read	E	-
	Processor write	M	-
	Bus read	SC	-
SC	Processor read	SC	-
	Processor write	M	Bus update ($\bar{S}$)
		SM	Bus update (S)
	Bus update	SC	Update
SM	Processor read	SM	-
	Processor write	M	Bus update ($\bar{S}$)
		SM	Bus update (S)
	Bus read	SM	Flush
	Bus update	SC	Update
M	Processor read	M	-
	Processor write	M	-
	Bus read	SM	Flush

Assume that Dragon protocol is also applied to the coherence problem that is defined in Figure 2.2. The solution to this problem is given Table 2.4. Initially, C_1 and C_2 caches have the block X in SC state since S state is not valid. C_1 broadcasts a bus update request when it receives write request from its processor. C_2 observes this bus update request for the block X and then updates its copy with previously written new data. New state of the block X becomes SM for C_1 and after this point C_1 is responsible for supplying

corresponding data for this block to other caches and the memory. Then, C_2 receives read X request from its processor. C_2 directly supplies data since it has the block. Finally, P_3 sends read request for block X and then C_3 broadcasts a bus read request. The data is supplied by C_1 that has modified the block.

Table 2.4. *Solution to the defined coherence problem with the Dragon protocol*

	States for block X				
Request	**C_1**	**C_2**	**C_3**	**Transaction**	**Data Supplied by**
-	SC	SC	-	-	-
P_1 writes to X	SM	SC	-	Bus update (S)	C_1
P_2 reads X	SM	SC	-	-	-
P_3 reads X	SM	SC	SC	Bus read (S)	C_1

2.3. Memory Consistency

Consistency, the other challenging problem of the shared memory multiprocessor, is related to correctness of the shared memory [10]. Figure 2.10 demonstrates a mutual exclusion protocol as an example of consistency problem [16]. The example includes two processes and each process contains a critical section. The protocol should assure that only one of the processes is in the critical section at the same time. However, both of them can be in critical section in following conditions. Assume that initial values of both variables a and b are zero. These variables are equalized to one by the Process 1 and Process 2, respectively, and then processors send write requests to the memory for these variables since these values are changed. Thereafter, Processor 1 and Processor 2 read variables b and a from the memory, respectively, to check the correctness of corresponding if statement. Nevertheless, both of processors are read the values as zero from memory since previous write requests to the memory are delayed. Both of the processes are eventually in the critical section. Consistency problems can be handled by implementing memory consistency model that defines the rules between processes and

the memory system [17]. In other words, this model indicates the order of read and write requests.

In proposed work, the memory model is based on uniprocessor memory model since the initial software program converted by the compiler is single threaded. Therefore, memory consistency problem does not occur as in multiprocessors' memories and implementing memory consistency model is not required.

Process 1	Process 2

a=1;
if (b==0){
critical section;
}
a=0;
	b=1;
	if (a==0){
	critical section;
	}
	b=0;

Figure 2.10. *An example of consistency problem*

2.4. Related Works

In literature, several hardware-based coherence protocols have been proposed. Most of them have been designed with write-invalidate policy. Moreover, both snooping [12, 18, 19] and directory-based [20-23] protocols have been used in these designs. Although several snooping with write-update protocol have also been implemented [14, 24], directory-based design with write-update has not been presented based on our researches. Proposed work is the first example of directory-based coherent cache design with write-update policy.

Several works have been presented to provide a memory system on FPGAs for parallel processing in addition to ASIC implementations. Besides works that does not require a coherence protocol such as [25, 26], a number of studies have also been presented to implement a coherence protocol on FPGAs.

In [27], symmetric multiprocessor model has been implemented on FPGA with softcore processors (Nios processors), and the Avalon bus. Since proposed model does not permit implementation of traditional snooping or directory-based protocols, a hardware component called as cache coherency module (CCM) has been added to design.

CCM snoops the bus and manages coherency between local caches. This kind of protocol is called as hybrid snooping protocol. Write-invalidate policy is provided by sending interrupt to all processors by CCM when invalidation is needed. Although defined model solves coherence problems, it has disadvantageous due to sending interrupt to all processors even if these processors are irrelevant to the corresponding request.

A coherent cache model that is suitable for both hard and soft processors have been presented in [28]. These processors provides strict consistency model. In proposed model, private caches are connected to Central Hub via FIFO based units (FSLs). The Central Hub provides communication like a bus. In this design, write-once protocol [18] has been chosen. This is a simple protocol that is implemented in snooping and write-invalidate. This work has not achieved a sustainable performance with hardware threads since number of threads is limited due to communication overhead on Central Hub.

In [29], a memory system has been designed for compilers that automatically converts high-level language to hardware for FPGAs. This memory system is connected to data path generated by a compiler. In proposed design, memory is partitioned into coherence clusters. Each coherence cluster have at most one write and at least one read ports. MARC II supports speculative memory operations on read ports. Each read and write ports includes a cache. Since each coherence cluster maps to different part of address space, coherence is only provided inside these clusters. Different coherence clusters are connected to TechMod via Shared Memory Bus. This TechMod provides an interface to the memory. Ports in a cluster are connected to each other and the Shared Memory Bus via Shared Coherence Bus. Both Shared Memory Bus and Shared Coherence Bus receive requests with dynamic priority. Write-invalidate or write-update policy can be chosen for each coherence cluster. States are different for read and write caches. Read caches have valid and invalid states only, while write caches have invalid, partially exclusive, exclusive and shared states.

In [30], proposed multiprocessor model is implemented with softcore processors (MicroBlaze processors). In this model, local caches are connected to these processors with Local Memory Buses (LMBs). Caches and the memory controller are communicated with each other using a cross bar interconnection network. FIFO based units (FSLs) are placed between each cache and interconnection network. Memory controller is also connected to interconnection network via an FSL. Coherence have been provided with snooping and modified version of MESI protocols. Proposed design also manages mutex

variables inside local caches. This design is not scalable since it cannot execute more than eight thread at the same time. In [31], a directory-based design is proposed to improve scalability in [30]. This design is similar to previous model. However, a directory has been connected to interconnection network and memory controller via FSLs. This directory has been implemented as a duplicate tag directory. MESIF protocol, i.e., MESI protocol with Forward state, has been chosen to provide coherence.

Memory system is required for FPGA based operating system in [32]. In this design, coherence is provided by coherent scratchpads. These scratchpads contain a marshaller, a cache and a router. The coherent scratchpads that belongs to same coherence domain and a coherent scratchpad controller are connected to each other with rings. Coherent scratchpads reach to lower level memory via coherent scratchpad controller. This controller also keeps data and owner bits in private scratchpads [33] for each memory address. The proposed coherence protocol is snooping with MOSI states.

The proposed design in this thesis is different from other FPGA based coherent caches in terms of both targeted system and coherence protocol. Proposed coherence mechanism targets to application specific supercomputer. This supercomputer is generated by a specific compiler that converts single-thread program written in high level language to RTL design executing in parallel. All of the previous works has been designed with snooping protocols except for [31], however, this design implemented with a central directory and does not provide multiple coherence domains unlike presented coherence protocol.

3. ESI PROTOCOL

3.1. System Overview

High-level synthesis (HLS) compilers are capable of converting a program written in a high-level language into a low level register-transfer level (RTL) design. A specific HLS compiler proposed in [6] automatically converts a single-threaded software program into an application-specific supercomputer. The supercomputer generated by the HLS compiler runs only its application and its hardware system contains at least one chip. The generated hardware executes in parallel and gives exactly same results with initial single-threaded input program of the compiler. Writing software programs manually in parallel is complicated [34] hence the compiler provides a great opportunity to programmers.

In this developed compiler method [6], each loop in the initial code fragment is converted automatically to a frequency-optimized finite state machine (FSM) called as thread unit. Thread units are connected to task and synchronization networks demonstrated in Figure 3.1. These networks mainly control the threads and determine the execution order. Thread units are also connected to a memory hierarchy via master ports to perform memory operations such as load and store. Each master port includes sending and receiving first-in first-out (FIFO) buffers to transfer data from the thread units to memory and from memory to the thread units, respectively.

Implementing an efficient memory hierarchy is quite important for any application-specific hardware design since the memory hierarchy is one of the major barriers on achieving high performance in many application workloads such as an image reconstruction application used in medical tomography. The implemented memory model is based on uniprocessor memory model, since the converted code fragment is single threaded. The supercomputer requires a memory system with a number of ports to transfer data and it possesses the following ports in Figure 3.1., where one chip model of the supercomputer is depicted. This memory system has one or more slave ports that are connected to hardware thread units. Memory requests sent by thread units are received via these ports. The memory system also has master ports to communicate with host. These ports are connected to the host communication network while this network is also connected to a PCI Express (PCIe) that provides communication with host. Furthermore, one or more master ports are connected to DDRn controller via one-to-one network. L2

caches send requests to the external DRAM unit that contains both tag and data of each L2 cache line via these ports.

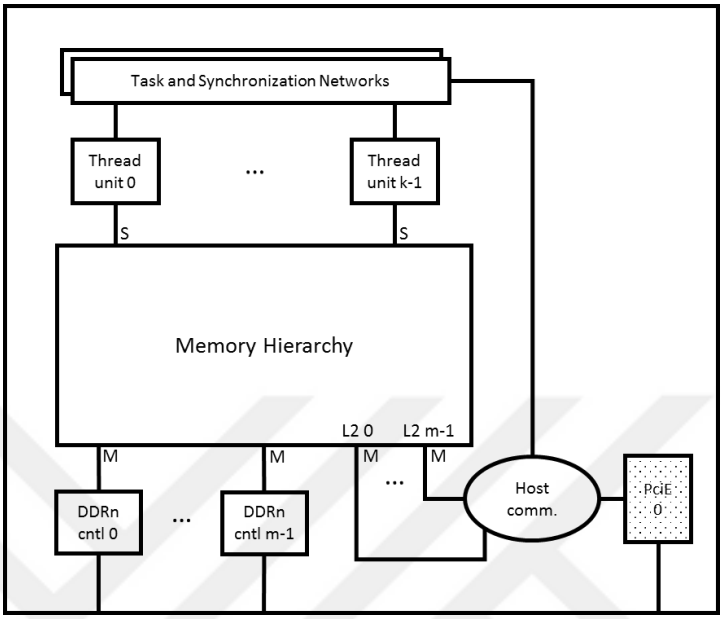

Figure 3.1. *One chip system model*

The compiler provides some conveniences to the memory system. The memory system does not receive some requests that occur in multi-threaded software, such as compare and swap or memory barrier instructions. Moreover, if two instructions refer to the same address and at least one of them is store, the compiler ensures that logically earlier one finishes its access before the logically later one starts. This synchronization is managed by acknowledgement responses sent by memory system as a response to store requests.

Two different memory models, i.e., bank-interleaved shared caches and coherent caches, have been presented for this supercomputer [6]. These memory models include two level of caches (L1 and L2 caches). Moreover, they should be written in Verilog language.

In bank-interleaved shared cache model, each bank is responsible for different sections of address space. Any coherence protocol is not required in this model since memory addresses do not overlap. However, an interconnection networks are required to connect thread units to L1 caches. It is simpler than coherent cache design, but the additional interconnection network increases the memory access time.

In coherent cache model, the coherence protocol is required and it uses directory-based write update protocol. L1 caches are connected to directories and each directory is connected to a L2 cache bank in this architecture. One of the advantages of this model is that each thread unit has point-to-point connection to the related L1 cache. Thus, memory access time is improved. The other advantage is that only caches that hold shared lines communicate with each other, therefore, communication is restricted. Roll-back or negative acknowledgement responses are not required in this model. The proposed work in this thesis is implementation of this coherence protocol and details are given and discussed in the following sections.

3.2. Protocol Details

As explained in Chapter 2, coherence protocol is specified by many options such as snooping or directory-based and write-update or write-invalidate. In the proposed model, directory-based design has been preferred to obtain a scalable architecture. Directory-based architectures include one central directory or distributed directories. Moreover, the number of directories is arbitrary in proposed design hence the design contains one central directory or distributed directories.

Directories are generally classified according to the kind of kept information, such as full-map, limited and chained directories [9]. The designed directory is full map directory. In this scheme, presence bits that indicate owner caches for each block are stored in directory entries.

In proposed model, write-update policy is implemented. Write-update policy is better than write-invalidate since write request that is received to a shared cache line is immediately sent to all sharer caches. However, write-update policy is not popular in commercial systems since it requires relatively more bandwidth. In proposed model, the directory always knows the correct set of sharer caches, therefore, the network traffic caused by write broadcasts is alleviated.

The proposed design has been implemented with ESI states. The ESI protocol includes only exclusive (E), shared (S) and invalid (I) states of the MESI protocol. Instead of using M state, extra one dirty bit is kept for each cache line. Shared line also can be dirty since the policy is write-update.

3.2.1. Requests

A thread unit sends load, store flush_all_L1 and flush_all_L2 requests to an L1 cache as listed in Table 3.1. A load request is sent by a thread unit to read a word, half word or a byte from the memory. Address of the requested data is included in this request. This request is responded by providing corresponding data. A store request is sent by a thread unit to write to a word, a half word or a byte to the memory. This request contains the address and the new data. The cache that receive store request sends an acknowledgement response at the end of the store process. A flush_all_L1 request is received by a cache when all load and store requests are responded. An acknowledgment is sent by a cache after all dirty lines are invalidated and flushed to the directory. One of the L1 caches receives a flush_all_L2 request and this cache is responsible for forwarding this request to the directory. The cache sends an acknowledgement to the thread unit, after this request is transferred. Flush_all_L1 and flush_all_L2 requests are sent to update host memory so that software and the accelerator have the same copy of the memory in order that software can resume its operation where accelerator completes its acceleration of the specific task.

Table 3.1. *List of requests coming from a thread unit to an L1 cache*

Request	**Meaning**	**Response**
Load	Read corresponding block	Data
Store	Write this data to corresponding block	Acknowledgement
Flush_all_L1	Flush and invalidate all dirty lines	Acknowledgement
Flush_all_L2	Transfer this request to L2 cache	Acknowledgement

Requests coming from an L1 cache to a directory are shown in Table 3.2. A line_read request is sent by an L1 cache to the directory when this cache receives a thread request corresponding to an invalid block. The line address is included in this request. The directory transfers this request to one of the other owners if there is any. Otherwise, the line is requested from L2 cache. After obtaining the line data, the directory provides this data to the requester cache. This response also includes the state of the line as exclusive or shared. A flush or an abandon request is sent by an L1 cache to the directory when a line is replaced by this L1 cache. Moreover, this request is flush, if the replaced

is dirty. Otherwise, it is abandon. A flush request contains the line address, the line data and the byte mask while abandon request includes only line address. The directory removes the requester cache among sharers of this line, after receiving one of these requests. Furthermore, the request is forwarded to the L2 cache by the directory if it is a flush. A remote_store request is sent by an L1 cache to a directory when this cache writes to the shared line. The remote_store request contains the new word and the address. The directory transfers this request to all of the sharer caches as write-update policy requires. The flush_all_L2 request is directly transferred to the L2 cache by the directory. An acknowledgement response is sent by a directory to the requester L1 cache after flush, abandon or flush_all_L2 process is completed.

Table 3.2. *List of requests sent by an L1 cache to a directory*

Request	**Meaning**	**Response**
Line_read	Read this line	Line data, state
Flush	Flush this data to L2 cache and remove this cache among sharers	Acknowledgement
Abandon	Remove this cache among sharers	Acknowledgement
Remote_store	Update copies of this block	Acknowledgement
Flush_all_L2	Transfer this request to L2 cache	Acknowledgement

Table 3.3 illustrates requests sent by a directory to an L1 cache. After a directory receives a line_read request to the line owned by at least one cache, it forwards this request to the one of the owner caches. This cache sends corresponding data as a response. After a remote_store request is received by an L1 cache, this request is sent by a directory to all of the sharer caches. All of these caches send an acknowledgement response to the directory after the copy of the line is updated.

Table 3.3. *List of requests sent by a directory to an L1 cache*

Request	**Meaning**	**Response**
Line_read	Read this line	Line data
Remote_store	Update your copy with this data	Acknowledgement

Requests sent by a directory to a L2 cache are shown in Table 3.4. In the case of receiving a line_read request to the line that is not owned by none of the caches, the directory transfers this request to the L2 cache. Corresponding line data is then provided by this L2 cache. After receiving a flush or flush_all_L2 request from an L1 cache, these requests are forwarded to the L2 cache. The L2 cache responds these requests by sending an acknowledgement.

Table 3.4. *List of requests sent by a directory to an L2 cache*

Request	**Meaning**	**Response**
Line_read	Read this line	Line data
Flush	Write this line	Acknowledgement
Flush_all_L2	Flush all dirty lines	Acknowledgement

3.2.2. State transfers

The ESI protocol is demonstrated in Table 3.5. Each L1 cache keeps state bits and a dirty bit for each one of its lines. Initially, all cache lines are in invalid state. This state indicates that the cache does not contain valid data for this line. The state of a block is E if the cache is unique owner of a clean or dirty line. Since the protocol does not contain modified state, the L1 cache determines dirtiness from the dirty bit. Shared state means that the line is potentially shared by at least one cache. A shared line can also be either clean or dirty. Moreover, more than one cache can separately write to the same line without invalidating other caches' copies due to the write-update policy.

Only requests received from the thread unit is accepted to the line in I state. The cache sends a line_read request to the directory after it receives a request corresponding to an invalid line. The directory responds this request by providing new state (E or S) and line data. The cache loads this line data in given state. In the case of receiving store request, the data sent by the thread unit is stored and the dirty bit is set after the line becomes hit. Moreover, the cache also sends remote_store request if the thread request is store and the state is shared. Other sharer caches, therefore, update their copy with recently written data.

A cache receives load, store and line_read requests to the exclusive line as shown in Table 3.5. A remote_store request is not received from the directory since the cache is

unique owner of an exclusive line. The cache immediately supplies the data to the thread without changing the state and the dirty bit when a load request is received from the thread unit. On the other hand, the state remains same and the dirty bit becomes one, after receiving store request is from the thread unit. The line data is sent to the directory when a line_read request is received. Therefore, the state becomes shared while and dirty bit is preserved.

The state of a shared line remains same after receiving requests such as load, store, line_read and remote_store. The dirty bit does not also change after a load request is received from the thread unit. In the case of receiving a store request, the cache sends a remote_store request to the directory. The new data is stored, and the dirty bit is set after the remote_store acknowledgement is received. The cache provides the line data to the directory when it receives a line_read request to the shared line. After receiving a remote_store request, new data is stored and then an acknowledgement response is sent to the directory Note that the dirty bit is not affected by a remote request.

Table 3.5. *The ESI Protocol*

Current		**Request**	**Next**		**Coherence Transaction**
State	**D. Bit**		**State**	**D. Bit**	
E	C	Load	E	C	-
	D			D	
	C	Store	E	D	-
	D			D	
	C	Line_read	S	C	Line Data
	D			D	
S	C	Load	S	C	-
	D			D	
	C	Store	S	D	Remote_store
	D			D	
	C	Line_read	S	C	Line Data
	D			D	
	C	Remote_store	S	C	Acknowledgement
	D			D	
I	-	Load	E	C	Line_read (E)
	-		S	C	Line_read (S)
	-	Store	E	D	Line_read (E)
	-		S	D	Line_read (S) & Remote_store

4. HARDWARE DESIGN

In proposed model demonstrated in Figure 4.1, each private L1 cache is connected to a hardware thread unit. These thread units are finite state machines that are generated by the compiler. L2 caches are partitioned through address space while directories are distributed over L2 caches. Note that this work does not include design of L2 caches, these caches have already been implemented for compiler library. Besides an implementation with connecting directories to L2 caches, the directories can be connected directly to the memories without L2 caches.

L1 caches and directories are connected to each other with two interconnection networks (Coherence 1 and Coherence 2) shown in Figure 4.1. These are butterfly networks specified in [6]. Coherence 1 is responsible for transmitting requests coming from L1 caches to the corresponding directory and then sending response that is received by this directory to the requester L1 cache. Requests are sent by a directory to an L1 cache via Coherence 2 network. Responses of L1 caches are also sent to the directory via this network. Cache to cache communication does not occur in proposed design since the directory manages the communication between caches.

In this model, L1 caches connected to the same directory can belong to different coherence domain. The directory determines caches that are in same coherence domain by masking and manages coherence transfers between these caches only.

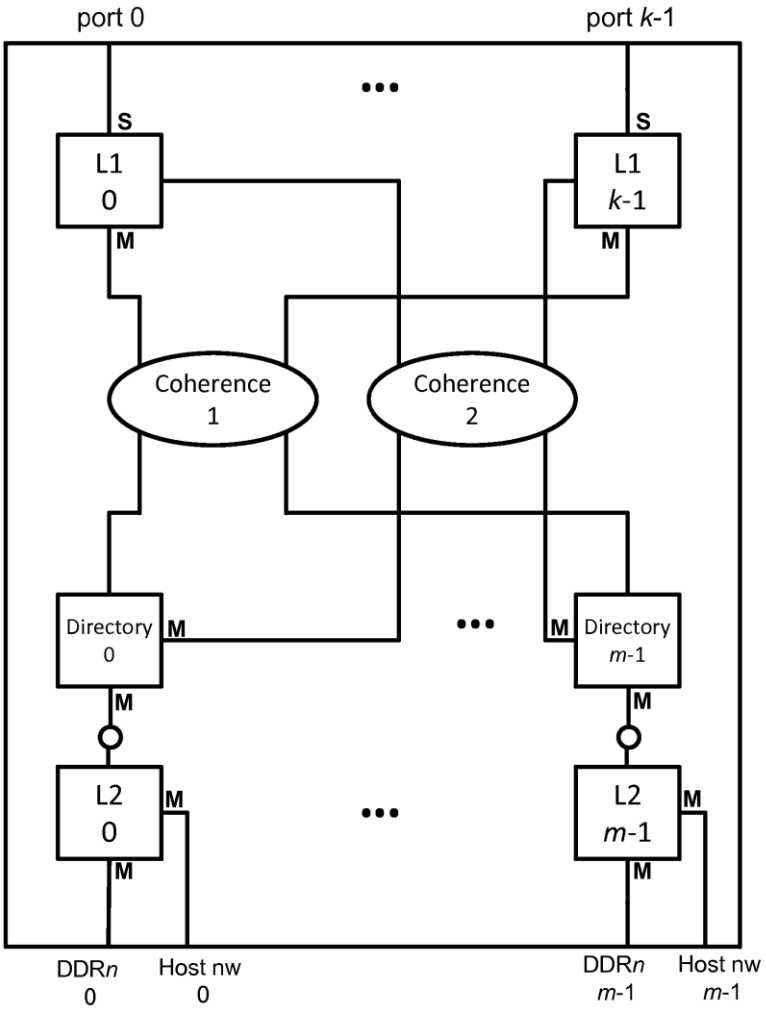

Figure 4.1. *The proposed model [6]*

4.1. Ports and Message Formats

Each directory and L1 cache have master and slave ports containing one sending and one receiving FIFO buffer, separately. In slave port, one of them is used for receiving requests while other one is used for sending responses while one of them is used for sending request while the other one is used for receiving responses in master port. In Figure 4.1, master and slave ports are shown by M and S letters, respectively.

Each private L1 cache has three ports as a slave thread port, a master coherence port and a slave coherence port. An L1 cache receives thread requests with the slave thread port. Slave thread receive message contains the data, the address and the opcode fields as shown in Figure 4.2. The opcode field indicates type of the sent request. Response message to the thread request is also demonstrated in Figure 4.2. This message contains data only. An L1 cache receives coherence requests coming from the directory via

Coherence 2 network from the slave coherence port. Message formats to receive coherence request and to respond them are illustrated in Figure 4.2. Slave coherence receive message contains data, address, opcode, cache id and directory id fields. Cache id and directory id fields indicate the id of the cache corresponding this request and the directory sending the request, respectively. Slave coherence send message includes line data, opcode and directory id fields. Coherence requests are sent by an L1 cache to a directory via Coherence 1 network with the master coherence port. Sending and receiving message formats of master coherence port are also shown in Figure 4.2. Master coherence send message contains domain id, byte mask and line data, opcode, tag and cache id. Coherence domain that the cache belongs to is indicated in domain id field. Byte mask indicates dirty bytes of the line. Tag is used to specify the acknowledgement responses. Finally, the master coherence receive message includes ESI state, line data, opcode, tag and cache id fields. ESI state is state of the requested line.

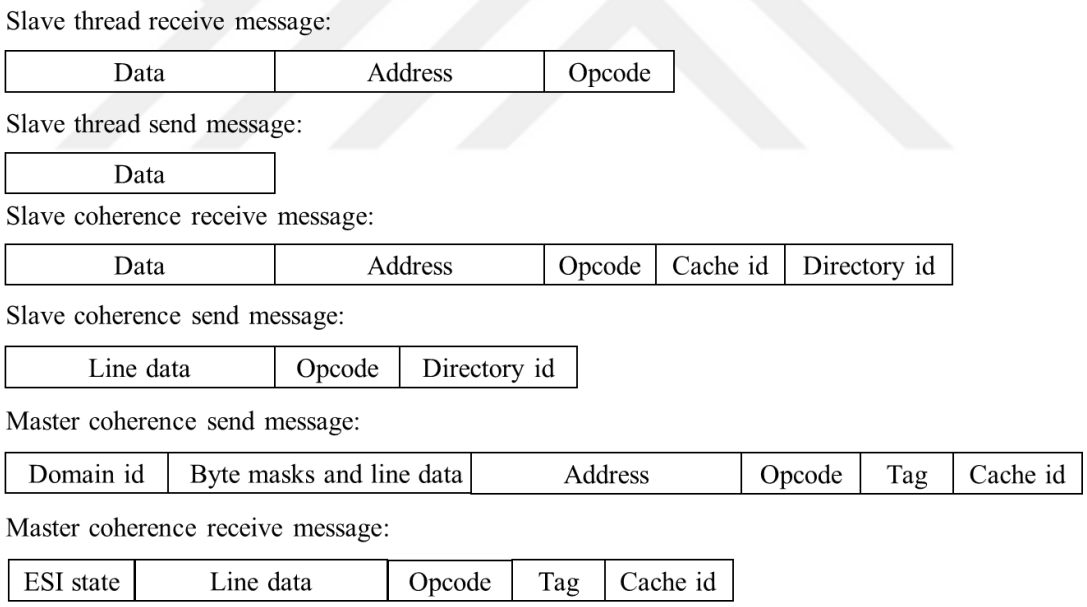

Figure 4.2. *Message formats of L1 caches' ports*

Each directory has three ports as slave coherence receive port, master coherence port and master L2 port. Message formats of directory ports are demonstrated in Figure 4.3. Directory receives requests sent by L1 caches and then sends responses via Coherence 1 network from slave coherence port. Slave coherence send and slave coherence receive message formats of the directory are analogous to master coherence

receive and master coherence send message formats of L1 caches, respectively. Requests are sent by a directory to L1 caches via Coherence 2 network through master coherence port. Master coherence receive and master coherence send message formats of the directory are also analogous to slave coherence send and slave coherence receive message formats of L1 caches, respectively. Master L2 port is used for sending requests and receiving responses from an L2 cache. Master L2 send message consists of line data with byte masks, line address, opcode and directory id fields. Moreover, master L2 receive message contains line data, opcode and directory id fields.

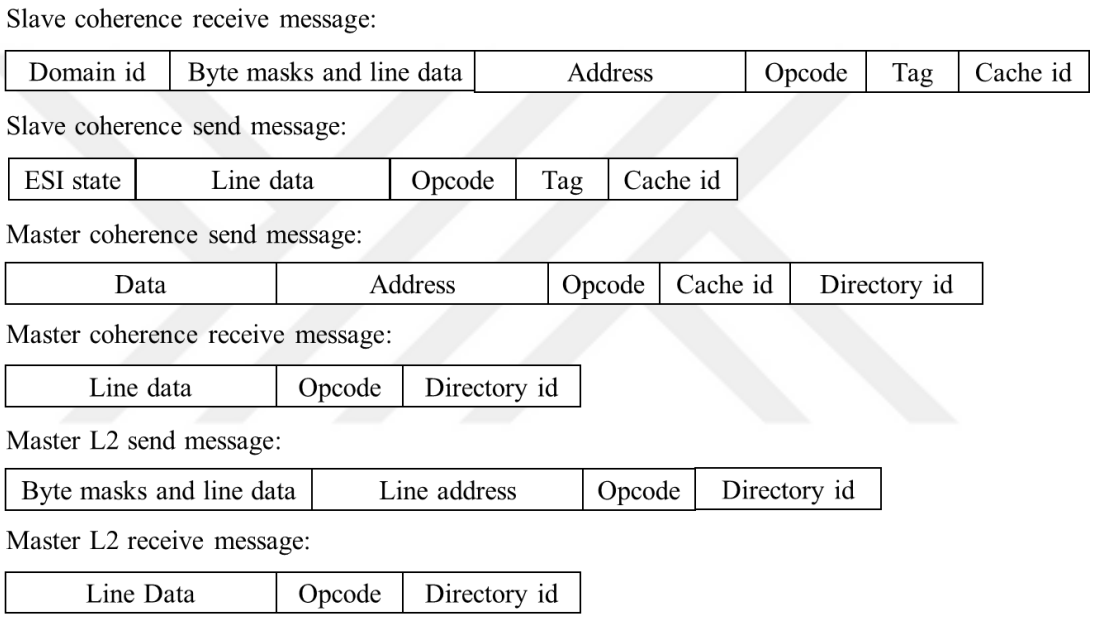

Figure 4.3. *Message formats of directories' ports*

4.2. Proposed Hardware Architecture of L1 Caches

Private L1 caches are implemented as direct-mapped write-back caches. Although direct-mapped caches are not effective in terms of hit rate, they are more suitable for FPGAs' structure and can easily be implemented using existing block memories in the target FPGA device. However, set associative caches are relatively expensive to build on FPGA in terms of area and performance since they require extra logic to handle associativity. Number of lines and number of words in a line are arbitrary parameters that are used to configure the direct-mapped cache and determined by the compiler. Number of words in a line is same for all caches although number of lines can be different. A

dirty line is written back to the L2 cache after it is replaced since caches follow write-back policy.

L1 caches are pipelined to improve performance and the pipeline is illustrated in Figure 4.4. This pipeline has four stages as access, compare, check and retire, respectively, from bottom to top in Figure 4.4.

As explained before, each L1 cache has slave thread, slave coherence and master coherence ports. In the access stage, both requests received from slave thread and slave coherence ports are accepted to the cache pipeline if there is no stall in the compare stage.

Access stage contains a block memory for tag and state and corresponding addresses of both coherence and thread requests are sent to this block memory. After one clock cycle, the tag and state information available at the output of the access stage that is fed into the next stage, which is compare state.

In the compare stage, tag and state information becomes ready for both requests. The information of being hit or miss of the corresponding line is also controlled in parallel. The line is hit if it is valid and read tag matches with the tag of the requested line.

The check stage consists of a cache controller (main FSM), a victim cache unit, block memories for data, dirty bit and byte mask information. The FSM guarantees that only one request is accepted to this stage at a time. This request can be waiting thread request, coherence request or new thread request. If the waiting thread request does not exist, a coherence request is prior than a new thread request. Normally, the coherence request is always hit, however, it can also be miss in some specific situations. Coherence requests that are miss are managed by using the victim cache. In the case of being hit, the data is read from or is written to data BRAM. In the case of receiving a thread request that is miss, this request is saved to waiting_thread_request register and a line_read request is sent to the directory. After line data is obtained, it is again saved to the register until the old line is flushed or abandoned if line replacement is required. The corresponding line is then become hit. Until waiting_thread_request is completed, coherence requests are maintained while new thread requests are stalled.

In the last stage, the retire stage, the response of both thread and coherence requests are sent through slave thread send port and slave coherence send port, respectively.

Details about cache components and cache's behavior after each request are given in the following paragraphs.

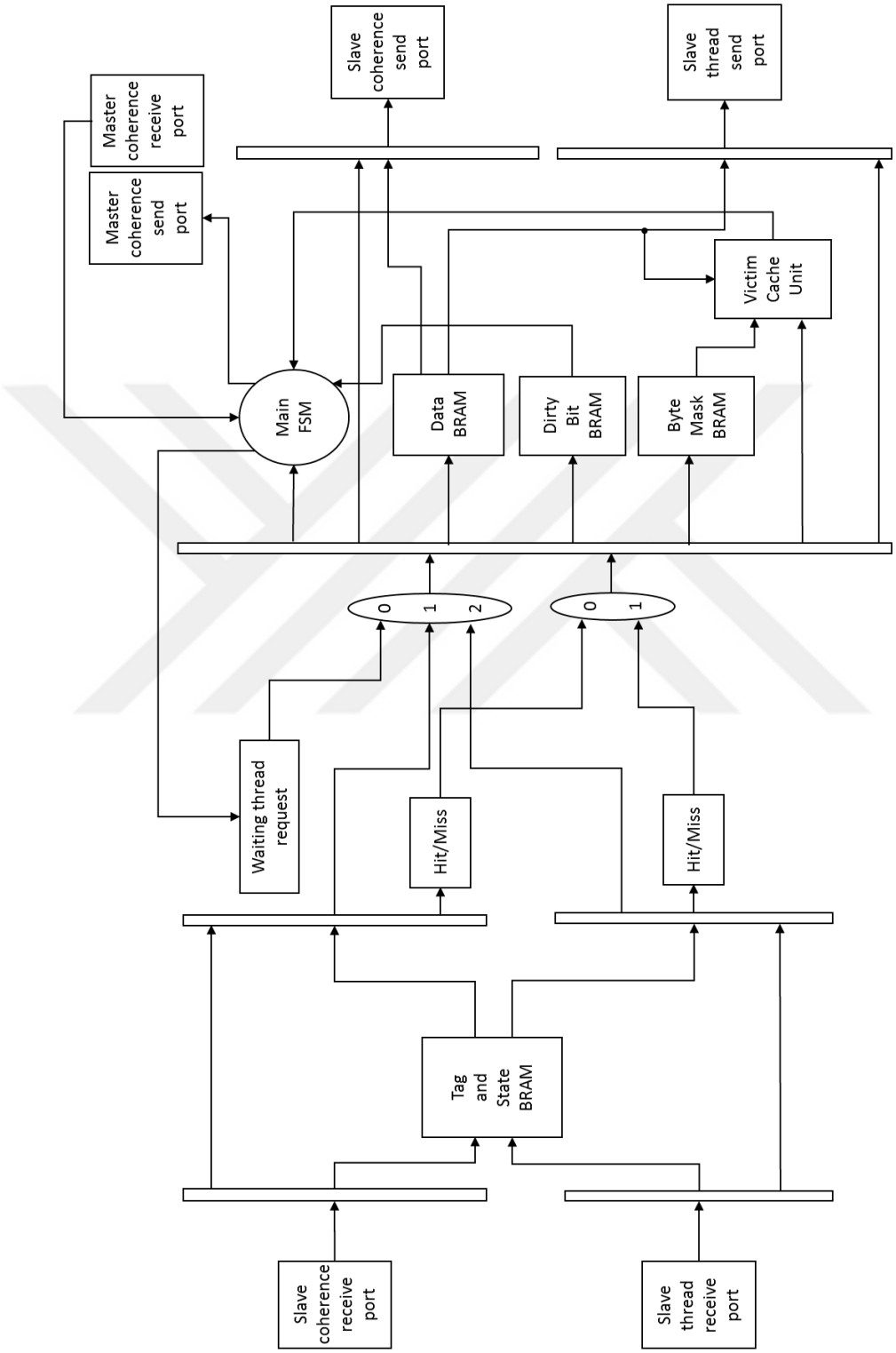

Figure 4.4. *The L1 cache pipeline*

Each L1 cache contains four block RAMs (BRAMs) for keeping tag and state, dirty bit, byte mask and data. Tags and ESI states are kept in tag and state BRAM for each cache line. Tag size is determined according to the number of lines in the cache and the line size. Two bits are enough to keep three different ESI states. A dirty bit is stored for each cache line in dirty bit BRAM to specify lines that are written before. A byte mask is kept for each word in byte mask BRAM. Dirty bytes in a word are indicated by these byte masks to prevent false sharing errors. Finally, data corresponding to each word is kept in each entry of data BRAM.

Each cache includes a special victim cache unit to manage flush, abandon, line_read and flush_all_L1 requests. A victim cache unit prevents some hazards. This unit reads words of a line from data BRAM when reading a whole line is required. The main cache sends related index of data BRAM when sending flush or abandon request is required. The victim cache unit reads the line data that is replaced and then store it to its victim cache. In the case of flush request, byte masks of this line are also read and then they are sent to the cache with the line data. If this flush request is sent due to receiving flush_all_L1 request, the line data is not stored in the victim cache. In the case of abandon request, the line data is only read to keep in the victim cache since the main cache does not need to line data. The victim cache unit receives line_read request that is hit or miss. This unit reads the line data from data BRAM and provides it to the main cache in the case of hit. Otherwise, the line data should exist in the victim cache. In this case, the line data is read from the victim cache and then it is sent to the main cache. The victim cache unit also receives remote_store requests if these requests correspond to a line in the victim cache. The victim cache unit sends the remote_store data to its victim cache.

Victim cache is a small fully associative cache. This cache keeps flushed and abandoned lines and receives line_read and remote_store requests only related to these lines. In the case of receiving line_read request, the victim cache reads the line data and provides it to the victim cache unit. Then, the victim cache unit supplies the line data to the main cache. In the case of receiving remote_store, the victim cache determines the corresponding word of the line and updates it with new data. Acknowledgement responses corresponding to flush and abandon requests are also transmitted to the victim cache. Therefore, these lines become invalid in the victim cache.

Each L1 cache contains a find dirty line unit. This unit determines indexes of dirty lines from dirty bits and provide them to the victim cache when a flush_all_L1 request is received.

The thread unit sends a load request to a cache to read the whole word or just a specific portion of a word such as a half word or a byte. This information is extracted from specific bits of the opcode field as shown in Figure 4.5. First and second bits of the opcode field indicates that the load request is a word, a half word or a byte. Right most two bits of the address field indicates of which half word or byte of word is requested. A mechanism inside the cache computes the requested data and fills the remaining bits with zero or sign bit. These bits are sign bit if the leftmost bit of opcode is one and vice versa.

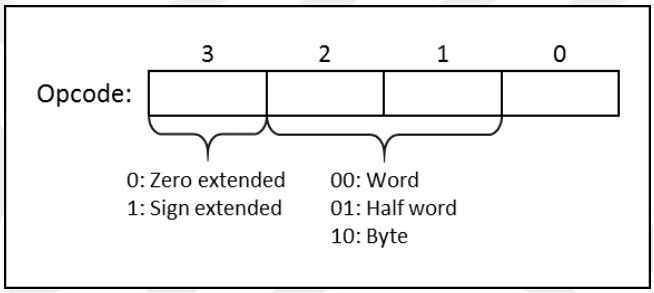

Figure 4.5. *An opcode of a load request*

A store request corresponds to a word, a half word or a byte. Specific bits of opcode field indicate the type of a store request as demonstrated in Figure 4.6. The rightmost two bits of the address field shows which portion of the word will be modified. A mechanism inside the cache computes the byte mask to indicate the bytes to be written. The new data is then accordingly stored to data BRAM by using this byte mask.

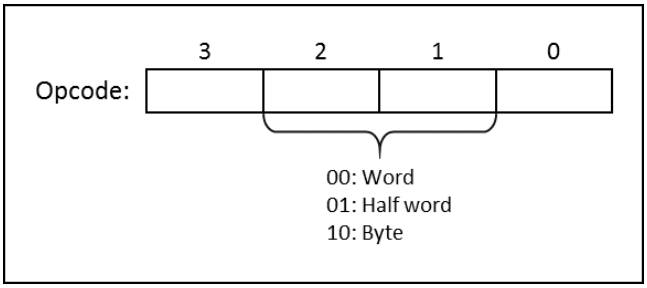

Figure 4.6. *An opcode of a store request*

Flush L2 request can be received from the thread, after all dirty L1 cache lines are written back. Only one of the coherent L1 caches receives this request and it is responsible for forwarding this request to its directory. An acknowledgement is sent to the thread after receiving acknowledgement from the directory.

Flush_all_L1 request is sent by the thread when all thread requests of all L1 caches are completed. After this request is received by the cache controller, the find dirty line unit is activated. This unit determines dirty lines and sends the addresses of the lines to the victim cache unit. The victim cache unit reads the corresponding words and the byte masks of these lines and then transmits them to the cache. Therefore, all dirty lines are written back to the L2 cache through the directory.

The flowchart that is illustrated in Figure 4.7 demonstrates how a load request is handled in the cache when it receives a load request from its thread. If this load request is hit, the requested data is computed and directly provided to the thread. Otherwise, a line_read request is sent to the directory to obtain the line data. The load request is kept in a register until receiving the line data from directory. Therefore, the cache controller continues to accept coherence requests. After the line data arrives, this new data and the state are stored in the corresponding BRAMs with new tag. Dirty bit and byte mask BRAMs are also updated to indicate that this line is clean. A flush or an abandon request is then sent to the directory if a replacement is required, in other words, the corresponding cache entry contains another valid line. Flush request is sent if the current line is dirty. Otherwise, the request is abandon. Flushed or abandoned line is stored in the victim cache before it is sent to provide temporary storage of the victim line in order to improve the hit latency. The cache continues to keep the request in register and coherence requests are accepted if a flush or an abandon request has been sent. After the acknowledgement response arrives, the cache sends new data to the thread.

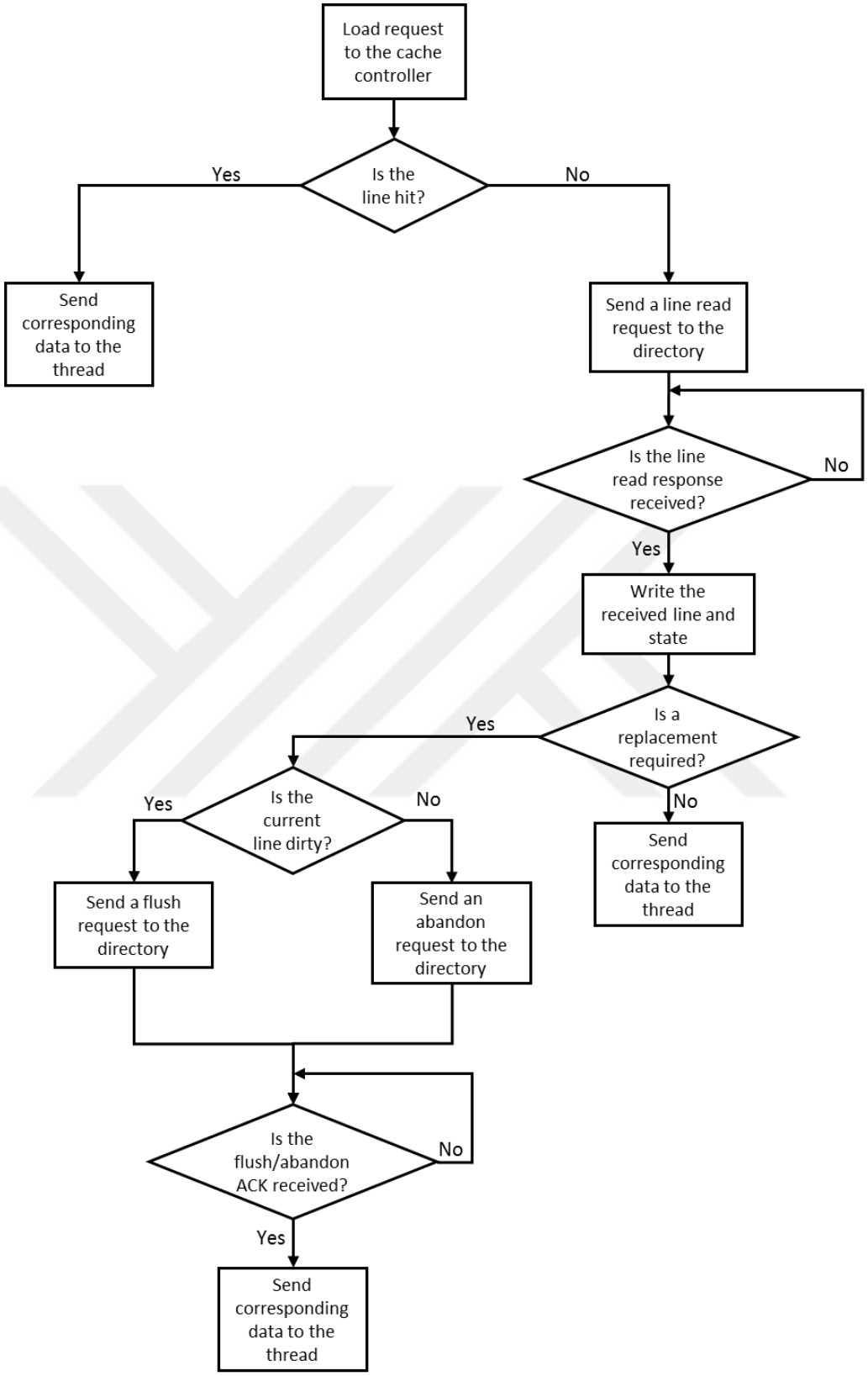

Figure 4.7. *The flowchart of a load request*

The flowchart is depicted in Figure 4.8 to show how the cache handles a store request that is received from its thread unit. When a store request is received from the thread unit, the cache controller first controls that the line is hit or miss. In the case of being hit, state of this line is determined by reading the state memory. A remote_store request is sent to the directory if the line is in S state. The cache controller accepts coherence requests until an acknowledgement is received from its directory. After an acknowledgement is obtained or if the cache line is in E state, the new data is written to the data BRAM. The dirty bit BRAM and byte mask BRAM are also updated to indicate the dirty line and dirty bytes of this line, respectively. Then, an acknowledgement is sent to the thread unit. In the case of being miss, same steps with the load miss processes, which is given in Figure 4.7, are followed. After the line data is provided and replacement operation is finished if flush or abandon is sent, the store hit process has been performed.

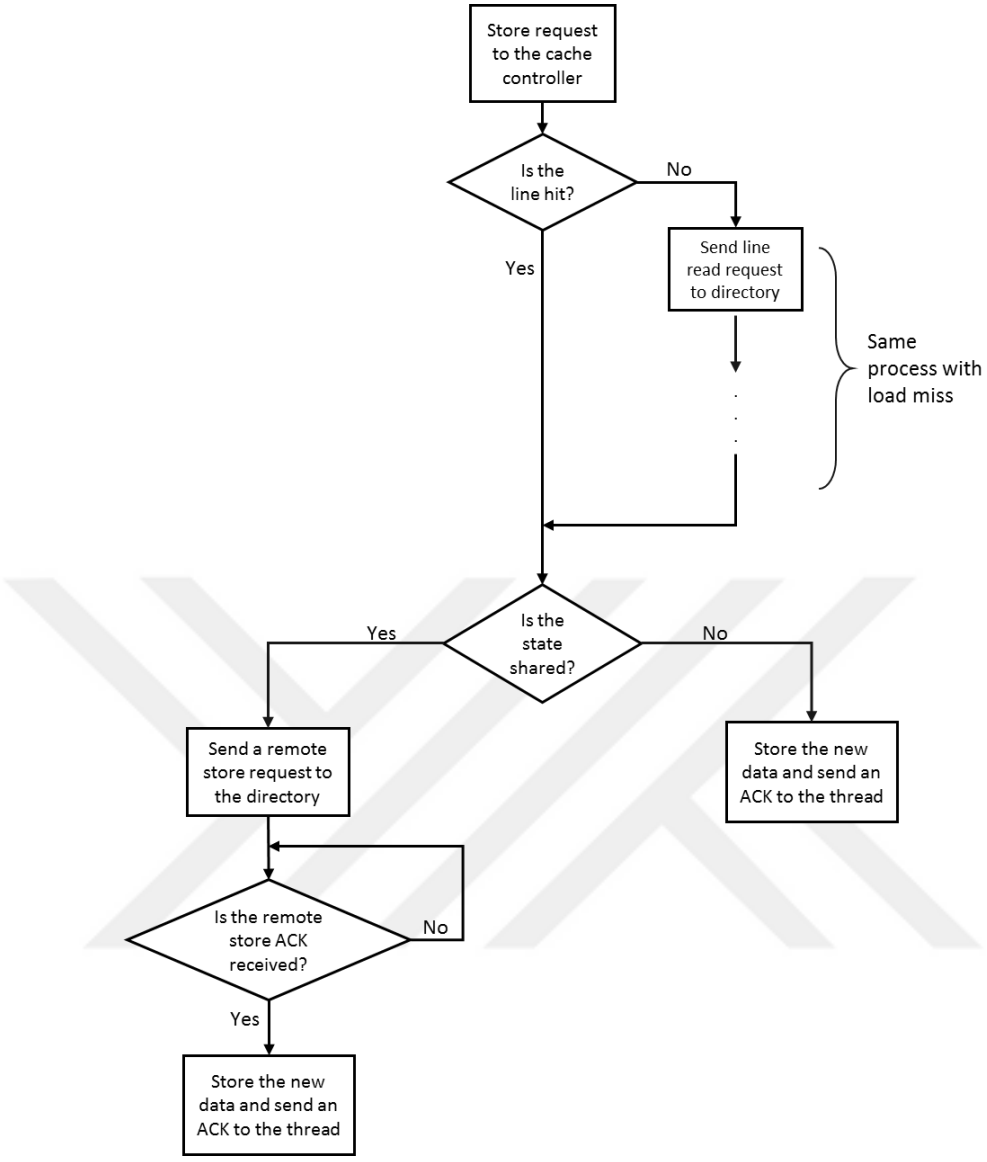

Figure 4.8. *The flowchart of a store request*

A cache receives line_read and remote_store request from its directory. Inherently, these requests must be always hit since the directory knows correct set of owner caches. However, they can be miss when a cache is replaced with a line and this request does not arrive to the directory yet. After receiving remote_store request that corresponds to hit line, the cache stores this data and then sends an acknowledgement to the directory. A remote_store request does not impact corresponding byte mask and dirty bit. In the case of being miss, the data is written to the victim cache without updating data BRAM. The remote_store request can be sent to the victim cache unit even if it is hit in one special condition. Assume that a thread request is stalled after a line_read and a flush or an abandon request is sent, and the line_read response is not arrived yet. After the cache

receives remote_store request for the flushed or abandoned line, it becomes hit since the new line data is not obtained yet. In this special condition, the cache should write this remote_store data to both the data BRAM and the victim cache.

Line_read requests are always managed by the victim cache unit. In the case of receiving line_read request to the line that is hit, this unit reads corresponding line from data BRAM word by word while sending each read word to the slave coherence port. This port collects these words and sends them to the directory as a line. In the case of being miss, the line is read from the victim cache and is then sent to the directory.

4.2.1. Possible hazards in an L1 cache and their solutions

Although the compiler itself prevents most of the hazards, the cache is exposed to some hazards that are discussed in this section. The compiler ensures that flush all and flush L2 request are sent by threads after all requests are completely responded by all caches. Hence these requests do not cause a hazard.

The compiler also ensures that any cache does not receive load or store requests before previous dependent store request is completed. This property solves many problems caused by receiving store request followed by dependent load or store request. The line_read and remote_store requests are also independent from previous store request since all caches do not receive dependent thread request until an acknowledgement to the store request arrives. This assurance provided by the compiler prevents hazards caused by receiving thread or coherence requests that is dependent to previous remote_store request.

Load request that is followed by dependent load, store, line_read or remote_store request does not cause a hazard. The cache responds these requests without requiring any additional mechanism. This case is also valid for line_read requests. After receiving line_read request, the cache responds all dependent thread and coherence requests.

Assume that the cache replaced a line and then received a load or store request for this line. In this situation, the cache requests this line from the directory. In the case of receiving line_read or remote_store request after replacement, these requests are handled via victim cache. The line data is read from victim cache when this request is a line_read. A remote_store data is written to the victim cache. Therefore, the cache supplies recent data if it receives line_read request to the same line.

4.3. Implementation Details of Directories

Each directory is designed as a set associative cache to increase hit rate. Remind that each address maps to a set and each set contains several locations called as way in a set associative cache. Number of sets and number of ways are arbitrary parameters determined by the compiler and each directory is configured with these parameters. Directory is not pipelined for this initial version. A detailed explanation of the directory and its design details are given in the following.

Each directory is responsible for all caches that are connected to it. These caches can belong to different coherence domains. The directory is designed as a full-map directory model. In full map directory schemes, presence bits (owner set) that indicate owner caches for each block are stored in directory entries. This owner set includes caches that belong to different coherence domain.

Each directory has three ports, slave coherence, master coherence and master L2. Each port contains sending and receiving FIFOs, separately. Slave coherence port is connected to the Coherence 1 network. The directory receives requests of the L1 caches via this port. Master coherence port is used for sending requests to L1 caches via the Coherence 2 network. The directory sends requests to L2 Cache via master L2 port.

Two type of BRAMs, tag BRAM and owner set BRAM, are contained in each directory. Tag BRAM keeps tags for each directory entry. Owner set BRAM includes presence bits for each cache and for each directory entry. Both BRAMs are replicated to number of ways.

The directory first initializes all BRAMs, before accepting any request. The directory reads all tag BRAMs to control that the requested line is inside one of them or not. If it is not inside the ways, one of the empty ways is chosen to save the new tag. In this model, we assume that an empty set is always available hence a replacement is not required. After computing the set, the directory reads corresponding owner bits.

A directory receives some requests such as line_read, remote_store, flush, abandon and flush L2 requests from L1 caches. A flush L2 request is directly sent to the L2 cache. Flush and abandon requests are also sent to the L2 cache after updating the owner set. Corresponding line data to line_read request is obtained from one of L1 caches or the L2 cache. Remote_store request is only transferred to sharer caches.

The directory reads the corresponding owner set, after receiving request from one of the L1 caches. This owner set contains presence bits of all caches that is connected to the directory even if they belong to different coherence domain. Coherence domain is first determined to find owner set. The information about which coherence domain of the requester cache belongs to is included in the request received. A mask is applied to the owner set by the directory to extract the owner set bits of corresponding coherence domain.

The directory first controls if this tag exists in one of the ways after receiving line_read request. If this tag is not included in any way, it means none of the caches own this line. In this case one of the empty ways is chosen to store the corresponding tag. The owner set bits corresponding to this way are initially all zeros. Even if the tag is matched, corresponding owner bits to coherence domain of the requester cache may be all zero. In both cases, since the line is not read by one of the caches, this line_read request is transferred to the L2 cache as demonstrated in Figure 4.9. This line data is sent in exclusive state to the requester and the presence bit corresponds to this cache is set when the line data response received from the L2 cache. In the case of receiving line_read request to the line that owned by at least one cache in the coherence domain, the line_read request is transferred to the one of these caches. In fact, the cache that is indicated by the rightmost bit of owner set bits is chosen in the case of more than one owner. The line data is sent to the requester cache in shared state and the owner set bit of the requester cache is set when line data is received.

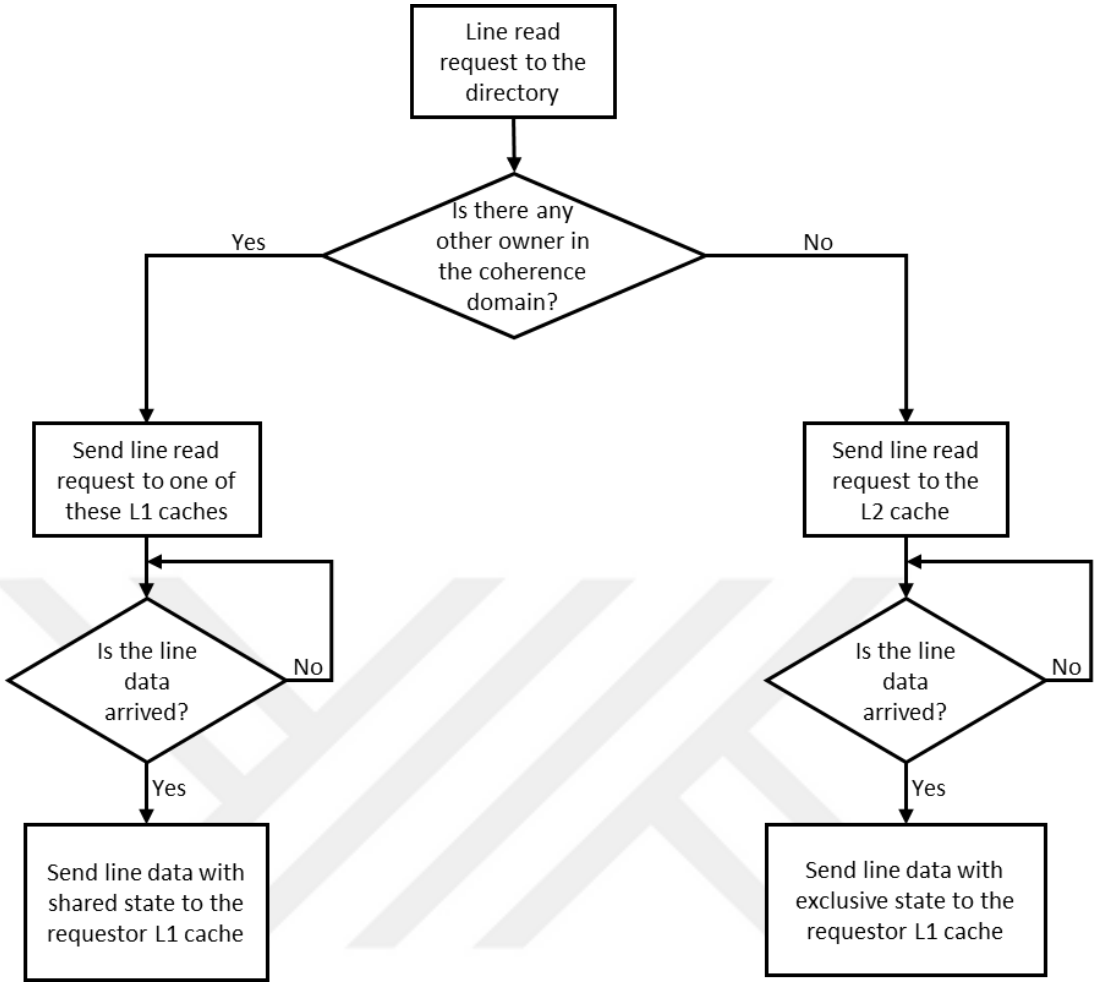

Figure 4.9. *The flowchart of a line_read request coming to the directory*

The processes after the directory receives a remote_store request from L1 caches is shown in Figure 4.10. The directory first controls if any sharer cache is in the corresponding coherence domain. The directory can then receive a remote_store request to an exclusive line since shared state is sticky. Moreover, the directory ignores this request and sends an acknowledgement to the requester if only the requester cache own this line. Otherwise, the remote_store request is transferred to one of the owner caches by the directory. This request is transmitted to another owner cache when an acknowledgement response is received. This process is continued until the remote_store request is transmitted to all sharers. An acknowledgement is sent to the requester cache when last the acknowledgement to the remote_store request is received.

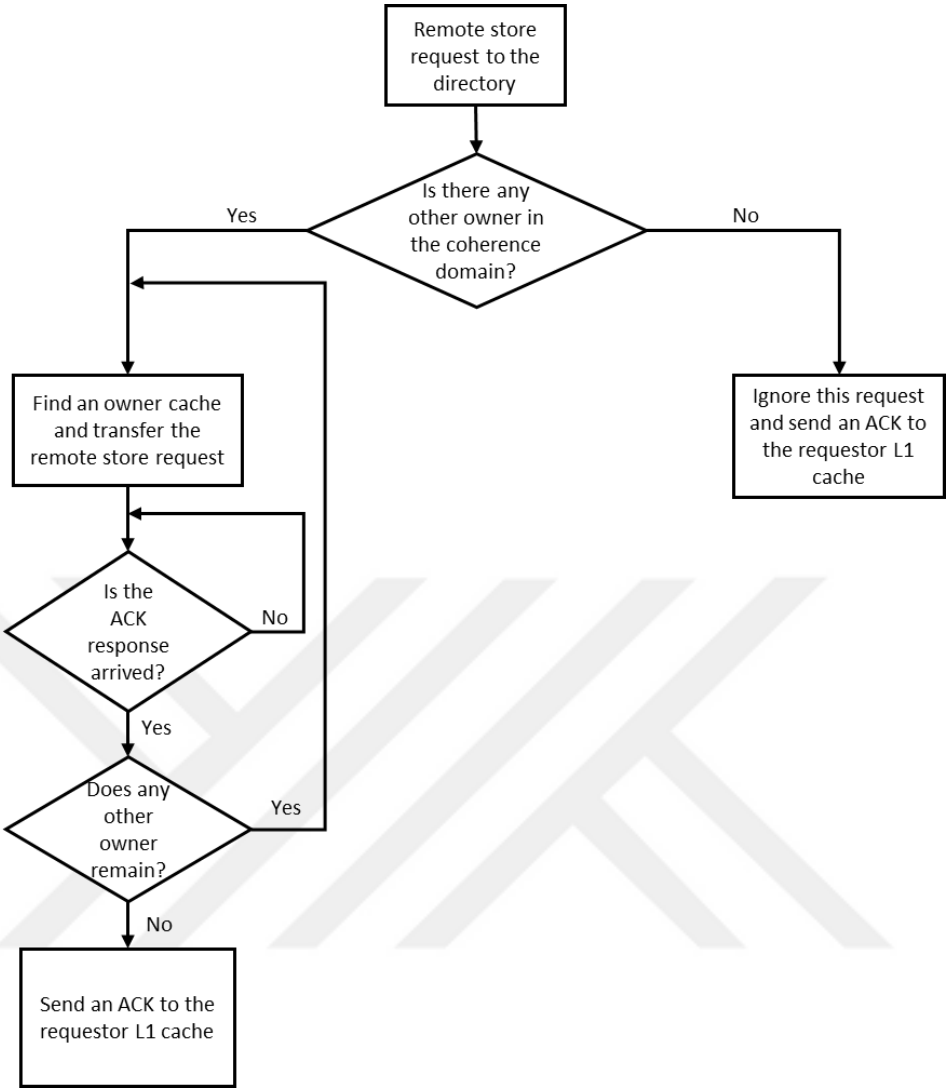

Figure 4.10. *The flowchart of a remote_store request coming to the directory*

4.4. Data Races and Their Solutions

Assume that presented design is implemented with N number of caches that are in the same coherence domain and all of them are connected to the same directory. Following paragraphs proves that the results are correct in all possible data races situations even if the order of requests changes before arriving to the directory.

Assuming that initially none of the caches have the requested line, C_1 and C_2 receive load or store request for this line, respectively. Line_read request of C_2 arrives to the directory before C_1's line_read request. The directory reads this line data from L2 cache and then sends to the C_2 in E state. When the line_read request of C_1 cache arrives, the

directory forwards this request to C_2 cache. C_2 cache changes the state of this line as shared and provides the line data. Then, the directory sends this line data in shared state to the C_2 cache. As a result, both caches have same data in the shared state, regardless of the order.

This time, assume that initially C_2 and C_3 have the same line. Assume that C_2 receives a store request and then sends a remote_store request to the directory. Then, C_1 receives a load or store request to the same line and it sends line read request to the directory. However, C_1's line_read request arrives to the directory before C_2's remote_store request. The directory forwards this line_read request to the C_2. C_2 sends newly written data to the directory instead of old data. After sending this data to C_1, the directory receives the remote_store request. Then, it sends the remote_store request to both C_1 and C_3. Remind that compiler ensures that these requests do not overlap hence the order becomes unimportant.

Assume that initially only C_2 has the line and then C_2 replaces this line with sending flush or abandon request to the directory. Then, C_1 sends line_read request to the directory. However, C_1's line_read request arrives to the directory before C_2's flush or abandon request. The directory forwards this line_read request to C_2 since C_2 is still owner. C_2 provides this line data from its victim cache and then the directory sends the line data in shared state. Then, the flush or abandon request arrives to the directory. The directory removes C_2 among owners and also writes the line to the L2 cache if the request is flush. Finally, an acknowledgement is sent to the C_2 cache, therefore, C_2 remove this line from victim cache. Although C_1 becomes exclusive owner of this line while it has the line in shared state, this situation does not cause any problem.

Suppose that initially C_1 and C_3 have the line and C_2 sends a line_read request to the directory after it receives a load or store request. Then, C_1 receives a store request and then it sends remote_store request to the directory. However, C_1's request arrives to the directory before C_2's request. The directory sends this remote store request to only C_3 since C_2 is not one of the owners yet. After the remote_store request is completed, the directory receives the line_read request. The newly written data is received from C_1 and then it is sent to the C_2 by the directory.

Assume that initially both C_1 and C_2 have the line. Assume that C_1 and C_2 receive a store request and then send a remote_store request, respectively. However, C2's remote store request arrives to the directory before C_1's remote_store request. Then, the directory transmits this request to C_1. C_1 stores remote_store data before the store request that is received from thread. Then, the other remote_store request arrives to the directory and then it is forwarded to C_1. Therefore, C_1's stores remote_store data after storing the data received from its thread unit. Remind that the compiler ensures that store requests are independent hence the order of store requests are not important

Assume that initially C_1 and C_2 have the line. Suppose that C_2 first replaces the line and then sends flush or abandon request to the directory. Then, C_1 stores to this line and then sends remote_store request to the directory. However, C_1's remote_store request first arrives to the directory, and then the directory forwards this request to C_2. Although the line is miss in C_2, it exists in C_2's victim cache. Therefore, the line in victim cache is updated with new data. The directory receives the flush or abandon request, after completing remote_store request. If the request is flush, old data is sent to the L2 cache. However, that does not emerge a problem since line_read requests is sent to the C_1 instead of the L2 cache.

Assume that initially only C_1 has the line. This time, C_2 sends a line_read request and then C_1 sends flush or abandon request. However, C_1's flush or abandon request arrives to the directory before C_2's line_read request. After completing flush or abandon request, the directory receives the line_read request. Since none of the caches have the line, the line data is requested from the L2 cache. The directory then provides this data to the cache C_2 in E state.

Assume that initially C_1, C_2 and C_3 have the line. Suppose that C_2 and C_1 sends a remote_store and flush or abandon request to the directory, respectively. However, C_1's flush or abandon request arrives first to the directory. After completing flush or abandon process, the directory accepts the remote_store request. This request is only forwarded to the C_3 cache since C_1 is not one of the owners anymore.

Assume that initially both C_1 and C_2 have the line and C_2 and C_1 caches send flush or abandon requests, respectively, but C1's flush or abandon request arrives first. Therefore, the directory receives and completes C_2's flush or abandon request after

completing C_1's flush or abandon request. The order of flush or abandon requests is immaterial.

5. VERIFICATION EFFORTS, EXPERIMENTAL RESULTS AND DISCUSSION

5.1. Functional Verification Tests before Integration to Compiler

Before integrating cache coherent memory to the compiler, many tests have been performed for verify functionality of proposed memory architecture. Simulation environment implemented for these tests is illustrated in Figure 5.1. To perform these tests, new components, such as test bench, thread emulator and L2 emulator, have been added to design.

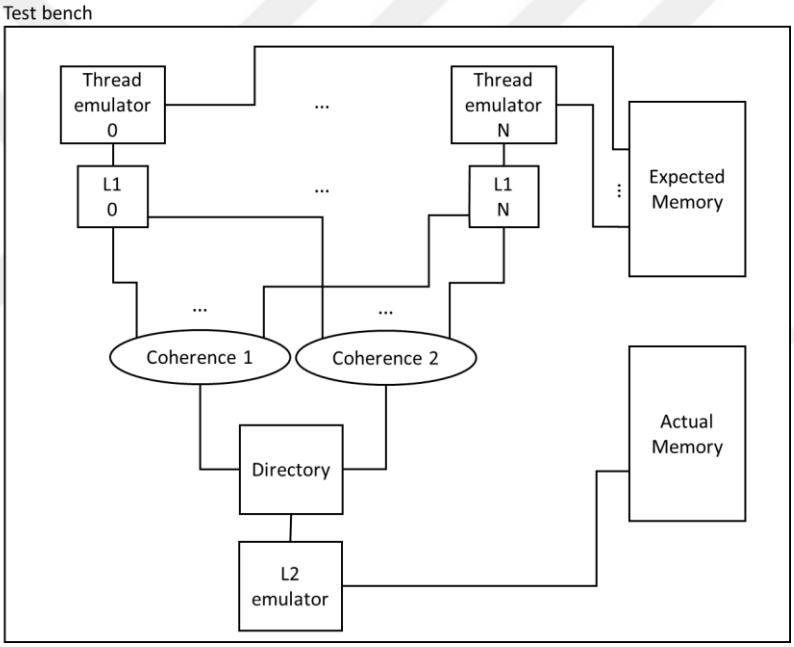

Figure 5.1. *Simulation environment*

Test bench unit is the top module that connects the coherent cache system to thread emulator units and a L2 emulator unit. Test bench also contains expected and actual memory footprints to verify the correctness of the proposed coherent cache system. These memories are initially filled with same random data in the beginning of test. Actual and expected memory are updated by the L2 and thread emulator units, respectively, during the test. These two memory units are compared at the end of test. In the case of being identical, simulation is successful meaning that coherent cache system works functionally correct.

The thread emulator unit is responsible for generating requests and sending these requests to the cache connected to itself as actual thread unit. In these requests, the opcode field is randomly selected as load or store and as a word, a half word or a byte. Data field and address field are also random to increase the coverage space of the infinitely many request possibilities. The randomly generated requests are sent to the coherent L1 caches after waiting during random clock cycle. In the case of receiving load request, this unit also controls correctness of the cache's response by comparing it with the response obtained by the expected memory. This unit sends flush_all_L1 request after all thread emulator units finish their load and store requests and all of these requests are responded. This unit also sends flush_all_L2 request if it is responsible for sending this request.

The L2 emulator unit is another component that emulates L2 cache. These unit responds line_read request by reading correct data from top unit. Flush and flush_all_L2 requests are also responded by sending acknowledgement response. These responses are sent after waiting for a random cycle to verify that cycles between responses does not matter in the correctness of the system.

The proposed memory architecture is tested with this simulation enviroment. These tests are repeated by changing parameters such as number of requests, caches' line size, directory's associativity and number of lines in caches and directory. The proposed memory architecture passed these random verification tests successfully.

5.2. Verification Tests of the Proposed Memory Hierarchy Integrated to Compiler

Proposed memory architecture presented in previous sections are tested with several algorithms written in C or C++ programming languages. These algorithms are converted to parallel hardware with our HLS compiler. Obtained results by generated hardware are verified by comparing the result of initial C or C++ program. The proposed memory architecture has been successful in 51 different extensive tests. Tested algorithms include some of the SPEC CPU2000 and SPEC CPU2006 benchmarks and some well-known algorithms such as greatest common divisor (GCD) and fast Fourier transform (FFT).

In all these tests, 2 L2 caches and 2 directories are generated as default. Each line of the L1 cache has 8 word and number of lines is 32 so each L1 cache capacity is 1Kbyte. Directories are 4-way set associative and number of sets is 64.

Number of generated coherent L1 caches depends on the test algorithm and they are in the range of 2 to 39. Number of coherence domains also depends on the algorithm. The generated number of L1 caches and coherence domains of selected tests are given in Table 5.1.

Table 5.1. *List of selected tests*

Test	Number of L1 Caches	Number of Coherence Domains
176.gcc	13	2
179.art	5	1
188.ammp	2	1
197.parser	2	1
254.gap	2	1
256.bzip2	30	9
403.gcc	4	2
445.gobmk	18	2
450.soplex	5	2
458.sjeng	39	10
471.omnetpp	11	1
998.specrand	2	1
FFT	2	1
GCD	2	1

5.3. Greatest Common Divisor Test

In GCD test, greatest common divisor of numbers 21 and 1020 is computed. The transfers during this test are shown in Figure 5.2. Both of L1 caches first receive a load request to read different words of the same line. Then, both of them send a line_read request to the directory_0 since all cache lines are initially miss. The request of L1_0 arrives first to the directory and then the directory forwards this request to L2_0. L2 cache reads corresponding page and sends the line data to the directory_0. The directory_0 transfers this line data to the L1_0 in exclusive state and set this cache as one of the owners of this line. L1_0 receives this exclusive line and provides thread requested word that is the first number 21. Simultaneously, directory0 receives line_read request of L1_1.

Directory_0 transmits this request to L1_0 since this cache is one of the owners of this line. L1_0 supplies corresponding line data to directory_0 and changes state of the line as shared. Directory_0 transfers the line data in shared state to L1_1 and set this cache as an owner of the line. L1_1 then receives the line data and supplies corresponding word to thread and this word is value of second number 1020. Finally, L1_0 receives store word request to a different line hence this line is miss. The cache sends a line_read request to directory_0. The directory provides line data from L2_0 and sends it to the requestor cache in exclusive state. The cache stores the new word after receiving line data and subsequently sends an acknowledgement to the thread. This new word is actually result of algorithm and the result value is 3. This cache writes the result to the memory via directory_0 when a flush_all request arrives.

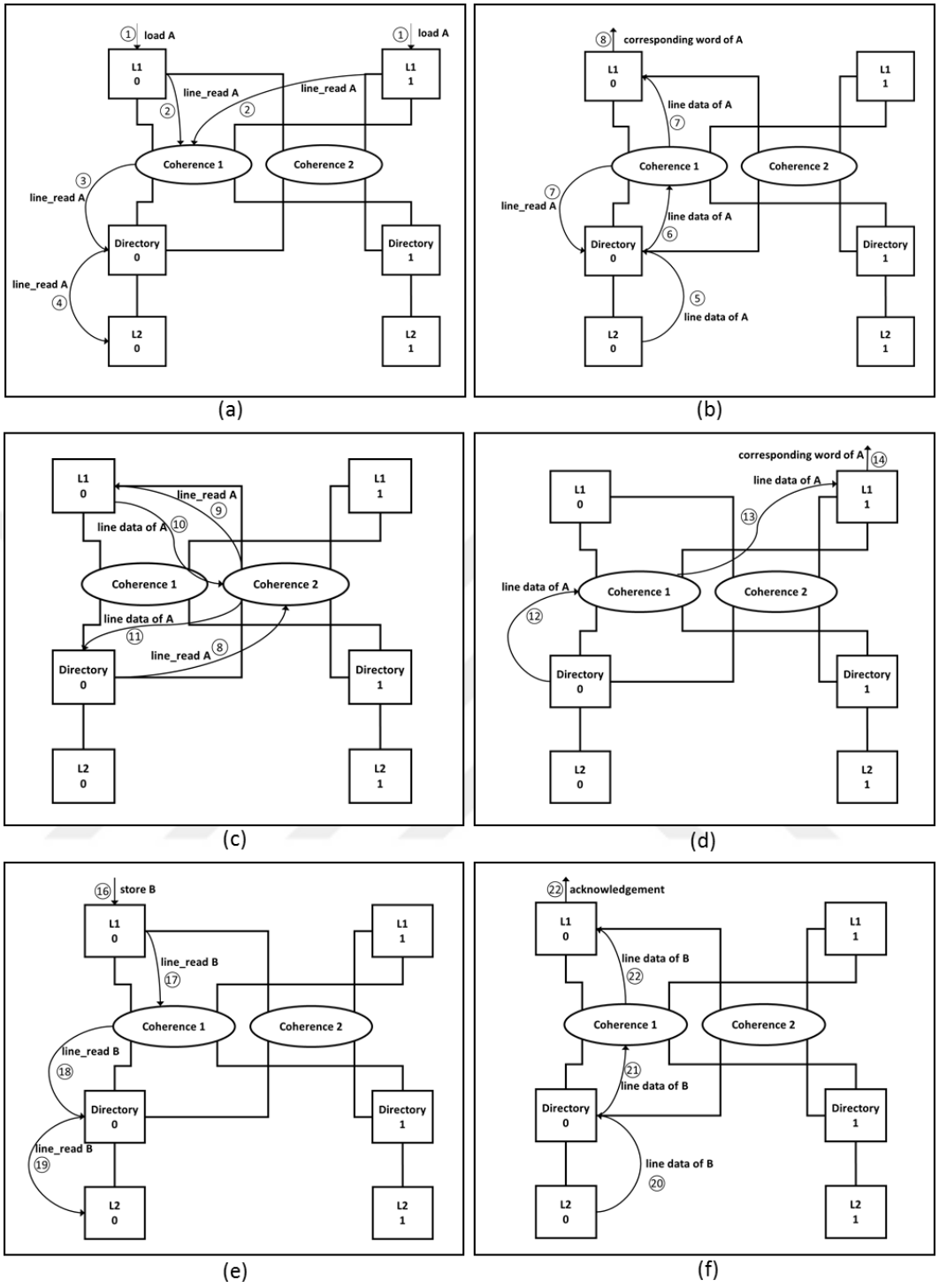

Figure 5.2. *Transfers during GCD test*

Table 5.2 shows the synthesis results of the GCD test. The synthesis is targeted to a Virtex7 xc7vx330t-3ffg1157 FPGA chip and ISE Design Suite 14.7 synthesis tool is used.

Table 5.2. *Synthesis results for GCD test*

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	38691	408000	9%
Number of Slice LUTs	35651	204000	17%
Number of fully used LUT-FF pairs	28375	45931	61%
Number of BRAM/FIFO	103	750	13%

5.4. Sjeng Test

458.sjeng test, one of the SPEC CPU2000 benchmarks, has the largest amount of coherent L1 caches. In this test, 39 coherent L1 caches which belong to different 10 coherence domains are generated. Table 5.3 shows distribution of L1 caches.

Table 5.3. *Distribution of L1 caches in 458.sjeng test*

Coherence Domain	Id of L1 Caches	Coherence Domain	Id of L1 Caches
0	38, 37	**5**	9, 8
1	36, 35	**6**	7, 6
2	34, 33	**7**	5, 4
3	32, 31, … ,13	**8**	3, 2
4	12, 11, 10	**9**	1, 0

Requests that are received to directories are analyzed in Table 5.4. Initially, several L1 caches send line_read requests to directories and these lines are supplied by L2 caches. Directory_0 receives 23 line_read requests. 15 of these requests supplied by L2_0 while other requests are provided by coherent L1 caches. Directory_1 receives 3 line_read request and all of them is sent to the L2_1. Directory_0 also receives 2 remote_store request to the different words of same line and this line is shared by 6 caches.

Table 5.4. *Analyze of requests that coming to directories*

Directory Id	Total Number of Requests	Number of Line-read Requests			Number of Remote-store Requests
		Total	Supplied by L2	Supplied by L1	
0	25	23	15	8	2
1	3	3	3	0	0

458.sjeng test is also tried with banked organization described in previous chapters. In this organization, 10 fully-associative L1 caches are used and these caches are connected to 2 L2 caches. Capacity of caches is identical to coherent caches organization. Number of line_read requests that arrives to L2 caches, in other words, number of misses is given in Table 5.5. In this organization, total miss rate is smaller since memory is partitioned. Besides, all of these requests are supplied by L2 caches since L1 caches do not communicate with each other.

Table 5.5. *Analyze of requests that coming to L2 caches in banked model*

	Number of Line-read Requests
L2_0	10
L2_1	10

Frequencies are measured for both model with Virtex7 xc7vx330t-3ffg1157 FPGA chip and ISE Design Suite 14.7. The frequency of cache coherent design is 261.433 MHz while the frequency of banked cache design is 261.146 MHz.

Figure 5.3 and Figure 5.4. show overall circuit of generated designs with coherent and banked model, respectively.

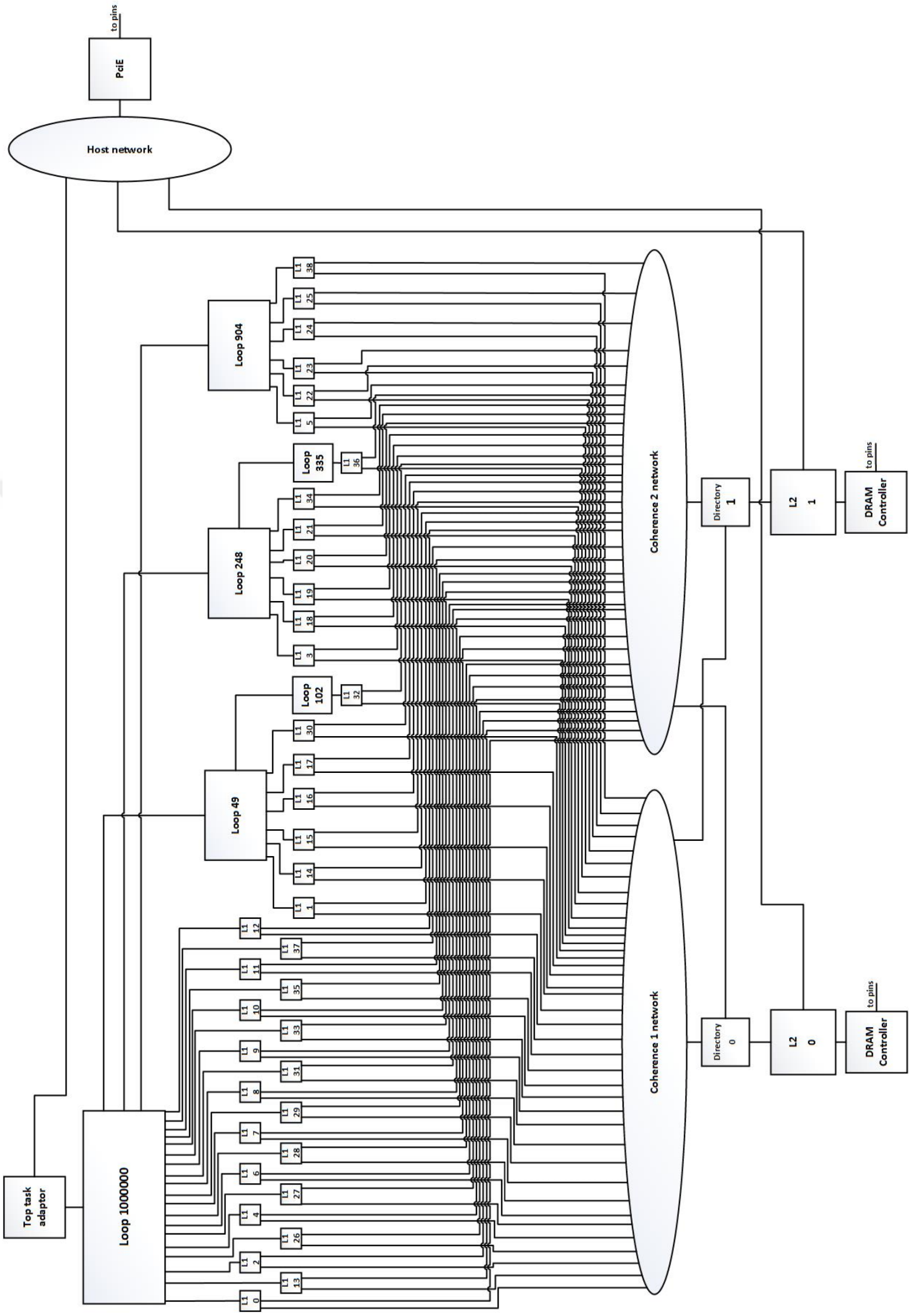

Figure 5.3. *Generated design for Sjeng test with coherent caches*

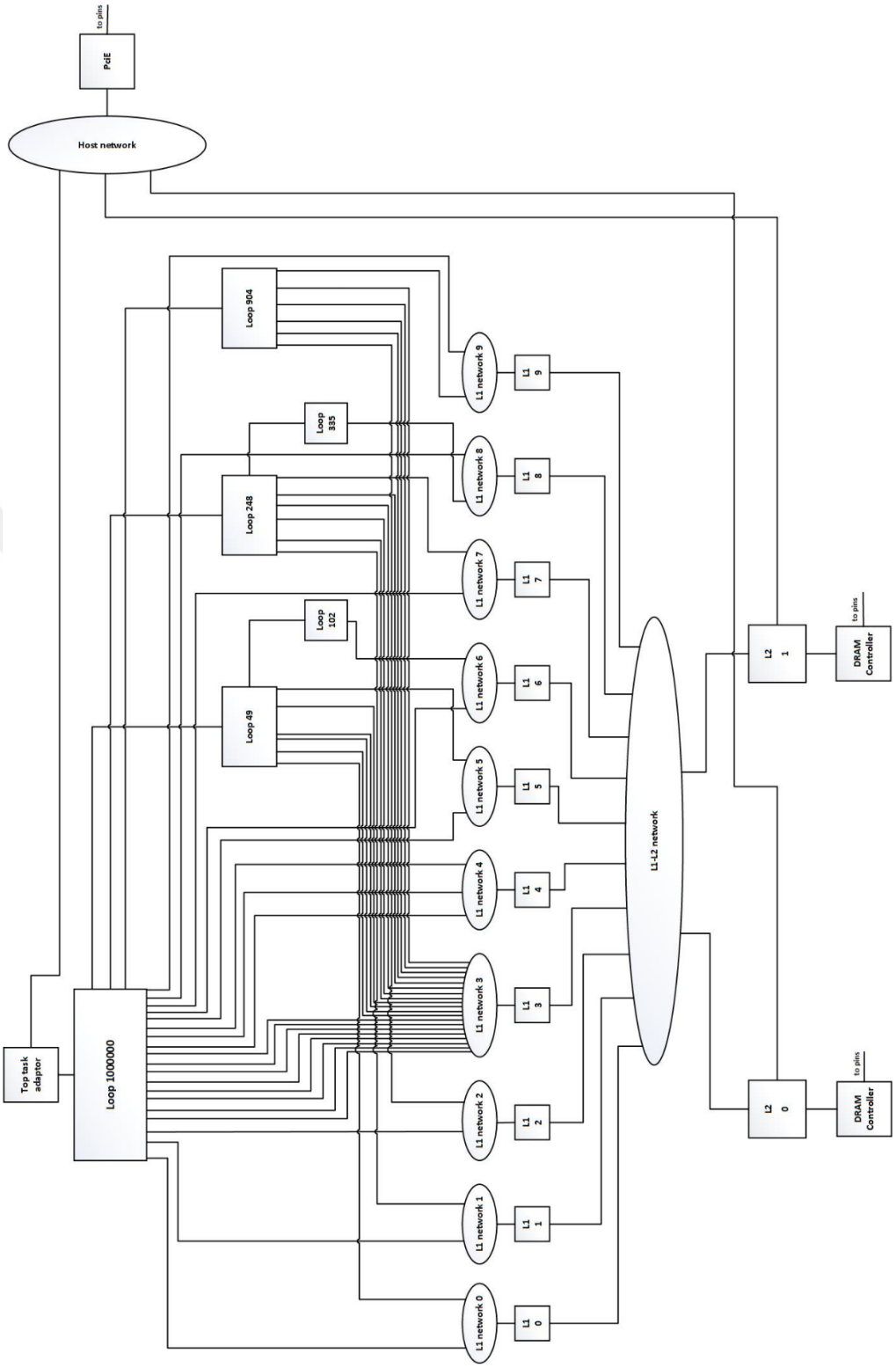

Figure 5.4. *Generated design for Sjeng test with banked caches*

5.5. Matrix Multiplication Test

One of the analyzed tests is matrix multiplication. Two NxN matrices in float type are multiplied and the result is written into third matrix in this algorithm. Dimension of matrix (N) is changed in original C code and the test is repeated. Requests received to directories and the implementation details are analyzed in Table 5.6. Number of generated coherent L1 caches is 4 and all caches belong to same coherence domain in test realized by using matrices of size 2x2. In other remaining tests, using matrices of different sizes 4x4, 6x6, …, 14x14, number of L1 caches are 19 and they belong to 4 different coherence domain. Number of line_read request increases while matrix dimension is increased, since the algorithm becomes more complex. The ratio of number of line_read request provided by L2 or L1 depend on dimension of matrices. At the end of algorithm, elements of result matrix are stored and all of these stores correspond to shared lines. Therefore, number of remote_store requests equals to number of elements in the matrix for all dimensions.

Table 5.6. *Analyze of requests and implementation details*

Matrix Dimension	**Number of L1**	**Number of Coherence Domain**	**Number of line_read requests**			**Number of remote_store requests**
			Total	**Supplied by L2**	**Supplied by L1**	
2x2	4	1	12	4	8	4
4x4	19	4	18	12	6	16
6x6	19	4	29	15	14	32
8x8	19	4	43	25	18	64
10x10	19	4	62	29	33	100
12x12	19	4	84	46	38	144
14x14	19	4	111	53	58	196

5.6. Advantages of Proposed Protocol

5.6.1. Advantages over write-invalidate policy

As mentioned before, write-update policy has advantages over write-invalidate policy since it suffers from the ping-pong effect. This advantage is clearly observed in experiments. For instance, all of the store requests correspond to shared lines in matrix multiplication tests. In 2x2 matrix multiplication, all L1 caches receive a store request to different words of same line. Then, all of these caches send remote_store requests to the directory. The directory receives these requests one by one and sends the new word to other sharer caches.

Suppose that write-invalidate policy is implemented with same model. This time, all caches send invalidate requests at the same time to the directory when they receive store requests. After receiving one of the invalidate requests, the directory transfers this request to other sharer caches. Then, the cache stores the word as an exclusive owner. The directory receives other invalidate requests that is stalled in network and the directory ignores these requests since these caches are not owner of this line anymore. However, these caches still have store requests, and they should first read the line since the line is not valid. The directory subsequently receives updated copy of the line from the cache that previously stores and sends it to the requestor cache as shared line. The cache that receives line_read sends invalidate request to be an exclusive owner. If the directory receives other caches' line_read requests before this invalidate request, this process becomes more complicated. Finally, the directory should handle with at least 3 more line_read requests. Occurrence of this situation increases as the size of the matrix algorithm increases. Moreover, this situation can be frequently observed for all algorithms due to locality.

5.6.2. Advantages of reading line from other L1 caches

In our model, line_read requests do not send to L2 if at least one of the other L1 caches has the line. The line transfer between caches provided by directory and accesses to L2 caches are reduced as much as possible. In experiments, cache to cache line transfer occurs frequently. These transfers provide advantages as compared to reading from L2 cache since tag and data of L2 cache are kept in DRAM.

5.6.3. Advantages over banked organization

As mentioned previous chapters, alternative memory architecture to coherent cache is banked memory model. This model designed as having at least miss ratio since memory is partitioned and fully associative caches are employed. However, additional networks are required between L1 caches and thread units and between L1 caches and L2 caches. In proposed coherent cache model, thread units and L1 caches are directly connected to each other. The proposed model has advantages since additional network delay emerges in other model. In sjeng tests, number of misses for both model is computed. Note that all of these misses are compulsory miss. Total miss count of banked and coherent models are 20 and 23, respectively. Besides, number of lines supplied by L2 caches is 18 for coherent model, so it is less than banked model. This situation shows that coherent model has another advantage for complex tests, since accessing to L2 cache is much slower. In terms of frequency, cache coherent model has better frequency although L1 caches and directories have complicated design, since Content-addressable Memories (CAMs) are used in banked organization.

6. CONCLUSION

In this thesis, a scalable memory architecture is implemented for a specific HLS compiler. This compiler converts a single threaded software program to application specific supercomputer and it requires a specific coherent cache system to decrease the memory access latencies. The memory architecture is coherent and the coherence protocol of this model is directory-based write-update. All of caches that share a line update their own data with a new copy when one of them stores to this shared line in write-update policy. A directory keeps set of owner caches, therefore, it always knows L1 caches that have a specific line. Directory is external component and it can be connected to either L2 caches or memory and number of directories is arbitrary. Directories are responsible for responding requests coming from L1 caches and managing the communication between them. In this model, L1 caches can belong to different coherence domain and only the caches that are in same coherence domain are communicated by directory. The synchronization between dependent memory operations are managed by the compiler.

In this work, the protocol and its implementation details are presented. This model is integrated to the compiler and the proposed memory system passed 51 tests. In these tests, number of generated L1 caches can be different according to the test and maximum number is observed as 39 in the Sjeng. The tests results show the performance potential of the model. However, the model could not been extensively analyzed in terms of performance since the proposed model is primitive. In next step, the model will be optimized to show performance, e.g., by adding a few pipeline stages to the directory.

One of the other future works is specializing the coherent caches according to input program by profiler feedbacks. In this way, for each algorithm, determining the most suitable parameters such as number of lines in each cache, will be possible.

The proposed memory model can be integrated to the multicore CPU (Central Processing Unit) system as a future work. However, some modifications are required such as rearranging the requests between memory and multicore CPU systems and implementing the memory consistency model.

REFERENCES

[1] Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). *Introduction to parallel computing: Design and Analysis of Algorithms*. Redwood City, CA, USA: Benjamin/Cummings Publishing Company.

[2] Koch, D., Hannig, F., and Ziener, D. (Eds.). (2016). *FPGAs for Software Programmers*. Switzerland: Springer International Publishing.

[3] Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1), 82-85.

[4] Herbordt, M. C., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., and DiSabello, D. (2007). Achieving high performance with FPGA-based computing. *Computer*, 40(3), 50-57.

[5] Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z. (2011). High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4), 473-491.

[6] Ebcioglu, K., Kultursay, E., and Kandemir, M. T. (2015). Method and system for converting a single-threaded software program into an application-specific supercomputer. Washington, DC, USA: U.S. Patent and Trademark Office. U.S. Patent No. 8,966,457.

[7] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9), 948-960.

[8] Hennessy, J. L., and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. San Francisco, CA, USA: Elsevier.

[9] Stenstrom, P. (1990). A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6), 12-24.

[10] Sorin, D. J., Hill, M. D., and Wood, D. A. (2011). A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3), 1-212.

[11] Jacob, B., Ng, S., and Wang, D. (2010). *Memory systems: cache, DRAM, disk*. Burlington, MA, USA: Morgan Kaufmann.

[12] Papamarcos, M. S., and Patel, J. H. (1984). A low-overhead coherence solution for multiprocessors with private cache memories. *ACM SIGARCH Computer Architecture News*, 12(3), 348-354.

[13] Culler, D., Singh, J. P., and Gupta, A. (1998). *Parallel computer architecture: a hardware/software approach*. Los Altos, CA, USA: Morgan Kaufmann.

[14] McCreight, E. M. (1985). The dragon computer system. *Microarchitecture of VLSI Computers*, 83-101.

[15] Archibald, J., and Baer, J. L. (1986). Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4), 273-298.

[16] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9, 690-691.

[17] Tanenbaum, A. S., and Van Steen, M. (2007). *Distributed systems: principles and paradigms*. Upper Saddle River, NJ, USA: Prentice-Hall.

[18] Goodman, J. R. (1983). Using cache memory to reduce processor-memory traffic. *ACM SIGARCH Computer Architecture News*, 11(3), 124-131.

[19] Katz, R. H., Eggers, S. J., Wood, D. A., Perkins, C. L., and Sheldon, R. G. (1985). Implementing a cache consistency protocol. *ACM SIGARCH Computer Architecture News*, 13(3), 276-283.

[20] Tang, C. K. (1976). Cache system design in the tightly coupled multiprocessor system. In: *Proceedings of National Computer Conference and Exposition*, New York, USA: ACM, pp. 749-753.

[21] Censier, L. M., and Feautrier, P. (1978). A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 12, 1112-1118.

[22] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. (1990). The directory-based cache coherence protocol for the DASH multiprocessor. In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, USA: ACM, pp. 148-159.

[23] Heinrich, J. (1997). Origin™ and Onyx2™ Theory of Operations Manual, Mountain View, CA, USA: Silicon Graphics, Inc. Document No: 007-3439-002.

[24] Thacker, C. P., Stewart, L. C., and Satterthwaite, E. H. (1988). Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8), 909-920.

[25] Putnam, A., Bennett, D., Dellinger, E., Mason, J., Sundararajan, P., and Eggers, S. (2008). CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In: *Proceedings of International Conference on Field Programmable Logic and Applications*, Heidelberg, Germany: IEEE. pp. 173-178.

[26] Chung, E. S., Hoe, J. C., and Mai, K. (2011). CoRAM: an in-fabric memory architecture for FPGA-based computing. In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* Monterey, CA, USA: ACM, pp. 97-106.

[27] Hung, A., Bishop, W., and Kennings, A. (2005). Symmetric multiprocessing on programmable chips made easy. In: *Proceedings of the Conference on Design, Automation and Test in Europe,* Munich, Germany: IEEE, pp. 240-245.

[28] Woods, D. (2009). *Coherent shared memories for FPGAs*. MSc Thesis, Toronto, Canada: University of Toronto: Graduate Department of Electrical and Computer Engineering.

[29] Lange, H., Wink, T., and Koch, A. (2011). MARC II: A parametrized speculative multi-ported memory subsystem for reconfigurable computers. In: *Proceedings of the Conference on Design, Automation and Test in Europe,* Grenoble, France: IEEE, pp. 1-6.

[30] Mirian, V., and Chow, P. (2012). Managing mutex variables in a cache-coherent shared-memory system for FPGAs. In: *Proceedings of International Conference on Field-Programmable Technology,* Seoul, South Korea: IEEE, pp. 43-46.

[31] Mirian, V., and Chow, P. (2012). An implementation of a directory protocol for a cache coherent system on FPGAs. In: *Proceedings of International Conference on Reconfigurable Computing and FPGAs,* Cancun, Mexico: IEEE, pp. 1-6.

[32] Yang, H. J., Fleming, K., Adler, M., and Emer, J. (2014). LEAP shared memories: Automating the construction of FPGA coherent memories. In: *Proceedings of the 22nd Annual International Symposium on Field-Programmable Custom Computing Machines,* Boston, MA, USA: IEEE, pp. 117-124.

[33] Adler, M., Fleming, K. E., Parashar, A., Pellauer, M., and Emer, J. (2011). Leap scratchpads: automatic memory and cache management for reconfigurable logic. In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* Monterey, CA, USA: ACM, pp. 25-28.

[34] Patterson, D. A., and Hennessy, J. L. (2008). *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishes Inc.

CURRICULUM VITAE

Name Surname : Gizem Yağan

Foreign Languages : English

Birth Place and Date : Eskişehir/ 17.05.1993

E-mail : gizemgulmez@eskisehir.edu.tr

Education:

- 2019, MSc, Eskişehir Technical University, Graduate School of Sciences, Electrical and Electronics Engineering Program
- 2016, BSc, Anadolu University, Department of Electrical and Electronics Engineering

Experience:

- 2017- , Research Assistant, Eskisehir Technical University, Department of Electrical and Electronics Engineering
- 2016- , Researcher, Erendiz Superbilgisayar Company

Projects:

- 2017-2018, Researcher, Erendiz Project, Design Feasibility for Compressed Imaging Supercomputer
- 2015-2016, Undergraduate Student Researcher, Decentralized Time-Delay Controller Design for Time-Delay Systems, TUBITAK
- 2015-2016, Undergraduate Student Researcher, Decentralized Control of Systems with Commensurate and Uncommensurate Time-Delays, TUBITAK

Awards:

- Graduated as High Honor Student

Skills:

- FPGA, Verilog, VHDL, C programming, MATLAB