Snooping protocol proposal to Improve Cache Performance via Reducing Memory Access Time

Rehab Flaih Hasan[*1], Maha Abdulkareem[1] and Abeer Diaa Al-Nakshabandi[2]

[1]Department of Computer Sciences – University of Technology – Baghdad
[2]Distribution Office at Ministry of Electricity – Baghdad

[*]surorh@yahoo.com

Received: 2 April 2018 Accepted: 9 May 2019

Abstract

Cache design in multiprocessor systems usually involves maintaining data consistency between these processors that are achieved through implementation one of most important protocols used for this purpose which are snooping protocol and directory-based protocol. It also includes improved memory access time by reducing the time spent in three cases which are: miss rate, miss penalty and time to hit in the cache. Generally, there exist three critical attributes that have an impact on the performance of any coherence protocol in the cache which are low-latency cache-to-cache misses, bandwidths efficiency and scalability challenges. In this research, a new protocol has been proposed for coherent caches named PMOESI protocol. This protocol has the same states of a standard MOESI protocol but the difference is in adding a new state named Premier "P" and also an exclusive reference buffer is designed to be added to Level1 cache. The MOESI protocol is a version of the snooping coherence protocol which each block in the cache memory can have one of five (Modified, Owned, Exclusive, Shared, Invalid) states. From using the proposed protocol, the performance is enhanced as a result of reducing latency time in comparison with MOESI protocol. The reason behind this improvement is in using low latency cache to cache transfer to deliver the desired block instead of fetching this block from main memory for responding to request writing of remote processors.

Keywords: Cache coherence problem, Snooping protocol, Directory-Based cache Protocols, MOESI, Cache Simulation, Dev. C++, Multiprocessor, Shared memory.

DIYALA JOURNAL FOR PURE SCIENCES

Snooping protocol proposal to Improve Cache Performance via Reducing Memory Access Time

Rehab Flaih Hasan, Maha Abdulkareem and Abeer Diaa Al-Nakshabandi

اقتراح بروتوكول الاستطلاع لتحسين أداء الذاكرة المخبئية عبر تقليل وقت وصول الذاكرة

رحاب فليح حسن[1]، مها عبد الكريم الراوي[1] و عبير ضياء النقشبندي[2]

[1]قسم علوم الحاسوب – الجامعة التكنولوجية – بغداد
[2]قسم التوزيع – وزارة الكهرباء – بغداد

الخلاصة

تصميم الذاكرة المخبئية في أنظمة المعالجات المتعددة عادة يتضمن الحفاظ على تطابق البيانات ما بين تلك المعالجات والتي تتحقق من خلال تنفيذ احدى أهم البروتوكولات المستخدمة لهذا الغرض والتي هي بروتوكول الاستطلاع والبروتوكول القائم على الدليل. ويشمل أيضا تحسين وقت الوصول للذاكرة من خلال تقليل الوقت الذي يقضيه في الحالات الثلاث والتي هي: عند عدم وجود كتلة البيانات في الذاكرة المخبئية، عند جلب كتلة البيانات من اقل مستوى فيها تلك الكتلة مضافا اليها وقت تسليم تلك الكتلة الى المعالج وكذلك في حالة وجود كتلة البيانات في الذاكرة المخبئية. عموما توجد ثلاث من الصفات الحرجة التي يكون لها تأثير على أداء اي بروتوكول ترابط في الذكرة المخبئية والتي هي تقليل الوقت المستغرق مابين اصدار الطلب من قبل احدى الذواكر المخبئية وتلقي الاستجابة من اخرى وكفاءة الخط الناقل للبيانات ومدى قابلية استخدام عدد كبير من المعالجات. في هذا البحث، لقد تم اقتراح بروتوكول جديد وذلك لتحقيق تطابق الذاكرة المخبئية والذي يسمى بروتوكول PMOESI. هذا البروتوكول له نفس حالات البروتوكول القياسي MOESI ولكن يختلف باضافة حالة جديدة تسمى "P" Premier وبتصميم مخزن مرجعي للحالات الحصرية ليتم اضافته في المستوى الاول من الذاكرة المخبئية. بروتوكول MOESI هو احد انواع بروتوكول الاستطلاع واسم البروتوكول مشتق من من خمس حالات اللي تمتلكها اي كتلة في الذاكرة هذه الحالات هي معدلة، ممتلكة، حصرية، مشتركة، غير صالحة. من استخدام البروتوكول المقترح، لقد تم تحسين الكفاءة كتيجة في تقليل وقت الاستجابة مقارنة مع بروتوكول MOESI.

السبب وراء هذا التحسين هو باستخدام الانتقال مابين الذواكر المخبئية لتسليم الكتلة المطلوبة بدلا عن جلب تلك الكتلة من الذاكرة الرئيسية عند الاستجابة لطلب الكتابة من المعالجات البعيدة.

الكلمات المفتاحية: مشكلة الترابط في الذاكرة المخبئية، بروتوكول الاستطلاع، البروتوكول القائم على الدليل، MOESI، محاكاة الذاكرة المخبئية، DEV C++، المعالجات المتعددة و الذاكرة المشتركة.

Introduction

Using of a multicore system, scientific and technical applications achieved better performance by scaling the core count [1]. Cache memory plays an important role in the design of these systems. It is used to access data instead of main memory which reduces the latency of delay time [2]. In such systems, when installing different caches in different processors in shared memory architecture, the difficulties will appear when there is a need to maintain consistency between the cache memories of different processors [16]. So, cache coherency protocol is very important in such kinds of systems; MSI, MESI, MOSI, MOESI, etc. are the most famous protocols to solve cache coherency problem [6, 8 and 13].

The Cache coherence problem appears when two different processors can have two different values for the same location. This problem is solved through coherency the caches such that any reader of the data item will return the most recently written value of that data item. In a coherent multiprocessor, the cache benefit, from both migration and replication of shared data [4]. Migration: moving data to a local cache to reduce both data latency to access shared data item that allocated remotely and the bandwidth demand on the shared memory. Replicating: copying the shared data that is being read to reduce both the latency of access and contention for a read shared data item [10].

Coherence Controller

There are two type of controller within multicore processor chip which are cache controller and memory controller. These controllers are state machines that include logic for implementing the coherence protocols and are communicated via queues [19]. The cache controller accepts loads and store from the core and returns load values to the core figure 1a. The controller initiates a coherence transaction by issuing coherence request to access the required block when a miss occurs in the cache. This coherence request is sent across interconnection network to one or more of coherence controller. The memory controller is a coherence controller at the Last Level Cache (LLC) figure 1b. A memory controller is similar to a cache controller but it differs that it has only a network side and it does not perform coherence request (loads or stores) or receive coherence response [7].

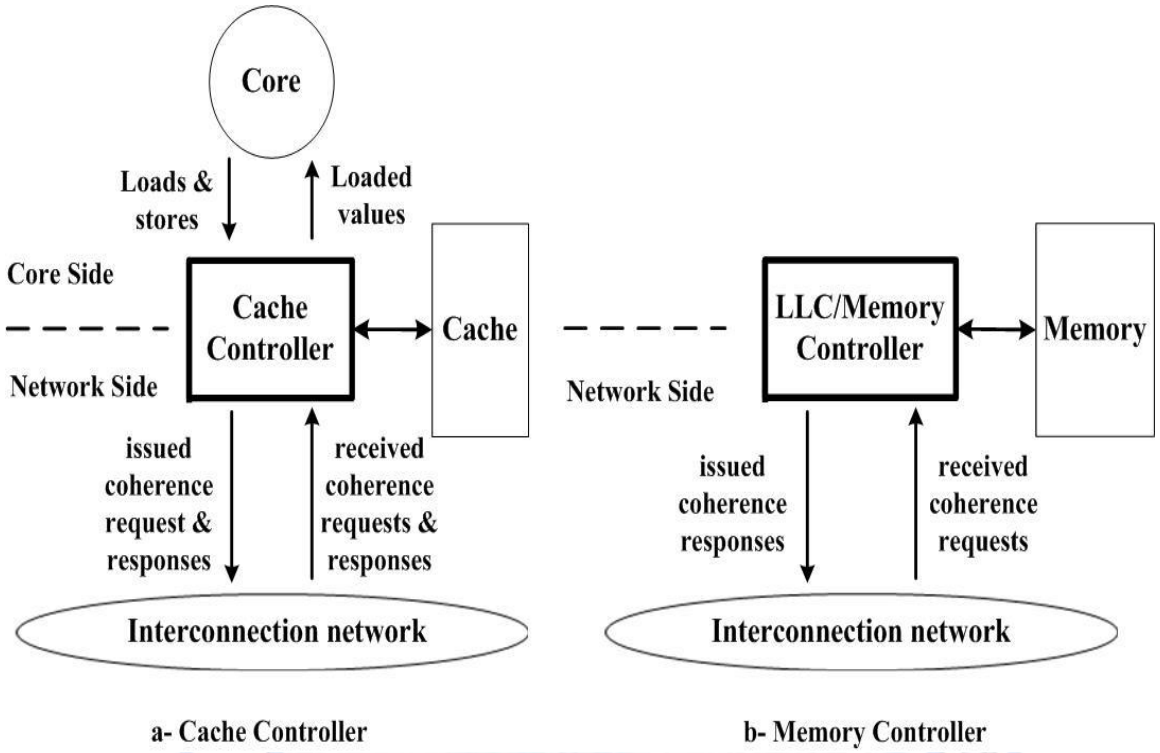

Figure 1: Cache and Memory Controller [11]

Protocols for Cache Coherence

There are two hardware-based protocols that achieve coherence between caches in multiprocessor systems which are snooping protocol and directory-based protocol [1 and 3]. The snooping protocol uses a single shared bus to solve the problem of cache coherence via broadcasting all requests for data to all processors and processors snoop to see if they have a copy and respond accordingly [9]. A shared bus is a high-speed link that is needed to transfer data between processors, caches, I / O interfaces, and memories within a multicore processor system in a coherent fashion figure 2 [5 and 20]. The ability to scale in directory-based schemes is better than snooping because it does not depend upon a shared bus for communication [4]. The directory which can be central or distributed keeps the state of all memory block shared between processors and then the cache controller uses point-to-point messages looking up directory instead of observing shared broadcast to get memory block state [18] figure 3.

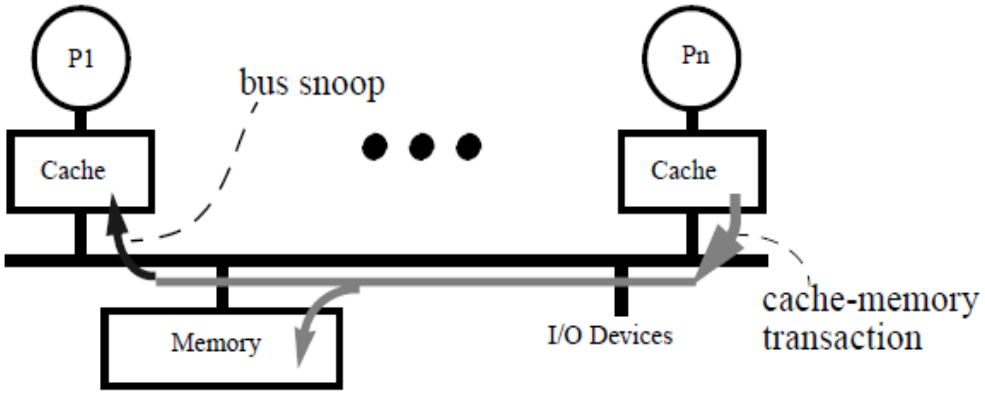

Figure 2: Snoopy Protocol [20]

Although the directory-based protocols will likely have to be employed for multi core architectures of the future, there exist a drawback that appears in a directory which are: storage overhead, frequent indirections, and are more prone to design bugs [15].

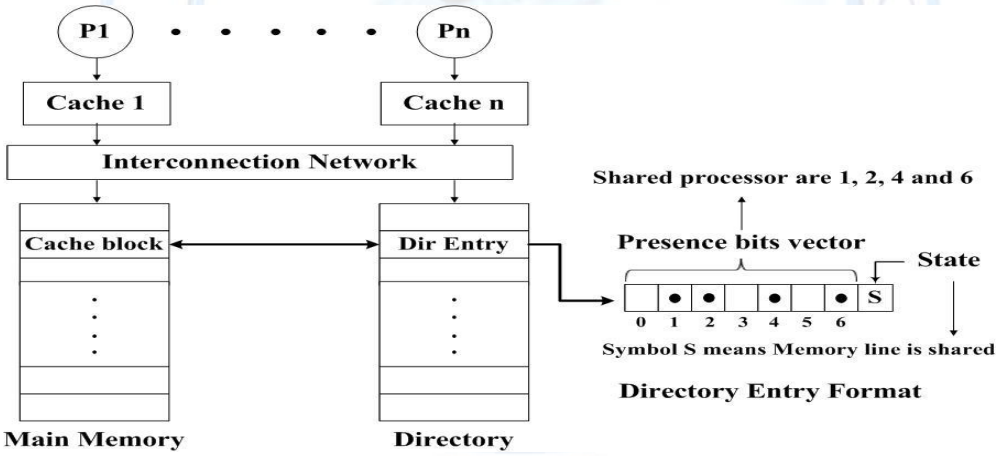

Figure 3: Directory Based Protocols [12 and 20]

Replacing Policies

The direct, fully associative, and set associative are three techniques which can be used in the mapping process to map the blocks of main memory into cache lines because these lines are fewer than main memory blocks [16]. In set and fully associative caches there exist various replacement algorithms that are used because these types of caches have several positions that the block may go, so there are different probabilities to choose a block that will be replaced [17]. The most used algorithms for replacement are: first in first out (FIFO) that replace the block that has been in the cache longest, least recently used

(LRU) that replace the block that has not referenced in cache for longest time, least frequently used (LFU) that replace block which has fewest hits and Random in which one block is selected randomly and replaced [10].

Measuring Cache Performance

The Average Memory Access Time (AMAT) gives us four metrics for cache optimizations: hit time, hit rate, miss rate, and miss penalty which expressed as in equation (1) [10, 12, 14, 17]:

$$\text{AMAT} = L1\ hit\ time * L1\ Hit\ rate + (L2\ Hit\ time * L2\ Hit\ rate + (L3\ Hit\ time * L3\ Hit\ rate + Access\ time\ of\ Main\ Memory\ * L3\ Miss\ rate) * L2\ Miss\ rate) * L1\ Miss\ rate \quad \textbf{(1)}$$

Where

- ***Hit*** -- the data requested by the processor appears in the cache.
- ***Miss*** -- the data is not found in the cache.
- ***Hit time*** – is how long it takes data to be sent from the cache to the processor. This is usually fast, on the order of 1-5 clock cycles at Level1, of 10-20 clock cycles at Level2, of 30-40 clock cycles at Level3, of 50-100 clock cycles at the main memory.
- ***Miss penalty*** – is the time required to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor.
- ***Hit ratio*** – the percentage of time the data is found in the higher cache.
- ***Miss ratio*** – is the percentage of misses and equal (100 - hit ratio).

Implementation Steps of Proposed Premier Modified Own Exclusive Shared Invalid (PMOESI) Protocol

The following sections illustrate main steps that needed to implement a proposed protocol using DEV C++ language:

1. Preparation Steps

This protocol has been implemented through the simulation process and some parameters must be preset to follow up the implementation methods accurately as shown in figure 4.

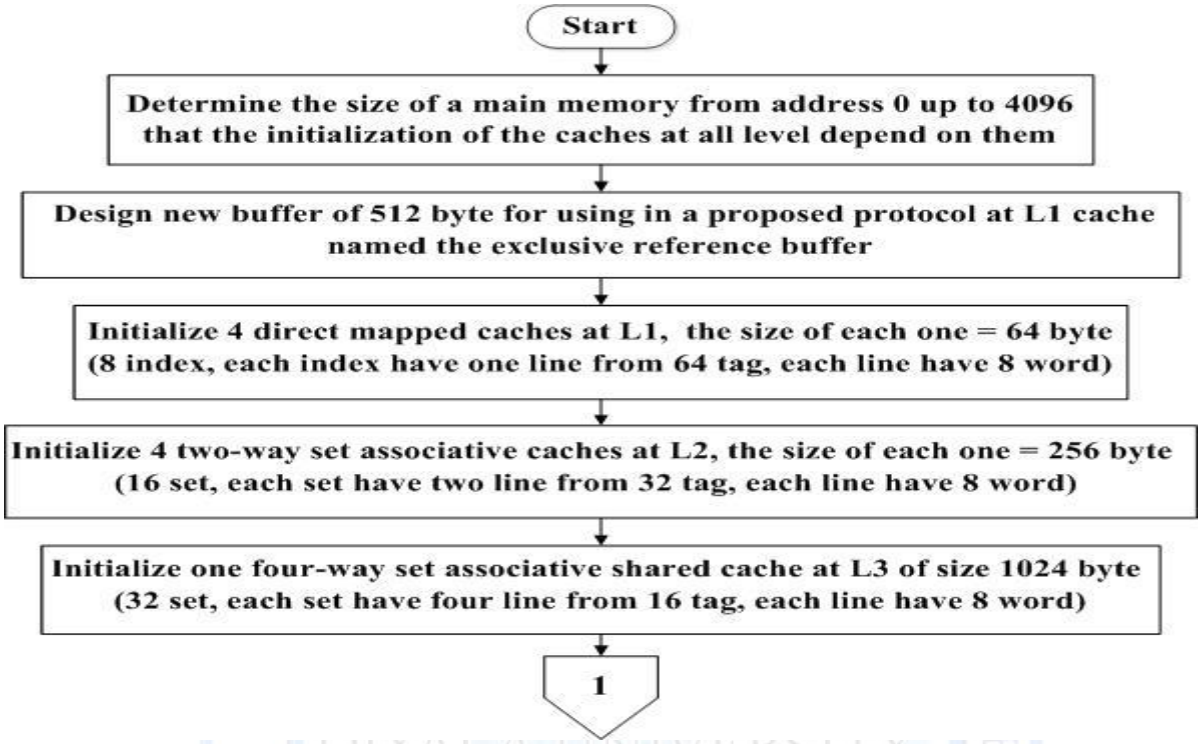

Figure 4: Steps Prepared Preset for a Proposed Protocol

2. Binary Representations of Memory Addresses

After defining the preparation parameters, the following functions should be implemented:

a) Binary Conversion Function

All main memory addresses are converted to a binary address.

b) Index, Tag and Offset Conversion Function

The tag, index and offset of address bits should be specified from a binary requested address depending on a chosen size of the cache and then each one is converted to decimal number via a binary to decimal conversion function in order to be used in the simulation process.

3. Cache Simulation from Instructions of Randomly Selected Addresses

All addresses of main memory have been simulated to all cache levels figure 5.

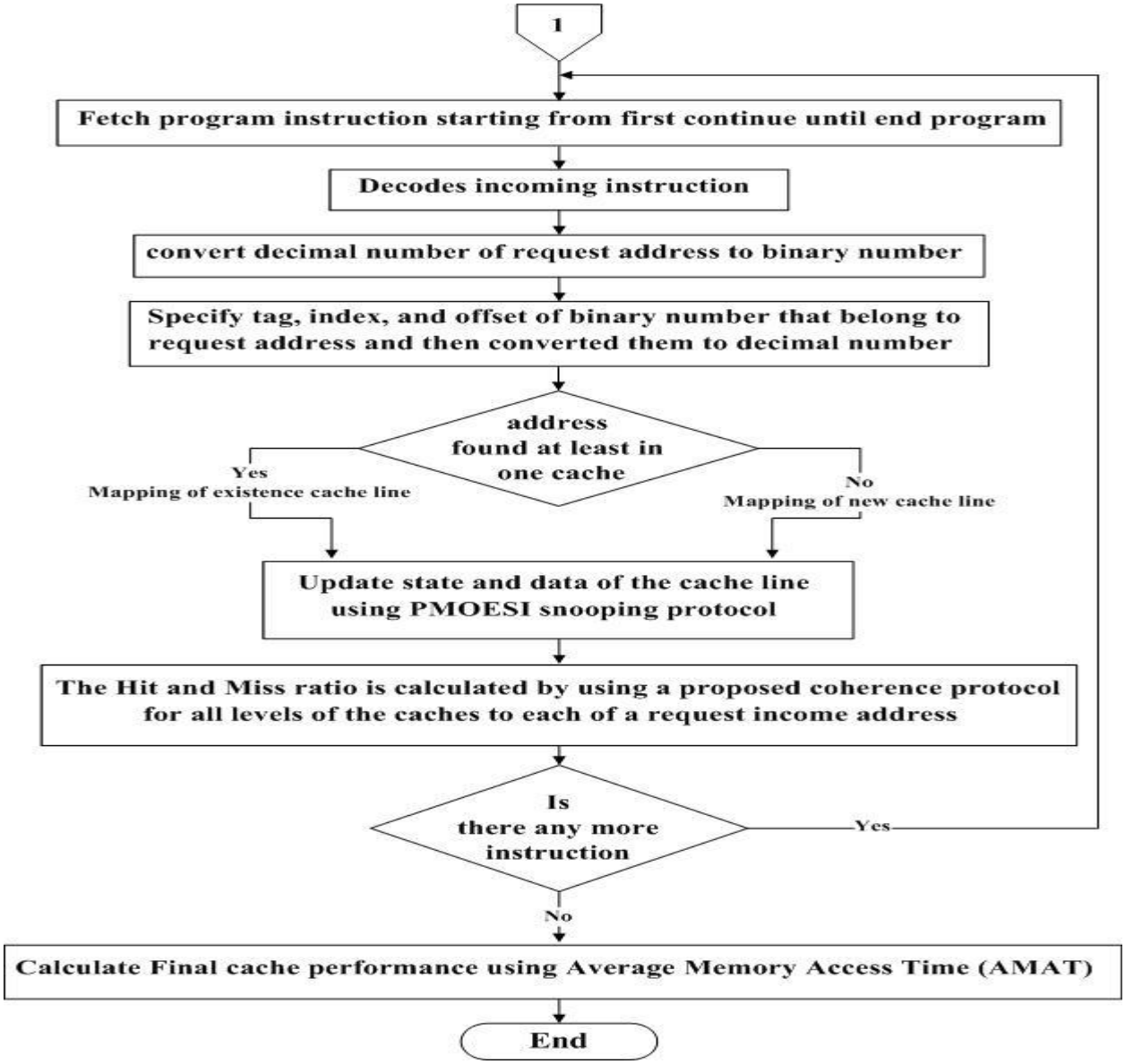

Figure 5: General Structure of Cache Simulation

4. (LFU + LRU) Replacement Algorithm

In this research LFU algorithm is merged within LRU algorithm. This algorithm is used to select one line from two lines at a set of L2 in the case of incoming other third line evicted from L1. The victim line is evicted to place at L3cache by using initially LFU algorithm if frequently of two line are not an equal and select line that least frequently used and if the frequently of

two line are equal then LRU algorithm is implemented for choosing least recently line to be evicted. Also, this algorithm is used to select one line from four lines at a set of L3 in the case of incoming other fifth line evicted from L2 and then victim line is evicted to put at the main memory.

5. The Transition States of Proposed PMOESI Cache Coherence Protocol

The transition between states of the proposed PMOESI protocol is shown in figure 6.

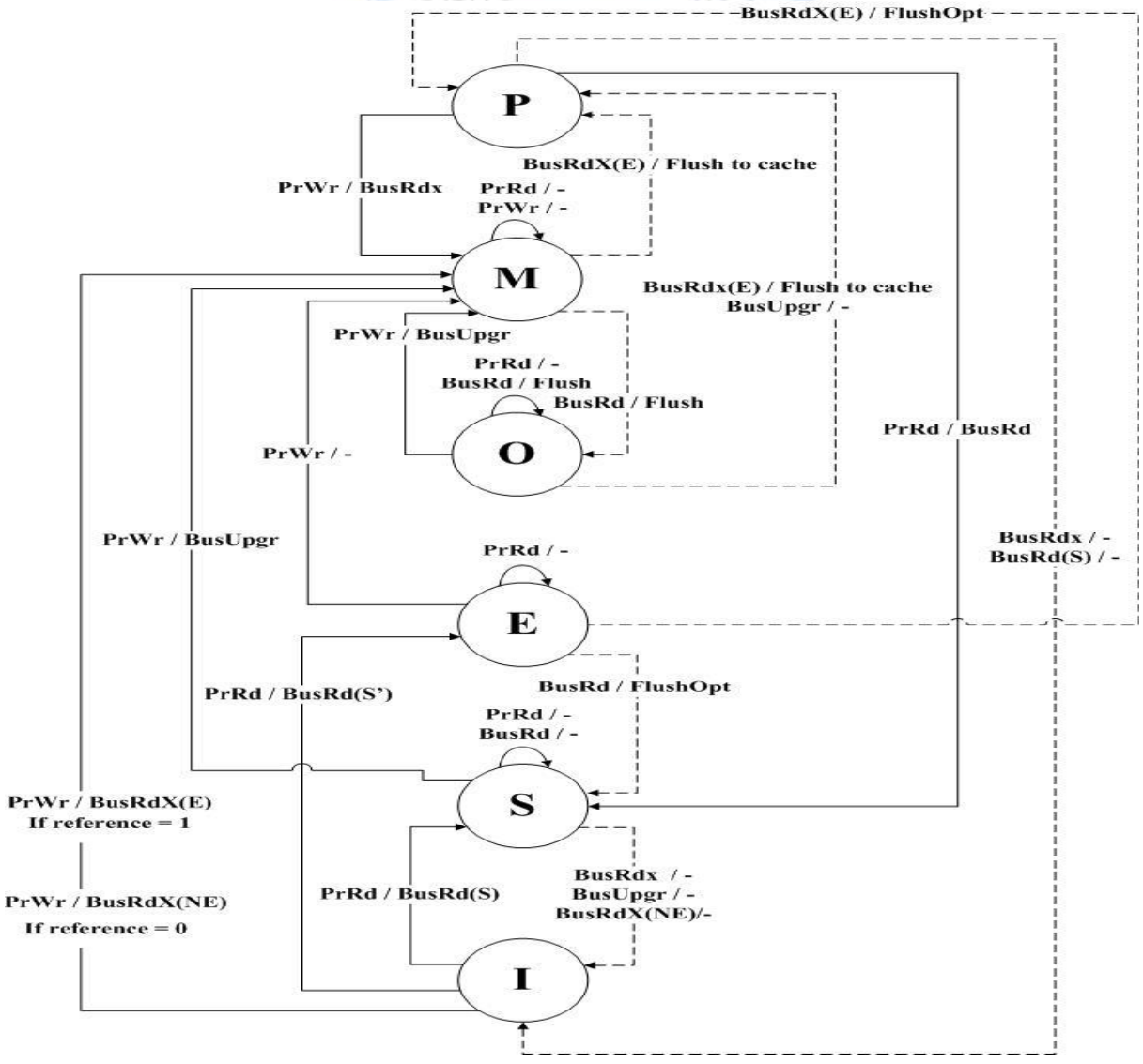

Figure 6: Buses to Coherent Caches using PMOESI Cache Coherence Protocol

Experimental Results

The proposal PMOSEI protocol design depends on low latency cache to cache misses in order to reduce the miss penalty, to approve this enhancement, several experiments have been applied for different cases of micro-instructions code, with different assumption concerning the main memory size, cache block size, and different degree of associative. The different results are obtained from different experiments, the following subsections present the specific assumptions for each experiment with a step by step tracing followed by performance evaluation.

1. Comparison between MOESI Protocol & PMOESI Protocol

This experiment compares between MOESI and PMOESI cache coherence protocol as shown in table 1 by using program of six instructions that two processor request address B1 and supposes that initially, B1 is not cached. The difference between these two protocols has been shown in blue when MOESI protocol is implemented and shown in red when the proposed protocol is implemented. The programs of 6 instructions are:

Table 1: MOESI Cache Coherence protocol Action

Seq	Event	Initially B1="I" in P1's Cache	Initially B1="I" n P2's Cache
1	P1 writes 10 to B1 (write miss)	B1 = 10 (Modified)	B1 = Invalid
2	P1 reads B1 (read hit)	B1 = 10 (Modified) **B1 is flush to main memory** **B1 is flush to cache2**	B1 = invalid
3	P2 writes 90 to B1 (write miss)	 **B1 = Invalid** **B1 = Premier**	B1 = 90 (Modified) **B1 is flush to main memory** **B1 is flush to cache1**
4	P1 writes 20 to B1 (write miss)	B1 = 20 (Modified)	**B1 = Invalid** **B1 = Premier**
5	P1 reads B1 (read hit)	B1 = 20 (Modified) **B1 is flush to main memory** **B1 is flush to cache2**	 **B1 = Invalid** **B1 = Premier**
6	P2 writes 30 to B1 (write hit)	**B1 = Invalid** **B1 = Premier**	B1 = 30 (Modified)

2. Memory Consistency

Two important disciplines are usually implemented that keep the memory consistent which are a memory consistency model and a cache coherence protocol. Cache coherence is an important, but incomplete piece of multiprocessor memory consistency. The consistency model is used to determine the behavior of reads and writes with respect to accesses to other memory locations while coherence determines the behavior of reads and writes to a single memory location. In table 2 a sample program has been ordered to complete coherency of 28 instructions using both direct mapped caches and set associative cache.

Initially, the cache simulation tracing using request addresses of a sample program in table 2 starting from the first address until ending program which shown in table 4 are mapped to direct mapped caches at all levels after determining the following parameters that are gaining from binary representation table 3:

Main Memory size = $2^8 byte = 256\ byte$,

Cache Block size = $2^i = 2^3 = 8$, where i=3 represents offset bits for all levels,

Cache Size = 2^j (j is (index + offset) bits), j = 5 at L1, j = 6 at L2, and j = 7 at L3

Cache Size at L1=$2^5 = 32$, Cache Size at L2= $2^6 = 64$, Cache Size at L3= $2^7 = 128$,

Number of tag at L1 = $2^3 = 8$, Number of tag at L2= 4, Number of tag at L3 =2,

Number of cache lines = Cache Size / Cache Block Size = $\frac{2^j}{2^i}$ then

Number of line at L1 = $\frac{2^5}{2^3} = 2^2 = 4\ line$, Number of lines at L2 = $\frac{2^6}{2^3} = 2^3 = 8\ line$,

Number of lines at L3 = $\frac{2^7}{2^3} = 2^4 = 16\ line$

Table 2: Sample Program of 28 instructions

Seq.	Core Name	Core request	Value	Address	Seq.	Core Name	Core request	Value	Address
1	p1	reads		35	15	p1	writes	10	41
2	p2	writes	100	135	16	p3	writes	69	164
3	p3	writes	80	135	17	p4	reads		100
4	p1	writes	20	35	18	p1	reads		41
5	p4	writes	80	228	19	p2	reads		193
6	p2	reads		164	20	p1	writes	33	135
7	p3	reads		41	21	p3	reads		41
8	p4	writes	30	135	22	p3	writes	8	164
9	p1	reads		135	23	p4	writes	55	100
10	p1	writes	11	52	24	p1	writes	93	135
11	p1	reads		228	25	p4	writes	77	100
12	p3	writes	99	100	26	p2	writes	200	80
13	p2	writes	77	193	27	p3	reads		80
14	p4	reads		193	28	p2	reads		80

Table 3: Binary Representation of Tag, Index and Offset at Three direct mapped Cache Levels

	Caches at Level1			Caches at Level2			Caches at Level3		
Seq.	Tag	Index	Offset	Tag	Index	Offset	Tag	Index	Offset
1	4	0	7	2	0	7	1	0	7
2	4	0	7	2	0	7	1	0	7
3	4	0	7	2	0	7	1	0	7
4	1	0	3	0	4	3	0	4	3
5	7	0	4	3	4	4	1	12	4
6	5	0	4	2	4	4	1	4	4
7	1	1	1	0	5	1	0	5	1
8	4	0	7	2	0	7	0	0	7
9	4	0	7	2	0	7	0	0	7
10	1	2	4	0	6	4	0	6	4
11	7	0	4	3	4	4	1	12	4
12	3	0	4	1	4	4	0	12	4
13	6	0	1	3	0	1	1	8	1
14	6	0	1	3	0	1	1	8	1
15	1	1	1	0	5	1	0	5	1
16	5	0	4	2	4	4	1	4	1
17	3	0	4	1	4	4	0	12	4
18	1	1	1	0	5	1	0	5	1
19	6	0	1	3	0	1	1	8	1
20	4	0	7	2	0	7	1	0	7
21	1	1	1	0	5	1	0	5	1
22	5	0	4	2	4	4	1	4	4
23	3	0	4	1	4	4	0	12	4
24	4	0	7	2	0	7	1	0	7
25	3	0	4	1	4	4	0	12	4

26	2	2	0	1	2	0	0	10	0
27	2	2	0	1	2	0	0	10	0
28	2	2	0	1	2	0	0	10	0

Table 4: Tracing of the Simulation of Direct-Mapped Caches using Table 1

index	Level1 Cache			
	P1	**P2**	**P3**	**P4**
0	1- **135** E 0 **P** & Flush Opt to P2 **I**	2- **135** **M** 100 Invalidate P1 **P** & Flush to P3	3- **135** **M** 80 Invalidate P1&P2 **P** & Flush to P4 **I**	5- **228** **M** 80
	4- **35** **M** 20	6- **164** **E** 0	12- **100** M 99	8- **135** **M** 30 Inv.P3 L1 &P2 L2 **O** 30
	9- **135** **S** 30 P1&P4 in L1	13- **193** **M** 77 19- **193** **O** 77 P2 in L1+P4 in L2	16- **164** **M** 69 Invalidate P2inL2 22- **164** **M** 8	14- **193** **S** 77 P2&P4 in L1
	11- **228** **S** 80 P1 in L1+P4in L2			17- **100** **S** 99 P4 in L1+P3 in L2 23- **100** **M** 55 25- **100** **M** 77
	20- **135** **M** 33 24- **135** **M** 93			
1	15- **41** **M** 10 Invalidate P3 18– **41** **M** 10 **O** 10		7- **41** **E** 0 **I** 21– **41** **S** 10 P1&P3 in L1	
2	10- **52** **M** 11	26- **80** **M** 200 **O** 200 28- **80** **O** 200	27- **80** **S** 200 P2&P3 in L1	
index	**Level2 Cache**			
	P1	**P2**	**P3**	**P4**
0	Evict result – 4 1- **135** **I**	Evict result – 6 2- **135** **P** **I**	Evict result – 12 3- **135** **I**	Evict result – 14 8- **135** **O** 30
	Evict result – 11 9- **135** **S** 30			Evict result – 17 14- **193** **S** 77
4	Evict result – 9 4- **35** **M** 20	Evict result – 13 6- **164** **E** 0 **P**	Evict result – 16 12- **100** **M** 99 **S** 99 **I**	Evict result – 8 5- **228** **M** 80 **O** 80
	Evict result – 20 11- **228** **S** 80			
index	**Level3 shared Cache**			
0	Evict result – 17 8- **135** **O** 30			
4	Evict result – 20 4- **35** **M** 20			

The evictions steps that occur through cache simulation of a direct mapping are:

- In step 4 evict line 128 to 135 from L1 to L2 in reaching line 32 to 39
- In step 6 evict line 128 to 135 from L1 to L2 in reaching line 160 to 167
- In step 8 evict line 224 to 231 from L1 to L2 in reaching line 128 to 135
- In step 9 move line 128 to 135 from L2 to L1 and as a result evict line 32 to 39 from L1 to L2
- In step 11 evict line 128 to 135 from L1 to L2 in reaching line 224 to 231
- In step 12 evict line 128 to 135 from L1 to L2 in reaching line 96 to 103
- In step 13 evict line 160 to 167 from L1 to L2 in reaching line 192 to 199
- In step 14 evict line 128 to 135 from L1 to L2 in reaching line 192 to 199
- In step 16 evict line 96 to 103 from L1 to L2 in reaching line 160 to 167
- In step 17 evict line 192 to 199 from L1 to L2 in reaching line 96 to 103 and evict line 128 to 135 from L2 to L3.
- In step 20 move line 128 to 135 from L2 to L1 and remove this line from L2 of P1 and also remove this line from L3 and then as a result of movement process the line 224 to 231 evict from L1 to L2 and the line 32 to 39 evict from L2 to L3.

The cache simulation tracing using table 2 that are mapped to two-way set associative caches at all levels are done in the same manner of table 4 after determining the following parameters: The main memory size, the block size, and the cache sizes are the same as in table 4 but the different is in using two lines in each set instead of one line. So, the number of tags is different as follow:

Number of tag at L1 = $2^4 = 16$, Number of tag at L2 $= 8$, Number of tag at L3 = 4,

Sets at L1 = Lines / number of way = $\frac{2^2}{2^1} = 2$, Sets at L2= $\frac{2^3}{2^1} = 2^2 = 4$, Sets at L3= $2^4/2^1 = 2^3$=8

The cache simulation tracing using table 2 that are mapped to direct mapped caches at L1 and two-way associative caches at L2 and four-way associative caches at L3 are done in the same manner of table 4 after determining the following parameters using 4096 of main memory addresses instead of 256 addresses:

Cache Block size at all levels = $2^3 = 8$, Cache Size at L1 = $2^6 = 64$,

Number of tag at L1 = $2^6 = 64$, Number of line at L1 = $\frac{2^6}{2^3} = 2^3 = 8\ line$,

Cache Size at L2 = $2^8 = 256$, Number of tag at L2 = $2^5 = 32$,

Number of lines at L2 = $\frac{2^8}{2^3} = 2^5 = 32\ line$, Sets at L2 = $2^5 / 2^1 = 2^4$

Cache Size at L3 = $2^{10} = 1024$, Number of tag at L3 = $2^4 = 16$,

Number of lines at L3 = $\frac{2^{10}}{2^3} = 2^7 = 128\ line$, Sets at L3 = $2^7 / 2^2 = 2^5$

Table 5 displays a result in applying PMOESI protocol on a sample program. Initially all states of an input addresses are Invalid, so when the processor P1 in step1 read address 135, the state is translated from "I" to "E" because the address is not found in the all levels of caches and as a result, the cache line that contains address gets by a read miss from main memory. All the next steps of the program are applied using the protocol in the same way.

Table 5: The Results of a Proposed Protocol using Table 2

Seq.	Core Name	Core Job	Data	Cache Line			Job	Sharer of Cores
				Address	State	Value		
1	P1	reads		135	E	0	R. M.	
2	P2	writes	100	135	M	100	W.M.	
3	P3	writes	80	135	M	80	W.M.	
4	P1	writes	20	35	M	20	W.M.	
5	P4	writes	80	228	M	80	W.M.	
6	P2	reads		164	E	0	R.M.	
7	P3	reads		41	E	0	R.M.	
8	P4	writes	30	135	M	30	W.M.	
9	P1	reads		135	S	30	R.M.S.	P1&P4 in L1
10	P1	writes	11	52	M	11	W.M.	
11	P1	reads		228	S	80	R.M.S.	P1 in L1+P4 in L2
12	P3	writes	99	100	M	99	W.M.	
13	P2	writes	77	193	M	77	W.M.	
14	P4	reads		193	S	77	R.M.S.	P2&P4 in L1
15	P1	writes	10	41	M	10	W.M.	
16	P3	writes	69	164	M	69	W.M.	
17	P4	reads		100	S	99	R.M.S.	P4 in L1+P3 in L2
18	P1	reads		41	M	10	R.H.	
19	P2	reads		193	S	77	R. H.	P2 in L1+P4 in L2
20	P1	writes	33	135	M	33	W.H.	
21	P3	reads		41	S	10	R.M.S.	P1&P3 in L1

22	P3	writes	8	164	M	8	W.H.	
23	P4	writes	55	100	M	55	W.H.	
24	P1	writes	93	135	M	93	W.H.	
25	P4	writes	77	100	M	77	W. H.	
26	P2	writes	200	80	M	200	W.M.	
27	P3	reads		80	S	200	R.M.S.	P2&P3 in L1
28	P2	reads		80	S	200	R.H.	P2&P3 in L1

Note: The words of the abbreviation symbols in table 5 are as follows:

R. M. = Read Miss, **W. M.** = Write Miss, **R. H.** = Read Hit

Number of hit = 8, Number of miss = 20,

Hit ratio = (number of hit / total number of addresses) * 100 = (8 / 28) * 100 = 29%

Miss ratio =100 - Hit ratio =100 - 28= 71%

Conclusions

From analyzing the results, the hit time is affected by using a different level of caches. The best case is the existence of the data in the first level and the worst case is the lack of data in the cache memories and being forced to fetch the data from the main memory causing miss penalty. The missed penalty has been reduced through organizing the caches to several levels and also via low-latency cache-to-cache misses. This research presents a new design of a cache coherence protocol that optimized memory access time.

The table 1 shows in steps 3, 4 and 6 the differences between MOESI and proposal protocol such that in a PMOESI protocol the write back to main memory does not occur but only need flush to cache that remotely writes to the same location. While in MOESI protocol the flush of the block is done to main memory to deliver to cache see table 5.

Table 5: Comparison between MOESI and PMOESI protocol

MOESI Cache Coherence Protocol	PMOESI Cache Coherence Protocol
In step3 P2 snoop request bus to invalidate P1 to change state from M to I and as a result P1 Flush cache block by transferring this block to main memory and fetched to do RWITM	In step3 P2 snoop request bus to invalidate P1 to change state from M to P and as a result P1 Flush cache block by transferring this block to cache2 to do RWITM
In step4 P1 snoop request bus to invalidate P2 to change state from M to I and as a result P2 Flush cache block by transferring this block to main memory and fetched to do RWITM	In step4 P1 snoop request bus to invalidate P2 to change state from M to P and as a result P2 Flush cache block by transferring this block to cache2 to do RWITM

In step6 P2 snoop request bus to invalidate P1 to change state from M to I and as a result P1 Flush cache block by transferring this block to main memory and fetched to do RWITM	In step6 P2 snoop request bus to invalidate P1 to change state from M to P and as a result P1 Flush cache block by transferring this block to cache2 to do RWITM

From steps 3, 4 and 6 of table 1, the performance of proposed protocol is increased because the latency of delay time is reduced in comparison with MOESI protocol. The reason behind this improvement is in using cache to cache transfer to deliver cache block to be used for Read with Intent to Modify (RWITM) as a result of a remote write instead of fetching a cache block from main memory.

Note: the hit ratio in applying proposed protocol table 5 is not differing from hit ratio of standard MOESI protocol but cache performance is enhanced by reducing latency time

The comparison between tracing in table 4 (using direct mapped caches at all levels) and tracing when two-way associative caches have been used at all levels in the case of using the same size of main memory in both:

- Eviction result of a direct mapped cache = 11 line that evicts from L1 to L2 while by using two-way set associative mapping the number of eviction line = 8.
- LFU+LRU algorithm is used in set-associative cache without using it with a direct mapped cache
- In using set associative cache, the addresses do not need to evict to L3 cache while by using direct mapped cache two line evict to L3 cache

The results after tracing in increasing the main memory size and the caches sizes and using a different degree of associative caches between levels, i.e., at L1 the direct mapped cache is used and at L2, L3 set associative cache are used are:

- The number of eviction lines from L1 to L2 became 5 and there are no lines evict to L3 and main memory
- the access time is reduced

Note: the hit ratio in applying proposed protocol table 5 in three cases is not differing but the hit time between levels is reduced when the degree of associativity and the cache size are increased.

Future Works

In future work the activities that have been proposed for reducing access to main memory include: Increasing cache block size to reduce compulsory miss that depends on a chosen cache sizes, selecting larger size of cache to reduce capacity miss, increasing associativity degree in order to reduce conflict misses that are limited from 2-way to 16-way for balancing between lower miss rates and higher costs, Adding new level to reduce miss penalty, and optimizing cache coherence protocol by modifying in an existence states.

References

1. N. Parvathy, B.R. Upadhyay, T.S.B. Sudarshan, Cache Coherence: A Walkthrough of Mechanisms and Challenges, In: International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), 2016, India, pp. 2251-2256.
2. C. Vivek, International Journal of Advance Research in Computer Science and Management Studies (IJARCSMS), 4(7),26-28 (2016).
3. G. Borowik, Z. Chaczko, W. Jacak, T. Łuba, Computational Intelligence and Efficiency in Engineering Systems, Vol. 595, (Springer, 2015).
4. X. Lian, X. Ning, M. Xie, F. Yu, Cache Coherence Protocols in Shared-Memory Multiprocessors, In: International Conference on Computational Science and Engineering (ICCSE), (2015), pp. 286-289
5. X. Song, Lecture6: System Integration and Performance", CIS 410 Hardware and Software Architecture – Department of Information Systems California State University, Los Angeles, 2015.
6. Y. Solihin, Fundamentals of Parallel Multicore Architecture, (CRC Press Taylor & Frabcis Group A Chapman & Hall Book, Boca Raton, 2015)
7. S. Helmi, A. Nguyen, P. Nguyen, H. Kodali, Cache Coherence Simulation, CSCI 5593-Advanced Computer architecture, 2015.

8. A. Saparon, F. N. B. Razlan, Cache Coherence Protocols in Multi-Processor, In: International conference on Computer Science and Information Systems (ICSIS), 17-18 Oct (2014), Dubai (UAE), pp. 129-134.

9. A.B. Abdallah, Multicore Systems On - Chip: Practical Software / Hardware Design, 2nd ed. (Atlanties press, Aizuwakamatsu, Japan, 2013).

10. J. L. Hennessy, D. A. Patterson, Computer Architecture A quantitative approach,5th ed. (Morgan Kaufmann, Elsevier, 2012).

11. D. J. Sorin, M. D. Hill, D. A. Wood, A Primer on Memory Consistency and Cache Coherence, (Morgan & Claypool Publishers, Wisconsin, 2011).

12. K. Hwang, N. Jotwani, Advanced Computer Architecture parallelism, scalability, programmability, (McGraw Hill Education, New Delhi, 2011).

13. S. Lametti, Cache Coherence Techniques, M.S.c. Thesis, University of Pisa, Pisa, Italy, (2010).

14. W. Stalling, Computer organization and architecture designing for performance, (Pearson Education Inc., New Jersey, 2010).

15. S. H. Pugsley, J. B. Spjut, D. W. Nellans, R. Balasubramonian, SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches, In: 19th international conference on Parallel architectures and compilation techniques, 11-15 September (2010), Vienna, Austria, 465-476

16. M. M. Mano, Computer System Architecture, 3rd ed. (Pearson Education, New Delhi, 2008).

17. T. Rauber, G. R¨unger, Parallel Programming for Multicore and Cluster Systems, (Springer, Heidelberg, 2007).

18. K. Strauss, Cache Coherence Embeded - Ring Multiprocessors, PH.D. Dissertation in Computer Science, University of Illinois at Urbana - Champaign (2007).

19. D. J Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. Martin, D.A. Wood, IEEE Transactions on Parallel and Distributed Systems, 13(6), 556-578(2002).

20. D. Culler, J. P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware / Software Approach,1st ed. (Morgan Kaufmann Publishers, San Francisco, California, 1997).