Institute of Software Technology
Department of Programming Languages and Compilers
University of Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Master thesis Nr. 3409

Analysis of Cache Usability on Modern Real-Time Systems

Ahmad N. Almheidat

Course of study:	INFOTECH
Examiner:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Supervisor:	Dipl.-Inf. Mikhail Prokharau
Commenced:	October 1, 2012
Completed:	July 29, 2013
CR-Classification:	B.3.2, B.3.3, C.1.2.vi, D.4.2.vii

ABSTRACT

Cache memories are used in the microprocessors to close the speed gap between the processor and the main memory. Caches can minimize the memory access time by keeping a copy of the highly demanded data closer to the processor. As a result, the overall program execution time is reduced. In safety-critical real-time systems, a worst-case analysis is required, and therefore the cache memories play an essential role in the estimation of the application's worst-case execution time. A simulation tool for the cache structure was developed to provide estimated measurements for both cache predictability and the worst-case memory access time based on the used architectural model. This may help to draw some conclusions about the actual cache operation. The simulation supports several modern uni-core and multi-core architectures, including some used in real-time systems. It also allows configuring different cache structures and hierarchies. The cache architecture, configuration and memory accesses from a simulated running application are specified by the user via an input file. The simulation provides a list of traces for every access. The cache predictability can be formulated as hit and miss rates. At the same time, the traces can be used to estimate total memory access time.

ACKNOWLEDGMENTS

I would like to thank Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder for giving me the opportunity to write my thesis at the Institute of Software Technology, Department of Programming Languages and Compilers at the University of Stuttgart.

I would like also to express my appreciation to my thesis supervisor Dipl.-Inf. Mikhail Prokharau for his guidance and significant review of the thesis work.

I am profoundly grateful to my family for their continuous support throughout the period of my studying.

Finally, I want to thank Nurgül Düzenli for her concern and support.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Cache predictability plays a significant role in determining the overall system performance during execution of an embedded application [43]. The cache predictability can be measured by the cache miss rate which is the percentage of the memory accesses not found in the cache taken from the total of all memory accesses. Since the cache miss rate depends on the cache configuration parameters, such as cache size, block size, and associativity, different cache configurations can result in different cache miss rates for a particular application [44]. To analyze different structures of the cache memory, a simulation of an embedded application's memory accesses can help by calculating the miss rates using different cache structures, thus allowing a choice of the best structure with the minimum cache miss rate for that embedded application [43].

This thesis aims to develop a simulator for the cache memories in the modern processors, both multi- and uni-core, especially those which are used in the embedded systems. The simulation supports different cache architectures such as directly mapped, fully associative and set-associative. Within the simulation a number of cache parameters such as cache size, block size, and associativity are adjustable. As part of the simulation three different cache replacement polices (FIFO, LRU, PLRU) have been implemented. The simulation supports multi-core architectures using multiple levels of private and shared caches with different cache hierarchies (inclusive, non-inclusive, or exclusive) at each level. It also offers a cache coherence scheme for private cache data consistency.

The rest of this thesis is structured as follows. Chapter 2 gives an introduction about the cache basics including cache structures, cache block allocation, cache block replacement, and multi-level cache hierarchies. Chapter 3 talks about cache memories in multi-core processors and their structure. It also provides examples based on modern multi-core processors. Besides, the chapter discusses the cache coherence problem and the coherence protocol scheme aimed at solving the problem. Finally, the chapter explains

the MSI protocol for cache coherence in multi-core processors. Chapter 4 shows the design and the detailed implementation of the cache simulator. Chapter 5 introduces and discusses a number of test case scenarios for the cache simulator. Finally, Chapter 6 concludes the entire work.

2 Cache Basics

2.1 Memory hierarchy

The memory system of a computer is not a single memory; it is organized as a hierarchy of connected memories of different technology. **Memory hierarchy** is a structure that uses multiple levels of memories; memories in the closer levels to the **CPU** are faster and smaller, and while the memory levels become farther from the CPU the size of the memory and access time both increase. This hierarchy offers high storage capacity and at the same time low average access time for the memory system. The level closer to the CPU is generally a subset of any level further away, and all the data are stored at the lowest level. A memory hierarchy can consist of multiple levels, but data are copied between only two adjacent levels at a time [1].

Fast memory technologies that provide high speed data access like SRAM (static random access memory) are expensive; therefore they are used only for small memories in the levels closer to the CPU. At the same time cheaper technology is slower, but it is used for large memories. This technology requires high access time like DRAM (dynamic random access memory). Main memory is implemented from DRAM technology [2]. The aim of the memory hierarchy is to provide memory system which gathers among big capacity, short access time, and cost effective characteristics. The big capacity can be obtained from using much memory which is slow and cheap in the lower levels of the hierarchy, while the fast access time can be obtained from using small and fast memory in the higher levels. Figure 1 shows the faster memory is closer to the CPU while the slower and cheaper is below it. In many embedded devices magnetic disks are replaced by flash memory [1].

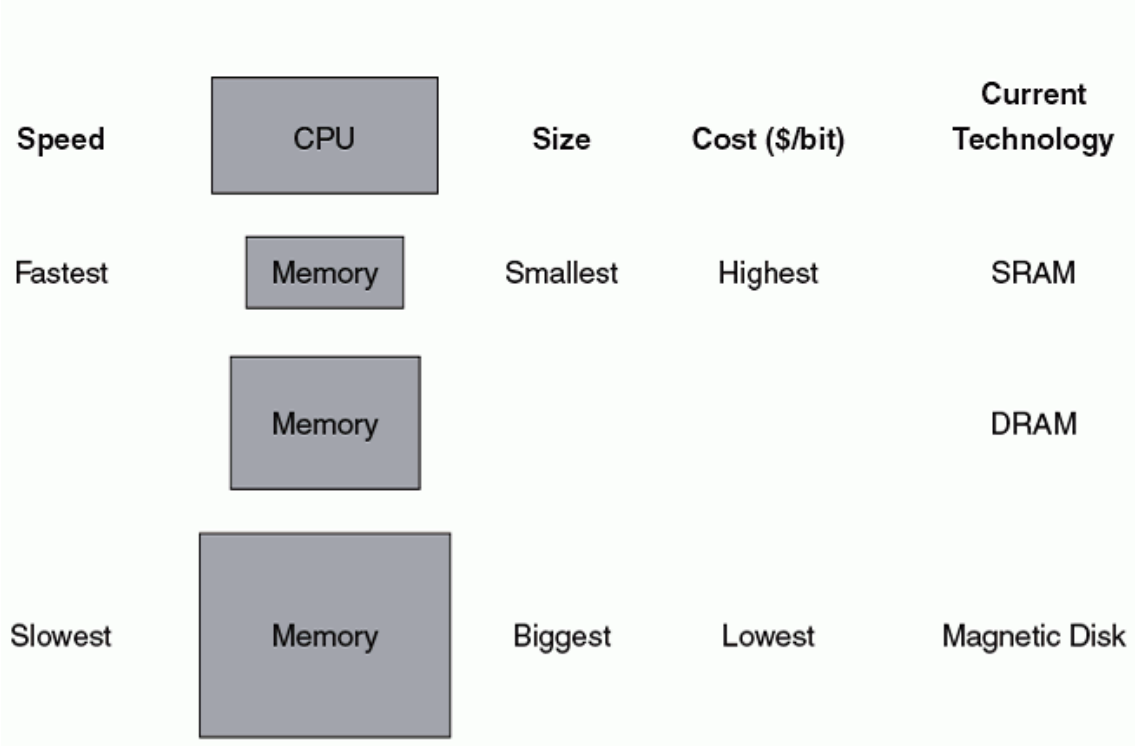

Figure 1 Basic structure of a memory hierarchy [1]

The reason for having a memory hierarchy is the principle of locality. It states that the computer programs access the memory data in certain patterns. There are two kinds of locality: **temporal locality** and **spatial locality**, the principle of temporal locality says if the program references a memory location, it will be more likely to re-reference that location than some other random location. For that reason the most recent referenced data should be stored in a place closer to the CPU. A smaller and faster memory called **cache** is used to store those data which are expected to be referenced next, so if the prediction is correct, the data referencing can be served faster. The principle of spatial locality says if the program references a memory location, it will be more likely to reference a location near to it than some random location. Therefore, the main memory is divided logically into a set of **memory blocks** which are equal in size; usually the size is a power of two. Each memory block is made out of adjacent words (bytes), it represents the minimum

unit of data that can be either present or not in the level of the hierarchy. When the processor references some memory location, the whole block which contains that location and the locations near to it is brought to the cache and stored in a **cache line** which is equal in size to the memory block. Afterward if the processor references any near address, it can be found in the cache, so the address referencing will be faster [1].

2.2 Processor Caches

The upper levels in the memory hierarchy represent the cache memories; they are small in size and built using fast memory (SRAM). The time to retrieve the requested data from the cache memory will be much smaller compared to the time needed to access the main memory, which is the major component of the memory access latency. Caches are used as fast storage to improve average access time to the slow main memory [1].

In modern architectures the cache memories are physically located in the CPU die (logically placed) between the processor register file and the main memory. They store a subset of memory data to hide the speed gap between the processor and the main memory by exploiting the locality principle in the memory accesses. All processor requests are served by caches first. As long as most memory accesses are found in the caches, the average latency of memory access will be closer to the cache latency than the latency of the main memory. In the recent processor architectures if the data are not found in the cache, it takes several hundreds of processor cycles to bring the data from the main memory. Therefore, cache predictability has a big influence on the whole system performance [7].

Modern computer microprocessors have not only one cache but several caches which are structured in a hierarchal way, one level of cache memory after the other. Most of the microprocessors have the caches that constitute up to three levels of the overall memory hierarchy and may be of different type [2]: separated cache, or unified cache. In the

separated cache the data and the instructions of the program are stored in different caches, so the cache is divided into instruction cache to hold the instructions and data cache to hold the data. A split instruction and data cache can increase the cache bandwidth. The unified caches hold both data and instructions. Figure 2 shows a system with two levels of cache, the first level is a separated cache and implemented in the same chip of the microprocessor, while the second level is a unified cache and implemented off-chip in a separate set of SRAMs.

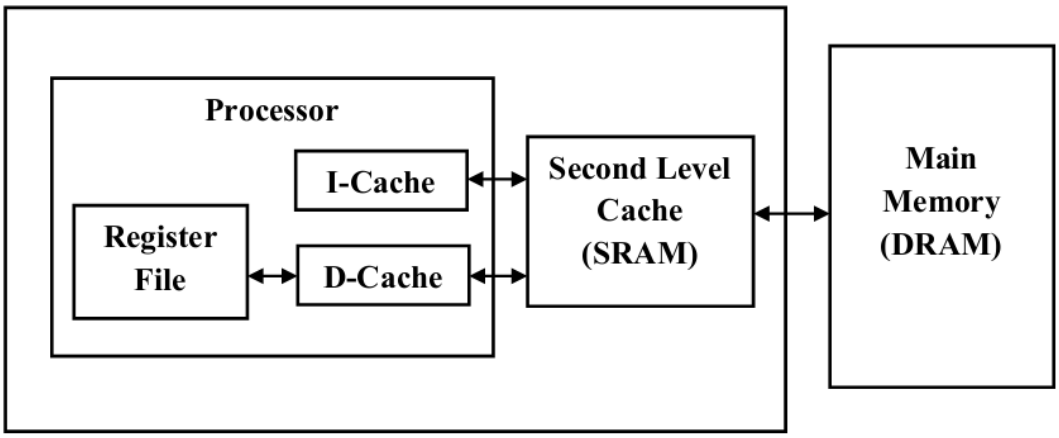

Figure 2 Memory hierarchy of a system that uses multi-level caches

All memory reference operations by the processor are sent first to the upper level of the caches. If the cache logic finds the requested data in a block presented in the cache, this means that the data are cached, this is called a **hit**. The time to access that data is called the **hit time**, which includes also the time needed to determine whether the access is a hit. In case of cache hit, the request can be directly serviced from the cache. If the data are not found in that upper level of the caches, the request is called a **miss**. The lower level of hierarchy (can be second level of cache, main memory, or disk) is then accessed to retrieve the block containing the requested data. The request is propagated from one level

to the next level in hierarchy until the requested block is found. When the requested block is found in one memory level, it is necessary to forward it to the upper level and store it there and so on through the intermediate hierarchy until the block is placed in the upper level of the caches, after that the processor request is serviced. The time taken to search for the requested block through the hierarchy until it is found, then bring it and store it back along the hierarchy to the upper levels of the caches is called **miss penalty**. It also includes the time needed for block replacement in each cache if it is needed [1]. Figure 3 shows how the data block is transferred from one level to the upper level in the hierarchy then to the processor.

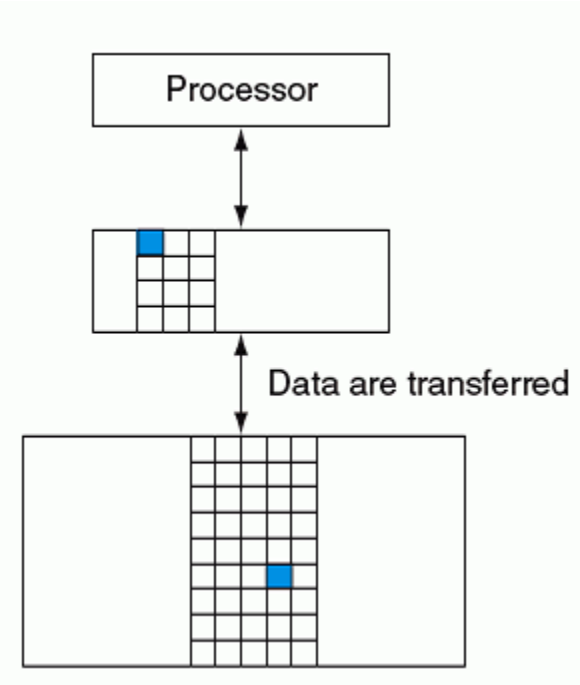

Figure 3 Data block transfer between every pair of levels in the memory hierarchy [1]

2.3 Cache Structure

Caches are partitioned into **cache sets** of equal size, usually power of two. Each cache set contains one or more of the cache lines. The number of blocks in the set is known as the

degree of **associativity**. For efficient lookup of a memory block in the cache, each memory block can map to only one cache set and be stored in one of set's blocks [3]. The cache is usually organized as a two-dimensional array, where the sets represent the rows and the blocks in each set represent the columns. Figure 4 shows the cache sets and set contents of blocks, here the associativity of the cache equals four.

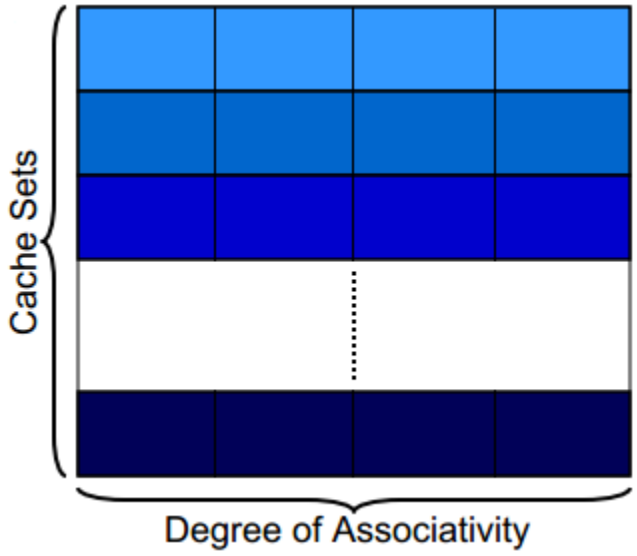

Figure 4 Associative cache structure

The number of blocks or lines in the cache is usually a power of two, and it can be calculated by

$$\frac{Cache\ Size\ (in\ bytes)}{Block\ Size\ (in\ bytes)}\ [3]$$

The number of cache sets is a power of two, and it can be also calculated by

$$\frac{Number\ of\ Cache\ Blocks}{Degree\ of\ Associativity}\ [3]$$

2.4 Cache Organizations

There are three different categories of cache organizations:

Direct-Mapped, every memory block brought to the cache has exactly one place (block) in the cache where it can be allocated. The degree of associativity in the direct-mapped cache organization equals 1, this means every cache set contains only one cache block, and therefore the number of cache sets in this case is equal to the number of blocks in the cache. Direct-mapped cache is cheap to implement in hardware, also the data look up in the cache is fast [1].

Fully-Associative, the block can be placed in any available cache line. The cache has only one big set which holds all cache blocks. The degree of the associativity in fully-associative cache is equal to the number of the cache blocks. As long as a new block is needed to be placed in the cache, it can be allocated in any available place in the cache. To find a block in fully-associative cache, all the blocks in the cache must be searched; therefore the implementation of the data look up is expensive [1].

Set-Associative, the block can be placed in a particular set in the cache. For *n*-way set-associative cache the degree of associativity equals n, so each set contains *n* blocks. Each memory block is mapped to a specific set, and then it can be allocated in any available place within that set. Every cache can be considered as a set-associative cache, because the set-associative placement combines both direct-mapped and fully-associative placement mechanisms. Set-associative cache organization offers the chance to increase performance by more flexible placement mechanism which reduces the number of cache misses and at the same time enables an efficient data look up [1].

2.5 How to Place a Block in the Cache?

The mapping between the addresses and cache sets can be done by modulo function [3]; the set number to where the block can be mapped is calculated by

$$(Block\ Address)\ modulo\ (Number\ of\ cache\ sets)$$

Figure 5 below shows an example for a block placement in the three different cache organizations. The cache contains 8 blocks, and the block number 12 in the memory or lower level of hierarchy needs to be placed in the cache. In fully-associative cache, the block can be placed in any of the 8 blocks. With direct-mapped cache, the block 12 can be mapped to cache block number: $(12\ mod\ 8) = 4$. The last cache organization is 2-way set-associative, therefore the cache has $(\frac{8}{2} = 4)$ sets and each set holds two cache blocks. The block number 12 is mapped to the set number: $(12\ mod\ 4) = 0$, and the block can be placed in any of the two blocks held by the set [3].

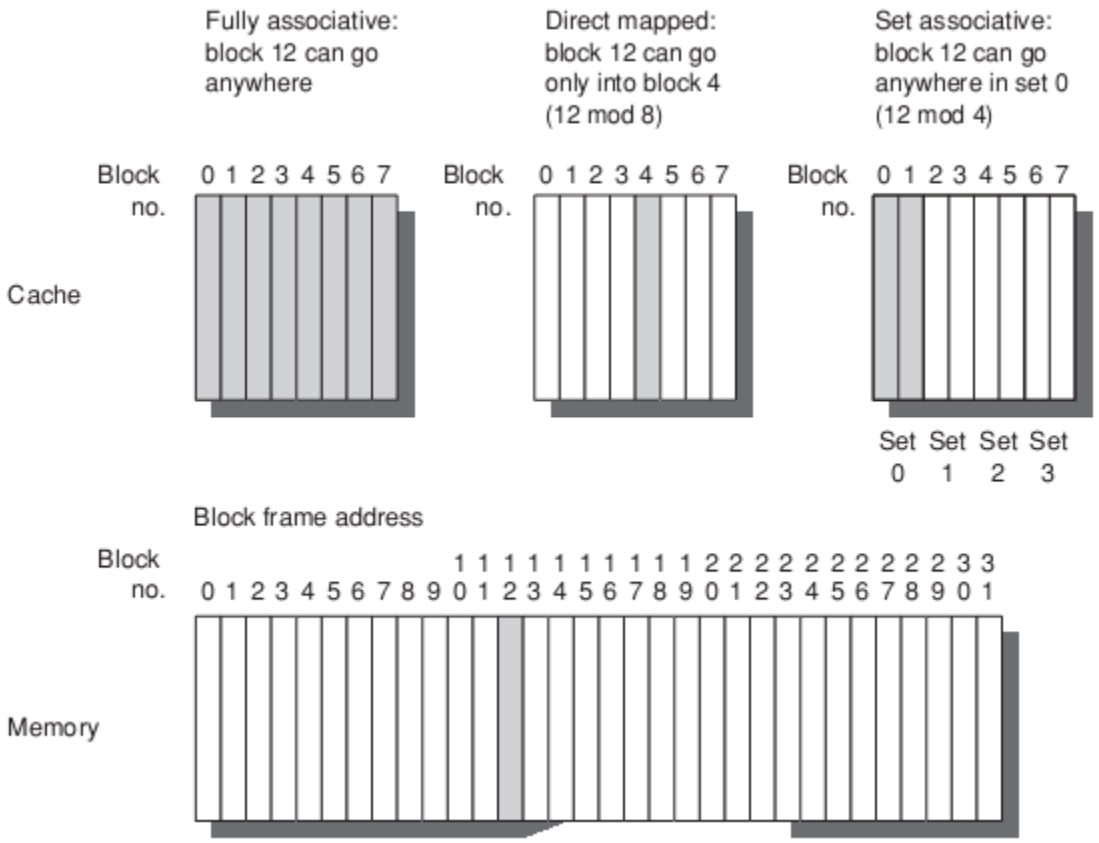

Figure 5 Three different cache organizations [3]

As for both direct-mapped and fully-associative strategies, they can be seen as variations on set-associative strategy. A direct-mapped cache is simply a one-way set-associative cache; each cache set holds one cache block. A fully-associative cache with m blocks is simply an *m*-way set-associative cache; it has one set with m blocks [3].

2.6 How to Find a Block in the Cache?

Each cache block has a tag field associated with it and stored along with the data, the block tag works as a unique identifier for the data mapped to that cache block, the tag contains some address information to identify whether the data in the cache block is the same as the requested data in the memory address, this means the data request is a hit in the cache. When the system starts up none of the cache blocks contains valid data, a **valid bit** associated with each block tag is needed to indicate whether the cache block contains valid data; so the bit is set to 1, else it is set to 0 [1].

The address of the memory access comes from the processor and is divided into two parts; the **block address** and the **block offset**. The block offset bits are the $\log_2(Block\ Size)$ least significant bits of the memory address and the remaining bits are for the block address field, if the block size is a power of two (which is usually the case). The block offset field represents the address of the accessed data within the block. The block address field is used to check whether the referenced block is present in the cache or not, it is divided again into two parts; **block tag** and **set index**. The set index bits are used to select the set which might contain the corresponding memory block. If the number of cache sets is a power of two (which is usually the case), then the set number can be found by taking the $\log_2(Number\ of\ sets)$ least significant bits of the block address and the rest bits are for the block tag. To check if the corresponding set is actually contained, the block tag filed in the memory address is compared with all block tags in the set, if it matches with one block tag and the valid bit is set to 1, then the request data are present in that cache block, and the block offset field is used to determine the data and return them to the processor. In a cache miss, the block is fetched from the lower level in the hierarchy [1].

The following figure 6 shows the three divisions of the memory address. The tag is used as a unique identifier for the cached memory block, to differentiate between the different blocks in the set. The index field determines the set number where the addressed data

should reside, it is only used in direct-mapped and set-associative caches. For the fully-associative cache there is no index field because the whole cache is organized as one set. The block offset is where the data can be found within the cache block [3].

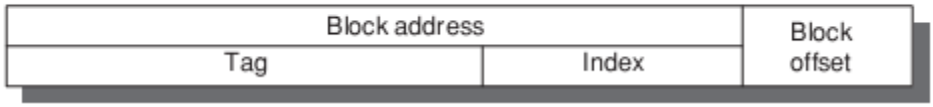

Figure 6 Three different portions of the memory address [3]

For a given memory address, the following procedure steps illustrate how the cache look up logic finds whether the addressed data are cached (cache hit) or not (cache miss):

1. The index is used to select the cache set where the address can be mapped. For fully-associative strategy the cache has effectively one set which contains all blocks.
2. For each block in the selected cache set, the tag from the memory address is compared with all tags associated with each block in the set. If a match is found, proceed to the next step, otherwise cache miss.
3. For the matching block, the valid bit is checked; if it is set to 1, cache hit, otherwise cache miss.
4. The block offset is used to determine the starting byte of the addressed data within the block.

Figure 7 shows an example of the four-way set-associative cache hardware implementation; four comparators are needed, because all the tags in the selected set are searched in parallel, a 4-to-1 multiplexer is used to select among the four data blocks. In direct-mapped cache only a single comparator is needed, because the set contains only

one block (one-way set-associative). In a fully-associative cache, a sequential search through the cache blocks would need a long time, or the search can be done in parallel with a comparator associated with each cache block, which increases the hardware cost. This makes the fully-associative placement practically usable only in a cache with small numbers of blocks [1].

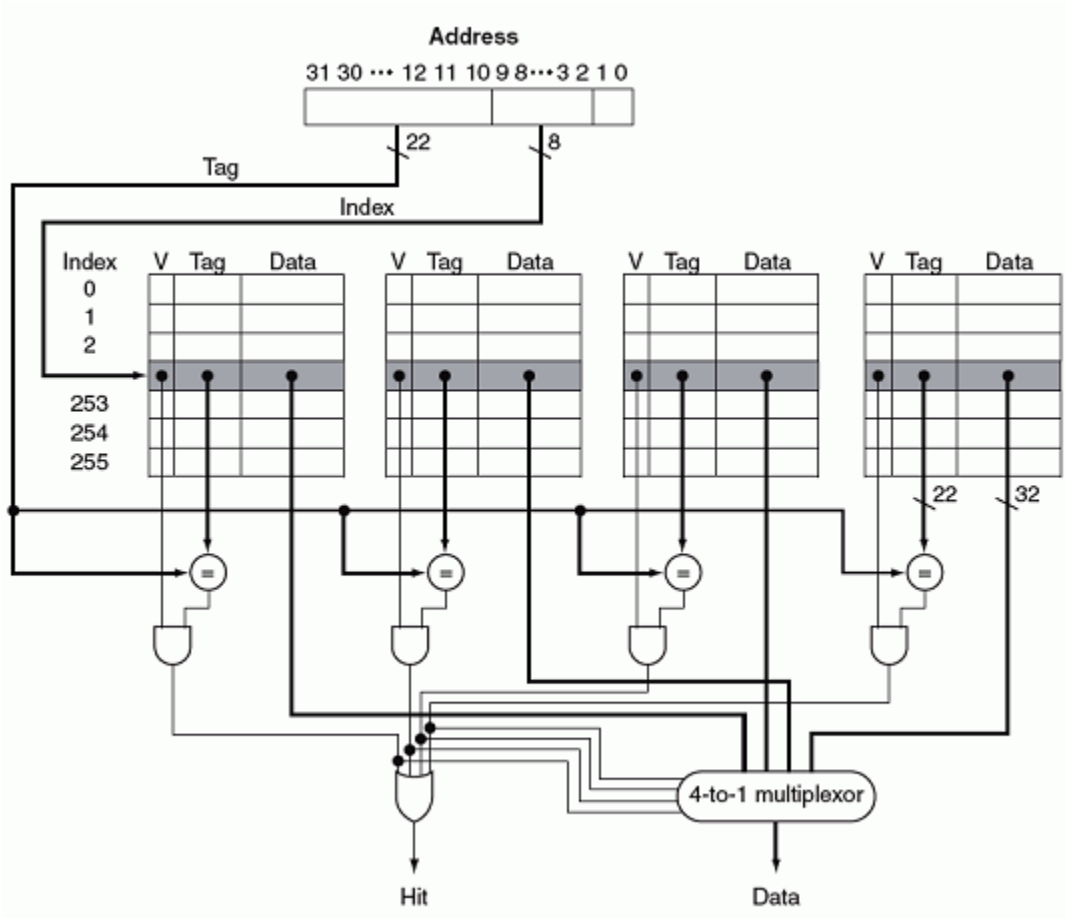

Figure 7 The hardware implementation of a 4-way set-associative cache [1]

2.7 Cache Organization Example

Consider a cache memory with 64 bytes size and with block size equaling 16 bytes. The memory address width is 16 bits. Table 1 below shows how the purposed cache is

structured using the different cache organizations (direct-mapped, fully-associative, and set-associative). For each cache structure it shows the number of sets in the cache, the number of blocks within each set, and the number of bits in each memory address division, the bits used for the block offset and the set index bits which are used to find the set to where the block maps and the tag bits which are stored in the cache as a block identifier.

Table 1 Cache organization example

Cache Organization	Number of sets	Number of blocks per set	Block offset bits	Index bits	Tag bits
Direct-mapped	$\frac{64}{16} = 4$	1	$\log_2 16 = 4$	$\log_2 4 = 2$	10
Fully-associative	1	$\frac{64}{16} = 4$	$\log_2 16 = 4$	0	12
2-way set-associative	$\frac{64}{16 \times 2} = 2$	2	$\log_2 16 = 4$	$\log_2 2 = 1$	11

Consider the following memory access sequence given as memory addresses: {0x1234, 0x123C, 0x1240, 0x1270, 0x1234, 0x1232, 0x1248, 0x12C8, 0x1248, 0x1244}. The sequence is applied to every cache structure described in table 1; assume every cache is initially empty. For the direct-mapped cache structure, table 2 shows the three divisions of the memory address (Block offset, Index, and Tag) and the result of each memory access. Table 3 shows the cache contents after applying the complete sequence of memory accesses.

Table 2 Memory access results for direct-mapped cache

Address	Block offset	Index	Tag	Hit/Miss
0x1234	0100	11	0001001000	Miss
0x123C	1100	11	0001001000	Hit
0x1240	0000	00	0001001001	Miss
0x1270	0000	11	0001001001	Miss
0x1234	0100	11	0001001000	Miss
0x1232	0010	11	0001001000	Hit
0x1248	1000	00	0001001001	Hit
0x12C8	1000	00	0001001011	Miss
0x1248	1000	00	0001001001	Miss
0x1244	0100	00	0001001001	Hit

Table 3 Direct-mapped cache content

Index	V	Tag
00	1	~~0001001001~~ ~~0001001011~~ 0001001001
01	0	
10	0	
11	1	~~0001001000~~ ~~0001001001~~ 0001001000

For the 2-way set-associative cache structure described in table 1, table 4 shows the three divisions of the memory address (Block offset, Index, and Tag) and the result of each memory access. Table 5 shows the cache contents after applying the complete sequence of memory accesses.

Table 4 Memory access results for two-way set-associative cache

Address	Block offset	Index	Tag	Hit/Miss
0x1234	0100	1	00010010001	Miss
0x123C	1100	1	00010010001	Hit
0x1240	0000	0	00010010010	Miss
0x1270	0000	1	00010010011	Miss
0x1234	0100	1	00010010001	Hit
0x1232	0010	1	00010010001	Hit
0x1248	1000	0	00010010010	Hit
0x12C8	1000	0	00010010110	Miss
0x1248	1000	0	00010010010	Hit
0x1244	0100	0	00010010010	Hit

Table 5 Two-way set-associative cache content

Index	V	Tag	V	Tag
0	1	00010010010	1	00010010110
1	1	00010010001	1	00010010011

For the fully-associative cache structure described in table 1, table 6 shows the two divisions of the memory address (Block offset, and Tag) and the result of each memory access. Table 7 shows the cache contents after applying the complete sequence of memory accesses.

Table 6 Memory access results for fully-associative cache

Address	Block offset	Tag	Hit/Miss
0x1234	0100	000100100011	Miss
0x123C	1100	000100100011	Hit
0x1240	0000	000100100100	Miss
0x1270	0000	000100100111	Miss
0x1234	0100	000100100011	Hit
0x1232	0010	000100100011	Hit
0x1248	1000	000100100100	Hit
0x12C8	1000	000100101100	Miss
0x1248	1000	000100100100	Hit
0x1244	0100	000100100100	Hit

Table 7 Fully-associative cache content

V	Tag	V	Tag	V	Tag	V	Tag
1	000100100011	1	000100100100	1	000100100111	1	000100101100

2.8 Cache Replacement Policies

The cache is much smaller in size than the main memory, and for this reason, the number of memory blocks that map to a particular cache set is greater than the size of the cache set [9]. On cache miss, the whole memory block which holds the requested data is fetched from the lower level, and forwarded to the higher level of the hierarchy. If the set is full (all blocks are valid), a **victim block** must be chosen and replaced (evicted). In direct-mapped cache, because there is only one position to check every time for a hit, it is trivial to choose the victim block, which is the block in that position. With a fully-associative or set-associative cache, there are many blocks to choose for replacement. In a fully-associative cache, all blocks in the cache can be candidates for replacement. In a set-associative cache, only the blocks within the selected set are candidates. The victim block is chosen among the candidates based on the **replacement policy** [2].

The most commonly-used known cache replacement policies are least-recently used (**LRU**), first-in first-out (**FIFO**), and a cost-efficient variant of LRU named pseudo-LRU (**PLRU**). Those policies work on each cache set individually, and use independent status bits per set which store information about previous set accesses. This information is used next time when a victim block needs to be chosen [10].

In **LRU** replacement, the victim block is the least recently used block in the set. That is the block which has been unused for memory reads or writes for the longest time. LRU strategy relies on the temporal locality; the recently used blocks are more likely to be used again. For this purpose, the LRU implementation keeps track with each block being

referenced in the set related to the other blocks within that set by maintaining a stack within the set to store the accessing sequence [4]. In caches, the replacement algorithm is implemented in hardware, which means that it should be easy to implement. LRU replacement policy has the best predictability properties [7], but it is costly to implement the logic for tracking the set usage information, especially when the degree of the associativity exceeds four [1].

LRU replacement policy can be implemented in software for simulation purposes by maintaining a queue for each cache set of length equaling to the associativity. The newly brought block to the set (after a cache miss) is placed in the front of the queue, also when referencing an existing block in the set (cache hit), the block is brought from its position to the front of the queue, so the blocks in the set will be ordered from most-recently to least-recently used [11]. As a result, for this procedure the last block in the queue is always the least-recently-used one among those blocks in the queue, and it will be removed to make a room for the block newly brought to the set when the set is full [2]. LRU replacement is used in the Intel Pentium I and the MIPS 24 K/34 K [9].

Consider a 4-way set-associative cache using LRU replacement policy and the following sequence of memory accesses (given as block addresses) {A, B, C, D, E, B, A, F}. Assume all blocks map to the same set and the set is initially empty. Table 8 shows all results of the cache accesses. The LRU policy queue is updated every time the set is accessed, and the referenced block is always placed in the front of the queue. The 0 values in the queue represent the invalid cache lines in the set which can be filled with the new blocks after cache misses. When a cache miss occurs while the set is full (i.e. referencing block *E*) the LRU block is chosen for replacement which is always the last block in the queue.

Table 8 LRU replacement policy

Block Address	Hit/Miss	LRU queue
A	Miss	{A,0,0,0}
B	Miss	{B,A,0,0}
C	Miss	{C,B,A,0}
D	Miss	{D,C,B,A}
E	Miss	{E,D,C,B}
B	Hit	{B,E,D,C}
A	Miss	{A,B,E,D}
F	Miss	{F,A,B,E}

In **FIFO** replacement, the victim block is the oldest block in the set. The implementation of the FIFO replacement policy is quite simple; it only requires a single round-robin counter per cache set which points to the next cache block to be replaced, and the counter is updated every time a block is added to the set. FIFO is cheaper in hardware implementation because it needs very little update logic, but it has less predictability compared to LRU [12].

In software, FIFO policy can also be implemented as a FIFO-queue where the new block to place in the set (after a cache miss) is placed in the front of the queue, but referencing an existing block in the set (cache hit) does not change anything in the queue order. According to this procedure, the last block in the queue is always the oldest one in the set which is also the next victim block to replace when the set is full. FIFO replacement policy is used in the Intel XSCALE, ARM9 and ARM11 processors [9].

Consider the same cache example used above, but now the cache uses FIFO replacement policy. Table 9 shows all results of the cache accesses. The FIFO policy queue is updated every time a cache miss occurs when the set is accessed, because a new block is brought to the cache and it needs to always be placed in the front of the queue. If the set is full, the first placed block in the set is chosen for replacement which is always the last block in the queue. In case of a cache hit the queue is kept unchanged as in second memory access to block *B*.

Table 9 FIFO replacement policy

Block Address	Hit/Miss	FIFO queue
A	Miss	{A,0,0,0}
B	Miss	{B,A,0,0}
C	Miss	{C,B,A,0}
D	Miss	{D,C,B,A}
E	Miss	{E,D,C,B}
B	Hit	{E,D,C,B}
A	Miss	{A,E,D,C}
F	Miss	{F,A,E,D }

PLRU replacement is a tree-based approximation of the true LRU policy. It arranges the cache blocks (ways) at the leaves of a binary tree [4]. For *n*-ways cache set, the PLRU tree has *(n - 1)* inner nodes pointing to the block to be replaced next, every node has a bit pointing to the sub-tree that contains at the end the PLRU candidate block. A '0' indicating the left sub-tree, a '1' indicating the right, figure 8 explains how to choose the victim block (way) for replacement when the cache set is full. The history bits $\{a_0, b_0, b_1\}$ represent the access order of the ways in the set, in the example figure if the value of the history bits $a_0b_0b_1 = 011$ the pseudo least recently used way is w_1 and it will be chosen as a victim next access miss to the set [6].

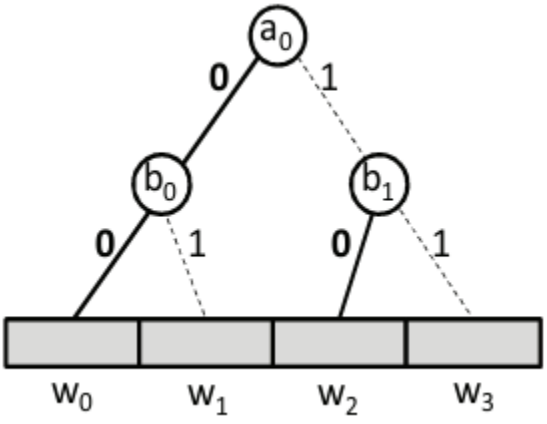

Figure 8 PLRU cache replacement policies for 4-way cache set [6]

On a cache hit, the binary tree of the relevant cache set is traversed starting from the tree root node to the referenced block, and on the way along the path the node values are flipped to point to the opposite branch where the referenced block is allocated [5]. On cache miss, PLRU tracks invalid blocks in the set first, the invalid lines are filled from left to right, ignoring the tree bits, but all the node values in the way forming the newly allocated block to the root node are flipped to point away from the block [8]. If the set is full (all blocks in the set are valid), the tree is traversed according to the node values to find the PLRU candidate block for replacement, while traversing to find the victim block all path node values are also flipped to point away from the newly allocated block.

As an example for PLRU replacement policy, consider the same example for the 4-way set-associative cache which is used above to discuss the replacement policies. Same memory block access sequence {A, B, C, D, E, B, A, F} is used and the same assumption that all blocks map to the same cache set is made. Figure 9 shows the state of the PLRU binary tree for the set after each access. In the initial state shown in figure 9a, the set is empty and the entire tree bits are set to '0'. The first reference to block *A* results in a cache miss, PLRU policy is searching first for an invalid line in the set where block *A* can be placed, because the set is empty the first line to the left is chosen, also the tree bits are set to point away from block *A*, shown in the figure 9b. For the following block accesses *B*, *C*, and *D* shown in the figures 9c, 9d, and 9e respectively also cause cache misses, the same procedure is used to place the new blocks as in the first block access. After those accesses the cache set is full and the tree bits point to the line containing block *A*. An access miss to block *E* evicts the memory block *A*, and all tree bits on the path to the root of the tree are made to point away from the new referenced block *E*, figure 9f shows the tree state. It is not necessary to change every bit in the path; the bit is kept not flipped when it already points away from the new referenced block. For the following hit reference to block *B*, the bits on the path from *B* to the root of the tree are made to point away from *B*, as shown in the figure 9g. Another access to *B* would not change the tree bits at all, as they already point away from it. The next access to block *A* evicts block *C* which is the least-recently-used block in the set, and an update to the binary tree is made

to point away of block *A*. However, the access to block *A* protects block *D* as well, because the root bit is flipped to point away from the two neighbor blocks, shown in the figure 9h. While block *D* is the least-recently-used block in the set and it should be the victim block for the next miss, accessing block *F* affects block *E* instead of *D* and this property reduces the predictability of PLRU replacement policy compared to the real LRU replacement policy. Finally, figure 9i shows the last state of the binary tree after the end of the access sequence, the tree bits now point to block *D*, which will be replaced on the next cache miss.

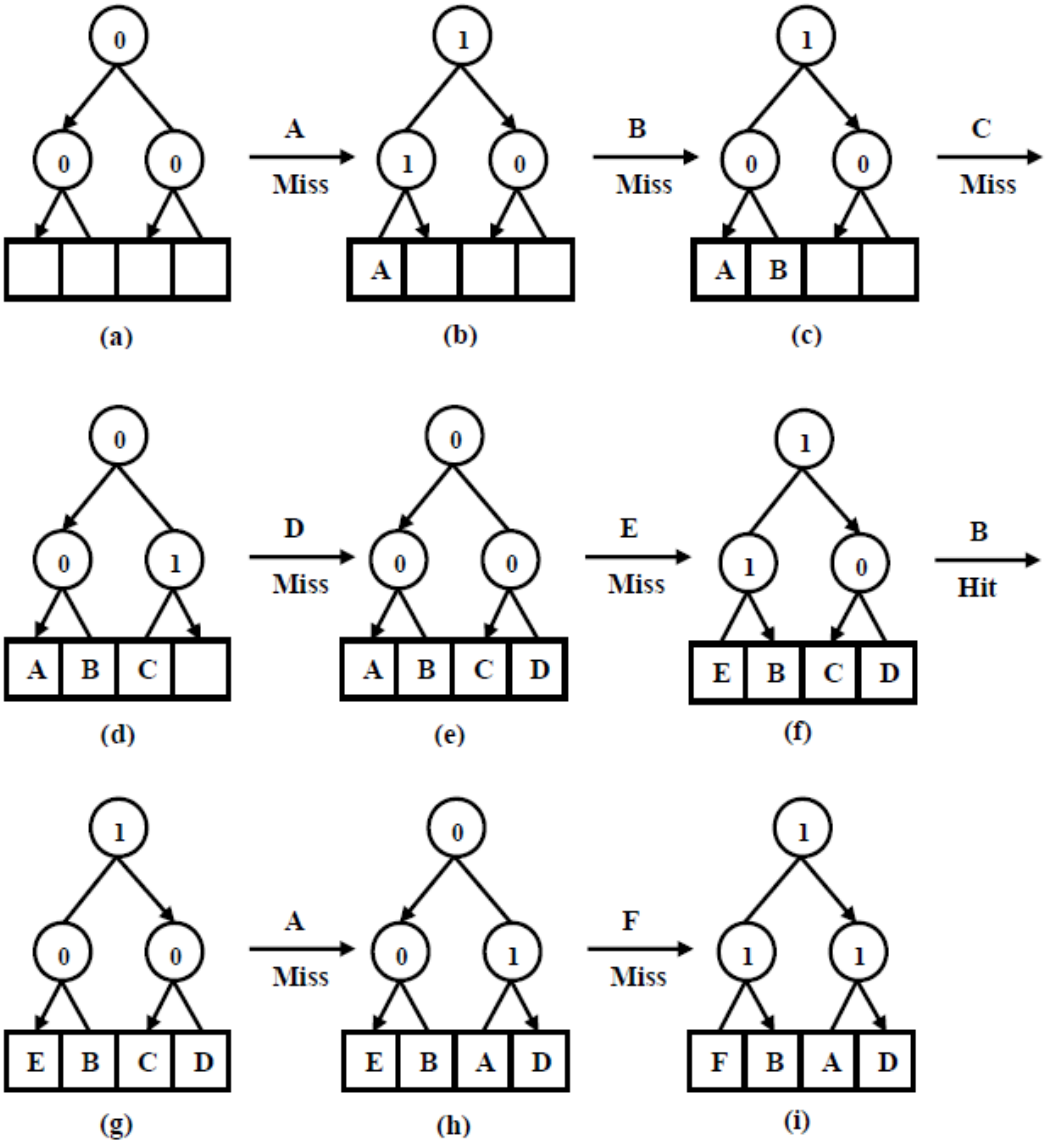

Figure 9 Updates of a PLRU tree of 4-way cache set

The advantage of PLRU replacement policy over real LRU replacement policy is that PLRU is much cheaper to implement, reducing the hardware overhead in terms of storage requirements and update logic. For *n*-way set-associative caches, PLRU requires only *(n - 1)* bits per set to track set reference information. But PLRU has a disadvantage that it

does not always replace the least-recently-used element, thus this property reduces its predictability [2]. PLRU is used in PowerPC series of microprocessors, Intel Pentium II-IV, Intel Core Processors and AMD microprocessors.

2.9 Cache Writing Policies

The write policy determines how the data is written to the different memories in the hierarchy. For the store instructions, the data from the processor is being written into only the data cache keeping it inconsistent with the lower levels in memory hierarchy, because they will have different values of the content data. For this concern, there are two schemes to perform cache writing:

Write through: Data is always written to the cache and immediately to the next lower level of memory hierarchy, ensuring that data is always consistent by updating the blocks in both of them. This takes a long time especially when the next level is always slow, while performing the write operation the processor must wait until the write is complete, this is called write stall and it slows down the processor considerably. Using a **write buffer** which holds the data while it is waiting to be written to the next level will solve the problem, so the processor can write the data to the buffer and continue execution after it. If the write buffer is full and the processor needs to add a write operation, then the processor must stall until there is an empty place in the buffer [1].

Write back: Data is first written to the cache so the block in the cache is only updated, and then the modified block is written to the lower level of memory hierarchy when the block is evicted from the cache. For this purpose and to distinguish the modified block, a sign bit called **dirty bit** is associated with each cache write back block, this bit is asserted whenever a word in the block is modified, so it gives an indication whether the block is clean (not modified) or dirty (modified). On a cache replacement, if the replaced block's dirty bit is asserted, the block is written back to the lower level of the hierarchy by

placing it in the buffer, otherwise it is overwritten by the newly placed block since an identical copy of the block is found in the lower level [3].

Write through policy has a number of advantages. First of all, it is easy to implement, because there is no need for an additional dirty bit per cache block to show that the block is modified. Also, in write through policy there is no need to write a complete block back to the lower level every time when a modified block is replaced by a cache miss: only the written data is sent to the lower level, and thus cache misses are cheaper with write though policy [3]. But write back policy has several advantages over write through policy. On such advantage is that memory write operations will be faster, because they always happen at the speed of the cache [1]. Another advantage is when several writes happen to the same block, it is needed only to write the block once to the lower level of memory hierarchy. Therefore less bandwidth of memory element in the memory hierarchy is used which makes write back policy more effective to use especially in multi-core processors [15]. Since write back does not use the memory hierarchy so often for writing, much power as compared to write through is saved, making write back more preferable to be used in embedded systems [14].

In case of a cache write miss, the data is not needed by the processor, so there are two options: either to allocate the block in the cache then perform a write hit to the cache which is called **write-allocate**, or not to allocate the block in the cache and modify the block only in the lower level memory which is called **no-write-allocate**. In write-allocate, write miss acts like read miss followed by write hit to the cache. For no-write allocate, write misses do not affect the cache [3].

Normally, write back policy is implemented together with write-allocate in the cache, because several writes may happen to the block, while for write back the write is done to the cache block, so it needs to be allocated in the cache. Caches which implement write through policy also use no-write-allocate together with it, because all writes to the block will be sent all the time to the lower level memory, hence no need to locate the block in the cache [2].

2.10 Three kinds of Cache Misses

Cache misses are classified into three categories:

Compulsory misses or cold start misses are the misses that happen upon the first access of the memory block that has never been in the cache. This kind of cache misses can appear in every cache organization. The cache size and the associativity of the set cannot make any improvement to the cold misses, but increasing the block size can somewhat reduce the cold misses, because it reduces the number of memory references to different memory blocks. However, it has a negative effect to the whole system performance by increasing the miss penalty [1]. The use of cache perfecting mechanism by bringing the next expected block to be accessed can reduce the cache misses caused by a cold start [13].

Capacity misses are misses that happen because of the small cache being unable to hold all blocks needed during the program execution. This kind of cache misses appears effectively even with full associativity, when the cache is full and some blocks are being replaced by other referenced blocks and later the replaced blocks are retrieved to the cache again, because they are accessed repeatedly [2]. Capacity misses comprise the majority of cache misses and are reduced by increasing the cache size [1].

Conflict misses or collision misses are misses that occur in direct-mapped or set-associative caches, when the cache is partitioned into sets with a specific associativity. The number of memory blocks that map to the set is always greater than the associativity of the set. Conflict misses happen when the number of blocks needed to be placed in the set at the same time exceeds the associativity, so the blocks which are replaced due to conflicts by other blocks map to the same set are referenced again later causing a conflict miss. The number of conflict misses can be reduced by increasing cache associativity, but this can also increase the access time of the cache, leading to overall system performance degradation [1].

2.11 Multilevel Cache Structure

Multilevel cache is a memory hierarchy with multiple levels of caches, rather than just a single cache and main memory [1]. Many processors use multilevel cache hierarchies to reduce both the latency of cache misses (miss penalty) and the cache miss rate. Modern processors usually support two to three cache levels.

Consider a processor with a multilevel cache which consists of two levels. The first level (L1) is the primary cache which is often smaller and faster cache to reduce the cache access time, while the second level (L2) is the secondary cache which is larger and uses higher associativity to reduce the cache miss rates. If a miss occurs in L1 cache, then L2 cache is accessed to search for the desired data. When the data is found in L2, the miss penalty of L1 is equal to the access time of L2 which is very small compared to the access time of main memory. If L2 does not contain the desired data, an access to the main memory is required, causing a larger miss penalty [1]. The miss rate of L1 becomes less important in presence of L2, thus L1 can be made smaller and faster reducing its access time. Also the access time of L2 becomes less critical in the presence of L1, thus L2 can be made larger reducing its miss rate [1].

2.12 Different Types of Cache Hierarchies

In a multilevel cache hierarchy, the levels which come after the first level in the cache hierarchy can be one of the following types: **Inclusive**, **Non-Inclusive**, or **Exclusive**. The inclusive cache requires that the content of all the smaller cache levels higher in the multilevel cache hierarchy (closer to the processor) is a sub-set of the inclusive cache. Then all memory blocks which are held by the higher levels are also included in the inclusive cache, and when a block is evicted from the inclusive cache, that block must be invalidated in all higher levels of cache (if it resides there) to guarantee the inclusion.

Those blocks which are invalidated or removed from the higher cache levels because of the inclusion property are called inclusion victims [17]. On the other hand, a non-inclusive cache allows memory blocks to be in the higher levels of the cache hierarchy without also being duplicated in it. Thus, it does not guarantee that the contents of the smaller cache levels are always a sub-set of the non-inclusive cache contents. In the exclusive cache model, the memory blocks in the cache are not present in the higher levels of the cache. So there is no intersection between the contents of an exclusive cache level and any higher cache level [20].

Consider a multilevel cache hierarchy with two levels, first level of cache (L1) and last level of cache (LLC). Figure 10 explains the different types of cache hierarchy between the two levels. In the inclusive hierarchy shown in figure 10a, any miss in L1 either hits in LLC or generates a miss in LLC, causing the memory block to be brought into both L1 and LLC. Likewise, when a memory block is evicted (invalidated) from the LLC, it must be sent to L1, where it will cause the block to be invalidated if it exists. In such case, the capacity of the cache hierarchy equals the size of LLC, because the content of L1 is always replicated. For non-inclusive hierarchy shown in figure 10b, no back invalidation is sent to L1 if a block is invalidated in LLC. Therefore, the removed block from LLC can be still present in L1, and due to this, the capacity of non-inclusive hierarchy is in the range between the size of LLC and the size of all levels in the hierarchy together. Figure 10c illustrates the cache fills in the exclusive hierarchy. In case of cache miss in both levels, the memory block is brought first to L1, afterward it is placed in LLC upon eviction from L1. When the block is referenced later, it is invalidated from LLC and placed again in L1. Thus, there is no memory block which is replicated in the two levels. In exclusive hierarchy, LLC works as a **victim cache** for the upper levels, and the capacity of the cache hierarchy equals the size of all cache levels in the hierarchy [16].

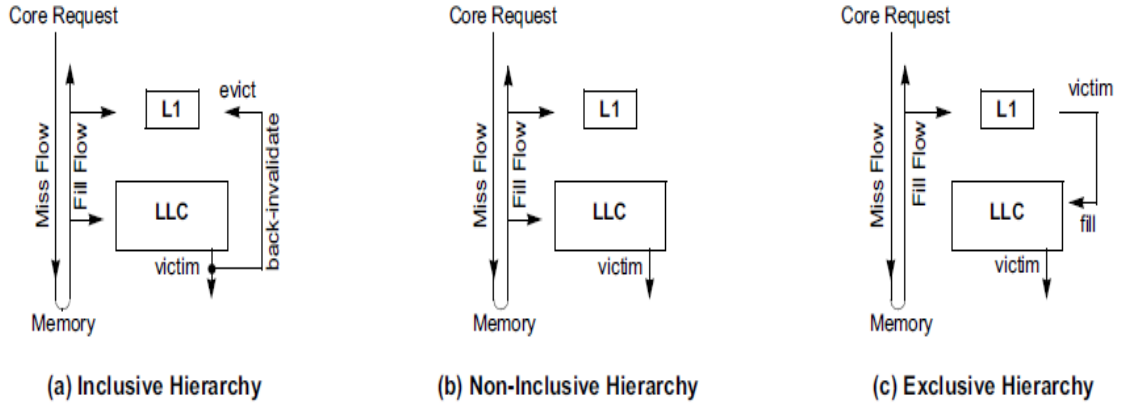

Figure 10 Multilevel cache hierarchies [16]

The different cache hierarchies have advantages and disadvantages. The inclusive cache suffers from effective space reduction as a result of data duplication to guarantee inclusion. Also, in case of back invalidation when a block is evicted from the lower level cache, that block should also be invalidated in the upper level cache. Those reasons cause a performance loss in inclusive cache, and to have a better performance, the size of the LLC should be larger than or equal to the sum of all upper levels of cache hierarchy [18]. On the other hand, inclusive property plays a significant role in multi-core cache coherence, making the inclusion property useful. Non-inclusive and exclusive caches enable higher capacity with the same cache size, but they make the implementation of multi-core processor cache coherence harder (see the next chapter). Moreover, exclusive cache requires higher bandwidth since every victim block from the higher level even those clean have to be written to the lower level [21]. When a cache miss at a higher level occurs, the new block is brought from the lower level to the higher level and at the same time a victim block (if exist any) is sent back to the lower level. Therefore, every cache miss may cause a two-block exchange between the two levels of cache.

3 Cache Structure in Multi-Core Processors

3.1 Multi-core Processor Cache Architecture

The multi-core processor or chip-multiprocessor (CMP) is a small number of symmetric **core** processors on a single chip, they use a centralized shared memory and all core processors have a symmetric equal access to it, this model of architecture is called symmetric multiprocessors (SMP). The number the processor cores in the multi-core chip is typically eight or fewer. Therefore, having a single shared memory for the core processors is possible. Sometimes, the SMP architecture is also called uniform memory access (UMA) multiprocessor, for the same reason of all processors having uniform access latency to the memory [3].

Most modern multi-core processors implement the SMP architecture. They employ multilevel cache hierarchies to reduce the cache access time during program execution, and also the latency of cache misses. Normally, the first levels in the cache hierarchy are small and **private caches**, so that every processor core has its own private cache, while the last level in the cache hierarchy (LLC) is a large and **shared cache** between all processor cores. This architecture is implemented in most of modern multi-core processors like Intel Nehalem [40], AMD Opteron [24], IBM Power 7 [38], Sun T2 [39], etc. Having a private cache within each core reduces the requests on the global interconnect among the cores, and thus, reduces the access latency to the data. On the other hand, shared cache is used for sharing the data among the cores and increasing the performance by reducing the core communication complexity. In short, it is used by the cores as a channel to communicate with each other. In Intel's Nehalem processor and

AMD's Barcelona processor, L1 and L2 are private caches per core and L3 is a shared cache [22].

Figure 11 shows the memory hierarchy for a multi-core processor with 4 cores. The processor uses a two level inclusive data cache hierarchy, where each processor core includes an L1 private cache and all cores share the L2 cache. Whenever the data is loaded to the private cache, it has to be placed in the shared L2 cache. Therefore, L2 cache contains the superset of the private caches [19].

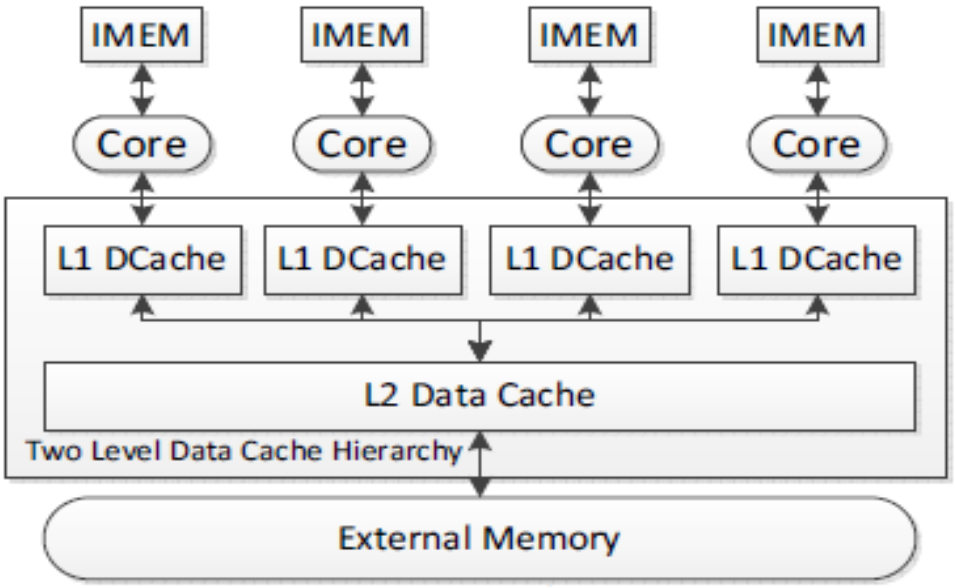

Figure 11 Two-level cache hierarchy for multi-core processor with 4 cores [19]

Multi-core processors are used to improve the performance of stand-alone applications. In order to do it and to exploit the power of the parallel computing provided by the multiple cores, the application must be split into multiple entities called **threads** which run simultaneously on the different cores [26].

The cache in a multi-core processor usually caches both shared and private data for the different threads. Private data is used by a single core, and when the data is cached, it is allocated in the core private cache. As the data is not used by the other cores, the running program on the core behaves the same as when it runs on a uniprocessor system. However, shared data is used by multiple cores. Basically, it is used to provide communication among the different cores through reads and writes. For this reason and to insure correctness, shared data has to be protected from being accessed concurrently by multiple threads with at least one writing thread. In software, the **mutual exclusion** principle is used in multi-threaded applications to protect the **critical sections** where the shared data is updated in the code. Mutual exclusion ensures that only one thread at a time accesses shared data. The implementation of critical sections in programming is done through the **synchronization primitives** which are software routines built over hardware synchronization instructions. Those instructions are uninterruptable, and thus, they provide exclusive access to shared memory location. As a result, it is guaranteed for one thread to have a mutual exclusive access to the critical section [27]. A number of synchronization mechanisms are used in the modern programming languages as synchronization mechanisms in multi-threaded programs such as semaphores [28], conditional variables, and monitors [29].

In the cache, when shared data is cached and used by multiple cores, it is replicated in the private caches of the multiple cores. This replication is good for reducing cache access latency. It also provides a reduction in contention that may happen to the shared last level cache when multiple cores are trying to access shared data item concurrently [30]. Shared data replication in the private caches can increase the performance, however, it causes a new problem called **cache coherence** when multiple cores are trying to update the shared data.

As mentioned before, in a typical multi-core processor design, caches are with multiple levels and the last level cache (LLC) is shared across the cores in order to improve scalability. The cache hierarchy between the shared LLC and the core private caches can

be inclusive. This requires that all memory blocks cached in all core private caches are also present in the shared LLC. Same thing is also true for inclusion property. When a memory block is brought to the processor, it is first allocated in the shared LLC, and then it is forwarded to the requesting core to be stored in its private cache. Afterward, when the cached block is evicted from the shared LLC, all cached copies of the same block in the private caches (if present) have to be invalidated. Inclusive cache hierarchies are widely used in multi-core processors since they can simplify the implementation of cache coherence and minimize the coherence traffic between the cores. Inclusive caches maintain a set of core valid bits called **snoop filter** [34] per cache block in the inclusive LLC. Each bit represents a core. If any of the core private caches may contain the same cache block, then core valid bit is set to 1. If no bits are set, no need to check the other cores. However, inclusion reduces cache capacity and also has backward invalidation effect, both resulting in performance reductions [31]. On the other side, if the cache hierarchy between the shared LLC and the core caches is exclusive, it requires that the contents of the core caches are not replicated in the shared LLC, but it is allowed to share data by more than one core. In an exclusive cache hierarchy, the shared LLC works as a victim cache for the cores' private caches, which means that the memory block newly brought to the processor is placed first in the requesting core private cache, then when it is evicted from the core it is stored in the shared LLC if the block is not cached by any other core. Exclusion cache hierarchy can increase the total on-chip cache memory capacity, but on the other hand, it cannot benefit from the natural snoop filter in the inclusive cache, because the contents of the LLC and the core private caches are always different. In exclusive cache structure, since snoop filters cannot be associated with the LLC, a new structure called **directory** [35] exists beside the LLC that is used to hold tags and snoop filters for all cached blocks in the cache hierarchy. This increases the hardware overhead and verification complexity [16].

Figures 12 and 13 show the differences between exclusive and inclusive caches with embedded snoop filters in the inclusive cache. The cache hierarchy is with three levels and the shared LLC is the third level (L3). A data request from core 0 misses both L1 and

L2 core private caches, and then it is forwarded to look up in L3 cache. In figure 12, the requested data is a hit in L3. For the exclusive cache, there is no need to check the other cores, because exclusion guarantees that the data would not be present in any of them. In the inclusive cache, the data can be also in another core's cache, but the snoop filter is can tell in which core the data is present, therefore, a core is checked only when its core valid bit is set, and only when the data is modified, so the L3 data copy is not updated. In figure 13, the requested data is missed in L3 cache. For an exclusive cache without snoop filter, the request must be forwarded to all other cores to look up the data. On the other hand, the inclusion property guarantees that when the requested data is not present in L3, then it is not cached anywhere else in the processor die.

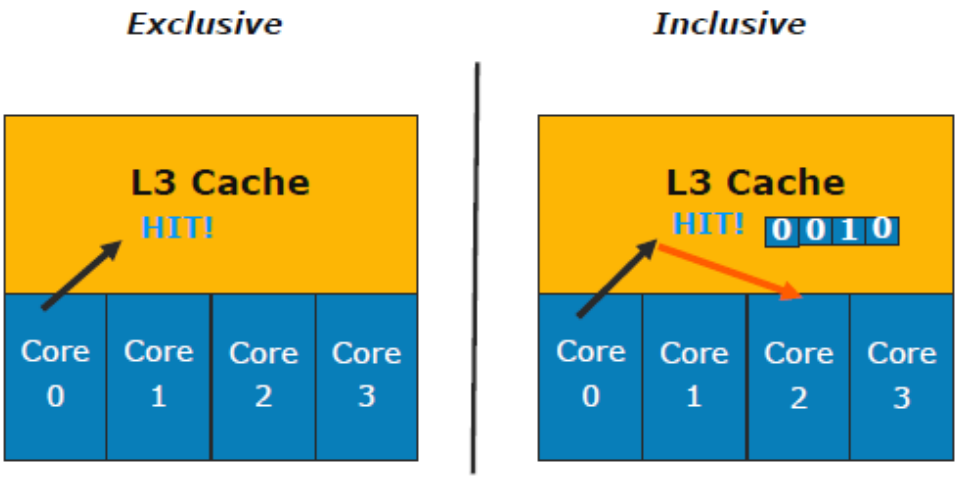

Figure 12 Exclusive vs. Inclusive cache hit [32]

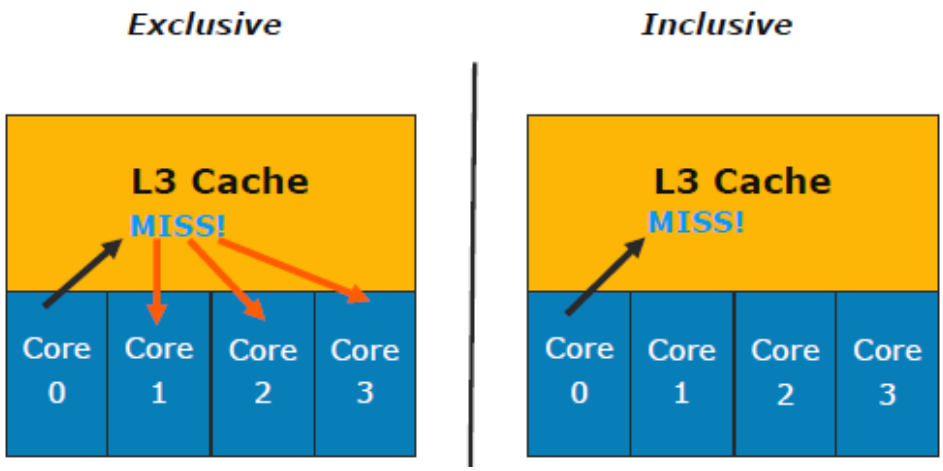

Figure 13 Exclusive vs. Inclusive cache miss [32]

3.2 Cache Hierarchies in the Intel Nehalem and AMD Opteron Processors

This section shows the cache hierarchies of two modern multi-core processors: the four-core Intel Nehalem (Core-i7) [36] processor and six-core AMD Opteron (Istanbul) [37]. Both have three levels of cache hierarchy, all are on the main processor die. The outermost cache in both is shared among cores, is inclusive in the Intel Nehalem and exclusive in the AMD Opteron. Both processors have on-chip memory controllers, which reduce the main memory latency.

3.2.1 Four-Core Intel Nehalem Core-i7

Intel Nehalem implementation contains four cores in a processor chip and supports hierarchy of up to three levels, figure 15 shows the Intel Nehalem core-i7 cache hierarchy. It divides the physical memory into blocks of 64 bytes, while all caches throughout the hierarchy have the same block size.

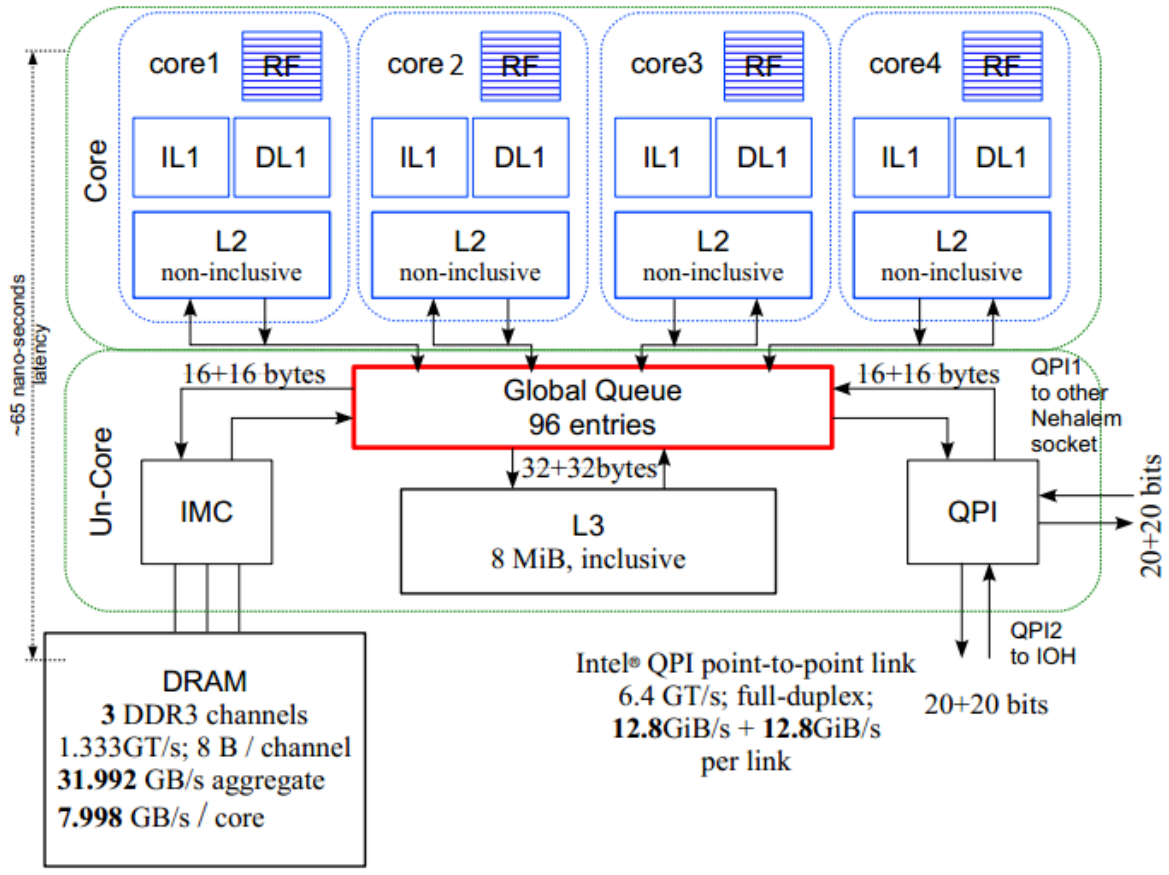

Figure 14 Intel Nehalem Core-i7 processor [23]

Each core has separate L1 caches for instructions and data, and its own unified (for both instructions and data) and non-inclusive L2 cache. The four cores share L3 cache, which is unified and inclusive of the core caches, meaning that every memory block that exists in either L1 data or instruction cache, or the L2 caches, is also existent in L3. Fills from the main memory are allocated first in the L3 cache, and then they are directed to the appropriate core, to be stored first in the L2 cache then forwarded to the L1 cache. The writing policy used in all cache levels is write-back, so whenever a modified block is evicted from any cache, the block is written back to the next cache in the hierarchy. The replacement policy which is used in L3 cache is a variant on pseudo-LRU, the replaced block is chosen based on PLRU but with an ordered selection algorithm. If a memory

block is replaced in L2 cache, no block invalidation message for the evicted block is sent to L1 cache, because L2 uses non-inclusive hierarchy. But when a block is evicted from the inclusive L3 cache, all existing copies of the block in L1 and L2 caches have to be invalidated to hold the inclusion. The inclusion property is implemented to minimize the snooping traffic among the cores. A 4-bit snooping filter associated with each L3 block indicates if the block is already cached in the L2 or L1 cache of a particular core. Therefore, block snooping is done through the L3 cache to track a particular block status, so there is no need to forward a broadcast snoop message to all cores [23].

3.2.2 Six-Core AMD Opteron (Istanbul)

The six-core AMD Opteron (Istanbul) is a multi-core processor that integrates six cores, and a shared 6 MB L3 cache on one die. The core has separate instruction and data caches backed by a large L2 cache. All caches throughout the hierarchy have 64-byte lines, and use write-back policy. Figure 16 shows the six-core AMD Opteron memory hierarchy. As in many other AMD processors, the L1 and L2 caches use an exclusive hierarchy, so that the fills from the main memory go directly into L1 cache in the appropriate core and are not placed in the L2 cache. When any memory block is evicted from the L1 cache, it moves into the L2 cache. Also, the other way around, the blocks which are hit in core's L2 cache are invalidated from L2 and placed into the requesting L1 cache. Here the L2 cache works as a victim cache for the L1 instruction and data caches. The 6 MB exclusive shared L3 cache works as a victim cache for cores' L2 caches. Also, a part of the exclusive cache is used to store the cache directory (snoop filter or probe filter) [24] for cache coherence. Every cached block in the hierarchy has an entry in the cache directory. The entry contains the block tag and cores' tracking bits, they show which of cores have a copy of the block and the status of that copy. All blocks that are evicted from cores' L2 caches, are stored in the L3 cache, the cache uses pseudo-LRU replacement strategy to place a new block into a full cache set. Having L3 as a victim cache allows caching more data, also back invalidation of the L2 caches is not needed when a block is victimized in L3 cache. AMD Opteron implements a semi-exclusive cache hierarchy at L3 cache [22]. When a core victimizes a particular block, it

is stored in L3 cache to detect a true sharing pattern to that particular block. If the next request to the block is from the same core where the block was located before, the data seems to be private to the core, therefore, the L3 does not keep a copy of the block. Else, the L3 cache can retain a line after providing a copy to a requesting core for further sharing [24].

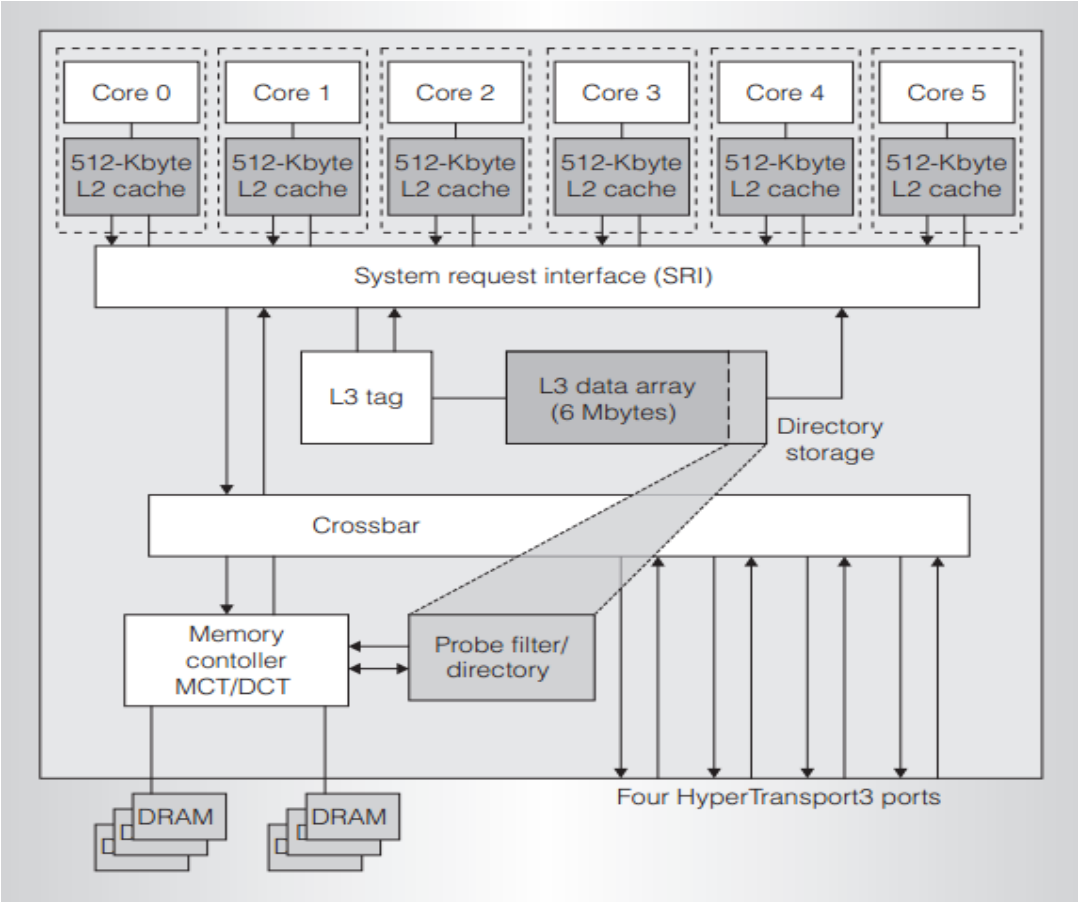

Figure 15 Six-Core AMD Opteron processor [24]

The following table 10 summarizes both Intel Nehalem Core-i7 and AMD Opteron (Istanbul) processors' cache characteristics.

Table 10 Three level cache in the Intel Nehalem and AMD Opteron processors [1]

Characteristic	Intel Nehalem Core-i7	AMD Opteron (Istanbul)
Number of cores	4	6
L1 cache organization	Separated cache per core	Separated cache per core
L1 cache size	32 KB each for (I/D) cache	64 KB each for (I/D) cache
L1 cache associativity	4-way (I), 8-way (D)	2-way
L1 replacement policy	Pseudo-LRU	LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time	4 cycles, pipelined	3 cycles
L2 cache organization	Unified cache per core	Unified cache per core
L2 cache size	256 KB	512 KB
L2 cache associativity	8-way	16-way
L2 cache hierarchy	Non-inclusive	Exclusive
L2 replacement policy	Pseudo-LRU	Pseudo-LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	10 cycles	12 cycles
L3 cache organization	Unified cache shared	Unified cache shared
L3 cache size	8 MB	6 MB
L3 cache associativity	16-way	16-way
L3 cache hierarchy	Inclusive	Semi-exclusive
L3 replacement policy	Variant of Pseudo-LRU	Pseudo-LRU
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	35-40 cycles	38 cycles

3.3 Cache Coherence Problem

In a shared-memory multiprocessor with private data caches, when a cached block is replicated in at least two private caches, modifying one of the block's copies will cause a cache coherence problem. In general, the coherence problem exists because there are two

states for the data, one is the global state, defined by the main memory or the shared last level of cache (for example L3 cache) in multi-core processors, and the other is local state defined by the cores' private caches (for example L1 and L2 caches) [15]. Consider a multi-core processor with two cores C1 and C2, each having a private cache which is connected to the memory. The two different private caches can have two different values for the same memory location, table 11 explains this. Consider the initial value at time 0 of $x = 1$ at time 0 which is not cached in any of the cores' private caches, also assume the writing policy that is used by the caches is write-back. At time 1, C1 reads *x* from the memory and places it in the core's private cache. At time 2, C2 reads also *x* and places it in the core's private cache. Now, both private caches and memory have the same value of $x = 1$. After that the value of *x* has been written by C1 at time 3, C1's private cache has the new value, while C2's private cache has the old value, and if C2 reads the value of *x* again, it will receive 1.

Table 11 Cache coherence problem

Time	Event	Value of *x*		
		C1 private cache	C2 private cache	Memory
0				1
1	C1 reads *x*	1		1
2	C2 reads *x*	1	1	1
3	C1 writes $x = 0$	0	1	1

To ensure consistency among multiple copies of the same memory location, a coherence scheme or protocol, is required. Cache coherence protocols are to maintain coherence among the individual caches in a system of multiple processors. The implementation of a cache coherence protocol tracks the state of any shared data block. There are two classes of protocols in use based on the way they locate multiple copies of the same block and the different techniques used to track the sharing state. One is **directory based** and the other is **snooping based**. In directory based protocols, the sharing status that maintains

the coherence among the different copies of a particular block of physical memory is kept in one location called directory. The directory acts as a filter, so when any processor/core needs to load a memory block, it goes through the directory. Also, when a cached block is modified in one cache, the directory either updates the other caches with the new value of the block or invalidates their copies of the block, depending on the implemented cache coherence method. In snooping protocols, every individual cache that has a copy of a particular block could monitor the sharing status of the block. All private caches are connected to a broadcast medium, i.e., a bus which connects the private caches to the shared cache or memory. The cache controller keeps listening to the broadcast medium and updates or invalidates the snooped block based on the cache coherence method and the request on the bus. In multi-core processors, if all cores share some level of the cache, the directory can be associated with the outermost cache, and a snooping scheme can be built over the directory to obtain the cache coherence. The directory is used to reduce snoop traffic in the broadcast medium by propagating the coherence messages only to the caches which have a copy of the block. This can reduce the overhead to the cache controller by consuming less medium bandwidth especially with an increasing number of cores, which makes this architecture more scalable. If the multi-core processor does not implement a shared last level of cache, a pure snooping scheme can be used to obtain cache coherence between cores' private caches [3].

Cache coherence protocols can implement two methods for propagating the modifications of the blocks among the different caches. One method is **write update** or write broadcast. The protocols which implement this method must broadcast the writes every time a write happens to a shared block. This consumes much bandwidth and for this reason it is not used in most of multi-core processors. The other method is **write invalidate**, which invalidates all other copies of the shared block once a write is performed on that block. This ensures that a core has exclusive access to a cached block before it writes to that block. Write invalidate protocols are commonly used cache coherence protocols in modern multi-core processors. To illustrate the idea, consider a multi-core processor with two cores, seen in figure 2. Both cores initially have a copy of block *X* cached in their

private caches, the figure shows how coherence actions work to keep local caches coherent in a write invalidate policy. When Core1 issues a write operation to the block *X*, a coherence invalidation request is sent through the bus to invalidate the copy of the block *X* in Core2's cache. Therefore the valid bit of the block *X* in Core2's cache is set to be invalid. Next time, when Core2 wants to read from the block *X* again, it will find the block not present in its local cache and it must obtain a new value of the block. A read miss request will be sent through the bus and make Core1 write back the updated value of the block to the external memory and forward a copy to Core2. If a multi-core processor uses a shared cache, the shared cache will act as the external memory, and the coherency must be handled for the local cache in each core [3].

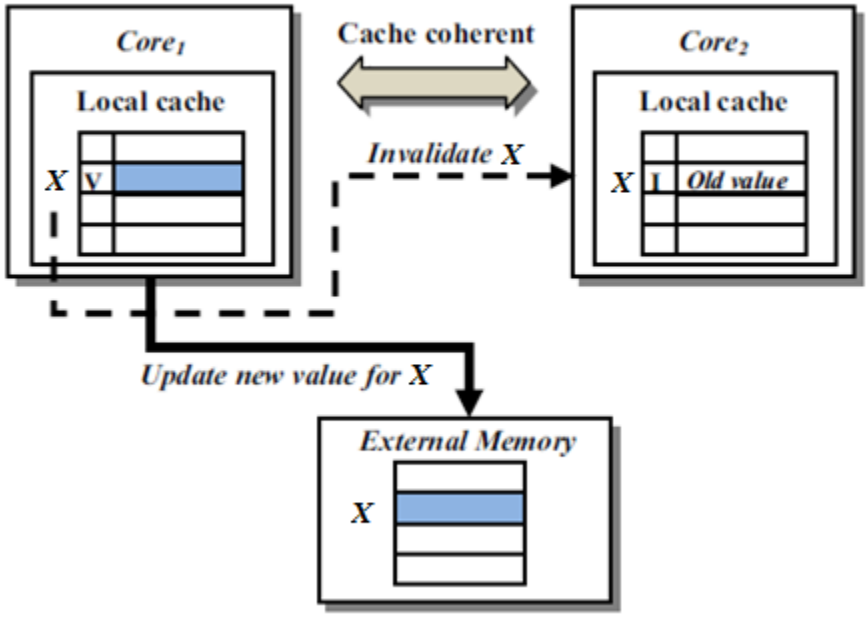

Figure 16 Write invalidate cache coherence method [25]

3.4 MSI protocol for Multi-core Cache Coherence

In multi-core processors where shared cache is present, the cores' private caches are connected to the shared outermost cache (for example L1 and L2 are private caches per-core and L3 is shared cache among the cores in Intel Nehalem and AMD Opteron) via a bus or interconnection network which provides a broadcast medium. The broadcast medium is used for sending coherence invalidation requests. When a core wants to perform a write to a shared block, it attempts to acquire access to the bus and sends a broadcast invalidation with the referenced memory address to the other cores on the bus. The cores keep snooping on the bus, and when they receive the address to be invalidated, all cores check their private cache for the corresponding block. If a match is found, the core invalidates the entire block. The bus offers sequential access and modification to the shared data. When two cores attempt to write to a shared block at the same time, first thing they will do is try to acquire the bus to send invalidation requests. Only one processor at a time can use the bus for sending requests, therefore, bus acquisition for the different cores is serialized. In case of a data miss in the private cache, the same snooping scheme can be used for allocating a data block in the cache when the used writing policy in the cache hierarchy is write-back, since the missed data can be modified in one of the private caches. All cores also snoop for every address placed on the bus, and if a core has the desired block modified in its private cache, then it provides the requesting core with the last updated copy of the block and stops the memory or shared cache access [3].

The simple MSI (**M**odified, **S**hared, and **I**nvalid) protocol can basically be used to implement the cache coherence. The protocol implementation assumes two things: the first assumption is that the cache writing policy which is used at least between the private caches and the shared cache or the memory is write-back and write-allocate, the second assumption is that the block size is the same in the entire memory hierarchy. These two assumptions are held in most multi-core processors implementations, since write-back

does not consume much bandwidth or much time to perform the write operation. The equal block size provides simplicity to the implementation [3]. The MSI protocol enforces one main rule to keep coherence. The rule says, "for a given block, at any given moment in time, there is either: only a single core with write (and read) permission to the block (in state M for modified) or, zero or more cores with read permission to the block (in state S for shared)" [33]. For this, every cache block in a core's private cache is in one of the following states: invalid, shared, and modified. The invalid state indicates that the contents of the block are invalid to use, because the block's contents are invalidated or the block is not yet occupied. The shared state indicates that the cache has an up-to-date and clean copy of the block which can be shared with the other private caches, and the access to the block is allowed for reads only. The modified state indicates that the cache has the only copy of the block which is up-to-date and modified, and the access to the block is allowed for both reads and writes. To track the block state, each cache block associates extra status bits to encode the block state [35].

The following tables and figures explain the MSI coherence mechanisms which are implemented in every private cache controller. The cache receives requests from the core to which the cache belongs and from the shared bus. The requests can be reads, writes, or data invalidations. The cache controller responds to the requests based on the request type and whether the data is present in the cache or not, also it depends on the state of the referenced block which contains the data [3].

Table 12 shows the read hit and write hit requests from the core to its local cache. For the first three requests in the table, the type of the cache action is normal hit and the requests are handled within the local cache. Even for the third request, a coherence action is not required because the referenced block state is modified, so the core has an exclusive copy of the block and no need for invalidation. But in the fourth request, because the referenced block state is shared, a coherence action is needed, since the other cores may have a copy of the block. An invalidate request is placed on the bus to invalidate the other copies (if any) and to obtain the block ownership. Figure 17 shows the state transitions of

the cache block based on the request and the state of the referenced block, it also shows the cache action as a response to the request.

Table 12 Requests from the core (hit in the local cache) [3]

Request	Block state	Type of cache action	Cache action
Read hit	Shared	Normal hit	Read data in the local cache
Read hit	Modified	Normal hit	Read data in the local cache
Write hit	Modified	Normal hit	Write data in local cache
Write hit	Shared	Coherence	Place invalidate on bus

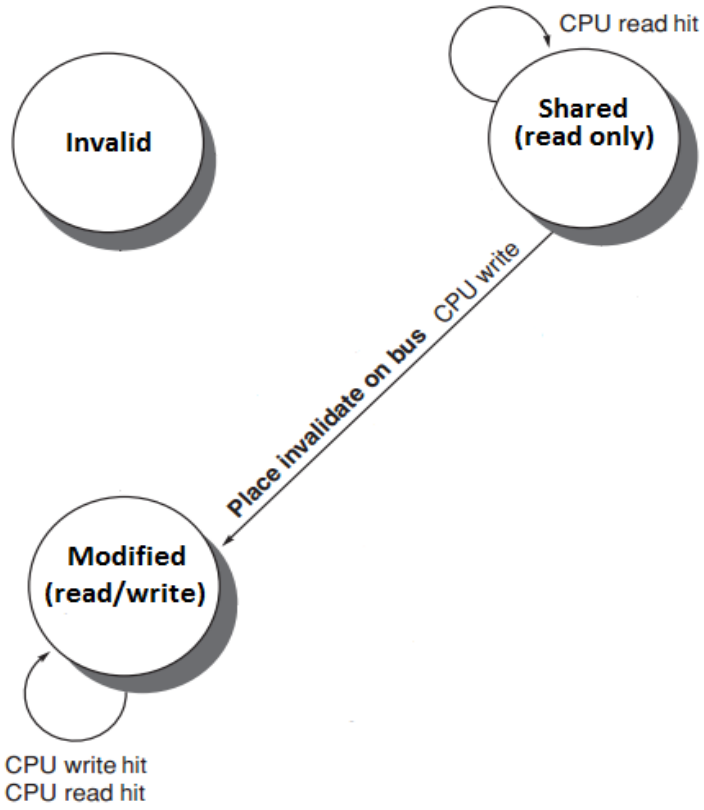

Figure 17 MSI protocol states transition for requests from the core (hit in the local cache) [3]

Table 13 shows the read miss and write miss requests from the core to its local cache. The referenced block here is a victim block which is chosen to be replaced by the new allocated block. If the victim block is in a modified state, then the block's contents are written back to the shared cache or memory before replacement, and after that, the request (read miss/write miss) is placed on the bus to obtain the block from the other private caches, or shared cache, or memory. The write miss request on the bus may cause invalidation in the other caches in order to grant the requesting cache an exclusive access to the new block. Figure 18 shows the state transitions of the cache block with the new allocated block based on the request and the state of the replaced block, it also shows the cache action as a response to the request.

Table 13 Requests from the core (miss in the local cache) [3]

Request	Block state	Type of cache action	Cache Action
Read miss	Invalid	Normal miss	Place read miss on bus.
Read miss	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write miss	Invalid	Normal miss	Place write miss on bus.
Write miss	Shared	Replacement	Address conflict miss: Place write miss on bus.
Write miss	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.

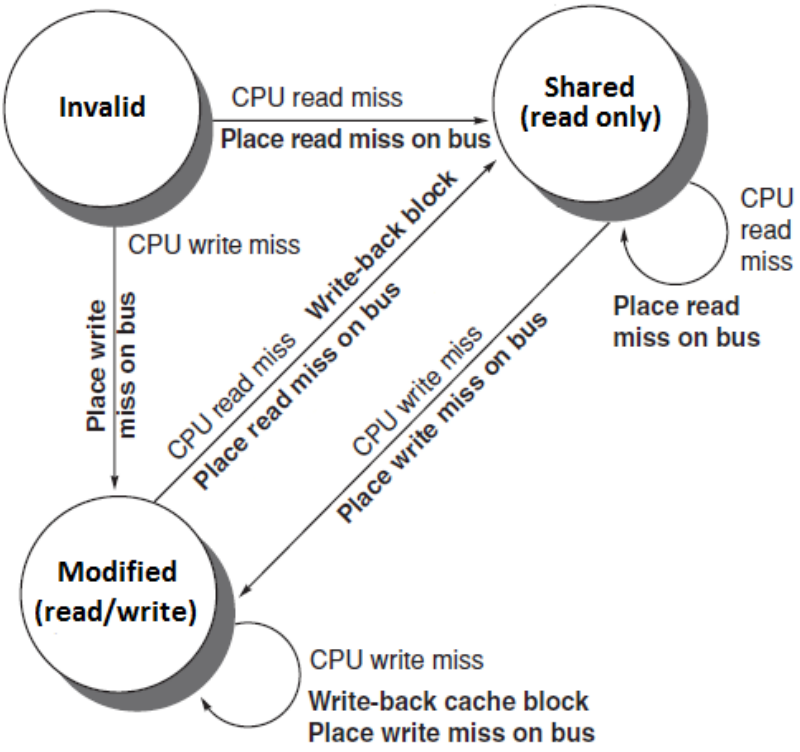

Figure 18 MSI protocol states transition for requests from the core (miss in the local cache) [3]

Table 14 shows the requests from the bus to the private cache. The private cache keeps snooping on the bus for requests like read miss, write miss, and invalidate. The cache responds to the request when it has a copy of the required block. For the first request in the table, no coherence action is needed, because the requesting cache tries to share a clean block which can be obtained from the shared cache or the memory. All other requests need a coherence action from the cache. If the state of the addressed block in the cache is modified, then block contents are written-back and the requesting cache is provided with a copy of the block. Both invalidate and write miss requests cause invalidation to the block contents. Figure 19 shows the state transitions of the addressed block based on the request and the state of the block, it also shows the cache action as a response to the request.

Table 14 Requests from the bus [3]

Request	Block state	Type of cache action	Cache Action
Read miss	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Modified	Coherence	Attempt to share data: write-back the cache block and make its state shared.
Invalidate	Shared	Coherence	Attempt to write shared block: invalidate the block.
Write miss	Shared	Coherence	Attempt to write shared block: invalidate the block.
Write miss	Modified	Coherence	Attempt to write block that is exclusive elsewhere in other local cache: write-back the cache block and make its state invalid in the local cache.

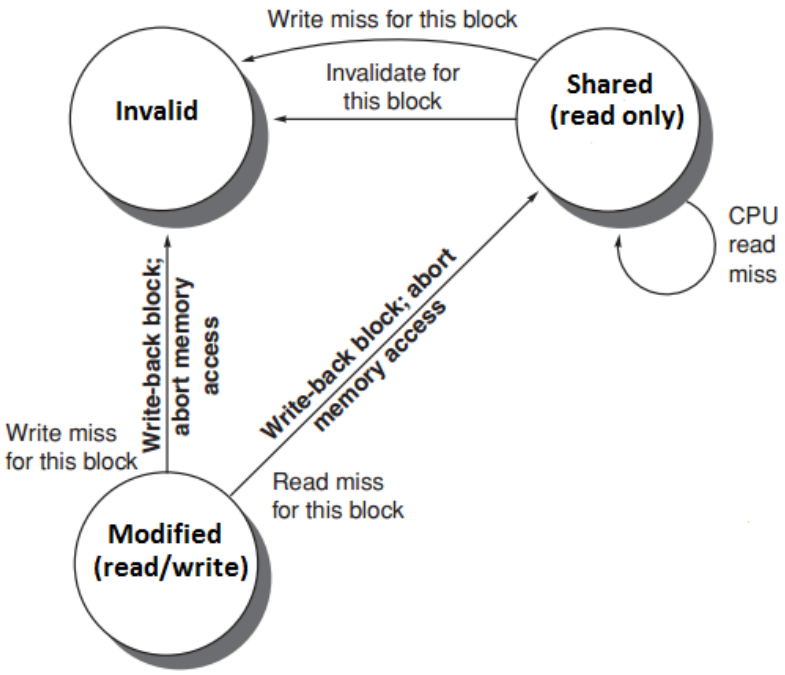

Figure 19 MSI protocol states transition for requests from the bus [3]

In those multi-core processors where the shared cache is inclusive, it includes the contents of all private caches. This allows implementing a straightforward directory scheme in the shared cache. This scheme can reduce the snooping traffic significantly and minimize the interference on the bus. Because inclusion ensures that each private block has a corresponding shared block, the coherence protocol can track the copies of the block in the private caches through the snoop filter (directory entry) in the shared cache. Each block in the shared cache is associated with several bits: two bits for coherence state and the rest for core tracking (one bit per core) showing which core caches the block [33]. Figure 20 shows a system model with inclusive shared cache and the block structure in both private and shared caches. Both blocks *A* (cached by Core 0) and *B* (cached by Core 1 and Core 2) must be present in the shared cache because of inclusion hierarchy. The core tracking bits in each block are set properly to point to the cores which cache the block ; *A{1,0,0,0}* and *B{0,1,1,0}*.

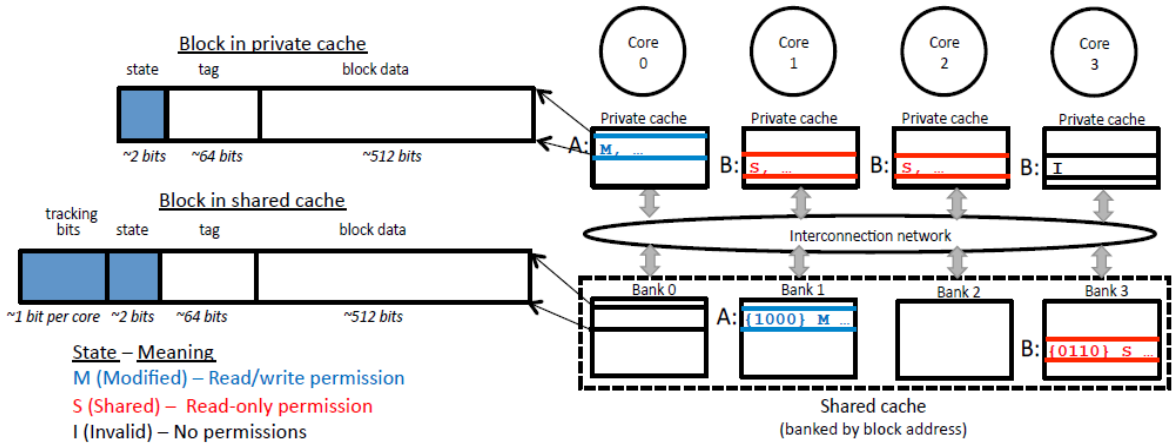

Figure 20 MSI cache coherence model using directory in the shared inclusive cache [33]

When a read or write misses in the private cache, the core sends a coherence request to the shared cache. The shared cache can respond to the request based on the referenced block's coherence state, either directly (when the state is shared), or by forwarding the request (when the state is modified) to the only core whose bit is set in the block's tracking list. The same thing happens when an invalidate coherence request is issued by a core because of writing to a shared block. The core sends the request to the shared cache which in its turn forwards the request only to the cores in the block's tracking list (if any) other than the requesting core. The core private caches then invalidate their copies of the block. The block's tracking list in the shared cache has to be updated when a block copy is evicted from a private cache [33].

This scheme (directory at the shared cache) avoids the need for snooping in the cores, it also avoids broadcast messages and uses point-to-point messages for communication. This minimizes the system overhead and improves the scalability allowing to add more cores to the system. This coherence model is implemented in Intel Nehalem [33].

In those multi-core processors where the shared cache is exclusive, the embedded snooping filter with the shared cache block cannot be used, because the blocks which are cached in the private caches are not cached in the shared exclusive cache. The solution to this problem is to use a directory structure with the shared cache which holds an entry for each cached block in the private caches. The entry contains the block's tag, coherence state, and tracking list [22]. Figure 21 shows the directory structure. This architecture is implemented in AMD Opteron [24].

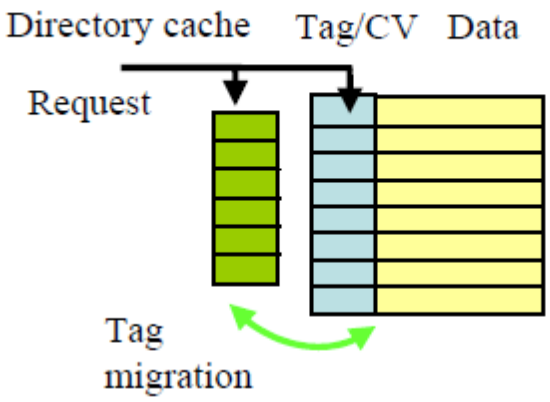

Figure 21 Separate directory with the shared exclusive cache [22]

The MSI protocol has some extensions that arise by adding additional states and transitions to improve the performance if possible, i.e., MESI and MOESI protocol extensions. In MESI a new state called Exclusive is added to MSI, the exclusive state indicates that the block is present in only one private cache but it is clean, this state is represented in the MSI protocol by a shared state with only one bit set in cores' tracking list. The core which has the block in exclusive state can write to it without sending an invalidate request, just the state of the block will change from exclusive to modified. A read miss from any other core to the block will change its state from exclusive to shared [17].

In MOESI a new state called Owned is added to the MESI extension. The owned state indicates that the block is modified in the private cache and out-of-date in memory. In MSI when the block is in modified state and there is an attempt to share the block from another core, the block is written back and its state changes to shared. But in MOESI protocol, the cache can share the latest copy of the block with the requesters, the state of the block changes from modified to owned while the state of the block in the other caches is shared. The state changes from owned to modified again when a core writes to the block after sending an invalidate request. This protocol extension (MOESI) is used by

AMD in Opteron processor [41]. The following table 15 and figure 22 show the MOESI protocol states and the states transition diagram.

Table 15 MOESI protocol states

State	Clean/Dirty	Unique	Read/Write
Modified	Dirty	Yes	R/W
Owned	Dirty	No	R
Exclusive	Clean	Yes	R/W
Shared	Clean	No	R
Invalid	NA	NA	NA

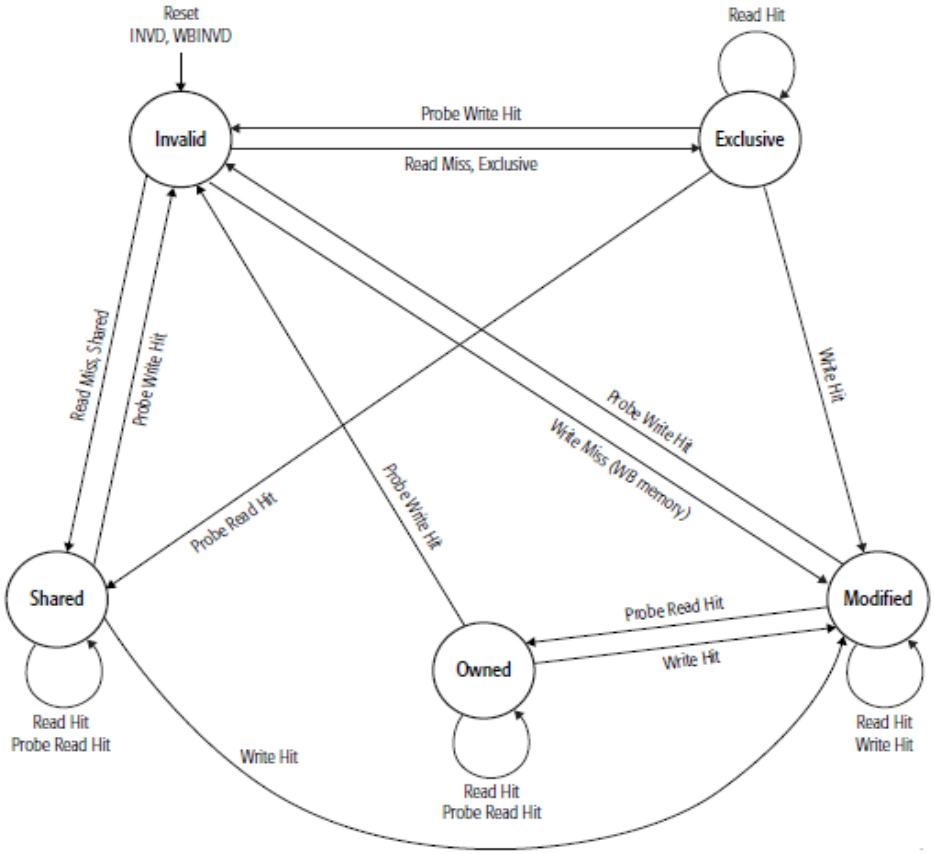

Figure 22 MOESI state transitions [41]

4 Design and Implementation of Cache Simulation

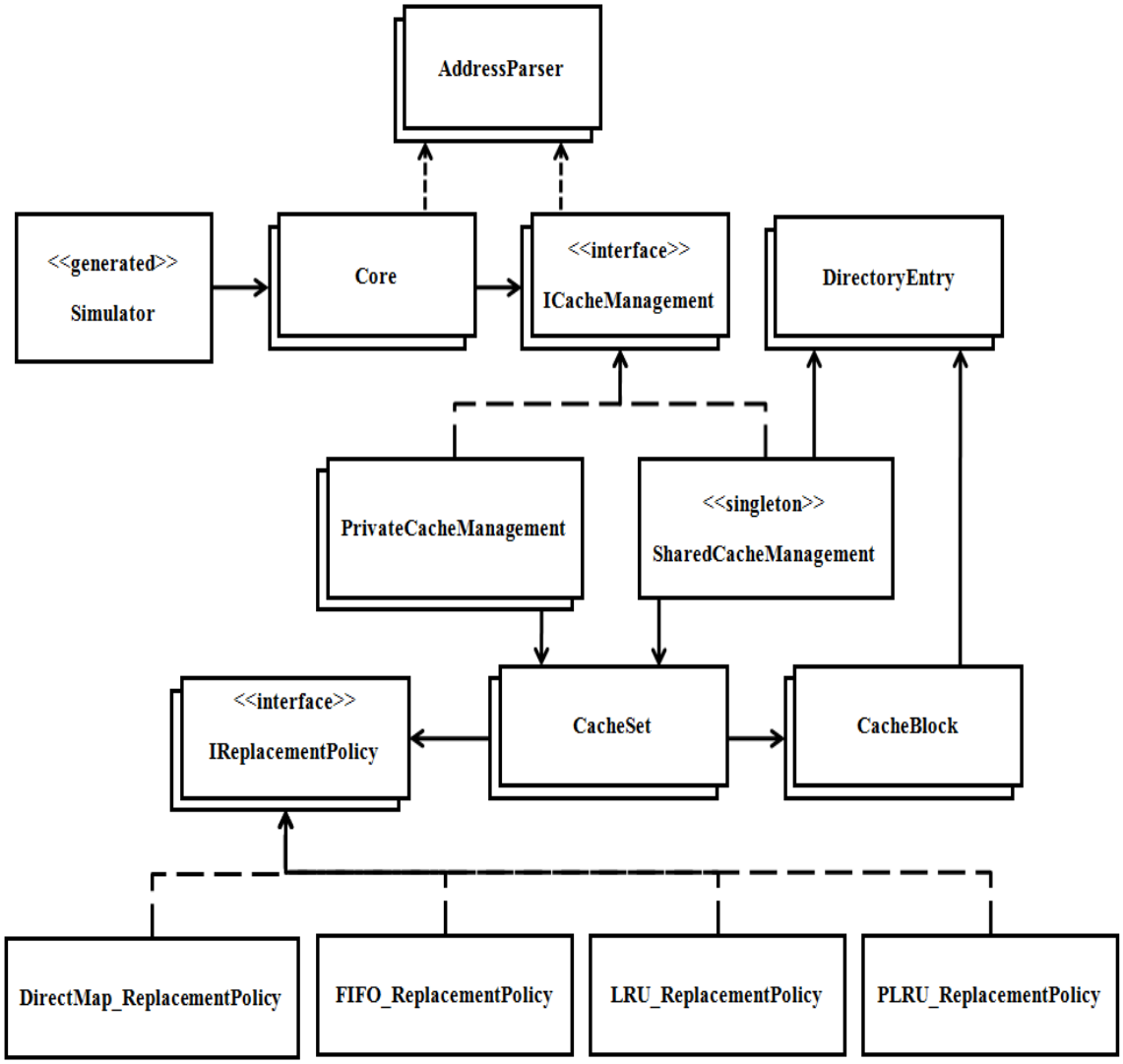

Figure 23 Cache simulation design

Figure 23 provides an overview of the cache simulation design. The classes are represented by boxes, the dashed arrows represent inheritance, the solid arrows represent referencing, and the dotted arrows represent messaging.

4.1 Cache Management

The cache management forms the most important part in the cache simulation design. In the multi-core processor cache hierarchy, we have two categories of caches based on the cache functionality. The first category is the private cache, which represents every single level of cache within the core, and the second category is the shared cache, which represents the shared last level of cache. Both caches share some functions which are common in every cache memory like request a cached block, allocate a cache block, resolve a cache coherence request, and write back a modified block. But the difference is in how each category implements these functions.

For the cache management, we implement an interface which defines all common cache functions, and both the private cache and the shared cache are implemented as two classes which have to implement the interface functions. Each class will implement the interface functions in a way suitable to the cache functionality. Figure 24 shows the UML class diagram from the cache management. The UML Lab-Yatta tool [42] was used to generate this and all further class diagrams. The interface `ICachemanagement` shown on top defines the common class management functions. The class `PrivateCacheManagement` represents the private cache category, and the class `SharedCacheManagement` represents the shared cache category.

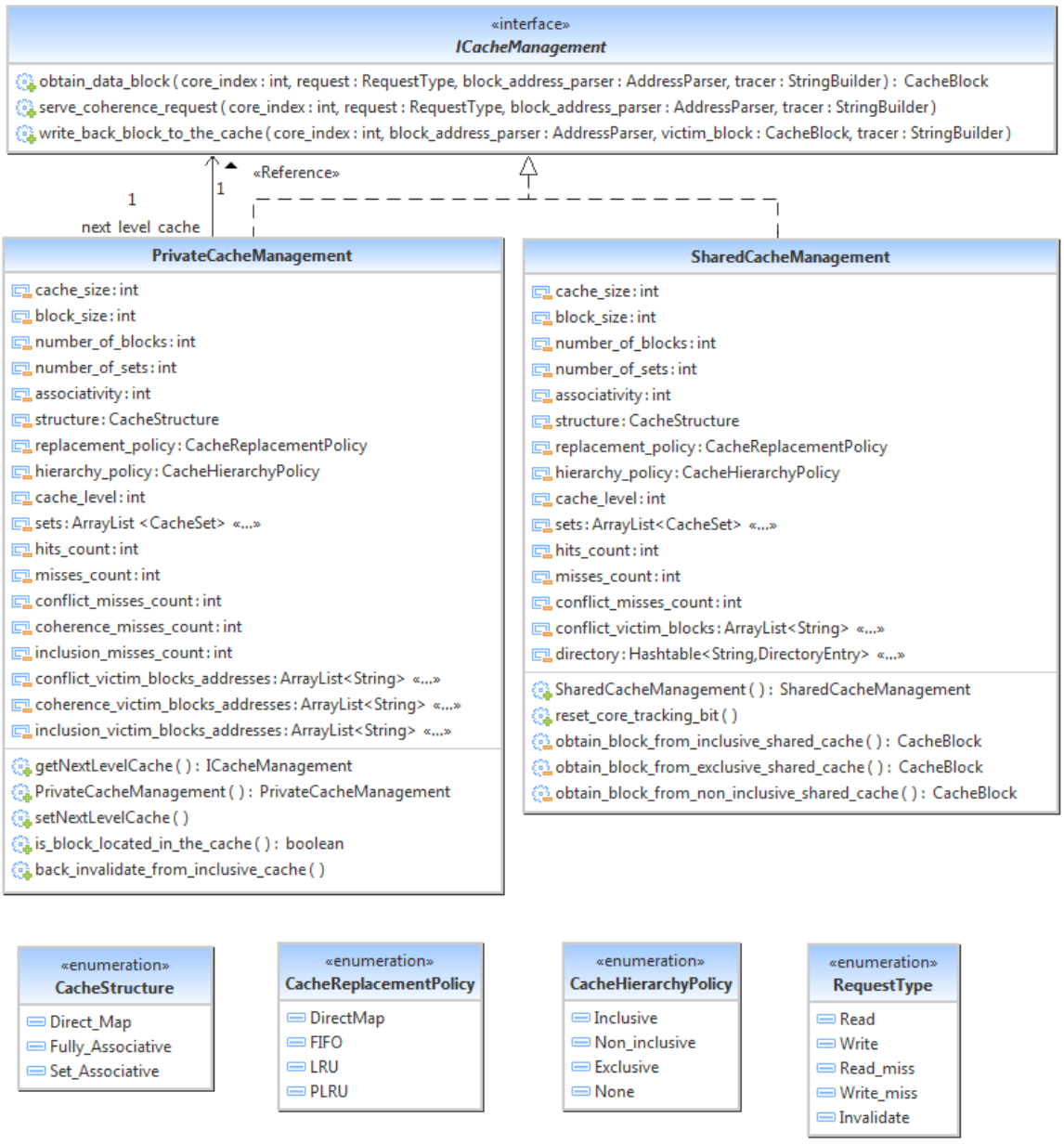

Figure 24 The UML class diagram of cache management

The following table 16 describes the cache management properties which are implemented in private and shared cache management classes.

Table 16 Cache management properties

Property	Type	Description
cache_size	int	The size of the cache
block_size	int	The size of the cache block
number_of_blocks	int	The number of blocks in the cache
number_of_sets	int	The number of sets in the cache
associativity	int	The associativity of the cache set
structure	Cache Structure	The cache structure or organization, Direct_Map, Fully_Associative, or Set_Associative
replacement_policy	Cache Replacement Policy	The replacement policy which is used to choose the next victim block in the cache set for replacement in the next miss, DirectMap where no replacement policy is used, FIFO, LRU, or PLRU
hierarchy_policy	Cache Hierarchy Policy	The type of cache hierarchy, Inclusive, Non_inclusive, Exclusive, or None as in the first level of cache hierarchy
cache_level	Integer	The level of the cache in the hierarchy
sets	ArrayList <CacheSet>	List of the cache sets which represents the cache sets, the index of the array is also used as the set index for mapping
hits_count	int	Counter for cache hits
misses_count	int	Counter for cache misses
conflict_misses_count	int	Counter for cache misses because of set conflict
coherence_misses_count	int	Counter for cache misses because of coherence protocol
inclusion_misses_count	int	Counter for cache misses because of inclusion property
conflict_victim_blocks_addresses	ArrayList <String>	List of victim block addresses due to set conflicts, if any block in the list is referenced again, the result is a conflict miss
coherence_victim_blocks_addresses	ArrayList <String>	List of victim blocks address due to coherence protocol, if any block in the list is referenced again, the result is a coherence miss
inclusion_victim_blocks_addresses	ArrayList <String>	List of victim blocks address due to inclusion property, if any block in the list is referenced again, the result is a conflict miss

The interface `ICacheManagement` defines three functions to manage cache referencing. The function `obtain_data_block` is used to get the cache block which contains the required data for memory access operations. The function `serve_coherence_request` performs cache coherence requests from the other cores or the shared cache. The function `write_back_block_to_the_cache` performs block write-backs for the evicted blocks from the upper level cache when the evicted blocks are modified or the hierarchy policy between the two cache levels is exclusive. Table 16 describes the functions' parameters.

Table 17 ICacheManagement functions' parameters

Parameter	Type	Description
`core_index`	`int`	The index of the core which initiates the request
`request`	`Request Type`	The request type to the cache, `Read`, `Write`, `Read_miss`, `Write_miss`, or `Invalidate`
`block_address_parser`	`Address Parser`	The address of the required block
`tracer`	`String Builder`	Cache access tracer which records the result of the access
`victim_block`	`Cache Block`	The victim block which is replaced from the upper level cache

4.1.1 Private Cache Management

Every single level of the core's private cache is represented in the simulation by an object of the class `PrivateCacheManagement`. The class constructor is used to initialize the cache parameters. Referring to the sections 2.3 and 2.4, the number of the cache blocks is calculated by dividing the cache size over the block size. If the cache structure is direct-mapped, then the number of sets in the cache equals the number of the cache blocks, so every set holds only one cache block. For full-associative cache structure, there is only one set, and it holds all cache blocks. Finally, for the set-associative cache structure, the number of blocks per set equals the associativity, so the number of sets is calculated by

dividing the number of blocks by the associativity. The indexed list `sets` is created afterwards with a number of `CacheSet` objects equaling to the calculated number of the cache sets, each object represents a cache set and contains an ordered list of `Cacheblock` objects equaling to the calculated number of blocks.

The class implements the interface `ICacheManagement`, so it has to implement all of the functions which are defined in the interface. The function `obtain_data_block` is used to get the required cache block, the request type to the cache is here either `Read` or `Write`. As explained in section 2.6, the address of the block is passed to the function as an object of `AddressParser` class, the `AddressParser` is used to parse the block address to get the `tag` and `index`. The `index` is used to get the cache set from the `sets` list and after it the `tag` is used to retrieve the block from the cache set through the set function `fetch_block(tag)`. If the fetch function returns the block, then the request hits in the cache, the `hits_count` is incremented and a cache hit message is added to the `tracer`. If the request type is `Write` and the block state is `Shared`, then an `invalidate` coherence request is sent through the `Core` (as we will see later) to the shared cache or to the other private caches when there is no shared cache, and then the block state is changed to `Modified`, referring to table 12, section 3.4. If the fetch function returns `null`, then the request misses in the cache, the `misses_count` is incremented and a cache miss message is added to the `tracer`. In case of a cache miss, the cache sends the request to the next level private cache to obtain the block. If the cache is the last level of the private caches, a `Read_miss` or `Write_miss` coherence request is sent through the `Core` to shared cache or to the other cores when there is no shared cache to obtain the block as shown in table 13, section 3.4. The new block is allocated to the related set through the set function `allocate_block(block)`. The allocate function returns either `null` when no victim block is replaced or a victim block chosen based on the replacement policy when the set is fully occupied by memory blocks. If the victim block state is `Modified` or the hierarchy policy with next level cache is `Exclusive`, the victim block is written back to the next level cache through the function `write_back_block_to_the_cache` (see section 2.9 for cache writing policies). The victim block address is added to

`conflict_victim_blocks_addresses` list, so when the block is referenced again, the miss will be considered as a miss because of set conflict. One more thing, a request is sent through the `Core` to the shared cache (if present) to set the core block tracking bit to 0 because the victim block is not any more present in the core. Considering the cache hierarchy here as explained in section 2.12, the new block brought to the cache is forwarded directly to the upper level cache and is not allocated in the intermediate cache when the hierarchy between the two cache levels is `Exclusive`. However, in `Inclusive` and `Non_inclusive` hierarchies the block is allocated on both cache levels. If a block is evicted from `Inclusive` cache, a back invalidation request is sent from the cache to all upper level caches, but for `Non_inclusive` and `Exclusive` hierarchies no back invalidation request is sent.

The class has to implement the `serve_coherence_request` function to serve the coherence requests sent to the cache from the shared cache or from the other cores when there is no shared cache. The block address is passed to the function as a parameter. The block tag and set index are calculated by the `AddressParser` object and then the related cache set is accessed to get the required cache block for the coherence request. If the block is found in the set, the cache responds to the request by the suitable action. As explained in the table 14 section 3.4, when the coherence request is `Read_miss` and the block state is `Modified`, the block content is written back to the shared cache (if present and not `Exclusive`) or memory, the block state is then changed to `Shared`. When the coherence request is `Write_miss` and the block state is `Shared`, the state is changed to `Invalid`. But if the block state is `Modified`, the block content is written back first to the shared cache or memory, then the block state is changed to `Invalid`. For the coherence request of type `Invalidate`, if the block state is `Shared`, the state is changed to `Invalid` and the block address is added to `coherence_victim_blocks_addresses` so when the block is referenced again, the miss will be considered as a miss because of coherence. Finally, the coherence request is propagated to all levels of the private cache within the core to make sure that all levels perform the coherence request.

The class has to implement the function `write_back_block_to_cache` which performs the write back for the replaced block from the upper level cache. The victim block is passed as a parameter to the function. The state of the victim block can be `Modified`, so its content needs to be written, or it can be `Shared` and the block needs to be allocated in the cache because of the `Exclusive` cache hierarchy between the two levels. Firstly, the block is fetched from the related set using the `fetch_block(tag)` function. If the block is present, then its state is updated with the victim block state, otherwise, the victim block is allocated to the set using the set function `allocate_block(block)`, which could result in a new victim block. The new victim block can be handled in the same way as explained in the function `obtain_data_block`.

The class member function `back_invalidation_from_inclusive_cache` is used to perform the back invalidation requests from the lower levels of cache when the cache hierarchy there is `Inclusive`. The function receives as a parameter the address of the block to invalidate. The block is fetched from the related set. If the block is present and its state is `Shared` the state is changed to `Invalid`. When the block state is `Modified`, the block content is written back first, then its state is changed to `Invalid`. The block address is added to `inclusion_victim_blocks_addresses` list, so when the block is referenced again, the miss will be considered as a miss because of inclusion property. Finally, the invalidate request is propagated to the next level invalidate function to make sure that the block is invalidated in all levels of the private cache within the core.

The class member function is_the_block_located_in_the_cache is used by Core to check if a block is present in any level of the core private cache. The block address is passed as a parameter to the function, then the block is fetched from the related set, if the block is present the function returns `true`, otherwise the block is not present and the function returns `false`.

4.1.2 Shared Cache Management

The shared last level of cache is represented in the simulation by a single object of the class `SharedCacheManagement`. As explained in section 3.4, the shared last level of cache is used in the coherence scheme to minimize the number of the sent coherence messages through the bus. The cache keeps a directory entry in the shared cache for every cached block in the cores' private caches. The `DirectoryEntry` object can be stored with the `Cacheblock` object at the shared cache when the cache hierarchy is Inclusive, or it can be stored in a separate `Directory` structure when the cache hierarchy is `Non_inclusive` or `Exclusive`.

The `SharedCacheManagement` class has to implement the functions of the interface `ICacheManagement`. The class constructor does almost the same things as in the `PrivateCacheManagement` class. The class implements the `obtain_data_block` function which is used by the cores to obtain the data block when the block is not present in the private cache. If the cache hierarchy is `Non_inclusive` or `Exclusive`, the `Directory` hash table is used to retrieve the block directory entry. The block `tag` is used as a hash `Key` and the hash `Value` is the directory entry object. If the block directory entry is found in the `Directory`, then the block is cached in at least one private cache. The directory entry contains the block state and `cores_tracking_bits` array. If the state of the block is `Shared` and the request type is `Read_miss` the requesting core's bit in the `cores_tracking_bits` array is set to 1. If the request type is `Write_miss`, then the cache forwards the `Write_miss` request to cores in the `cores_tracking_bits`, after that, the state in the directory entry is changed to `Modified` and only the requesting core's bit is set to 1. If the state in the entry is `Modified` and the request type is `Read_miss`, the cache forwards the `Read_miss` request to the only core which has the block, the block state is changed to `Modified` and the requesting core's bit in the `cores_tracking_bits` array is set to 1. If the request is `Write_miss`, the cache also forwards the `Write_miss` request to the only core which has the block and only the requesting core's bit is set to 1. When the `Directory` does not have an entry for the block, the related cache set is searched for the

block. If the block is present in the cache, then it is forwarded to the requesting core, and invalidated in the shared cache if the cache hierarchy is `Exclusive`, otherwise, the block is fetched from the memory if it is not present in the cache. A new directory entry for the block is added to the `Directory` hash table with `Shared` state if the request type is `Read_miss`, or `Modified` if the request type is `Write_miss`, also the requesting core's bit in the `cores_tracking_bits` array is set to 1. For the `Inclusive` cache hierarchy, the block directory entry is stored within the block. Firstly, the block is fetched from the cache set, then the block directory entry is retrieved using the block function `get_snoop_filter()`. After getting the directory entry, the request is served in the same way as explained before with the `Exclusive` cache. If the block is not present in the cache, it is fetched from the memory, allocated in the cache, and then forwarded to the requesting core. If the block allocation results in a victim block, and the victim block state is Shared, the cache forwards an `Invalidate` request to cores in the `cores_tracking_bits` of the victim block then the victim block is written back to the memory if the `Dirty` bit is set to 1.

The class has to implement the `serve_coherence_request` function which is used by the core to send invalidate requests to the other cores' private caches. The block's directory entry can be obtained either from the `Directory` hash table or from the block depending on the cache hierarchy as explained before. The invalidate request is forwarded to cores in the `cores_tracking_bits`, after that, the state in the directory entry is changed to `Modified` and only the requesting core's bit is set to 1.

The class has to implement the `write_back_block_to_cache` function which is used to write back the evicted blocks from the private caches to the shared cache. If the cache hierarchy is `Inclusive` and the victim block state is `Modified`, the cache block `Dirty` bit is set to 1. If the cache hierarchy is `Exclusive` and no other private cache has a copy of the victim block, then the victim block is allocated in the shared cache.

4.1.3 Cache Set and Cache Replacement Policies

Every cache set is represented in the simulation by an object of the class `CacheSet`. The class implements an ordered list of `blocks` which represent the blocks held by the set. As explained in section 2.8, the order is used by the `ReplacementPolicy` to select a victim for replacement when the cache is full. Figure 25 shows the class diagram for the `CacheSet` class and the `IReplacementPolicy` interface, the figure also shows the class diagrams for all implemented replacement policies in the simulation, all replacement policy classes implement the `IReplacementPolicy` interface. The class constructor function is used to create the `blocks` list with the calculated number of blocks and the `replacement_policy` object which can be an object of any replacement policy classes. The replacement policy class updates the order of the blocks in `blocks` list based on the implemented replacement policy.

The class member function `fetch_block(tag, request)` is used by the cache management to fetch a cache block from the set. The `blocks` list is searched to look for the block. When a block with the same `tag` and with state other than `Invalid` is found in the list, a copy of that block is returned. The `replacement_policy` updates the order of the blocks list through the function `update_replacement_policy(block)` only when the `request` is `Read` or `Write`.

The class member function `allocate_block(block)` is used by the cache management to allocate a cache block to the set. The `replacement_policy` is used to get a victim block through the function `get_victim_block` to replace it with the new block. Invalid blocks are replaced first, and when the list is full a valid block is chosen for replacement based on the replacement policy. The data of the new block is filled to the victim block and a copy of the victim block is returned to the cache management.

The class member function `update_block(newBlock)` is used by the cache management to update the data of an existing block in the set with a new block data.

The class member function `is_block_present(tag)` is used to check whether a block is present in the set: if it is, `true` is returned, otherwise `false`.

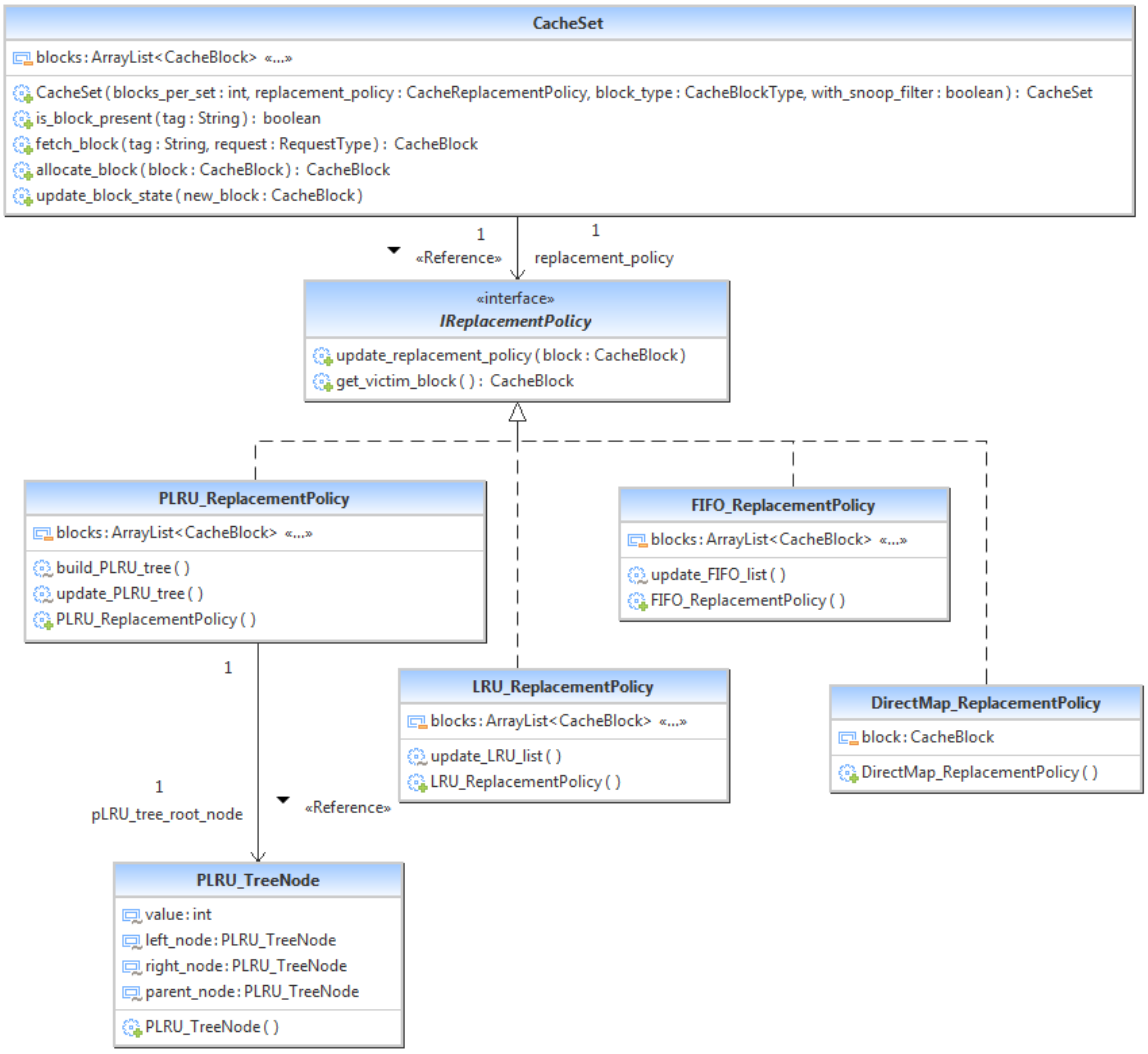

Figure 25 Cache set and replacement policies class diagram

4.1.4 Cache Blocks

Every cache block is represented in the simulation by an object of the class `CacheBlock`. Figure 26 shows the class diagram for both `CacheBlock` and `DirectoryEntry` classes. If

the block belongs to a shared inclusive cache, then the block's directory entry is stored within the block through `snoop_filter` class local variable.

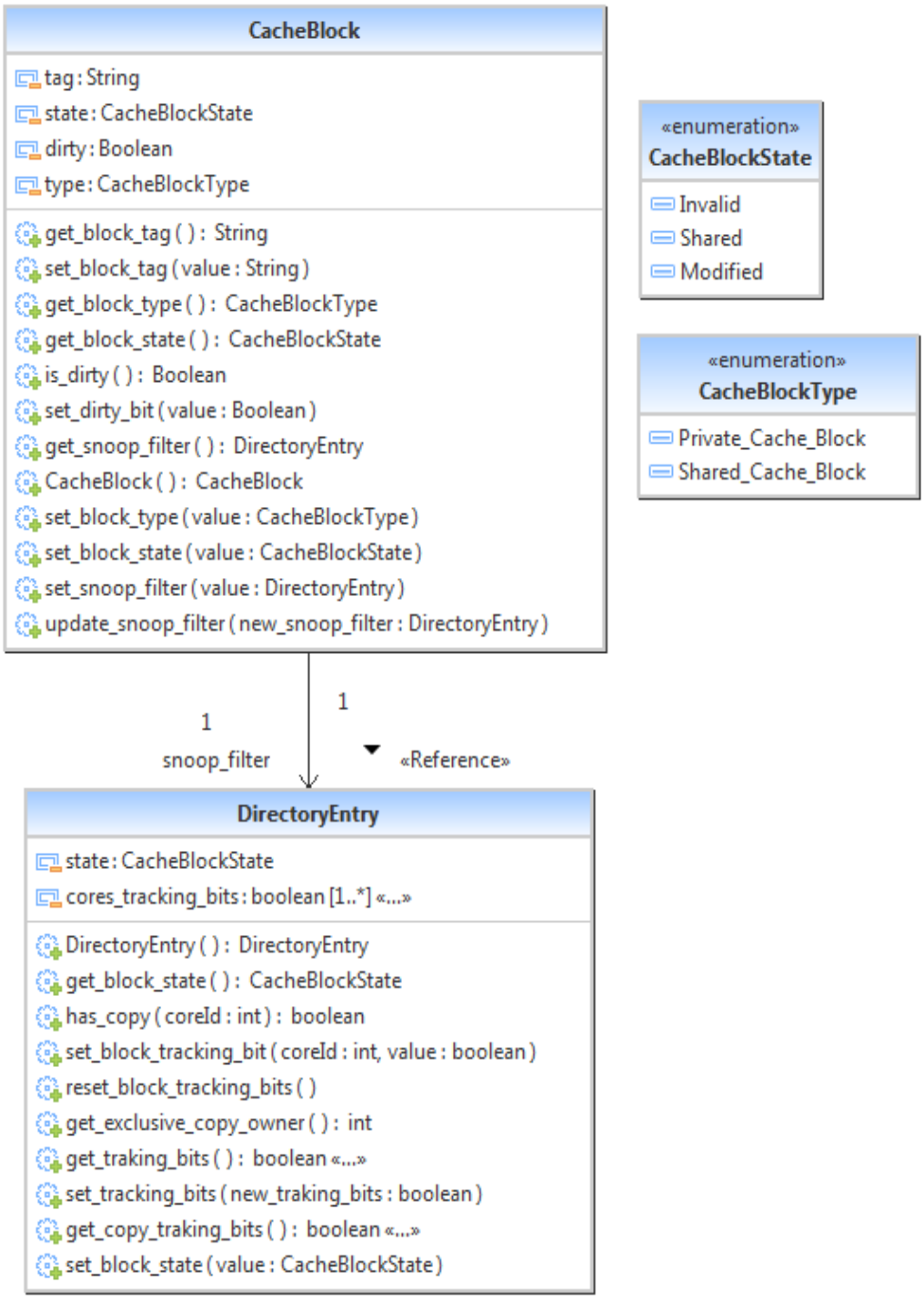

Figure 26 Cache block and directory entry class diagram

4.1.5 Address Parser

The `AddressParser` class is used to parse the block address to get the block tag and the set index. To parse a block address within a cache the function `parse(block_size, number_of_sets, structure)`. The parsing method is explained in section 2.6.

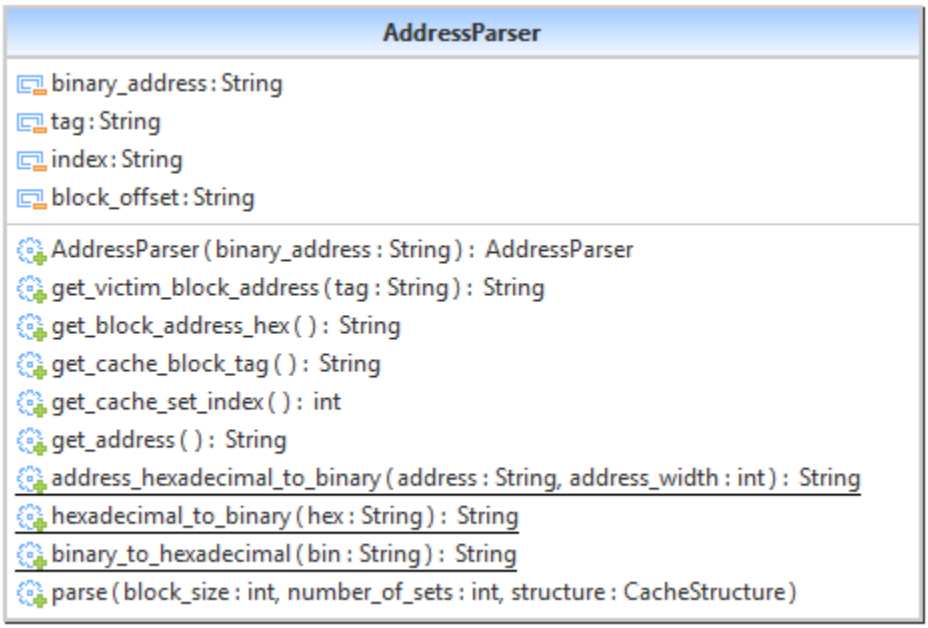

Figure 27 Address Parser class diagram

4.2 Processor Cores

The `Core` class represents the core processor in the simulation. The core has `private_cache` which points to the first level of private cache and `shared_cache` which points to the shared last level of caches (if present). The following figure 27 shows the class diagram of the Core class.

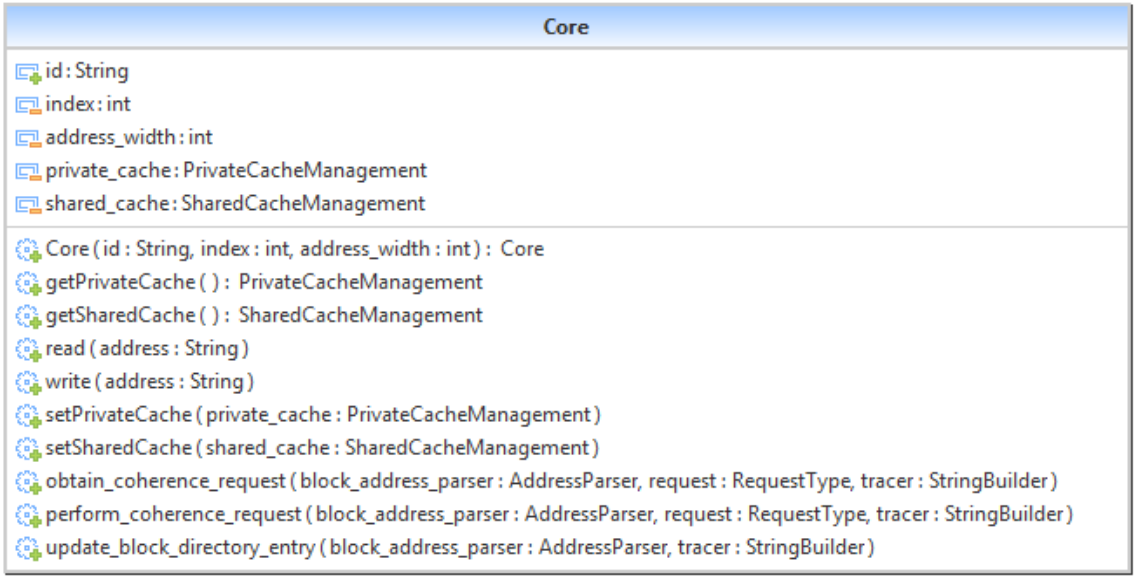

Figure 28 Core class diagram

The class member functions `read(address)/write(address)` are used by the Simulator class to perform core memory read operation. The memory address is passed to the function as a parameter. First the function checks the memory address using the address parser class. When the address is valid, the function `obtain_data_block()` of the core private cache is called to perform the memory access. The obtain function takes as parameters the core index, the block address, the request type `Read/Write`, and the memory access tracer. If the core does not have private cache, the shares cache `obtain_data_block()` function is called to obtain the block.

The class member function `obtain_coherence_request()` is used by the core private cache levels to send a coherence requests to the shared cache (if present) or to broadcast it to the other cores.

The class member function `perform_coherence_request()` is used by the other cores or the shared cache (if present) to forward the coherence requests to the core's private cache.

The member function update_block_directory_entry() is used by the core private cache levels to send coherence request to the shared cache (if present) to reset the core's tracking bit to 0, for some block evicted from the private cache.

4.3 Simulator

The `Simulator` class is used to construct the cache simulation objects and to run the simulation. Fugire 29 shows the class diagram for the simulator class.

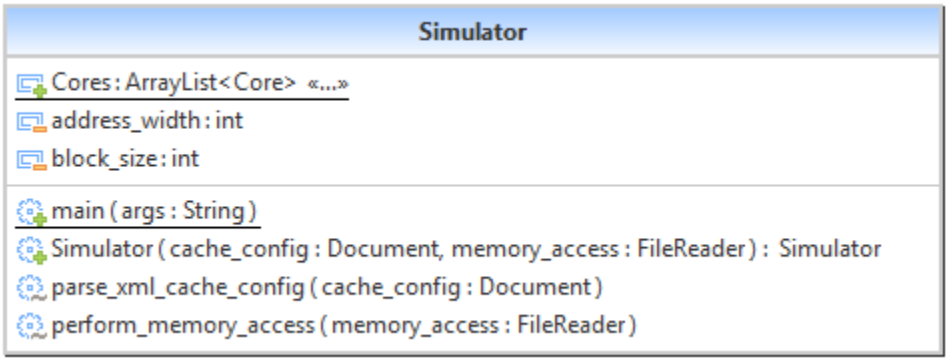

Figure 29 Simulator class diagram

The simulator uses XML format input to define the cache structure. It also applies a predefined XML-schema for structuring and validating the XML cache structure input format. The `main()` function creates a new object of the Simulator class and calls the class constructor. In the constructor the function `parse_xml_cache_config( xmlDocument)` is called first to validate the input `xmlDocument` for cache structure with the predefined cache structure XML-schema, if the input XML matches the XML-schema, the input XML is parsed to build the cache structure. The function `perform_memory_access(FileReader)` is called to read the simulator memory access

list input file, then parse it and start the simulation. The following listing 1 shows the cache structure XML-schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
elementFormDefault="qualified">

<xs:element name="simulator">
<xs:complexType>
<xs:sequence>
      <xs:element name="cores" type="coresType" minOccurs="1"
      maxOccurs="1"  />
      <xs:element name="caches" type="cachesType" minOccurs="1"
      maxOccurs="1" />
</xs:sequence>
<xs:attribute name="address_width" type="xs:integer" use="required" />
<xs:attribute name="block_size" type="xs:string" use="required" />
</xs:complexType>
</xs:element>
<xs:complexType name="coresType">
<xs:sequence>
      <xs:element name="core" type="coreType" minOccurs="1"
      maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
<xs:complexType name="coreType">
<xs:attribute name="id" type="xs:ID" use="required" />
</xs:complexType>
<xs:complexType name="cachesType">
<xs:sequence>
      <xs:element name="private_cache" type="privateCacheTypye"
      minOccurs="0" maxOccurs="1" />
      <xs:element name="shared_cache" type="sharedCacheTypye"
      minOccurs="0" maxOccurs="1" />
</xs:sequence>
</xs:complexType>
<xs:complexType name="privateCacheTypye">
<xs:sequence>
      <xs:element name="cache" type="cacheType" minOccurs="1"
      maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>
<xs:complexType name="sharedCacheTypye">
<xs:sequence>
      <xs:element name="cache" type="cacheType" minOccurs="0"
      maxOccurs="1" />
</xs:sequence>
</xs:complexType>
```

Listing 1 Cache structure XML-schema

```
<xs:complexType name="cacheType">
<xs:attribute name="level" type="xs:integer" use="required" />
<xs:attribute name="cache_size" type="xs:string" use="required" />
      <xs:attribute name="cache_structure" type="cacheStructureType"
      use="required" />
      <xs:attribute name="associativity" type="xs:integer"
      use="optional"/>
      <xs:attribute name="replacement_policy"
      type="replacementPolicyType" use="optional" />
      <xs:attribute name="cache_hierarchy" type="hierarchyType"
      use="required" />
</xs:complexType>
<xs:simpleType name="cacheStructureType" final="restriction">
    <xs:restriction base="xs:string">
     <xs:enumeration value="DM" />
     <xs:enumeration value="FA" />
     <xs:enumeration value="SA" />
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="replacementPolicyType" final="restriction">
    <xs:restriction base="xs:string">
     <xs:enumeration value="FIFO" />
     <xs:enumeration value="LRU" />
     <xs:enumeration value="PLRU" />
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="hierarchyType" final="restriction">
    <xs:restriction base="xs:string">
      <xs:enumeration value="inclusive" />
      <xs:enumeration value="non-inclusive" />
      <xs:enumeration value="exclusive" />
      <xs:enumeration value="none" />
    </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Listing 1 Cache structure XML-schema (continued)

4.3.1 Simulator Input

As we mentioned before, the simulator has two input files. The first one is the cache structure in XML format. The second one is the simulator memory access list, every line in the list has a memory operation in a predefined format. Listing 2 shows an example of

cache structure in XML while listing 3 shows an example of memory access list. Notice that the `core_Ids` in the cache structure XML is the same as in the memory access list.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator address_width="16" block_size="4 B">

<cores>
      <core id="c0" />
      <core id="c1" />
</cores>

<caches>

<private_cache>
      <cache level="1" cache_size="32 B" cache_structure="SA"
            associativity="2" replacement_policy="LRU"
            cache_hierarchy="none" />
</private_cache>
<shared_cache>
      <cache level="2" cache_size="64 B" cache_structure="SA"
            associativity="4" replacement_policy="LRU"
            cache_hierarchy="inclusive" />
</shared_cache>

</caches>
</simulator>
```

Listing 2 Cache structure XML format

```
C0 Read [0x1210]

C1 Read [0x1213]

C0 Read [0x2352]

C1 Write [0x1213]

C0 Read [0x1210]
```

Listing 3 Memory access list

4.3.2 Simulator Input Validation

The following java code in listing 4 is used to match the input cache structure in XML with the XML-schema. If the input XML is not matched with the schema, the code throws an exception.

```
try {
// define XML schema object to validate the cache simulator XML input
configuration
SchemaFactory schemaFactory =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);

// the name of schema file is cache_system_schema.xsd
Schema schema = schemaFactory.newSchema(
      new File("simulator_xml_schema.xsd"));
DocumentBuilderFactory parserFactory =
      DocumentBuilderFactory.newInstance();
parserFactory.setSchema(schema);
parserFactory.setIgnoringElementContentWhitespace(true);
parserFactory.setIgnoringComments(true);
DocumentBuilder documentBuilder = parserFactory.newDocumentBuilder();

// the name of the XML input configuration file
Document document = documentBuilder.parse(
      new File("cache_config.xml"));
}
catch (Exception e) {
      System.out.println(e.getMessage());
}
```

Listing 4 Java code for cache structure XML validation with the XML-schema

When the XML input of the cache structure is valid, the XML is parsed to get the information about the simulated cache structure. The XML includes the list of cores and cache levels (private/shared) with the cache parameters of each level like the level number, the cache size, the cache structure (DM: direct-mapped, FA: fully-associative, or SA: set-associative), the associativity, the replacement policy (FIFO, LRU, or PLRU), and the cache hierarchy policy (inclusive, non-inclusive, exclusive, or none for the first level of cache).

After parsing the XML, the simulation objects (cores and caches) are created to start the simulation. Figure 30 shows the simulation object diagram for the cache structure XML shown in listing 2.

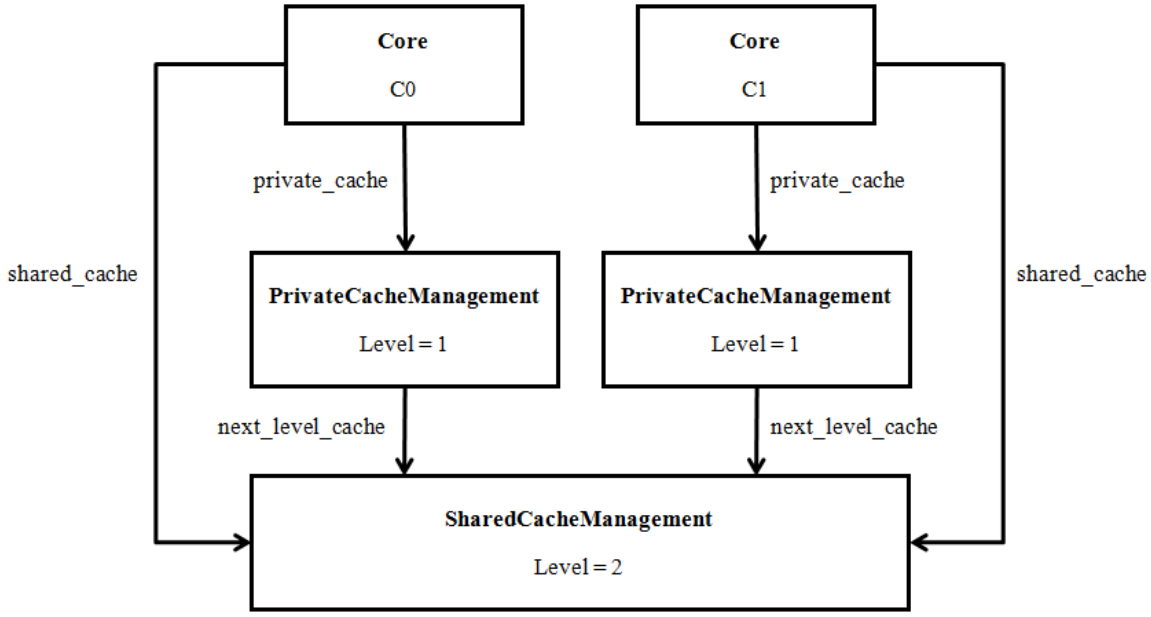

Figure 30 Simulation object diagram

4.3.3 Simulator Output

The simulation output is a list of traces for every memory access operation. Each trace contains the operation itself and the result of every cache level access (Hit/Miss). The trace includes also the block state in the cache when it is a hit, and the block allocation, and victim block replacement when the block is a miss. The sent coherence messages are also included. Listing 5 shows the simulation output for the cache structure and memory access list shown respectively in listing 2 and listing 3. The simulation also provides counts of hits and misses in each cache level.

```
core C0 Read [0x1210]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, allocate block, block address: 0x0484, new block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x0484, block state: Shared
*****************************
core C1 Read [0x1213]:
C1 L1 private cache Miss
L2 shared cache Hit, block address: 0x0484, block state: Shared , core tracking
bits: 10
L2 shared cache, block address: 0x0484, new block state: Shared , core tracking
bits: 11
C1 L1 private cache, allocate block, block address: 0x0484, block state: Shared
*****************************
core C0 Read [0x2352]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, allocate block, block address: 0x08D4, new block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x08D4, block state: Shared
*****************************
core C1 Write [0x1213]:
C1 L1 private cache data request Hit, block address: 0x0484, block state:
Shared
C1 send block invalidate request to shared cache, block address: 0x0484
L2 shared cache, block address: 0x0484, block state: Shared , core tracking
bits: 11
L2 shared cache, forward invalidate request to core C0, block address: 0x0484
C0 L1 private cache, invalidate block, block address: 0x0484, block state:
Shared
C0 L1 private cache, block address: 0x0484, block new state: Invalid
L2 shared cache, block address: 0x0484, new block state: Modified , core
tracking bits: 01
C1 L1 private cache, block address: 0x0484, block new state: Modified
*****************************
core C0 Read [0x1210]:
C0 L1 private cache Miss
L2 shared cache Hit, block address: 0x0484, block state: Modified , core
tracking bits: 01
L2 shared cache, forward Read miss request to core C1, block address: 0x0484
C1 L1 private cache, write back block, block address: 0x0484, block state:
Modified
C1 L1 private cache, block address: 0x0484, block new state: Shared
L2 shared cache, block address: 0x0484, new block state: Shared , core tracking
bits: 11
C0 L1 private cache, allocate block, block address: 0x0484, block state: Shared
*****************************
```

Listing 5 Simulation output

5 Tests and Simulation Results

5.1 Test Cache Configurations

For testing the cache configuration, we will use the example which is discussed in section 2.7. The used cache size in all cases is 64 byte, the block size is 16 byte, and the address width is 16 bits. The following memory access list in listing 6 is used in the simulation.

```
C0 Read [0x1234]
C0 Read [0x123C]
C0 Read [0x1240]
C0 Read [0x1270]
C0 Read [0x1234]
C0 Read [0x1232]
C0 Read [0x1248]
C0 Read [0x12C8]
C0 Read [0x1248]
C0 Read [0x1244]
```

Listing 6 Memory access list for testing cache configurations

5.1.1 Direct-Mapped Cache

Listing 7 shows the direct-mapped cache structure and listing 8 shows the simulation output.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator address_width="16" block_size="16 B">

<cores>
<core id="c0" />
</cores>

<caches>
<private_cache>
<cache level="1" cache_size="64 B" cache_structure="DM"
       cache_hierarchy="none" />
</private_cache>
</caches>

</simulator>
```

Listing 7 Direct-Mapped cache XML

```
core C0 Read [0x1234]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x123, block state: Shared
****************************
core C0 Read [0x123C]:
C0 L1 private cache data request Hit, block address: 0x123, block state: Shared
C0 L1 private cache, block address: 0x123, block new state: Shared
****************************
core C0 Read [0x1240]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x124, block state: Shared
****************************
core C0 Read [0x1270]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x127, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x123, block
state: Shared
****************************
core C0 Read [0x1234]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x123, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x127, block
state: Shared
****************************
core C0 Read [0x1232]:
C0 L1 private cache data request Hit, block address: 0x123, block state: Shared
C0 L1 private cache, block address: 0x123, block new state: Shared
****************************
core C0 Read [0x1248]:
C0 L1 private cache data request Hit, block address: 0x124, block state: Shared
C0 L1 private cache, block address: 0x124, block new state: Shared
****************************
core C0 Read [0x12C8]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x12C, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x124, block
state: Shared
****************************
core C0 Read [0x1248]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x124, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x12C, block
state: Shared
****************************
core C0 Read [0x1244]:
C0 L1 private cache data request Hit, block address: 0x124, block state: Shared
C0 L1 private cache, block address: 0x124, block new state: Shared
****************************
```

Listing 8 Direct-Mapped cache simulation output

The simulation results are the same as in the table 2 on page 27.

5.1.2 Fully-Associative Cache

Listing 9 shows the fully-associative cache structure and listing 10 shows the simulation output.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator address_width="16" block_size="16 B">

<cores>
<core id="c0" />
</cores>

<caches>
<private_cache>
<cache level="1" cache_size="64 B" cache_structure="FA"
       replacement_policy="LRU" cache_hierarchy="none" />
</private_cache>
</caches>

</simulator>
```

Listing 9 Fully-Associative cache XML

```
core C0 Read [0x1234]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x123, block state: Shared
****************************
core C0 Read [0x123C]:
C0 L1 private cache data request Hit, block address: 0x123, block state: Shared
C0 L1 private cache, block address: 0x123, block new state: Shared
****************************
core C0 Read [0x1240]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x124, block state: Shared
****************************
core C0 Read [0x1270]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x127, block state: Shared
****************************
core C0 Read [0x1234]:
C0 L1 private cache data request Hit, block address: 0x123, block state: Shared
C0 L1 private cache, block address: 0x123, block new state: Shared
****************************
core C0 Read [0x1232]:
C0 L1 private cache data request Hit, block address: 0x123, block state: Shared
C0 L1 private cache, block address: 0x123, block new state: Shared
****************************
core C0 Read [0x1248]:
C0 L1 private cache data request Hit, block address: 0x124, block state: Shared
C0 L1 private cache, block address: 0x124, block new state: Shared
****************************
```

Listing 10 Fully-Associative cache simulation output

```
core C0 Read [0x12C8]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x12C, block state: Shared
****************************
core C0 Read [0x1248]:
C0 L1 private cache data request Hit, block address: 0x124, block state: Shared
C0 L1 private cache, block address: 0x124, block new state: Shared
****************************
core C0 Read [0x1244]:
C0 L1 private cache data request Hit, block address: 0x124, block state: Shared
C0 L1 private cache, block address: 0x124, block new state: Shared
****************************
```

Listing 10 Fully-Associative cache simulation output (continued)

The simulation results are the same as in the table 6 on page 28.

5.1.3 Two-way Set-Associative Cache

Listing 11 shows the two-way set-associative cache structure and listing 12 shows the simulation output.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator address_width="16" block_size="16 B">

<cores>
<core id="c0" />
</cores>

<caches>
<private_cache>
<cache level="1" cache_size="64 B" cache_structure="SA"
associativity="2" replacement_policy="LRU" cache_hierarchy="none" />
</private_cache>
</caches>

</simulator>
```

Listing 11 Two-way set-associative cache XML

```
core C0 Read [0x1234]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x123, block state: Shared
****************************
core C0 Read [0x123C]:
C0 L1 private cache data request Hit, block address: 0x123, block state: Shared
C0 L1 private cache, block address: 0x123, block new state: Shared
****************************
```

Listing 12 Two-way set-associative cache simulation output

```
core C0 Read [0x1240]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x124, block state: Shared
****************************
core C0 Read [0x1270]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x127, block state: Shared
****************************
core C0 Read [0x1234]:
C0 L1 private cache data request Hit, block address: 0x123, block state: Shared
C0 L1 private cache, block address: 0x123, block new state: Shared
****************************
core C0 Read [0x1232]:
C0 L1 private cache data request Hit, block address: 0x123, block state: Shared
C0 L1 private cache, block address: 0x123, block new state: Shared
****************************
core C0 Read [0x1248]:
C0 L1 private cache data request Hit, block address: 0x124, block state: Shared
C0 L1 private cache, block address: 0x124, block new state: Shared
****************************
core C0 Read [0x12C8]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x12C, block state: Shared
****************************
core C0 Read [0x1248]:
C0 L1 private cache data request Hit, block address: 0x124, block state: Shared
C0 L1 private cache, block address: 0x124, block new state: Shared
****************************
core C0 Read [0x1244]:
C0 L1 private cache data request Hit, block address: 0x124, block state: Shared
C0 L1 private cache, block address: 0x124, block new state: Shared
****************************
```

Listing 12 Two-way set-associative cache simulation output (continued)

The simulation results are the same as in the table 4 on page 28.

5.2 Test Cache Replacement Policies

In replacement policy tests we will consider the examples which are discussed in section 2.8. The following memory access list in listing 13 is used in the simulation.

```
C0 Read [0x0A00]
C0 Read [0x0B00]
C0 Read [0x0C00]
C0 Read [0x0D00]
C0 Read [0x0E00]
C0 Read [0x0B00]
C0 Read [0x0A00]
C0 Read [0x0F00]
```

Listing 13 Memory access list for testing replacement policies

5.2.1 LRU Replacement Policy

Listing 14 shows the four-way set-associative cache structure with LRU replacement policy and listing 15 shows the simulation output.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator address_width="16" block_size="16 B">

<cores>
<core id="c0" />
</cores>

<caches>
<private_cache>
<cache level="1" cache_size="1 KB" cache_structure="SA"
       associativity="4" replacement_policy="LRU" cache_hierarchy="none" />
</private_cache>
</caches>

</simulator>
```

Listing 14 Cache structure XML with LRU replacement policy

```
core C0 Read [0xA00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0A0, block state: Shared
*****************************
core C0 Read [0xB00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0B0, block state: Shared
*****************************
core C0 Read [0xC00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0C0, block state: Shared
*****************************
core C0 Read [0xD00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0D0, block state: Shared
*****************************
core C0 Read [0xE00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0E0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0A0, block
state: Shared
*****************************
core C0 Read [0xB00]:
C0 L1 private cache data request Hit, block address: 0x0B0, block state: Shared
C0 L1 private cache, block address: 0x0B0, block new state: Shared
*****************************
```

Listing 15 LRU cache simulation output

```
core C0 Read [0xA00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0A0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0C0, block
state: Shared
****************************
core C0 Read [0xF00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0F0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0D0, block
state: Shared
****************************
```

Listing 15 LRU cache simulation output (continued)

The simulation results are the same as in the table 8 on page 31.

5.2.2 FIFO Replacement Policy

The same cache structure is used to simulate FIFO replacement policy. Listing 16 shows the simulation output.

```
core C0 Read [0xA00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0A0, block state: Shared
****************************
core C0 Read [0xB00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0B0, block state: Shared
****************************
core C0 Read [0xC00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0C0, block state: Shared
****************************
core C0 Read [0xD00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0D0, block state: Shared
****************************
core C0 Read [0xE00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0E0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0A0, block
state: Shared
****************************
core C0 Read [0xB00]:
C0 L1 private cache data request Hit, block address: 0x0B0, block state: Shared
C0 L1 private cache, block address: 0x0B0, block new state: Shared
****************************
```

Listing 16 FIFO cache simulation output

```
core C0 Read [0xA00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0A0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0B0, block
state: Shared
****************************
core C0 Read [0xF00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0F0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0C0, block
state: Shared
****************************
```

Listing 16 FIFO cache simulation output (continued)

The simulation results are the same as in the table 9 on page 32.

5.2.3 PLRU Replacement Policy

The same cache structure is also used to simulate PLRU replacement policy. Listing 17 shows the simulation output.

```
core C0 Read [0xA00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0A0, block state: Shared
****************************
core C0 Read [0xB00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0B0, block state: Shared
****************************
core C0 Read [0xC00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0C0, block state: Shared
****************************
core C0 Read [0xD00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0D0, block state: Shared
****************************
core C0 Read [0xE00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0E0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0A0, block
state: Shared
****************************
core C0 Read [0xB00]:
C0 L1 private cache data request Hit, block address: 0x0B0, block state: Shared
C0 L1 private cache, block address: 0x0B0, block new state: Shared
****************************
```

Listing 17 PLRU cache simulation output

```
core C0 Read [0xA00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0A0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0C0, block
state: Shared
*****************************
core C0 Read [0xF00]:
C0 L1 private cache Miss
C0 L1 private cache, allocate block, block address: 0x0F0, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0E0, block
state: Shared
*****************************
```

Listing 17 PLRU cache simulation output (continued)

The simulation results are the same as in the Figure 9 on page 35.

5.3 Test Cache Structure on Multi-Core Processors

A multi-core processor with two cores is used to test the cache hierarchy at the shared last level cache. The following listing 18 shows the cache structure of the multi-core processor.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator address_width="16" block_size="4 B">

<cores>
	<core id="c0" />
	<core id="c1" />
</cores>

<caches>

<private_cache>
	<cache level="1" cache_size="32 B" cache_structure="SA"
		associativity="2" replacement_policy="LRU"
		cache_hierarchy="none" />
</private_cache>
<shared_cache>
	<cache level="2" cache_size="64 B" cache_structure="SA"
		associativity="4" replacement_policy="LRU"
		cache_hierarchy="inclusive" />
</shared_cache>

</caches>
</simulator>
```

Listing 18 Cache structure XML for multi-core processor with two cores

The following memory access list in listing 19 is used in the multi-core cache simulation.

```
C0 Read [0x1210]
C1 Read [0x1213]
C0 Read [0x2352]
C1 Write[0x1213]
C0 Read [0x1210]
C0 Read [0x2213]
C1 Read [0x2351]
C0 Read [0x2522]
C1 Read [0x3452]
C1 Read [0x1210]
```

Listing 19 Memory access list for testing cache hierarchy

5.3.1 Inclusive Cache Hierarchy at the Shared Last Level of Cache

Using the memory access list defined in listing 19 for the simulation, listing 20 shows the simulation result of a system containing a shared inclusive cache.

```
core C0 Read [0x1210]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, allocate block, block address: 0x0484, new block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x0484, block state: Shared
****************************
core C1 Read [0x1213]:
C1 L1 private cache Miss
L2 shared cache Hit, block address: 0x0484, block state: Shared , core tracking
bits: 10
L2 shared cache, block address: 0x0484, new block state: Shared , core tracking
bits: 11
C1 L1 private cache, allocate block, block address: 0x0484, block state: Shared
****************************
core C0 Read [0x2352]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, allocate block, block address: 0x08D4, new block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x08D4, block state: Shared
****************************
core C1 Write [0x1213]:
C1 L1 private cache data request Hit, block address: 0x0484, block state:
Shared
C1 send block invalidate request to shared cache, block address: 0x0484
L2 shared cache, block address: 0x0484, block state: Shared , core tracking
bits: 11
```

Listing 20 Inclusive cache hierarchy simulation output

```
L2 shared cache, forward invalidate request to core C0, block address: 0x0484
C0 L1 private cache, invalidate block, block address: 0x0484, block state:
Shared
C0 L1 private cache, block address: 0x0484, block new state: Invalid
L2 shared cache, block address: 0x0484, new block state: Modified , core
tracking bits: 01
C1 L1 private cache, block address: 0x0484, block new state: Modified
*****************************
core C0 Read [0x1210]:
C0 L1 private cache Miss
L2 shared cache Hit, block address: 0x0484, block state: Modified , core
tracking bits: 01
L2 shared cache, forward Read miss request to core C1, block address: 0x0484
C1 L1 private cache, write back block, block address: 0x0484, block state:
Modified
C1 L1 private cache, block address: 0x0484, block new state: Shared
L2 shared cache, block address: 0x0484, new block state: Shared , core tracking
bits: 11
C0 L1 private cache, allocate block, block address: 0x0484, block state: Shared
*****************************
core C0 Read [0x2213]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, allocate block, block address: 0x0884, new block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x0884, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x08D4, block
state: Shared
L2 shared cache, update core tracking bits, block address: 0x08D4, new block
state: Shared , core tracking bits: 00
*****************************
core C1 Read [0x2351]:
C1 L1 private cache Miss
L2 shared cache Hit, block address: 0x08D4, block state: Shared , core tracking
bits: 00
L2 shared cache, block address: 0x08D4, new block state: Shared , core tracking
bits: 01
C1 L1 private cache, allocate block, block address: 0x08D4, block state: Shared
*****************************
core C0 Read [0x2522]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, allocate block, block address: 0x0948, new block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x0948, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0484, block
state: Shared
L2 shared cache, update core tracking bits, block address: 0x0484, new block
state: Shared , core tracking bits: 01
*****************************
```

Listing 20 Inclusive cache hierarchy simulation output (continued)

```
core C1 Read [0x3452]:
C1 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, allocate block, block address: 0x0D14, new block state: Shared
, core tracking bits: 01
L2 shared cache, victim block for replacement, block address: 0x0484, is dirty:
true, block state: Shared , core tracking bits: 01
C1 L1 private cache, back invalidate block (inclusion), block address: 0x0484,
block state: Shared
L2 shared cache, write back block to memory, block address: 0x0484
C1 L1 private cache, allocate block, block address: 0x0D14, block state: Shared
****************************
core C1 Read [0x1210]:
C1 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, allocate block, block address: 0x0484, new block state: Shared
, core tracking bits: 01
L2 shared cache, victim block for replacement, block address: 0x0884, is dirty:
false, block state: Shared , core tracking bits: 10
C1 L1 private cache, back invalidate block (inclusion), block address: 0x0884,
block state: Shared
C1 L1 private cache, allocate block, block address: 0x0484, block state: Shared
C1 L1 private cache, victim block for replacement, block address: 0x08D4, block
state: Shared
L2 shared cache, update core tracking bits, block address: 0x08D4, new block
state: Shared , core tracking bits: 00
****************************
C0 L1 private cache
cache hits: 0
cache misses: 5
cache conflict misses: 0
cache coherence misses: 1
cache inclusion misses: 0
****************************
C1 L1 private cache
cache hits: 1
cache misses: 4
cache conflict misses: 0
cache coherence misses: 0
cache inclusion misses: 1
****************************
L2 shared cache
cache hits: 3
cache misses: 6
cache conflict misses: 1
****************************
```

Listing 20 Inclusive cache hierarchy simulation output (continued)

The simulation results show each core was first sending the coherence requests to the shared cache, then the shared cache forwarded them to the core. The core C_0 had one coherence miss happening at the operation `C0 Read [0x1210]`, because the block was invalidated by C_1 after the operation `C1 Write [0x1213]`. The core C_1 had one inclusion miss resulting from the operation `C1 Read [0x1210]`, because the block was invalidated

by the shared cache after it was evicted from the shared cache (inclusion property) at the operation `C1 Write [0x1213]`.

5.3.2 Exclusive Cache Hierarchy at the Shared Last Level of Cache

Using the memory access list defined in listing 19 for the simulation, listing 21 shows the simulation result of a system containing a shared exclusive cache.

```
core C0 Read [0x1210]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, block address: 0x0484, new directory entry block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x0484, block state: Shared
****************************
core C1 Read [0x1213]:
C1 L1 private cache Miss
L2 shared cache, block address: 0x0484, directory entry block state: Shared ,
core tracking bits: 10
L2 shared cache, block address: 0x0484, new directory entry block state: Shared
, core tracking bits: 11
C1 L1 private cache, allocate block, block address: 0x0484, block state: Shared
****************************
core C0 Read [0x2352]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, block address: 0x08D4, new directory entry block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x08D4, block state: Shared
****************************
core C1 Write [0x1213]:
C1 L1 private cache data request Hit, block address: 0x0484, block state:
Shared
C1 send block invalidate request to shared cache, block address: 0x0484
L2 shared cache, block address: 0x0484, directory entry block state: Shared ,
core tracking bits: 11
L2 shared cache, forward invalidate request to core C0, block address: 0x0484
C0 L1 private cache, invalidate block, block address: 0x0484, block state:
Shared
C0 L1 private cache, block address: 0x0484, block new state: Invalid
L2 shared cache, block address: 0x0484, directory entry new block state:
Modified , core tracking bits: 01
C1 L1 private cache, block address: 0x0484, block new state: Modified
****************************
core C0 Read [0x1210]:
C0 L1 private cache Miss
L2 shared cache, block address: 0x0484, directory entry block state: Modified ,
core tracking bits: 01
L2 shared cache, forward Read miss request to core C1, block address: 0x0484
C1 L1 private cache, write back block, block address: 0x0484, block state:
Modified
C1 L1 private cache, block address: 0x0484, block new state: Shared
L2 shared cache, block address: 0x0484, new directory entry block state: Shared
, core tracking bits: 11
C0 L1 private cache, allocate block, block address: 0x0484, block state: Shared
****************************
```

Listing 21 Exclusive cache hierarchy simulation output

```
core C0 Read [0x2213]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, block address: 0x0884, new directory entry block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x0884, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x08D4, block
state: Shared
L2 shared cache, directory entry update core tracking bits, block address:
0x08D4, new block state: Shared , core tracking bits: 00
L2 shared cache, remove directory entry , block address: 0x08D4
L2 shared cache, allocate block (exclusion), block address: 0x08D4
*****************************
core C1 Read [0x2351]:
C1 L1 private cache Miss
L2 shared cache Hit, block address: 0x08D4
L2 shared cache, invalidate block (exclusion), block address: 0x08D4
L2 shared cache, block address: 0x08D4, new directory entry block state: Shared
, core tracking bits: 01
C1 L1 private cache, allocate block, block address: 0x08D4, block state: Shared
*****************************
core C0 Read [0x2522]:
C0 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, block address: 0x0948, new directory entry block state: Shared
, core tracking bits: 10
C0 L1 private cache, allocate block, block address: 0x0948, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0484, block
state: Shared
L2 shared cache, directory entry update core tracking bits, block address:
0x0484, new block state: Shared , core tracking bits: 01
*****************************
core C1 Read [0x3452]:
C1 L1 private cache Miss
L2 shared cache Miss
L2 shared cache, block address: 0x0D14, new directory entry block state: Shared
, core tracking bits: 01
C1 L1 private cache, allocate block, block address: 0x0D14, block state: Shared
C1 L1 private cache, victim block for replacement, block address: 0x0484, block
state: Shared
L2 shared cache, directory entry update core tracking bits, block address:
0x0484, new block state: Shared , core tracking bits: 00
L2 shared cache, remove directory entry , block address: 0x0484
L2 shared cache, allocate block (exclusion), block address: 0x0484
*****************************
core C1 Read [0x1210]:
C1 L1 private cache Miss
L2 shared cache Hit, block address: 0x0484
L2 shared cache, invalidate block (exclusion), block address: 0x0484
L2 shared cache, block address: 0x0484, new directory entry block state: Shared
, core tracking bits: 01
C1 L1 private cache, allocate block, block address: 0x0484, block state: Shared
C1 L1 private cache, victim block for replacement, block address: 0x08D4, block
state: Shared
```

Listing 21 Exclusive cache hierarchy simulation output (continued)

```
L2 shared cache, directory entry update core tracking bits, block address:
0x08D4, new block state: Shared , core tracking bits: 00
L2 shared cache, remove directory entry , block address: 0x08D4
L2 shared cache, allocate block (exclusion), block address: 0x08D4
****************************
C0 L1 private cache
cache hits: 0
cache misses: 5
cache conflict misses: 0
cache coherence misses: 1
****************************
C1 L1 private cache
cache hits: 1
cache misses: 4
cache conflict misses: 1
cache coherence misses: 0
****************************
L2 shared cache
cache hits: 2
cache misses: 5
cache conflict misses: 0
****************************
```

Listing 21 Exclusive cache hierarchy simulation output (continued)

The simulation results show that each core was first sending the coherence requests to the directory within the shared cache, and then the directory forwarded them to the core. When the block (address: 0x08D4) was evicted from the private cache, it was allocated in the shared cache (shared exclusive cache is considered as a victim cache to the private caches). Because of the exclusion property exclusive cache has better hit rate than the inclusive cache.

5.3.3 Multi-Core Processor without Shared Last Level Cache

In this kind of architecture the private caches are connected together and to the memory through a bus. Listing 22 shows the cache structure and listing 23 shows the simulation result of a system containing no shared cache.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator address_width="16" block_size="4 B">

<cores>
	<core id="c0" />
	<core id="c1" />
</cores>

<caches>

<private_cache>
	<cache level="1" cache_size="32 B" cache_structure="SA"
		associativity="2" replacement_policy="LRU"
		cache_hierarchy="none" />
</private_cache>

</caches>
</simulator>
```

Listing 22 Cache structure XML for multi-core processor with no shared cache

```
core C0 Read [0x1210]:
C0 L1 private cache Miss
C0 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x0484
C0 L1 private cache, allocate block, block address: 0x0484, block state: Shared
****************************
core C1 Read [0x1213]:
C1 L1 private cache Miss
C1 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x0484
C0 L1 private cache, block address: 0x0484, block state: Shared
C0 L1 private cache, block address: 0x0484, block new state: Shared
C1 L1 private cache, allocate block, block address: 0x0484, block state: Shared
****************************
core C0 Read [0x2352]:
C0 L1 private cache Miss
C0 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x08D4
C0 L1 private cache, allocate block, block address: 0x08D4, block state: Shared
****************************
core C1 Write [0x1213]:
C1 L1 private cache data request Hit, block address: 0x0484, block state:
Shared
C1 send broadcast coherence request to other cores private caches, request
type: Invalidate, block address: 0x0484
C0 L1 private cache, invalidate block, block address: 0x0484, block state:
Shared
C0 L1 private cache, block address: 0x0484, block new state: Invalid
C1 L1 private cache, block address: 0x0484, block new state: Modified
****************************
```

Listing 23 No shared cache simulation output

```
core C0 Read [0x1210]:
C0 L1 private cache Miss
C0 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x0484
C1 L1 private cache, write back block, block address: 0x0484, block state:
Modified
C1 L1 private cache, block address: 0x0484, block new state: Shared
C0 L1 private cache, allocate block, block address: 0x0484, block state: Shared
*****************************
core C0 Read [0x2213]:
C0 L1 private cache Miss
C0 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x0884
C0 L1 private cache, allocate block, block address: 0x0884, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x08D4, block
state: Shared
*****************************
core C1 Read [0x2351]:
C1 L1 private cache Miss
C1 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x08D4
C1 L1 private cache, allocate block, block address: 0x08D4, block state: Shared
*****************************
core C0 Read [0x2522]:
C0 L1 private cache Miss
C0 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x0948
C0 L1 private cache, allocate block, block address: 0x0948, block state: Shared
C0 L1 private cache, victim block for replacement, block address: 0x0484, block
state: Shared
*****************************
core C1 Read [0x3452]:
C1 L1 private cache Miss
C1 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x0D14
C1 L1 private cache, allocate block, block address: 0x0D14, block state: Shared
C1 L1 private cache, victim block for replacement, block address: 0x0484, block
state: Shared
*****************************
core C1 Read [0x1210]:
C1 L1 private cache Miss
C1 send broadcast coherence request to other cores private caches, request
type: Read_miss, block address: 0x0484
C1 L1 private cache, allocate block, block address: 0x0484, block state: Shared
C1 L1 private cache, victim block for replacement, block address: 0x08D4, block
state: Shared
*****************************
```

Listing 23 No shared cache simulation output (continued)

```
C0 L1 private cache
cache hits: 0
cache misses: 5
cache conflict misses: 0
cache coherence misses: 1
****************************
C1 L1 private cache
cache hits: 1
cache misses: 4
cache conflict misses: 1
cache coherence misses: 0
****************************
```

Listing 23 No shared cache simulation output (continued)

The simulation results show that each core was sending the coherence requests as broadcast messages through the bus to the other core.

5.3.4 Multi-Core Processor with Only Shared Cache and No Private Caches

Listing 24 shows the cache structure and listing 25 shows the simulation result of the stated case.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator address_width="16" block_size="4 B">

<cores>
	<core id="c0" />
	<core id="c1" />
</cores>

<caches>

<shared_cache>
	<cache level="1" cache_size="64 B" cache_structure="SA"
		associativity="4" replacement_policy="LRU"
		cache_hierarchy="inclusive" />
</shared_cache>

</caches>
</simulator>
```

Listing 24 Cache structure XML for multi-core processor with no private cache

```
core C0 Read [0x1210]:
L1 shared cache Miss
L1 shared cache, allocate block, block address: 0x0484
****************************
core C1 Read [0x1213]:
L1 shared cache Hit, block address: 0x0484
****************************
core C0 Read [0x2352]:
L1 shared cache Miss
L1 shared cache, allocate block, block address: 0x08D4
****************************
core C1 Write [0x1213]:
L1 shared cache Hit, block address: 0x0484
L1 shared cache, block address: 0x0484, set dirty: true
****************************
core C0 Read [0x1210]:
L1 shared cache Hit, block address: 0x0484
****************************
core C0 Read [0x2213]:
L1 shared cache Miss
L1 shared cache, allocate block, block address: 0x0884
****************************
core C1 Read [0x2351]:
L1 shared cache Hit, block address: 0x08D4
****************************
core C0 Read [0x2522]:
L1 shared cache Miss
L1 shared cache, allocate block, block address: 0x0948
****************************
core C1 Read [0x3452]:
L1 shared cache Miss
L1 shared cache, allocate block, block address: 0x0D14
L1 shared cache, victim block for replacement, block address: 0x0484, is dirty:
true
L1 shared cache, write back block to memory, block address: 0x0484
****************************
core C1 Read [0x1210]:
L1 shared cache Miss
L1 shared cache, allocate block, block address: 0x0484
L1 shared cache, victim block for replacement, block address: 0x0884, is dirty:
false
****************************
L1 shared cache
cache hits: 4
cache misses: 6
cache conflict misses: 2
****************************
```

Listing 25 No private cache simulation output

To the best of our knowledge, there is no architecture like this. However, it shows that some experimental architectures are supported by the simulation.

5.4 Measuring Cache Performance

The miss rate calculated by the simulation can provide a good measurement to the cache predictability. The simulation provides statistics for every single cache level after performing all memory accesses. The statistics include the number of hits and the number of misses, including the number of misses per miss type. Using these statistics, the miss rate can be calculated, while the memory access time can be approximated as well. In general, a number of cache parameters like the cache size, block size, associativity, cache coherence, and cache inclusion property can affect the miss rate.

6 Conclusion and Future Work

In this thesis work, the cache architectures in the modern multi-core processors were first analyzed in order to find a better way to adapt as many architectures as possible to the simulation. Then, a software cache simulator was designed, implemented and tested to simulate the behavior of predefined cache architectures according to the provided memory accesses. The simulator takes the cache structure as an input in XML format and then uses a predefined XML-schema to validate it. The simulator also takes the memory accesses of the simulated software as an input list file. In this file the user can define the number of the cores and the levels in the cache hierarchy. The upper levels in the hierarchy can be considered private, while the last level of cache can be either private or shared among the cores. Within each cache level, cache configuration parameters and replacement policy can be specified. Every cache is considered in the simulation as a standalone cache, receiving block requests, allocating new blocks, handling the evicted blocks, and dealing with the cache coherence across the cache hierarchy. The simulator then starts executing the provided sequence of memory accesses. Every memory access becomes a block request, which is sent first to the first level in the cache hierarchy. The simulator records a memory access trace for all caches throughout the hierarchy. The trace contains every operation that was executed at every cache level affected by the request. The trace can be used to calculate the required time to perform each memory access and every cache level counts the number of hits and misses, which can be used in cache predictability analysis.

We can propose as future work to implement additional replacement policies, i.e., Random (RAND) and Pseudo-Round-Robin (PRR) replacement policies [2]. The simulation can be extended to support a system consisting of several separate blocks of cache coherent cores [45]. The simulator can also be extended to include additional hardware features, i.e., crossbar interconnect, for more realistic system simulation.

References

1. Patterson, David A., and John L. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2008.
2. Grund, Daniel. *Static Cache Analysis for Real-Time Systems*. epubli, 2012.
3. Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.
4. Zhang, Ke, et al. "PAC-PLRU: A Cache Replacement Policy to Salvage Discarded Predictions from Hardware Prefetchers." *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE, 2011.
5. Roy, Sourav. "H-NMRU: a low area, high performance cache replacement policy for embedded processors." *VLSI Design, 2009 22nd International Conference on*. IEEE, 2009.
6. Perez, W. J. H., et al. "Functional test generation for the pLRU replacement mechanism of embedded cache memories." *Test Workshop (LATW), 2011 12th Latin American*. IEEE, 2011.
7. Cullmann, Christoph, et al. "Predictability considerations in the design of multi-core embedded systems." *Proceedings of Embedded Real Time Software and Systems* (2010): 36-42.
8. Grund, Daniel, and Jan Reineke. "Toward precise PLRU cache analysis." *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Vol. 15. Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik, 2010.
9. Reineke, Jan, et al. "Timing predictability of cache replacement policies." *Real-Time Systems* 37.2 (2007): 99-122.
10. Wilhelm, Reinhard, et al. "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 28.7 (2009): 966-978.
11. Ferdinand, Christian, and Reinhard Wilhelm. "Efficient and precise cache behavior prediction for real-time systems." *Real-Time Systems* 17.2-3 (1999): 131-181.
12. Grund, Daniel, and Jan Reineke. "Abstract interpretation of FIFO replacement."*Static Analysis*. Springer Berlin Heidelberg, 2009. 120-136.

13. Wallin, Dan, and Erik Hagersten. "Miss penalty reduction using bundled capacity prefetching in multiprocessors." *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003.
14. Jeyapaul, Reiley, and Aviral Shrivastava. "Smart cache cleaning: energy efficient vulnerability reduction in embedded processors." *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2011.
15. Mounes-Toussi, Farnaz, and David J. Lilja. "Write buffer design for cache-coherent shared-memory multiprocessors." *Computer Design: VLSI in Computers and Processors, 1995. ICCD'95. Proceedings., 1995 IEEE International Conference on*. IEEE, 1995.
16. Jaleel, Aamer, et al. "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies."*Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010.
17. Qian, Bin-feng, and Li-min Yan. "The research of the inclusive cache used in multi-core processor." *Electronic Packaging Technology & High Density Packaging, 2008. ICEPT-HDP 2008. International Conference on*. IEEE, 2008.
18. Li, Lingda, et al. "Improving inclusive cache performance with two-level eviction priority." *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE, 2012.
19. Haque, Mohammad Shihabul, et al. "TRISHUL: A Single-pass Optimal Two-level Inclusive Data Cache Hierarchy Selection Process for Real-time MPSoCs."
20. Subha, S. "A two-type data cache model." *Electro/Information Technology, 2009. eit'09. IEEE International Conference on*. IEEE, 2009.
21. Zheng, Ying, Brian T. Davis, and Matthew Jordan. "Performance evaluation of exclusive cache hierarchies." *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 2004.
22. Zhao, Li, et al. "Ncid: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies." *Proceedings of the 7th ACM international conference on Computing frontiers*. ACM, 2010.
23. Thomadakis, Michael E. "The architecture of the Nehalem processor and Nehalem-EP smp platforms." *Resource* 3 (2011): 2.
24. Conway, Pat, et al. "Cache hierarchy and memory subsystem of the AMD Opteron processor." *Micro, IEEE* 30.2 (2010): 16-29.

25. Fu, Cheng-Yang, Meng-Huan Wu, and Ren-Song Tsay. "A shared-variable-based synchronization approach to efficient cache coherence simulation for multi-core systems." *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011.
26. Suleman, M. Aater, et al. "Accelerating critical section execution with asymmetric multi-core architectures." *ACM Sigplan Notices*. Vol. 44. No. 3. ACM, 2009.
27. Nikolopoulos, Dimitrios S., and Theodore S. Papatheodorou. "Fast synchronization on scalable cache-coherent multiprocessors using hybrid primitives." *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE, 2000.
28. Zuberi, Khawar M., and Kang G. Shin. "An efficient semaphore implementation scheme for small-memory embedded systems." *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*. IEEE, 1997.
29. Garg, Vijay K. *Concurrent and distributed computing in Java*. Wiley-IEEE Press, 2005.
30. Hossain, Hemayet, Sandhya Dwarkadas, and Michael C. Huang. "POPS: Coherence protocol optimization for both private and shared data." *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011.
31. Tian, Yingying, and Daniel A. Jiménez. "Sampling Temporal Touch Hint (STTH) Inclusive Cache Management Policy." *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011.
32. Semin, Andrey. "Inside Intel Nehalem Microarchitecture." (2009).
33. Martin, Milo MK, Mark D. Hill, and Daniel J. Sorin. "Why on-chip cache coherence is here to stay." *Communications of the ACM* 55.7 (2012): 78-89.
34. Al-Mouhamed, Mayez A., and Khaled A. Daud. "Experimental Analysis of SMP Scalability in the Presence of Coherence Traffic and Snoop Filtering." *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 2012.
35. Lis, Mieszko, et al. "Memory coherence in the age of multicores." *Computer Design (ICCD), 2011 IEEE 29th International Conference on*. IEEE, 2011.
36. Intel® Microarchitecture (Nehalem), http://www.intel.com/.
37. AMD® Advanced Micro Devices (Opteron), http://www.amd.com/.

38. Kalla, Ron, et al. "Power7: IBM's next-generation server processor." *Micro, IEEE* 30.2 (2010): 7-15.
39. Čakarević, Vladimir, et al. "Characterizing the resource-sharing levels in the UltraSPARC T2 processor." *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009.
40. Molka, Daniel, et al. "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system." *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 2009.
41. Devices, A. Micro. "AMD64 architecture programmer's manual volume 2: System programming." (2006).
42. UML Lab-Yatta, http://www.uml-lab/en/uml-lab/.
43. Haque, Mohammad Shihabul, Jorgen Peddersen, and Sri Parameswaran. "CIPARSim: Cache intersection property assisted rapid single-pass FIFO cache simulation technique." *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2010.
44. Zang, Wei, and Ann Gordon-Ross. "T-spacs: a two-level single-pass cache simulation methodology." *Proceedings of the 16th Asia and South Pacific Design Automation Conference*. IEEE Press, 2011.
45. Blake, Geoffrey, Ronald G. Dreslinski, and Trevor Mudge. "A survey of multicore processors." *Signal Processing Magazine, IEEE* 26.6 (2009): 26-37.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Stuttgart, 29. July 2013 _____