LC-SIM:

A SIMULATION FRAMEWORK FOR EVALUATING LOCATION CONSISTENCY BASED CACHE PROTOCOLS

by

Pouya Fotouhi

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering

Spring 2017

LC-SIM:

A SIMULATION FRAMEWORK FOR EVALUATING

LOCATION CONSISTENCY BASED CACHE PROTOCOLS

by

Pouya Fotouhi

Approved: _____

Guang R. Gao,Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

I would like to thank Professor Gao for giving me the opportunity of joining CAPSL and multi-dimensional learning experience.

With special thanks to Dr. Stéphane Zuckerman for guiding me step by step over the research, and my colleague Jose Monsalve Diaz for deep discussions and his technical help.

Very special thanks to my wife Elnaz , and also my parents for their support and love.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

New high-performance processors tend to shift from multi to many cores. Moreover, shared memory has turned to dominant paradigm for mainstream multicore processors. As memory wall issue loomed over architecture design, most modern computer systems have several layers in their memory hierarchy. Among many, caches has become everlasting components of memory hierarchies as they significantly reduce access time by taking the advantage of locality.

Major processor vendors usually rely on cache coherence, and implement a variant of MESI, *e.g.*, MOESI for AMD, to help reduce inter-chip traffic on the fast interconnection network. Supposedly, maintaining coherence should help with keeping parallel and concurrent programmers happy, all the while providing them with a well-known cache behavior for shared memory. This thesis challenge the assumption that Coherence is well-suited for large-scale many core processors. Seeking an alternative for coherence, LC cache protocol is extensively investigated.

LC-Cache is a cache protocol weaker than Coherence, but which preserves causality. It relies on the Location Consistency (LC) model. The basic philosophy behind LC is to maintain a unique view of memory only if there is a reason to. Other ordinary memory accesses may be observed in any order by the other processors of the computer system.

The motivation to stand against cache coherence, relies on underestimated limitations implied on system design by coherence. Observations presented in this thesis, demonstrates that coherence eliminates the possibility of having a directory based protocol in practice since size of such directory grows linearly with number of cores. In addition, coherence adds implicit latency in many cases to the protocol.

This thesis presents LCCSim, a simulation framework to compare cache protocol based on location consistency against cache coherence protocols. A comparative analysis between the MESI and MOESI coherence protocols is provided, and pit them against LC-Cache. Both MESI and MOESI consistently generate more on-chip traffic compared to LC cache since transitions in LC cache are done locally. However, LC cache degrades total latency of accesses as it does not take the advantage of cache to cache forwarding. Additionally, LC cache cannot be considered a true implementation based on LC since it does not behave according to the memory model. The following summarizes the contributions of this thesis:

- Detailed specification of LC cache protocol, covering the missing aspects in the original paper.
- A simulation framework to compare cache protocols based on LC against cache coherence protocols.
- Extensive analysis of LC cache protocol, leading to discovery of several weaknesses.
- Demonstrating features for an efficient cache protocol, truly based on location consistency.

Chapter 1

INTRODUCTION

Traditionally in computer systems, speedup used to be gained through advances in miniaturization process and increase in the clock rate of processors. After a while, mainstream chip manufactures faced a challenged which had been speculated before. On one hand, feature size in the fabrication process were reaching the borders of quantum physics and on the other hand, due to the high frequencies where chips were processing at, power and energy consumption turned into serious concerns. To satisfy the need for performance in the market, designers had no choice other than having more than just one core on chip. But having multiple cores will rise several challenges.

An important issue to tackle was the need for communication between the cores. In other words, in order to hire the computation power of several cores concurrently a medium for communication was necessary. To address that, two solutions was proposed. One offered communication through messages between cores. MPI program execution model[15] is inspired by the idea of message passing between cores. Another solution was to share data among the cores and there are several program execution models following this approach [10][22][9][7]. Comparing the ideas discussed above is beyond the scope of this thesis and we refer to [12] for a detailed discussion. In a shared memory based system, cores can share data using shared memory. However, since such access can be made simultaneously, a set of rules needs to be defined to specify the orderings in such cases.

Another problem to overcome for system architects, was the cover the huge gap between the latency of accesses to main memory on one hand, and the time needed processing the data on the other hand. This issue is often referred as memory wall issue [34]. This gap has been increased as applications tend to require more space

to keep their data and having a bigger memory requires more time to decode the address. Use of a memory hierarchy, with smaller sizes of memory as it get closer to the processor, was the solution proposed. In addition, there are common features in data access patterns where we can take the advantage of and the most important one is called locality. Since the results of recent computations will often used in near future (*i.e.* temporal locality), and it is likely to access the elements next to the element which has just accessed (*i.e.* spatial locality), computer architects has proposed the idea of caching the data.

When it comes to the presence of caches on a multi-core system using shared memory, we can potentially have multiple copies of a location in different caches and we need a mechanism to deal with such cases. That is, the cache protocol of the system. It worth mentioning that a cache protocol is tightly coupled with memory consistency model of the system since it affects the ordering in case of accesses to shared data.

Major processor vendors often rely on cache coherence, and implement a variant of MESI, *e.g.*, MOESI for AMD. Supposedly, maintaining coherence should help with keeping parallel and concurrent programmers happy, all the while providing them with a well-known cache behavior for shared memory. This thesis challenge the assumption that Coherence is well-suited for large-scale many core processors. Seeking an alternative for cache coherence protocols, LC cache protocol is extensively investigated.

LC-Cache is a cache protocol weaker than coherence, but which preserves causality. It relies on the Location Consistency (LC) model. The basic philosophy behind LC is to maintain a unique view of memory only if there is a reason to. Other ordinary memory accesses may be observed in any order by the other processors of the computer system.

This thesis also presents LCCSim, a simulation framework to compare cache protocol based on location consistency against cache coherence protocols. A comparative analysis between the MESI and MOESI coherence protocols is provided, and pit them against LC-Cache. Both MESI and MOESI consistently generate more on-chip traffic compared to LC cache since transitions in LC cache are done locally. However,

LC cache degrades total latency of accesses as it does not take the advantage of cache to cache forwarding. Additionally, LC cache cannot be considered a true implementation based on LC since it does not behave according to the memory model. The following summarizes the contributions of this thesis:

- Detailed specification of LC cache protocol, covering the missing aspects in the original paper.
- A simulation framework to compare cache protocols based on LC against cache coherence protocols.
- Extensive analysis of LC cache protocol, leading to discovery of several weaknesses.
- Demonstrating features for an efficient cache protocol, truly based on location consistency.

The organization of this paper is as follows. Chapter 2 presents a brief discussion on memory consistency models and cache coherence protocols as the background required. The motivation of this thesis and problem formulation is presented in chapter 3. LC cache protocol is discussed in chapter 4, and the simulation framework is described in chapter 5. Experimental setup and results of experiments are presented in chapter 6. Lastly, the related work is presented in chapter 7 followed by conclusions of this thesis discussed in chapter 8.

Chapter 2

BACKGROUND

Most computer systems and multicore chips provide hardware support for shared memory. While defining correct behavior for a shared memory system may look simple at first glance, there would be several corner cases and challenges for an implementation of such behavior. In a shared memory system, addressing mode and consistency model together form the memory model of system. A memory model defines what correctness means for a shared memory system.

In this thesis, I discuss memory consistency model aspect of a memory model. For the rest of this thesis, a memory consistency model may be referred as memory model or consistency model.

Shared memory has turned to dominant paradigm for mainstream multicore processors. Meanwhile, due to the large gap between the latency of processor and main memory (also known as memory wall issue), most modern computer systems have several layers in their memory hierarchy. In addition, the performance of system can be improved by taking the advantage of locality[14]. Therefore, caches has become everlasting components of most computer systems.

However, the complexity of cache design increases in a shared memory system. Since several processors may access a shared location at the same time, cache protocols need to be coupled with proper mechanisms in order to maintain a *coherent* value for the shared locations if necessary. This problem is known as *cache coherence*.

It worth mentioning that the problem of coherence can potentially exist even in the absence of shared memory. The best example to provide would be *false-sharing*. False-sharing is the case where two private locations X and Y are mapped into one

cacheline, and X is being accessed by processor P_1 and Y is accessed by P_2. While locations X and Y are not truly shared between P_1 and P_2, one may illegally over write (and discard) the value produced by the other one.

Cache coherence can be considered as part of memory consistency models of a system. It will indeed affect the coherence protocol, and also imposes many constrains on the hardware implementation of the system. It worth keeping in mind that cache coherence and consistency model of the system are tightly tied concepts. But they are often considered as two separate issues as a divide-and-conquer strategy.

The reset of this chapter goes as follows. Section 2.1 presents a brief survey of consistency models. Cache coherence, the techniques for implementing coherence, and a few protocols as examples are discussed in section 2.2.

2.1 An Introduction to Memory Consistency Models

Memory consistency model implies rules on ordering of the accesses (*i.e.* loads and stores) to a given location. In other words, memory consistency model defines the correct behavior of a shared memory system. To illustrate why correct behavior must be defined, consider the example shown in figure 2.1. As can be seen, P_1 updates the value of location x to 1 and then writes 1 to location y. P_2 busy waits until the value of location y is 0, and the tries to load the value of location x into r1. Most programmers expect the value of r1 to be 1 after this execution, but running this snippet of code on most of modern computer can potentially make r1 equal to 0.

Initially : x = y = 0	
P1	P2
x = 1	while(!y){}
y = 1	r1 = x

Figure 2.1: Motivating example for memory consistency models

Instruction reordering is the main reason for such behavior. Nowadays, instruction reordering is used by compilers at compile time, and also by out-of-order

processors at execution time. The goal is to lower the total time of execution, provide more flexibility for scheduling, and avoid as many stalls as possible within the processor's pipeline. However, in the example shown in figure 2.1, since there is no data dependency between two write operations in P_1, these two instructions can be reordered leading to an execution trace similar to the trace shown in figure 2.2.

P1	P2
y = 1	
	while(!y){}
	r1 = x
x = 1	

Figure 2.2: One possible execution trace of the example presented in fig 2.1

The memory model of the system, can create (or prevent) many opportunities for optimizations. Over the past decades, several models has been introduced in academia and industry [1]. In this section I divide consistency models to following two categories, uniform and hybrid. This classification relies on types of accesses to shared location in a memory model. In subsection 2.1.1, Sequential Consistency [21], Total Store Ordering [32], and Coherence [18] are discussed as a few examples of unified memory consistency models. Subsection 2.1.2 includes Weak Ordering [4][13], Release Consistency [4], and Location Consistency [16][17] as examples of hybrid memory consistency models.

2.1.1 Uniform Memory Consistency Models

As I discussed memory models can be divided into two categories, unified and hybrid models. To have a better understanding of this classification, types of memory accesses should be discussed.

In a shared memory system, memory accesses are either private accesses or shared ones. Private accesses are read and write operations performed on private (*i.e.* not shared) memory locations. On the other hand, read and write operations

performed on shared location are considered shared accesses. Shared accesses to a given location can be either competing (*i.e.* racy) or non-competing[26]. A pair of accesses is considered competing if the ordering of two accesses is not defined, and at least on of them is a write operation. For instance, accesses to shared locations within a critical section are non-competing since ordering for them is guaranteed by mutual exclusion. A competing access can be synchronizing or non-synchronizing. Synchronizing accesses are those use to enforce ordering. For instance, a synchronizing access may delay the access and wait for all previous accesses to finish. However, not all competing accesses are synchronizing. Non-synchronizing accesses, defined as competing accesses without ordering constrains imposed, can be found in chaotic relaxation algorithms.

Uniform memory models are models which do not distinguish access categories. Three examples of such models are presented in the rest of this section.

2.1.1.1 Sequential Consistency

For most of the programmers, Sequential Consistency (SC) is considered the most intuitive model. Lamport formalized SC [21] back in 1979. According to Lamport, if "the result of an execution is the same as if the operations had been executed in the order specified by program" the such core is considered *sequential*. A multiprocessor system is called *sequentially consistent* if "the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor in this sequence if the order specified by its program".

Going back to the example presented in figure 2.1, any possible execution of such program in a system using SC will result 1 as the final value for r1. Storing value 0 in r1 is simply not possible since the program order in P_1 should be respected and therefore, the write operation on x should always preformed before the write operation on y.

To compare sequential consistency with other uniform consistency models presented in this section, consider the accesses presented in figure 2.3. In this example,

P_1 writes to location x then reads location y, and P_2 writes to location y and reads location x. The question is to see if a model allows both read operations, values in r1 and r2, to be equal to 0 under a valid set of accesses.

Initially : x = y = 0	
P1	P2
x = 1	y = 1
r1 = y	r2 = x

Figure 2.3: A simple example for uniform consistency models. The question is to see if a model allows both r1 and r2 to be equal to 0.

In a system with SC as its memory model, r1 and r2 can not be equal to zero at the same time after execution of the example shown in figure 2.3. SC does not allow reordering of memory accesses. If any of read operations returns 0, it implies that the preceding write is observed by memory and return value for the other read operation should be 1. Therefore, at most one of the read operations can return 0.

Sequential consistency is considered intuitive for most programmers. But from a design prospective, it impose several restrictions and limitations. In particular, obeying the program order for all memory accesses prohibits reordering of instructions even in the absence of data dependencies. Instruction reordering is the key to many compiler optimizations nowadays, and also a presumed fact for out-of-order processors. In addition, the fact that SC requires all memory accesses from all processors to be observed by all processor in a particular order, significantly limits the scalability of this model. Therefore, SC is often assumed stronger than needed and expensive to implement for modern processor's design.

2.1.1.2 Total Store Ordering

According to the manual for SPARC-V8 provided by designer in 1992 [32], the standard memory model is called Total Store Ordering (TSO) and all SPARC implementations must provide at least the TSO model.

Total Store Ordering guarantees that the store (S), FLUSH (F), and atomic load-store([L,S]) instructions of all processors appear to be executed by memory serially in a single order called the memory order. Furthermore, the sequence of these instructions in the memory order for a given processor is identical to the sequence in which they were issued by the processor. Intuitively, each processor interacts with memory (assuming a single port memory) through a port followed by a switch (a multiplexer). Every port consists of a First Input First Output (FIFO) store buffer and a load line. Restrictions of memory model applied through signals between ports of processors.

TSO, likewise SC, does not allow reordering for two consecutive load or store operations, and a load operation followed by a store. But it allows reordering of a store operation followed by a load operation (obviously, in the absence of data dependencies). Therefore, for the set of accesses in figure2.3, TSO allows both r1 and r2 to be equal to 0. Read operations can be reordered, and executed before the write operation in each processor. Such reordering can allow both read operations to return the initial value for location x and y. One valid order of accesses under TSO is shown in figure 2.4.

P1	P2
r1 = y	r2 = x
x = 1	y = 1

Figure 2.4: One possible order of accesses in a system using TSO

SC and TSO are fairly close models, especially compared with more relaxed consistency models. However, TSO offers better performance compared to SC. This performance gap can covered in speculative processors. Any valid execution/implementation using SC, is a subset of valid executions/implementations using TSO. SPARC also provided two other memory models in their next generation called Partial Store Ordering (PSO) and Relaxed Memory Ordering (RMO). An interested reader may refer to [33] for detailed specifications of each protocol.

2.1.1.3 Coherence

Coherence [18], as a memory consistency model, can be seen as a location-relative weakness of SC. As I discussed in section 2.1.1.1, sequential consistency required all processors to agree on some sequential order for execution of all memory accesses. Whereas in coherence, accesses are required to be sequentially consistent on a per-location basis. In other words, for each memory location x, there is a total order of all the memory operations dealing with x and memory operations on x follow their program order.

Coherence is significantly weaker compared to SC and TSO. It eliminates the program order constrain imposed by SC and TSO on accesses to different locations in memory. Clearly, a system obeying SC or TSO respects coherence but not vice versa. In the context of example presented in figure 2.3, coherence allows r1 and r2 to be both hold the value 0 after executing corresponding memory accesses. Figure 2.5 shows one valid ordering under coherence.

x	y
r2 = x	r1 = y
x = 1	y = 1

Figure 2.5: A valid trace of accesses under coherence

Coherence also inspires cache protocols since it naturally fits within the subject. Cache coherence protocols are later discussed in section 2.2.

2.1.2 Hybrid Memory Consistency Models

Contrary to uniform memory models, hybrid models distinguish different access categories. In hybrid models, ordering constrains are defined depending of the category of the access. For instance, one model may have strict rules for synchronizing accesses while providing less restrictions for non-synchronizing memory operation.

Similar to the previous section, three examples of hybrid memory consistency models are presented.

2.1.2.1 Weak Ordering (Weak Consistency)

Dubois et al proposed Weak ordering (WO) [13] in 1986. A system using WO should implement the following.

- Accesses to synchronization locations are sequentially consistent.
- All previous memory accesses access to a synchronizing location in a processor should be performed before another memory access is issued.
- No memory access is issued by a processor before an issued memory access to a synchronizing variable has been performed.

Putting all together, synchronizing accesses in WO can be seen as memory *fence* operations. In other words, no regular access can *pass* a synchronizing access (*i.e.* to be reordered before or after, according to the program order). Moreover, synchronizing accesses are sequentially consistent with respect to each other.

2.1.2.2 Release Consistency

As an improvement on WO, Gharachorloo et al proposed Release Consistency (RC) [18] in 1990.Their approach was to impose fewer constrains on ordering memory accesses by exploiting information about properties of shared-memory accesses. Therefore, RC divides accesses to shared locations to *acquire*, *release*, and *non-synchronizing* accesses. Acquire accesses are similar to synchronizing accesses in WO, except that only future accesses are delayed upon an acquire. Likewise, a release operation works similar to a synchronizing access in WO, but issuing is delayed until all previous accesses have been performed. To formally describe the model, a system follows RC if:

- Before an *ordinary* Load/Store access is allowed to perform (become visible) with respect to any other processor, all previous acquire accesses must be performed.
- Before a release access is allowed to perform with respect to any other processor, all previous ordinary Load/Store accesses must be performed.
- Special accesses are sequentially consistent with respect to one another.

Gharachorloo et al proved that RC is equivalent to, and will produce same results as, sequential consistency at least for a particular category of programs called *Properly-Labeled* [1].

2.1.2.3 Location Consistency

In December 1999 Gao and Sarkar proposed a memory model called Location Consistency (LC). In this model, ordering constrains are captured at location level on a per location basis. For a given location, a set of valid values called value set is defined for any read operation based on the *state* of that location. The state obtained through Partially Ordered Multi-set (POMset) for every location. POMsets are updated upon performing a write, acquire, or release operation. Due to importance of LC in this thesis, this model is discussed further in detail.

A POMset, representing the state of location L, is defined as $state(L) = (S, \prec)$. Where S is a multi-set and $\prec \subseteq S \times S$. An element in S is either an acquire, a release, or a write operation. In order to compute the new POMset $(S_{new}, \prec_{new})$ after operation e (either a release or a write operation) based on the old POMset $(S_{old}, \prec_{old})$, the following rules should be applied.

$S_{new} := S_{old} \cup \{e\}$

$\prec_{new} := \prec_{old} \cup \{(x, e) \mid x \in S_{old} \wedge processorset(x) \cap processorset(e) \neq \emptyset\}$

Where $processorset(e)$ is defined as the set of processors involved in operation e. Upon an acquire operation e, following rules is used to impose desired orderings.

$S_{new} := S_{old} \cup \{e\}$

$\prec_{new} := \prec_{old} \cup \{(x, e) \mid x \in S_{old} \wedge processorset(x) \cap processorset(e) \neq \emptyset\} \cup \{(e', e) \mid e' = most_recent_release(e) \wedge e\prime \neq \emptyset\}$

Note that $most_recent_release(e)$ is non-empty if location L is previously release by some processor P. To obtain the value set $V(e)$, the set of permissible values for

[1] The definition of properly-labeled programs is beyond the scope of this thesis and could be found in the original paper [18]

read operation e, first an extended POMset should be defined. An extended POMset $\prec'$ definition is as follows.

$$S' = S \cup \{e\}$$

$$\prec' = \prec \cup \{(e', e) \mid e' \in S \wedge P_i \in processorset(e')\}$$

Now the value set $V(e)$ can be defined as follows.

$$V(e) = \{v \mid \exists w \in S' \wedge \{condition1 \vee condition2 = True\}\}$$

$$condition1 : w = write(P_i, v, L) \wedge w \prec' e \mid \{\nexists w' \mid w' \prec' e \wedge w \prec w'\}$$

$$condition2 : w = write(P_j, v, L) \wedge w \not\prec' e$$

Condition 1 defines set of most recent processor writes (MRPW)[2]. Condition 2 points to all the write operations from other processors without any specified orderings related to read operation e. Intuitively, the value set $V(e)$ for a read operation e on location L by P_i contains one or more values from the following.

- The value of most recent write operation to location L by processor P_i.
- The value of most recent write operation to location L by some processor P_j if and only if *proper* acquire and release operations are used.
- Any value produced by write operations (without any specified orderings, *i.e.* racy write operations) on location L by processor $P_{j|j\neq i}$.

Note that in case of programs without data-race, where *proper* acquire and release operations are used for accesses to shared memory, the value set $V(e)$ only contains one value (as is expected by most programmers).

In support of their work, authors proved weakness (the model is weaker than RC), equivalence (it is equivalent to RC for *data-race free* programs), monotonicity (same value set is valid for a more parallel version of the program), and non-intrusiveness (reads would not affect the value set) for the LC model as four measures of usefulness and robustness [17][16].

[2] Refer to [17] for a detailed definition.

2.2 Cache Coherence

In a uniprocessor system, caches are *functionally* invisible for programmers. In a shared memory system, several processors may accesses a shared location. Since each processor can store a *local* copy of location in their caches, any updates on the local copy potentially can lead to an *incoherent* access by other processors. As discussed in section 2.1, the memory model of system is in charge of defining *correct* orderings in case of concurrent memory accesses. Therefore, a cache protocol is required to apply rules defined by the memory model of system. Such protocol should carefully follow the memory consistency model of system, to avoid creating new functional behavior by caches. In this section, cache coherence protocols are discussed. But it worth keeping in mind that non-coherent protocols also exist, and they bring new trade-offs to design space of a memory system.

Implementation details of caches are beyond the scope of this work since this thesis aims the protocols. A cache coherence protocol is often seen as a finite state machine. State transitions[3] are defined depending on processor's request. Whilst a coherent protocol can have several states, the can be classified based on the following measures.

- **Read/Write Permissions** The operations which a processor is allowed to perform on a location depends on the state of that location. Serving a read operation is often straightforward since it does not impose any modifications and a *valid* copy of location would server the purpose. On the other hand, a write operation usually requires more considerations since it potentially can create incoherency.
- **Dirty/Clean Data** Each location in cache is a *local* copy of the original value in main memory[4]. The value of this local copy can be either *Clean* or *Dirty* compared to the original value. A local copy is considered dirty if it holds a different value from the original copy, and it is considered clean if it contains a value identical to the original copy.

[3] This thesis does not discuss details of transient states, which can be found in [31].

[4] Considering several layers in the memory hierarchy will not change such analysis, and intentionally avoided to keep away the unnecessary complexity

Coherence can be maintained at different granularities, ranging form one byte to several bytes. But most often the granularity of coherence is a cache line. Designing a coherence protocol can be several different ways. Even a given state machine (*i.e.* set of states and transitions) can lead to many different protocols. Among many design options regarding a coherence protocol, there is a primary decision with major impact on design. Coherence protocols can be divided into two classes: snooping and directory. These classes are briefly discussed in the following.

- **Snooping protocols** In snooping protocols, cache controller broadcasts requests to all other cache controllers. Other coherence controllers, repeatedly snooping for coming requests, will take required actions upon receiving the request. Most snooping protocols rely on an interconnection network (*i.e.* a shared medium) which delivers messages in a total order.
- **Directory protocols** In directory protocols, a cache controller send requests to directory which contains identities of sharers for each location. This way the necessary communication can be done through point-to-point communication with sharers.

Choosing between snooping and directory protocols is a design trade-off decision. Snooping protocols are less complex to implement, but suffer from poor scalability (as the number of cores increases) since broadcasting does not scale well. Directory protocols provide a better scalability, but they have three main shortcomings. First, implementing directories in hardware requires logic and the size of such directories grows linearly as the number of cores increases. Second, directory protocols make many two-hop transactions into three-hop transaction since they require an extra message to be sent to directory. This will increase the latency of accesses. Third, the directory will become the bottleneck of the system since it is the single point of contact for all processors. These shortcomings are mitigated to some extend through use of dynamic distributed directories [29][25].

As coherence is widely used in modern computer systems [25], the scalability issue of cache coherence protocols remains a challenge for computer architects, at least in near future. In the rest of this section, three examples of cache coherence protocols are presented.

2.2.1 MSI

MSI is the first model discussed in this section and is considered as the base model for other models discussed later. A cacheline in a system using MSI can have any of the following states:

- **Modified (M)** A cacheline with state M contains dirty data and therefore can not be evicted silently. Performing read and write operations on a cacheline with state M results in a cache hit. No other processor is allowed to have a copy of a cacheline in state M.
- **Shared (S)** Data in a cacheline with state S is clean. One protocol may or may not allow silent eviction of such line. A read request from processor results in a cache hit. But performing a write operation requires a state transition from S to M, in order to obtain the write permission. Other processors may have a copy of a cacheline in state S.
- **Invalid (I)** An invalid cache line contains out-of-date information and it can be silently evicted from cache. Obviously, both read and write operations result in a cache miss and the line should be fetched from higher memory.

Figure 2.6 reflects the state machine for MSI. Arrows in black shows transition upon read (PrRd) and write (PrWr) operations and required messages (GetM/GetS) to be communicated. Red arrows on right side of figure shows the state transitions upon receiving a messages. Note that this figure is intentionally simplified and implementation details of for snooping or directory protocols are not included.

To illustrate how MSI protocol works, an example of state transition is shown in figure 2.7. Initially, both P_1 and P_2 have location[5] x in their caches with state I. Upon operation 1, P_1 sends a read request (GetS) for location x and loads x with state S. Performing operation 2 requires P_1 to obtain write access for location x. A write request (GetM) is issues by P_1 and the state of line is changed to M afterwards. The read operation on P_2, instruction 3, requires the state of location x in P_1 to be downgraded to S by sending a read request (GetS) and P_2 loads the location x with

[5] Here, a location is assumed to be in the minimum size which coherence is maintained at.

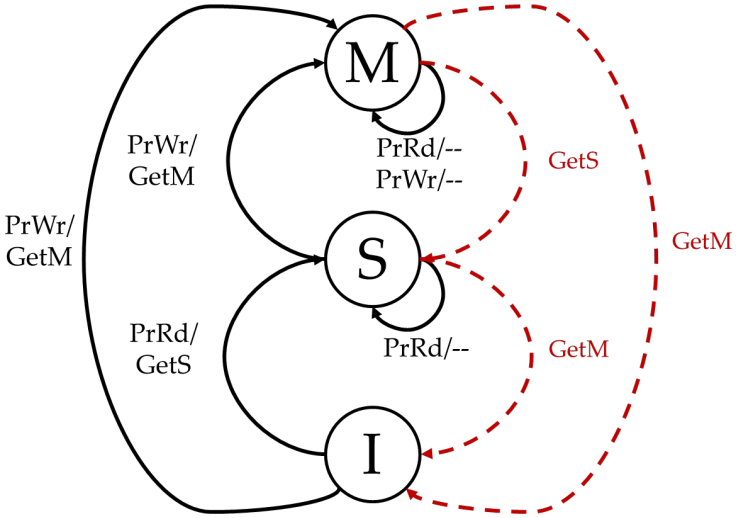


Figure 2.6: State diagram for MSI cache coherence protocol

state S. Upon instruction 4 in P_1, a write request (GetM) is needed since P_1 does not have write permission for a cache line with state S.

Instruction	P1(x)	P2(x)
0 : -	I	I
1 : P1 : RD (x)	S	I
2 : P1 : WR(x,5)	M	I
3 : P2 : RD(x)	S	S
4 : P1 : WR(x,10)	M	I

Figure 2.7: Operation of MSI protocol

Among the communications needed for the example shown in figure 2.7, some of can be eliminated by applying optimization on the protocol itself. Two of such techniques are discussed in the rest of this section.

2.2.2 MESI

MESI [27] protocol can be seen as an improvement on MSI protocol. MESI adds E state to MSI protocol to enable a processor to read and then write to a location

with a single coherence message. A cacheline in MESI protocol can be in one of the following states.

- **Modified (M)** Similar to MSI protocol, a cacheline with state M contains dirty data and therefore can not be evicted silently. Performing read and write operations on a cacheline with state M results in a cache hit. M is an exclusive state, meaning that no other core is allowed to have a copy.
- **Exclusive (E)** Data in a cacheline in state E is clean. As for a cacheline in state S, a read operation results in a cache hit. But unlike S state, a write operation also results in a cache hit. Most implementations allow silent eviction of a cacheline in state E. Similar to M state, E is an exclusive state as well.
- **Shared (S)** Like MSI, a cacheline with state S hold clean data. A read request from processor results in a cache hit and a write operation requires a state transition from S to M, in order to obtain the write permission. Other processors are allowed to have a copy of a cacheline in state S.
- **Invalid (I)** Data within an invalid line can not be used and should be replaced upon both read and write operations.

Figure 2.8 shows the state machine for MESI protocol. State transition and required messages upon read/write operations are shown in black while red arrows show state transitions required based on the incoming messages.

To highlight the advantage of MESI over MSI, figure 2.9 shows state transitions in a MESI cache protocol for the same instruction as used in figure 2.7. As can be seen, the state of location x after performing instruction 2 is E. This will eliminate the need for a communication upon performing instruction 3 since P_1 has exclusive access to location x.

2.2.3 MOESI

MOESI protocol improves MESI by enabling *dirty-sharing* of data by adding the O state. This improvement can significantly affect the protocol in case of single-producer-multiple-consumer access patterns. Each cacheline can have one of the following state in MOESI protocol.

- **Modified (M)** A cacheline in M state hold update data. The copy in maim memory is *stale* and should be updated upon eviction. No other processors are allowed to have a copy. Both read and write operations result in a cache hit.

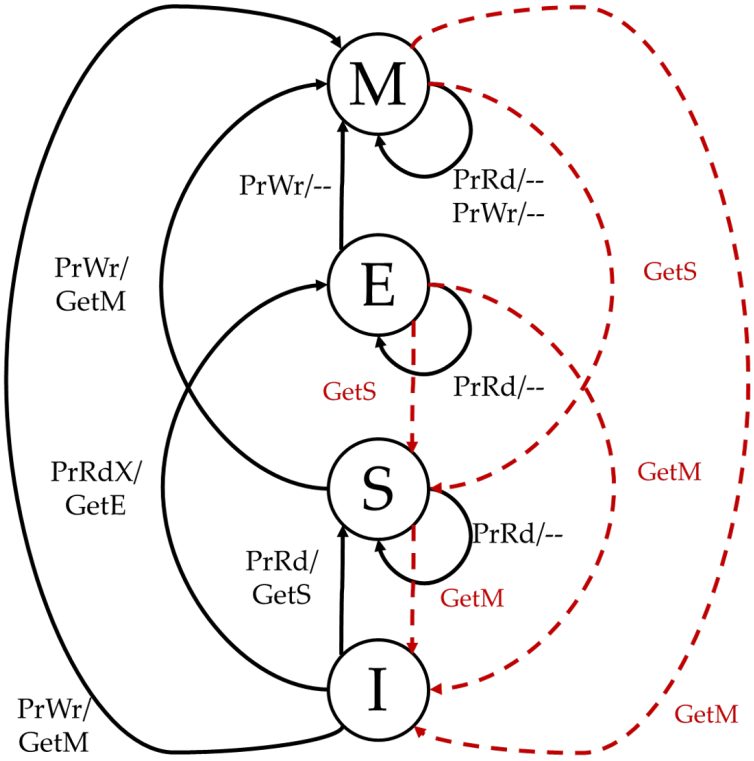


Figure 2.8: State diagram for MESI cache coherence protocol

Instruction	P1(x)	P2(x)
0 : -	I	I
1 : P1 : RD (x)	E	I
2 : P1 : WR(x,5)	M	I
3 : P2 : RD(x)	S	S
4 : P1 : WR(x,10)	M	I

Figure 2.9: Operation of MESI protocol

- **Owned (O)** Data within a cacheline in state O is update. O state is similar to S state, in the sense that it allows other processors to have a copy of the cache line. But it is different since the copy in main memory is stale and should be updated. Only one processor is allowed to be in state O and other processors may have a copy of cacheline in state S. Both read and write operations result in a cache hit, but a write operation requires invalidation of other copies.

- **Exclusive (E)** A cacheline in state E holds clean and update data. As for a cacheline in state S, a read operation results in a cache hit. But unlike S state, a

write operation also results in a cache hit. The copy in memory is valid and no other processor has a copy of cacheline.

- **Shared (S)** A cacheline in state S hold update data. Other processor may have a copy of cacheline. The copy in memory is valid unless some other processor has the cacheline in state O. Read operation results in a cache hit but a write operation requires coherence messages.
- **Invalid (I)** Data within an invalid line can not be used and should be replaced upon both read and write operations.

Similar to previous protocols, the state machine for MOESI protocol is shown in figure 2.10. Arrows in black shows transition upon read (PrRd) and write (PrWr) operations and required messages (GetM/GetE/GetS) to be communicated. Red arrows on right side of figure shows the state transitions upon receiving a messages. Note that this figure is intentionally simplified and implementation details of for snooping or directory protocols are not included.

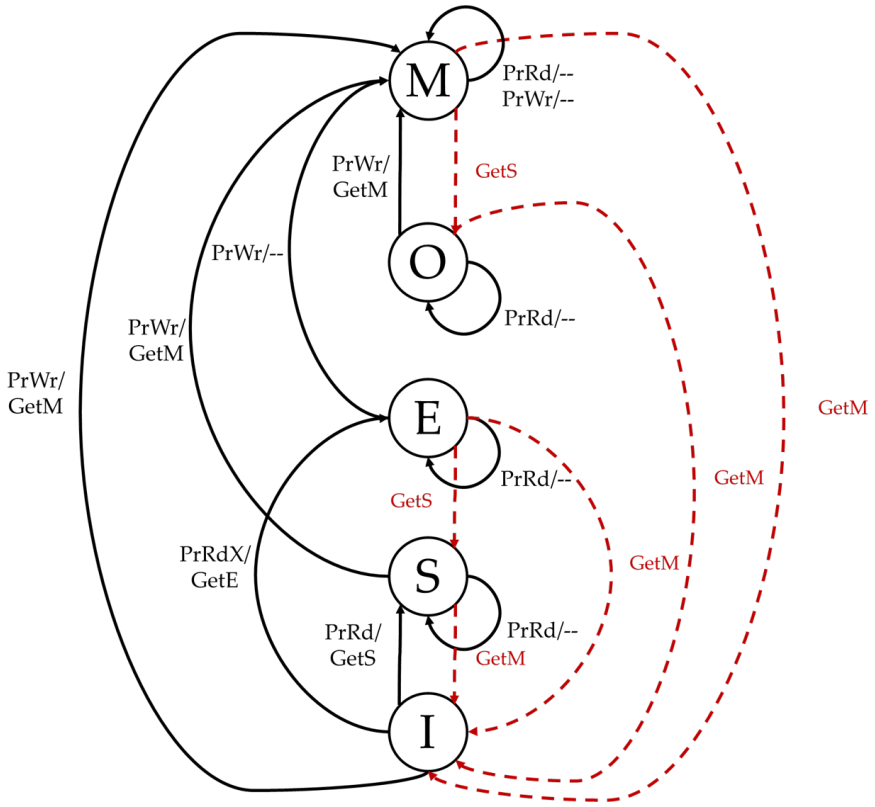


Figure 2.10: State diagram for MOESI cache coherence protocol

Figure 2.11 shows the state transitions in a MOESI cache protocol for the same instruction of figure 2.7. Adding the O state improves the protocol by allowing P_1 to perform instruction 4 without any coherence messages required since P_1 has the cacheline in state O.

Instruction	P1(x)	P2(x)
0 : -	I	I
1 : P1 : RD (x)	E	I
2 : P1 : WR(x,5)	M	I
3 : P2 : RD(x)	O	S
4 : P1 : WR(x,10)	M	I

Figure 2.11: Operation of MOESI protocol

Chapter 3

THE PROBLEM WITH COHERENCE

This thesis challenges the assumption that cache coherence protocols are well-suited for large-scale multi or many core processors. The motivation to stand against cache coherence, relies on underestimated limitations implied on system design by coherence.

First, despite the existence of better, more refined, and efficient coherence protocols such as MOESI, their implementation leads to significant amount of data transfers both on and off chip. Moreover, they do not address the need for atomic operations (such as compare-and-swap), which are crucial components of efficient lock-free data structures, or building blocks to efficient locking mechanisms in operating systems and runtime environments.

To design a cache coherence protocol for large-scale multi or many core processors, snooping protocols cannot be considered as snooping does not scale well. On the other hand, the size of directory in a directory protocol grows linearly with the number of cores. Lastly, cache coherence protocols impose implicit latency in many cases to the accesses.

An alternative for cache coherence protocols, should ideally generate less traffic on chip and require smaller size for a directory based protocol. To this end, LC cache protocol were found as a good candidate. This protocol is illustrated in the following chapter.

Chapter 4

LC-CACHE: TOWARD A SCALABLE CACHE PROTOCOL FOR EXASCALE SYSTEMS

As discussed earlier in 2.1.2.3, Location Consistency is a interesting relax hybrid memory model. However, since the need for caches is undeniable, a cache protocol should be attached to the memory model. It is imperative since it enhance the synergy between underlying hardware and the memory model of the system.

A cache protocol based on LC model would be different from other protocols discussed in section 2.2 since it violates coherence. Other protocols maintain coherence to respect the "correctness" defined by memory model of the system. Like any other consistency model, LC also provides semantics for a valid interleaving of the accesses but as explained in section2.1.2.3, since LC defines a value set for every location, there is no need for two distinct logical cores to agree on a single value for a given location.

In this section, location consistency cache protocol is discussed. The protocol was proposed by Gao and Sarkar. For the rest of this chapter, it is assumed that each cacheline represent one location in memory to avoid unnecessary discussions. A variety of methods, like what proposed by Gao and Sarkar, can be used for multi-word cache lines.

First the original model proposed by Gao and Sarkar followed by a brief discussion on the weaknesses of their solution is presented.

4.1 LC-Cache Protocol

Gao and Sarkar presented Location Consistency (LC) memory model along with an implementation for a cache protocol based on LC [17]. Protocol diagrams are

presented in figure 4.1 and they illustrate state transitions for Read and Write operation as well as Acquire and Release operations.

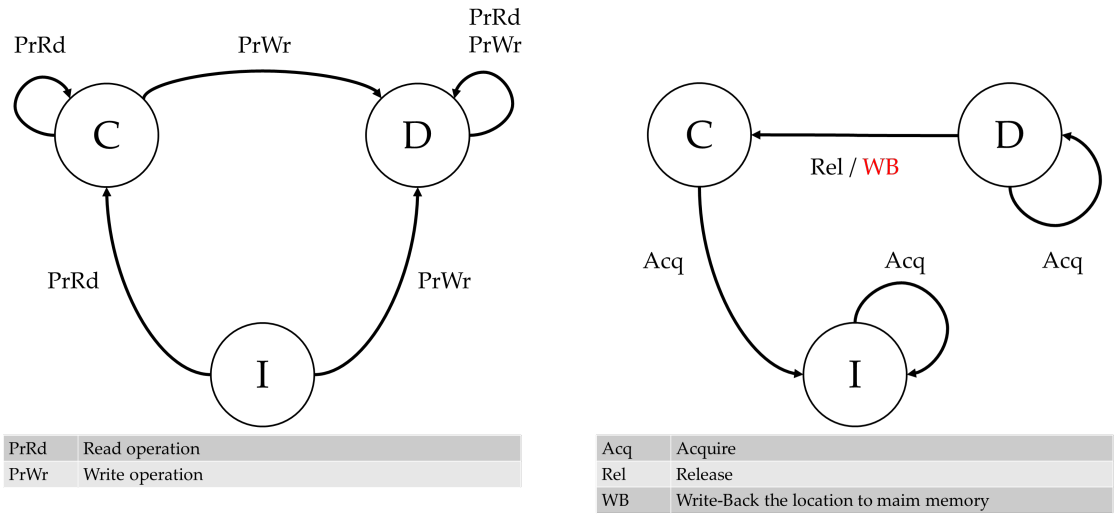


Figure 4.1: State transitions - LC cache protocol

As can be seen, cache line X can have one of the following three states:

- **Dirty (D)**
  The information within a cache line X in D state is valid. Line X will remain in D state unless it gets evicted from cache or a Release operation is applied on X. Both Read and Write operations accessing X will result in a hit.

- **Clean (C)**
  A cache line X in state C, holds valid information. While both Read and Write operations on X will result in a hit, the state of cache line will change form C to D upon the Write operation on X. In addition, the state of cache line will change from C to I by performing an Acquire operation on X. We refer to this action as self-invalidation. Note that, in all cases, there is no need to update directory or broadcast since protocol does not imply the need for exclusive ownership for Write operations.

- **Invalid (I)**
  An invalid cache line X contains information which is not valid. Contrary to previous cases, both Read and Write operation accessing X will result in a miss and the line should be fetched form higher level memory. Performing either an acquire operation or a release operation on an invalid cacheline does not imply any state transitions.

To explain the transitions between the states, the set of actions required upon each operation is presented.

- **Read**
  Reading a cache line in either D or C states will result in a hit. Performing a Read operation on a cache line in state I will result in a miss and the line should be fetched form higher level memory and stored in cache in state C.

- **Write**
  Similar to Read operations, writing to a cache line in D or C states will result in a hit. However, writing to an Invalid cache line (state I) will result in a miss. Upon a miss, the line should be fetched form higher level memory and stored in cache in state D.

- **Acquire**
  To obtain the lock, an atomic operation is performed. Atomic operations are supported by hardware in the current architectures and different implementations of them are beyond the scope of this work. The state of cache line would not change if it is in either D or I states. For cache lines in state C, however, a *self-invalidation* should be performed to obtain a fresh value since the value can possibly be updated by other processors.

- **Release**
  Releasing a cache line in C or I states would not imply any state transitions while releasing a dirty cache line requires updating the higher level memory (write-back) and changing the state to clean.

To illustrate how the protocol works, first I divide memory accesses into three categories and I will compare the proposed protocol with MESI protocol under each category. Memory accesses are divided into the following categories:

- **Regular accesses**
  Regular accesses are accesses to private locations (not shared with other processors). In case of regular accesses, LC Cache protocol would behave similar to MESI protocol, as an example of coherence protocols, form a performance prospective. However, the proposed protocol would be cheaper in terms of logic on chip and more efficient from an energy consumption point of view since it does not require snoops to other processors or a directory to consult with upon each access. In case of MESI, for private accesses, the cache line would be in either E (assuming only read operations) or M (assuming both read and write operations) state. Using LC Cache, the state of cache line would be either C (only read operations) or D (both read and write operations).

P0	P1	P2
atomic x++;		
	atomic x++;	
		atomic x++;

Figure 4.2: Atomic increments on shared location x

Note that, going from E to M state in MESI can be done silently (i.e. there is no need for a snoop or a consult with directory) but going form I to E state implies an exclusive snoop request or a consult with directory based on the implementation of the protocol. On the other hand, LC Cache will perform all state transition silently and save some traffic on chip as well as the energy consumed by sending those messages. Regarding the correctness of the program being executed, since regular accesses are accesses to private locations, there is no need for further explanations and program would behave exactly the same under both protocols.

- **Synchronous accesses**
  Synchronous accesses are accesses to shared locations which has been labeled with proper synchronization operations (*i.e.* acquire-release pairs). In other words, memory accesses within critical sections are called synchronous. The ordering of accesses to shared memory are defined by Memory Consistency Model of the system. However, any Memory Consistency Model will agree that there should be (at least) a total order on synchronous store operations for data race free programs.

  Note that in MESI, as an example of coherence protocols, synchronous accesses are not differentiated from regular accesses. The total order on write operations required by Memory Consistency Model is achieved through invalidations or updates based on the implementation of the protocol. In addition, coherence does not provide the semantics required for atomicity of accesses or creating critical sections and it needs to be coupled with additional mechanisms. In practice, system developers tried to create such support by proposing various solutions such as Lock/Unlock for creating critical sections in programs, or atomic operations such as fetch_and_increment or compare_and_swap for single atomic accesses. In case of synchronous accesses, LC Cache would perform on par compared to coherence protocols from a performance viewpoint. The total order of writes is maintained by write-backs upon release operation. The atomicity of access is guaranteed through acquire operation.

  To illustrate the state transitions in both cases, consider the example of figure 4.2 where three processors try to increment the shared location x atomically in the order shown in figure 4.2. We assume that each processor has one level of private cache with write-allocate and write-invalidate [14] policies.

– **MESI**

In case of MESI, atomic instructions are most often translated to Load-Linked Store-Conditional operations for hardware in order to fit them in pipelined processors. Load-Linked flag is used to guarantee the atomicity of the access.

The first processor (P0) accessing location X will load the cacheline in state E by sending a control message (a snoop request or a consult with directory) and later on will write to the location X silently (with no control messages) and changes the state of the cacheline to M. When the second processor (P1) loads the cacheline, it will send one control message and triggers the write-back of location X in P0s cache which causes an access to DRAM. By the time the write-back is completed, P0 will have the cacheline in state S. Then P1 will load a copy of location X from DRAM and will have it in state S. Upon the write operation in P1, a control message is required to invalidate the copy of location X in P0s cache. Similar steps will be taken for the access by the third processor P2. Assuming that the cacheline X will be evicted after the execution of the above example in P2, MESI implies five control messages and six accesses to DRAM in total.

– **LC Cache**

In LC Cache, P0 will read the location X first and will have it with state C. Note that in here, similar to MESI, I do not account for control messages required for atomicity of the access. Upon the write operation, the state of cacheline is changed to D and a write-back is performed on release operation. After that, P1 acquires the location and loads the cacheline from DRAM with state C. Upon write, the state of line is changed to D and no control messages are required. Note that the copy in P0s cache is not invalidated and P0 is allowed to keep his local copy. If the updated value is needed, the access should be wrapped with an acquire-release pair. In that case, the protocol guarantees that the updated value is loaded by preforming a self-invalidation right after the acquire operation. Release operation in P1 will trigger a write-back of location X to DRAM. Similar steps will be taken by P2 accessing location X. In total, LC Cache will have six DRAM accesses, like MESI, and no control messages. However, it implies that accesses to shared location should be wrapped by acquire-release pairs.

P0	P1	P2
	$r_1 = x;$	$r_1 = x;$
atomic x++;	$r_1 = x;$	$r_1 = x;$

Figure 4.3: Single producer with multiple asynchronous consumers

- **Asynchronous accesses**
  Contrary to synchronous accesses, asynchronous accesses are accesses to share locations which has not been labeled with proper synchronization operations. These accesses are considered as data race in most cases while they can be valid in relaxed algorithms [3]. For such accesses, MESI (and coherence in general), provides unnecessary restrictions that programmer has not asked for. Those restrictions, imply taking unnecessary actions which can affect the system form a performance prospective as well as energy consumption.

  I firmly believe that, hardware should prioritize energy consumption and performance of the system by taking as few actions as possible. To illustrate, consider the example of single producer with multiple asynchronous consumers shown in figure 4.3. Processor P0 is incrementing the value of location x atomically while two other cores, P1 and P2, are asynchronously reading the value of location x twice. In a system using MESI protocol, upon write in P0 other copies of data should be invalidated and then, a write-back should be performed and the cache-line needs to be reloaded upon the next read operation. On the other hand, LC would only perform one write back upon release operation, one access to DRAM for read operation(s) in each core and no control messages.

  To further explain, consider the example of figure 4.3, and let's assume that the atomic operation in P0 is executed between the two load operations in P1 and P2. In case of MESI, the cache line is first loaded in P1s and P2's caches in state S. Then P1 fetches the line from DRAM with state S. Then, P0 performs the write operation and it requires all other copies to be invalidated. Needless to say, upon the next read operation on either P1 or P2, a write-back on P0s copy should be performed and cacheline should be loaded form DRAM. In total, for the example given in figure 4.3, 6 DRAM accesses and 7 control messages are implied using MESI. In case of a system using LC cache, all the three cores will load cacheline form DRAM with state C and make 3 DRAM accesses. After that, upon write operation in P0, the state of cacheline in P0s cache become D and no invalidation or control messages are required. Read operations in P1 and P2 will result in hits. In total, LC implies 4 DRAM accesses and no control messages for the given example. Note that, the correct value to be returned by accesses to shared memory is determined by the memory model of the system. But in case of asynchronous accesses, as discussed in [REFERENCE], programming languages such as C++ [8] and Java [24] do not guarantee that the return value of racy read operations in P1 and P2 will be the most updated value produced by P0. Therefore, unlike what coherence does, there is no need for invalidating other copies.

As discussed above, the potential for an efficient implementation of LC Cache is undeniable. Coherence, as a memory consistency model, is extremely stronger than

memory consistency models currently used in programming languages like C++ and Java. In other words, a cache protocol based on coherence implies more restriction than what programmer asks for. In addition, lack of semantics for providing mutual exclusion, or creating critical sections, in coherence further highlights the need for a new cache protocol.

4.2 Weaknesses of LC Cache

The implementation of LC Cache proposed by Gao and Sarkar has several weaknesses. It would not take the advantage of cache to cache forwarding which has been used in most cache protocols nowadays. Additionally, it does not behave differently form a cache implementation based on Release Consistency [18] and most importantly, it cannot be considered a true implementation base on LC since it does not behave as LC expects. For instance, consider figure 4.4 which shows the example provided by authors in [17] to illustrate how Location Consistency behaves differently compared to Release Consistency.

P1	P2
w1: L := 1	
	acquire(L)
	w2: L := 2
	release(L)
acquire(L)	
r1: A := L	
release(L)	
r2: B := L	

Figure 4.4: An example to highlight the difference between LC and RC

The purpose of provided example is to see whether or not it is possible for two consecutive read operations, r1 and r2, in thread0 to return different values. As explained in [17], return values would be identical in a system using Release Consistency under all possible scenarios but it is possible to have different values through

Location Consistency. However, the cache protocol presented in [17] would not behave accordingly for the example presented.

Figure 4.5 shows the state transitions in a system using the original LC cache protocol, upon each instruction of the example presented in figure 4.4. Two scenario are considered in figure 4.5. The first one, shown in the second column, is the case where location L remains in P1's cache after the first write operation w1. In other words, L would be not be evicted form cache between w1 and the acquire instruction in thread0. The second scenario is the contrary case where L is evicted from P1's cache after w1. In the first scenario, the value of location L is 1 right before the acquire operation in P1 and the state of L is D. Therefore, according to figure 4.1, performing the acquire operation in P1 does not change the state of location L and the read operation r1 would return the value 1. On the other hand, in the second scenario, P1 needs to obtain the value of L since it has been evicted from its cache. Consequently, r1 will return 2 which fetched from maim memory. In both cases, the value of location L in main memory would be the same as the return value of r1 since performing a release operation on a Dirty line requires updating the memory[17]. In other words, the value produce due to the racy operation w1 is either covered by an eviction or will be propagated to the main memory.

As I discussed, return values would be identical in any possible case under a system using the implementation of LC cache proposed by Gao and Sarkar. This is clearly against the motivation of having a cache protocol based on LC. I believe such protocol should maintain a total order on writes (per location) for data race free accesses. For programs with data races, such as the example provided in figure 4.4, the protocol should let the process performing the racy access to preform operations locally, but prevent the propagation of data produced to others due to the race condition.

Instruction	L remains in P1 cache	L evicted from P1 cache
w1	L_{P1} = 1 (D) L_{P2} = (I) L_{MM} = 0	L_{P1} = 1 (D) L_{P2} = (I) L_{MM} = 0
acquire(L)	L_{P1} = 1 (D) L_{P2} = (I) L_{MM} = 0	L_{P1} = (I) L_{P2} = (I) L_{MM} = 0
w2	L_{P1} = 1 (D) L_{P2} = 2 (D) L_{MM} = 0	L_{P1} = (I) L_{P2} = 2 (D) L_{MM} = 0
release(L)	L_{P1} = 1 (D) L_{P2} = 2 (C) L_{MM} = 2	L_{P1} = (I) L_{P2} = 2 (C) L_{MM} = 2
acquire(L)	L_{P1} = 1 (D) L_{P2} = 2 (C) L_{MM} = 2	L_{P1} = (I) L_{P2} = 2 (C) L_{MM} = 2
r1	L_{P1} = 1 (D) L_{P2} = 2 (C) L_{MM} = 2	L_{P1} = 2 (C) L_{P2} = 2 (C) L_{MM} = 2
release(L)	L_{P1} = 1 (C) L_{P2} = 2 (C) L_{MM} = 1	L_{P1} = 2 (C) L_{P2} = 2 (C) L_{MM} = 2
r2	L_{P1} = 1 (C) L_{P2} = 2 (C) L_{MM} = 1	L_{P1} = 2 (C) L_{P2} = 2 (C) L_{MM} = 2

Figure 4.5: State transitions and values after each instruction in figure 4.4. $L_{P1} = 1$(D) means the value of location L in processor P1 is equal to 1 and is in state D. L_{MM} refers to the value of location L in main memory.

Chapter 5

LCCSIM: A SIMULATION FRAMEWORK TO COMPARE CACHE PROTOCOLS

In this chapter, Location Consistency Cache Simulator (LCCSim) is presented. In order to compare LC based cache protocols to well-known coherence protocols, an analytic simulation framework has been developed in Python [28] programming language. It aims simulation of cache behavior and it relies on traces of memory accesses. Such traces can be obtained using any profiling tools such as Intel's pin [23] or simulators such as gem5 [6] but for the experiments presented in this thesis, they were generated manually using the trace generator. Trace generator is also developed in Python and it takes several arguments such as total number of instructions, number of cores, the ratio of read and write operations, the total number of location being accessed and so on. Using inputs, several scenarios were created in order to test the simulator as well as comparing LC cache protocol with other coherence protocols.

5.1 Implementation details

To avoid adding unnecessary complexity to the framework, LCCSim implements on level of caches per each core all connected to one higher level of memory which is simply called main memory. The size of each processor's cache is 256 lines. Write-backs to main memory are inserted into write buffers of size 16 cachelines (words). Write buffers will merge two writes to the same location into one write operation within the buffer. After the insertion of a write-back operation into write buffer, the write buffer is flushed if it has reached the maximum fir its capacity.

Coherence protocols are all implemented as directory based protocols using one central directory which is implemented using Python's dictionaries. However, all the

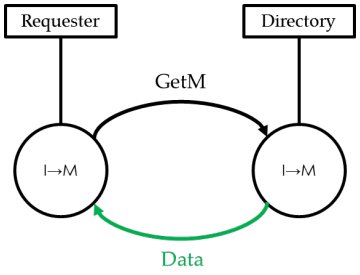
existing techniques such as can be hired for an actual implementation of LC cache in hardware. Implementations of all coherence protocols in LCCSim use write allocate and write invalidate [14][19] policies. For the replacement policy in case of evictions in caches, Least Recently Used (LRU) is implemented. Implementation detail for each cache protocol, similar to the simple implementations discussed in [31], is as follows. Note that in all figures used to explain details of coherence protocols, first action upon request is shown in black text and arrows. Green arrows and text show the second communication needed for each operation. Red arrows and messages reflect third part of a coherence transaction.

- **MSI** Transitions depending on the current state of a cache line is shown in figure 5.1. Note that three cases in a directory protocol cause three step operations as shown in figure 5.1b, 5.1c and 5.1e. Evicting a cacheline in state S can not be done silently and requires a consult with directory. The response by directory in figure 5.1a, and response from owner in figure 5.1b contains ac Ack count of zero.
- **MESI** Figure 5.2 shows state transitions for MESI directory cache coherence protocol. Like MSI, figure 5.2b, 5.2c and 5.2f show three step transitions in MESI protocol. Transition from E to M is done silently. A cacheline in state S can not be evicted silently as shown in figure 5.2i.
- **MOESI** Transitions in MOESI protocol is shown in figure 5.3 and 5.4. Note the sharing of dirty data shown in figure 5.4d and 5.3d. Similar to MSI and MESI, transition form state E to state M is silent, evicting a cacheline in state S can not be done silently and requires a consult with directory as shown in figure 5.4h.

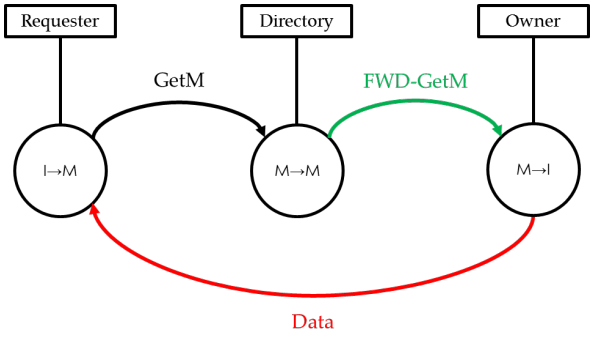
5.2 Simulator features

As we discussed, LCCSim is designed to compare cache coherence protocols to cache protocols based on Location Consistency. Therefore, it considers different aspects of the systems influenced by those cache protocols. For that purpose, the following measurements are made.

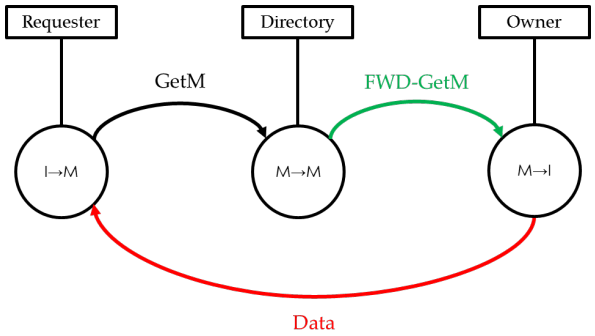
- **Access time**
  In general, one of the aspects of the system which is affected directly by cache protocol is the time it takes for the cache to satisfy the request from CPU. I call this the access time which varies significantly if the access is either a hit or a miss. In order to measure the access time, latencies for different actions needed



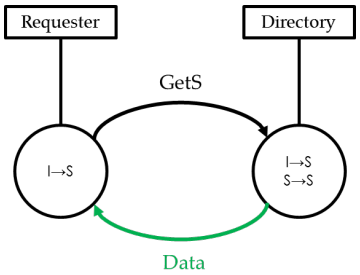
(a) I to M - No other sharer



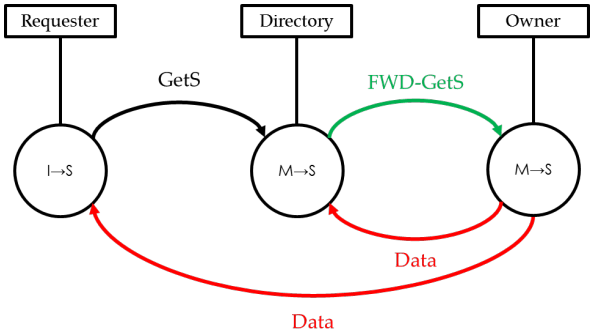
(b) I to M - Owned by some other processor



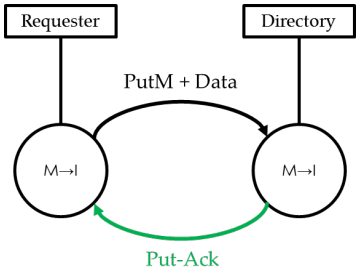
(c) I/S to M - with other sharer(s)



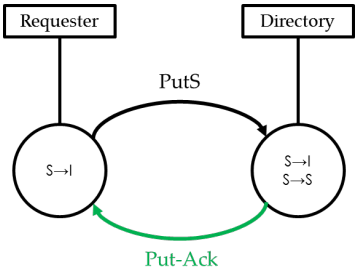
(d) I to S - No other sharer



(e) I to S - Owned by some other processor

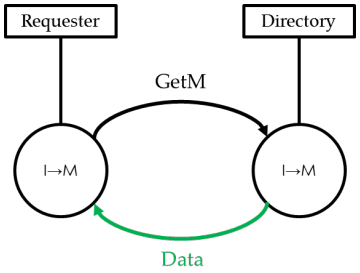


(f) M to I

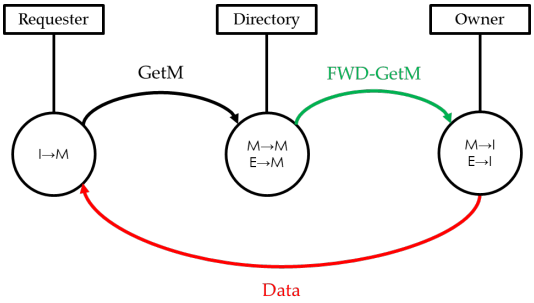


(g) S to I

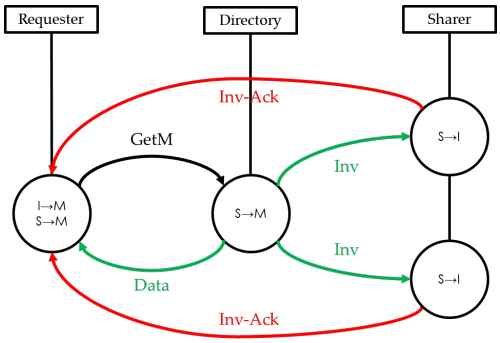
Figure 5.1: Transitions in MSI protocol



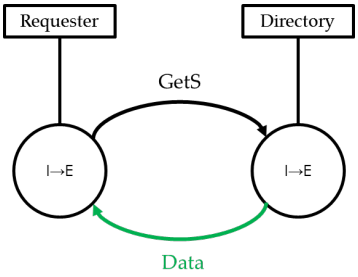
(a) I to M - No other sharer



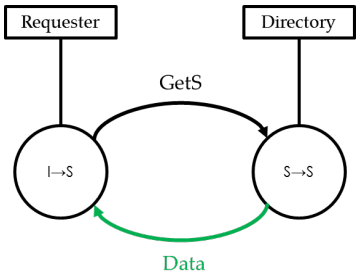
(b) I to M - Owned by some other processor



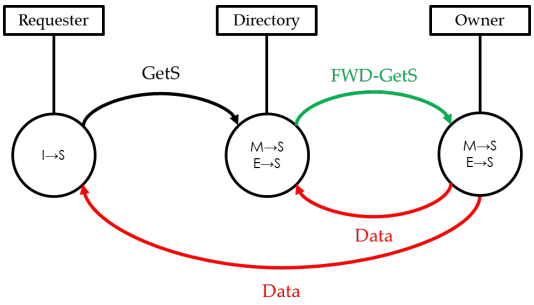
(c) I/S to M - with other sharer(s)



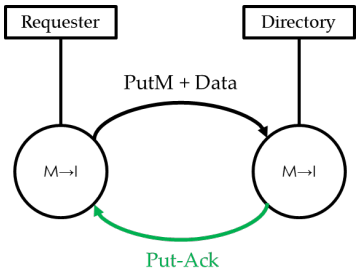
(d) I to E



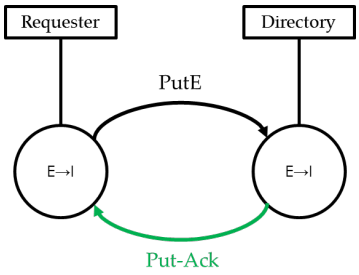
(e) I to S - No other sharer



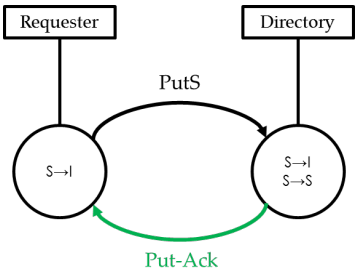
(f) I to S - Owned by some other processor



(g) M to I

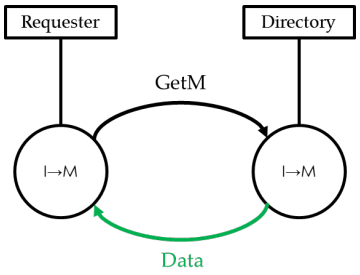


(h) E to I

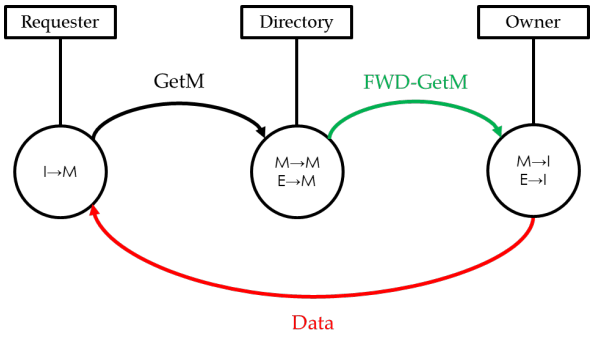


(i) S to I

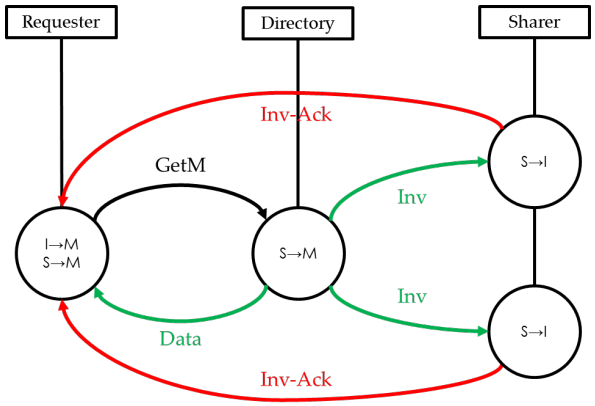
Figure 5.2: Transitions in MESI protocol



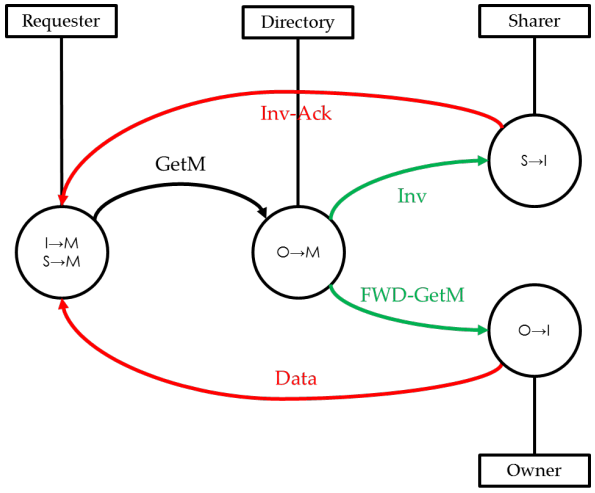
(a) I to M - No other sharer



(b) I to M - Owned by some other processor

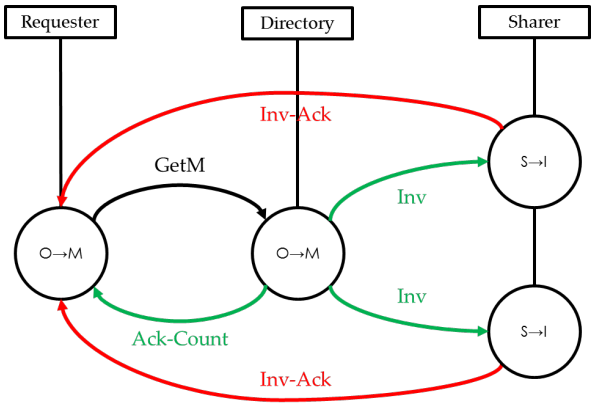


(c) I/S to M - with other sharer(s)

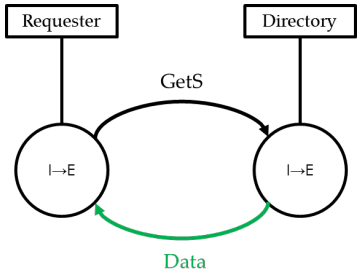


(d) I/S to M - Owned by some other processor

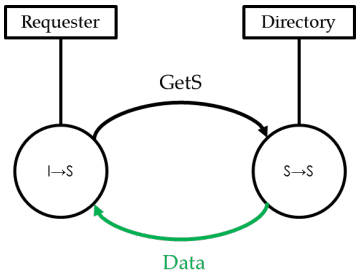
Figure 5.3: Transitions in MOESI protocol



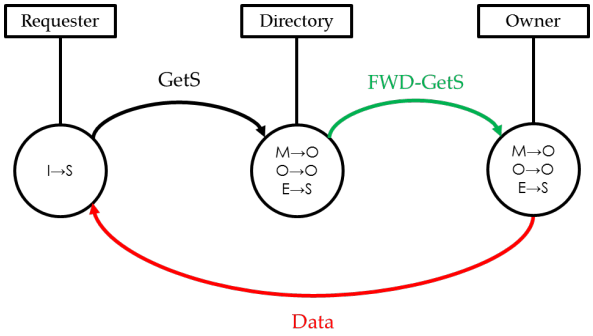
(a) M to O



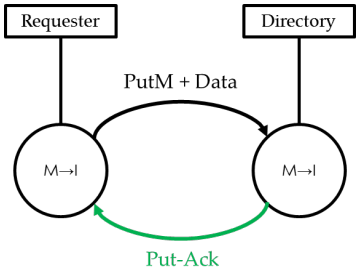
(b) I to E



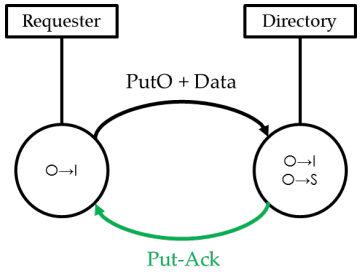
(c) I to S - No other sharer



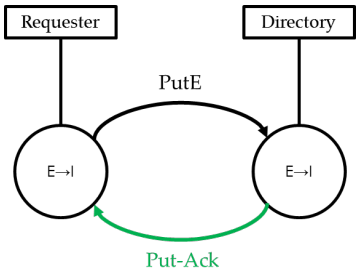
(d) I to S - Owned by some other processor



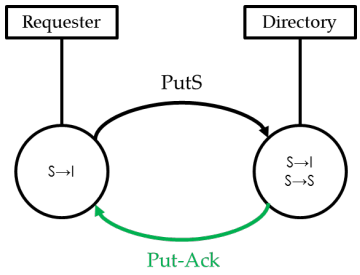
(e) M to I



(f) O to I



(g) E to I



(h) S to I

Figure 5.4: Transitions in MOESI protocol - Cont'd

to be defined. If the access is serviced by cache, *i.e.* a hit, the latency of such access is equal to 1 cycles similar to what actually happens in real hardware.

For the accesses to main memory, *i.e.* a miss which needs to be serviced by main memory, the latency is set to 50 cycles for each line. This is justified by the fact that the latency of accesses to D-RAM is around 200 cycles nowadays but every access usually fetches 4 lines from D-RAM. For data transfers between caches, *i.e.* a miss which can be serviced by a copy in some other core's cache, the latency is set to 10 cycles. Latter represents accesses known as three-hop accesses in directory protocols. For two-hop operation, where data is provided by the directory, the latency of accesses is set to 5 cycles. These numbers where chosen based on simple implementation of directory based protocols in gem5 simulator [6], but can be adjusted to reflect more realistic scenarios as explained in section 5.3.

Total access time is calculated by accumulating the access time for each of the instructions and reported at the end of a simulation.

- **On-chip traffic**
  Another important factor affected by the cache protocol is the amount of traffic on-chip generated because of the cache protocol. To take that into account, following measures were implemented within the simulator.

  - **Data transfers**
    Some cache coherence protocols allow data transfers between caches in order to lower the latency of access. The improvement is achieved by providing the line through another processor's cache rather than sending the request to the main memory. However, as in most cases, this improvement is all about a trade off.

    The access time is improved since the latency of obtaining a copy from some other core's cache is lower compared to the latency of a request from main memory. But on the other hand, this requires the data to be transfered on-chip. In addition, directory protocols transfer data between directory and cache controller in some cases[1].

    For data transfers, on-chip traffic is calculated by counting the number of transfers and multiplying them to the size of a cacheline which is considered 64 Bytes.

  - **Control messages**
    Cache protocols, whether they are snoopy or directory based protocols, imply control messages to be transfered on-chip. Those messages should be

[1] Details of such cases are explained in section 5.1

transfered between processors for snoopy protocols, and between each processor and directory for directory based protocols. Such messages also contribute to the amount of traffic on chip.

Figure 5.5 shows the control messages counted by LCCSim and the size of each messages in order to calculate the traffic on-chip. GetX, FWD-GetX, PutX, Inv, and X-Ack messages contain the address of the location and therefore 8 bytes would be large enough to fit the address in. Ack-Count should contain a number up to the number of cores in system. Thus 2 bytes can hold values for up to nearly 64 thousands of cores. Flush, FlushAll, and Flush-Ack only need to contain the identifier of the core initiating the request and 2 bytes can perfectly fit the identifier.

Control message	Size in Byte(s)
GetM	8
FWD-GetM	8
GetS	8
FWD-GetS	8
PutM	8
PutO	8
PutS	8
Put-Ack	8
Inv	8
Inv-Ack	8
Ack-Count	2
Flush	2
Flush-All	2
Flush-Ack	2

Figure 5.5: Control messages in LCCSim

5.3 Limitations of LCCSim

LCCSim has several limitations as the main purpose of its design is to investigate and compare the behavior of cache protocols. That is, the affect of the protocols on system and the state transitions for a given input rather than an accurate and detailed

simulation of caches. However, several improvements can be applied to improve the quality of measurements.

One of the limitations is having a single level of cache instead of multiple levels. Although a single level of caches may considered unrealistic but it does not affect the fairness of comparisons, which is considered as the main goal for LCCSim's design. In addition, adding multiple levels of cache will amplify the effects of cache protocol under investigation. Therefore, the result of comparisons are correct, reliable, and fair.

Another shortcoming of LCCSim is the way that processors are tiled. Currently, all processor have the same latency for their access (to other processor's cache as well as the main memory). In a more realistic scenario, latencies should be modified so that an actual configuration could be represented. For instance, in case of a 2D mesh with one centralized directory, processors should have lower latencies for communicating their neighboring processors and also different latencies for their communications with directory based on their distance form directory. However, this will not hurt the main purpose of LCCSim and all such shortcomings will be considered as part of the future work as explained in section **??**.

Chapter 6

EXPERIMENTAL RESULTS

LCCSim was discussed in details in chapter 5. This chapter presents the results of experiments made in order to compare LC Cache protocol against cache coherence protocols.

For all the experiments presented in this chapter, a random pattern of access is used. In total, four scenarios were used for running experiments. Three main categories according to the classification of memory accesses discussed earlier in section 2.1, and a fourth scenario as combination of different access types. For each scenario, following three sets of experiments were designed.

- **Scaling over the number of Cores** To observe scalability, cache protocols were evaluated as core count were doubling up starting from 1, up to 512.
- **Scaling over the ratio of write operations** To reveal the effect of write operations on cache protocols, ratio of write operations changed between %5 and %50.
- **Scaling over number of locations** Total number of locations changed from 256 to 8196, in order to measure the effect of capacity misses on each cache protocol.

Figure 6.1 shows experiments details for each of the experiment sets described above. Note that in all cases, LC Cache protocol does not generate any traffic on-chip as explained in section 4.1.

Scaling over	Number of cores	Number of locations	Ratio of writes	Number of instructions
Number of cores	1 – 512	8196	20%	10^6
Ratio of write operations	64	8196	5% - 50%	10^6
Number of locations	64	256 - 8196	20%	10^6

Figure 6.1: Experiment details

6.1 Private Accesses

This section presents the results for scenario of private accesses. That is, an access pattern without any sharing of data. Figure 6.2 shows the total latency of access for each cache protocol as number of cores is increasing. As can be seen, total latency of accesses is independent of number of cores in case of accesses to private locations. In the absence of sharing, transitions in each cache are autonomous. Total latency of accesses for MSI protocol is marginally higher compared to other protocols since writing to a cacheline in state S requires a two-step communication with directory as shown in figure 5.1a.

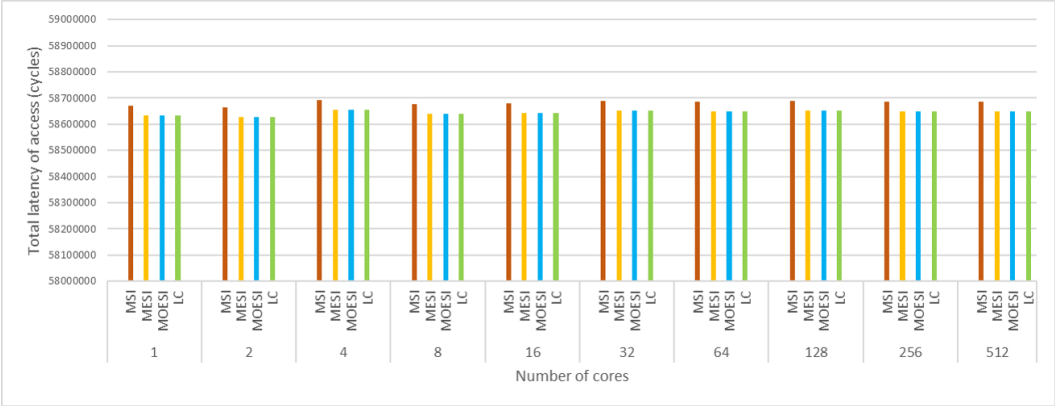


Figure 6.2: Private accesses - Total access latency over number of cores

On-chip traffic for cache protocols in case of private accesses is reported in figure 6.3. Similar to total latency, on-chip traffic do not depend on number of cores in case of private accesses. MSI protocol generates more on-chip traffic since a read-write operation requires two consults with directory.

Figure 6.4 and 6.5 show total access latency and on-chip traffic as ratio of write operations is increasing. Total latency increases as more write operations are performed since number of write-back operations required upon capacity misses is increasing. Similarly, traffic on-chip is increased since a data transfer to directory is required when a cacheline in state M is evicted (as shown in figure 5.1f).

Scaling over number of locations is shown in figure 6.6 and 6.7. Cache size in LCCSim is set to 256 lines. Therefore, for 256 locations all the memory location will be

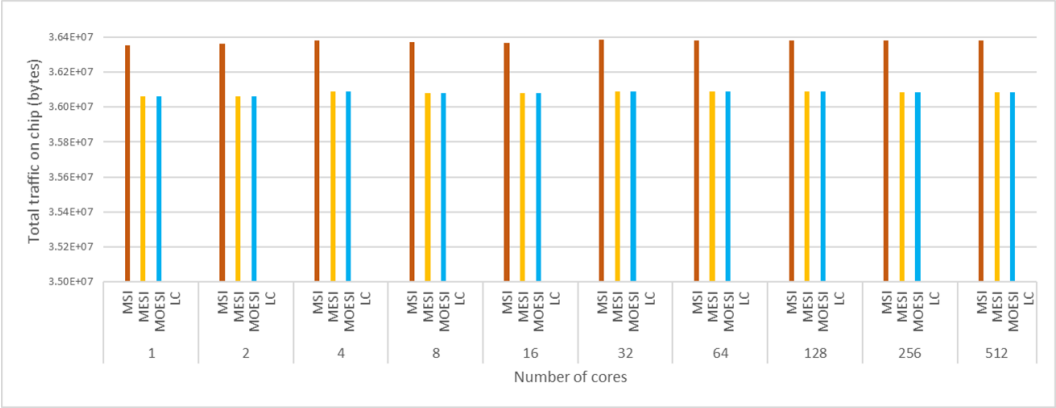


Figure 6.3: Private accesses - Traffic on-chip over number of cores

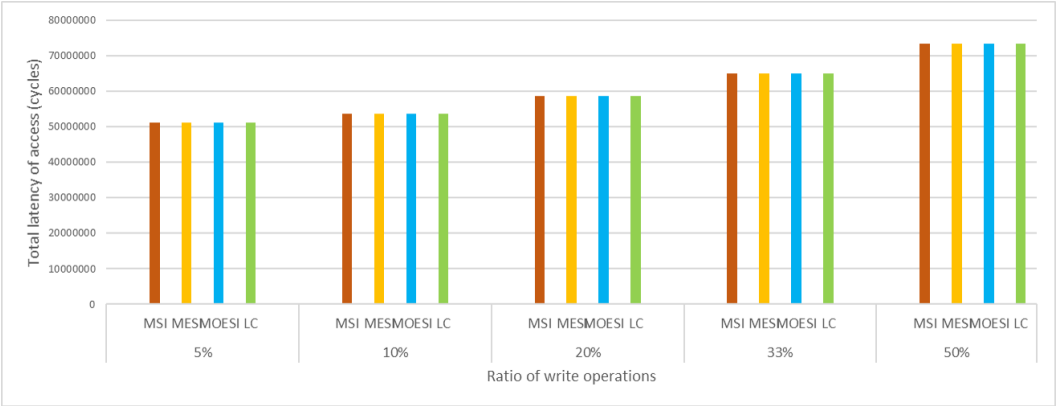


Figure 6.4: Private accesses - Total access latency over ratio of write operations

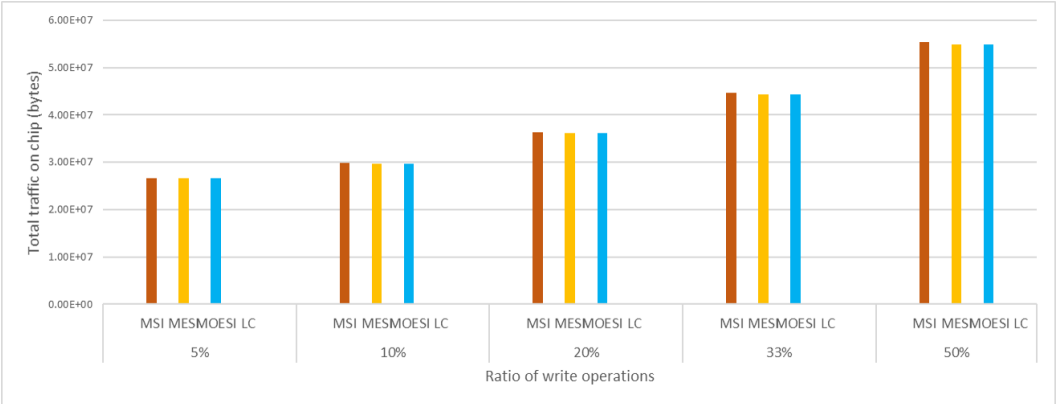


Figure 6.5: Private accesses - Traffic on-chip over ratio of write operations

present in cache and both latency and traffic cause by accesses is significantly decreased. As number of locations is increasing, latency and on-chip traffic is increased due to

reduction in locality of data. This effect is protocol independent, with the exception of MSI protocol which requires two consults with directory for a read operation followed by a write. As can be seen, this effect is amplified in accesses with higher locality (*i.e.* lower miss rates).

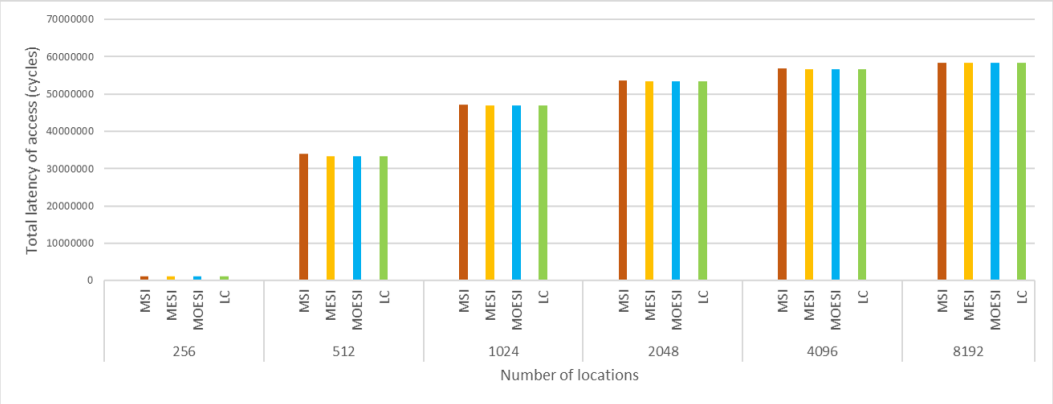


Figure 6.6: Private accesses - Total access latency over number of locations

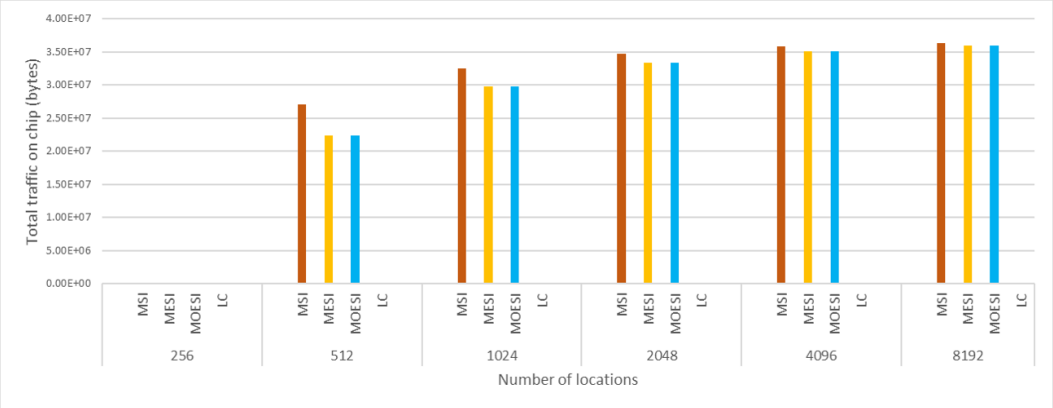


Figure 6.7: Private accesses - Traffic on-chip over number of locations

To summarize behavior of cache protocols in case of private accesses, coherence protocols shown to have similar performance considering both latency and on-chip traffic with a marginal overhead in case of MSI protocol. LC cache have the same total access latency as coherence protocols while it does not generate any traffic on-chip as discussed earlier in section 4.1.

6.2 Shared accesses - Non-synchronizing

The results for scenario of non-synchronizing shared accesses are presented in this section. Note that all accesses in under this scenario are non-synchronizing (*i.e.* racy). Figure 6.8 describes scaling of total latency of access for each cache protocols as number of cores increases. For coherence protocols, the total latency of accesses decreases by nearly 20% as number of cores doubles up. This improvement is achieved through use of cache to cache forwarding. As number of cores increases, more locations will be stored in caches and a request for those locations can be serviced through cache to cache data transfers. Data transfers has lower latency of access compared to accesses to main memory. In contrast with coherence protocols, LC does not take the advantage of cache to cache data transfers. Therefore, using LC cache protocol, the total latency of access remains unchanged as number of cores increases. This shortcoming of LC cache is discussed in section 4.2.

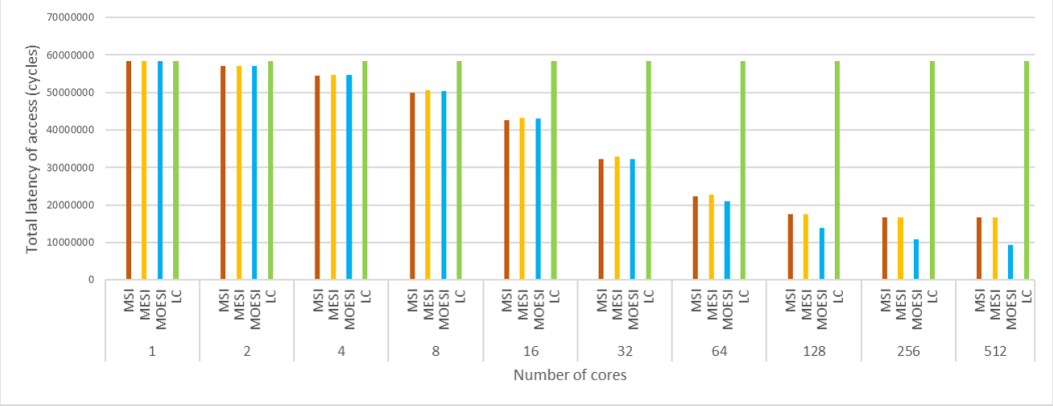


Figure 6.8: Non-synchronizing shared accesses - Total access latency over number of cores

Total traffic on chip generated by each protocol is shown in figure 6.9. The trade-off done by coherence protocols can be observed clearly. As number of cores grows, coherence protocols improve the total latency of accesses by using cache to cache data transfers. That is, more on-chip traffic as number of cores is increasing.

One observation on figure 6.8 and 6.8 is that improvements on latency, and

increase in total traffic on chip, flattens as number of cores reaches to 64. This observation does not counter previous analysis. The total number of locations for this scenario is 8192. Having 64 cores each with a cache size of 256 locations, the aggregated size of caches would be twice as the size of input data. Therefore, after handling cold misses, most accesses can be satisfied through cache to cache data transfers. In other words, the improvement technique meets its upper-bound.

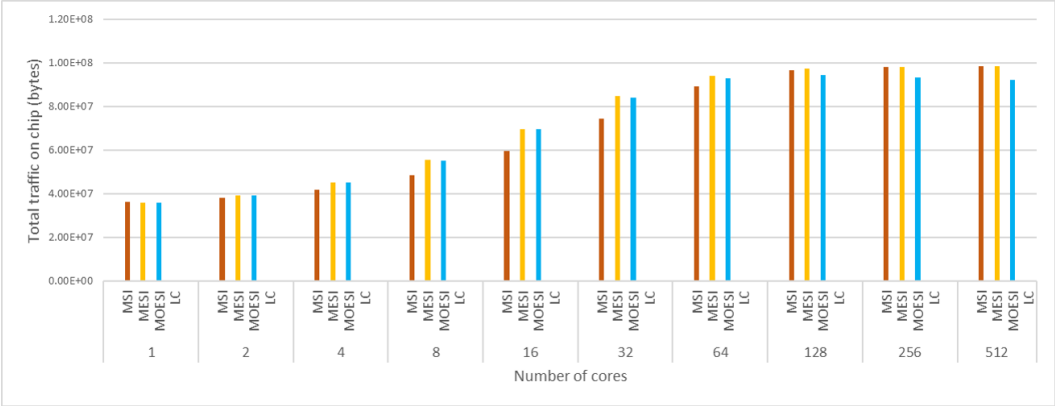


Figure 6.9: Non-synchronizing shared accesses - Traffic on-chip over number of cores

The impact of write operations on cache protocols is presented in figure 6.10 and 6.11. Total latency of accesses using LC cache protocol is significantly higher compared to cache coherence protocols since it does not use local copies in other caches. However, cache coherence protocols are more sensitive to write operations, since they impose unnecessary orderings. Note that non-synchronizing accesses require no orderings and any possible orderings should be allowed. The total latency of accesses increases by an average of %176 for coherent protocols and %41 for LC cache protocol. Moreover, MOESI protocol provides lower latency of accesses as more write operations are performed, since it allows sharing of dirty data.

With respect to on-chip traffic, MSI and MESI protocols generate %17 more traffic on average as the ratio of write operations increases from %5 to %50 while oc-chip traffic generated by MOESI protocol grows %4.

Results for scaling over number of locations is shown in figure 6.12 and 6.13. For access on more than 512 locations, LC cache has significantly higher total latency of

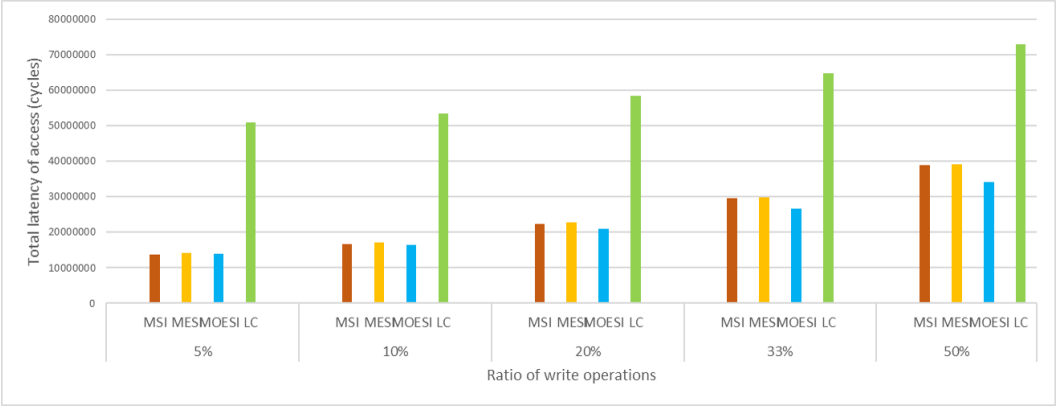


Figure 6.10: Non-synchronizing shared accesses - Total access latency over ratio of write operations

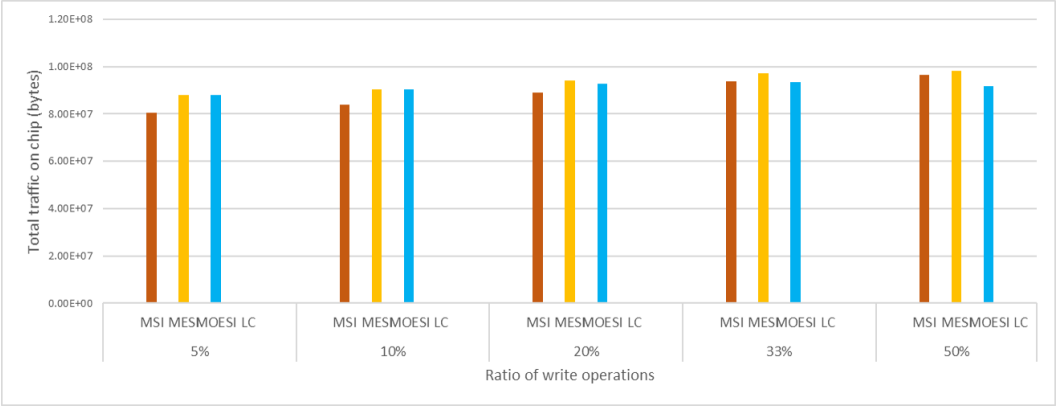


Figure 6.11: Non-synchronizing shared accesses - Traffic on-chip over ratio of write operations

accesses compared to cache coherence protocols since it does not take the advantage of cache to cache data transfers. MOESI protocol offers lower latency for accesses through dirty data sharing, and less traffic on chip as it requires less updates to directory as explained in section 2.2.3.

Data access patterns used for experiments intentionally have low locality in order to evaluate each protocol in extreme cases. LC cache protocol have higher total latency of access as it does not take the advantage of local copies in other caches. However, LC cache protocol outperforms cache coherence protocols where the locality of accesses is considerable. For instance, consider the case of 256 locations in figure 6.12 which is presented in figure 6.14.LC cache protocol offers significantly lower access time by

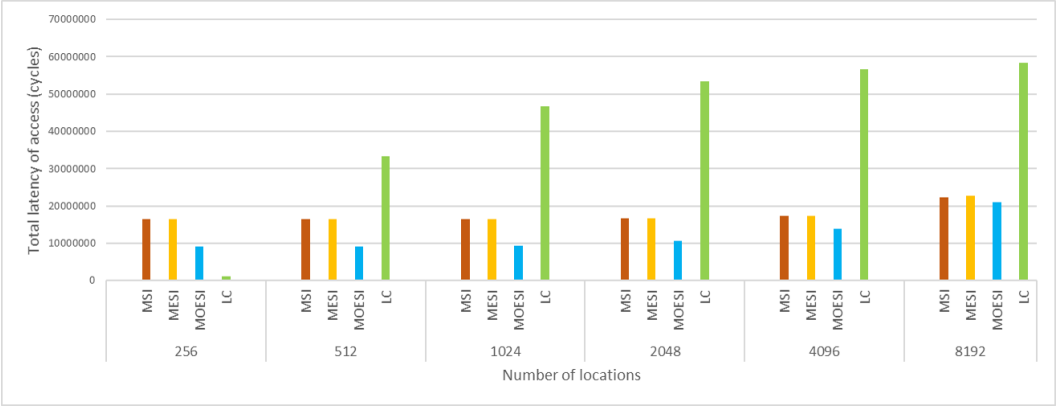


Figure 6.12: Non-synchronizing shared accesses - Total access latency over number of locations

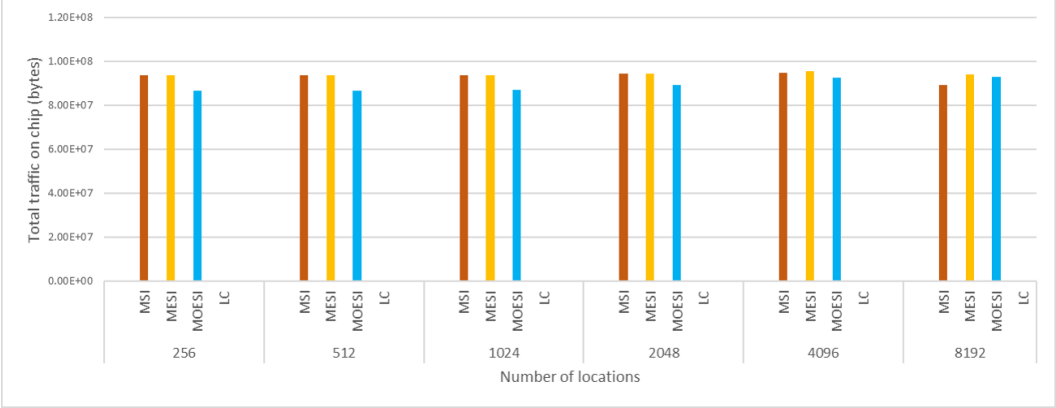


Figure 6.13: Non-synchronizing shared accesses - Traffic on-chip over number of locations

avoiding unnecessary orderings in case of non-synchronizing accesses. For 256 locations, all locations fit within one processor's cache. Therefore the advantage of cache to cache forwarding used by cache coherence protocols, is eliminated since all accesses will result in cache hits after cold misses. In this scenario, the LC cache's potential for scaling can be observed.

To summarize the behavior of cache protocols for non-synchronizing accesses, LC cache offers lower access latency compared to cache coherence protocols by avoiding unnecessary orderings (implied in case of coherent protocols) where locality of accesses is notable. However, as locality of accesses decreases, total latency of accesses for cache coherence protocols slightly increases by taking the advantage of cache to cache data

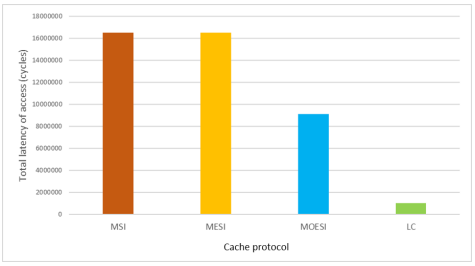


Figure 6.14: Non-synchronizing shared accesses - Total access latency for 256 locations

transfers. On the other hand, the total latency of accesses grows drastically as locality is decreasing proving the critical need for use of cache to cache forwarding in an efficient implementation of LC cache.

6.3 Shared accesses - Synchronizing

This section presents results for synchronizing shared accesses. That is, all accesses are made to shared locations, and write operations are performed as atomic operations. Figure 6.15 shows scaling of total latency of access for each cache protocols as number of cores increases. Similar to non-synchronizing accesses, cache coherence protocols achieve lower latency of accesses as core count is increasing by taking the advantage of cache to cache data transfers. LC cache protocol has higher total latency for a single core experiments as it does not rely on a directory containing a copy of locations. Moreover, as number of cores increases, LC does not achieve any improvements since it does not take the advantage of cache to cache data transfers.

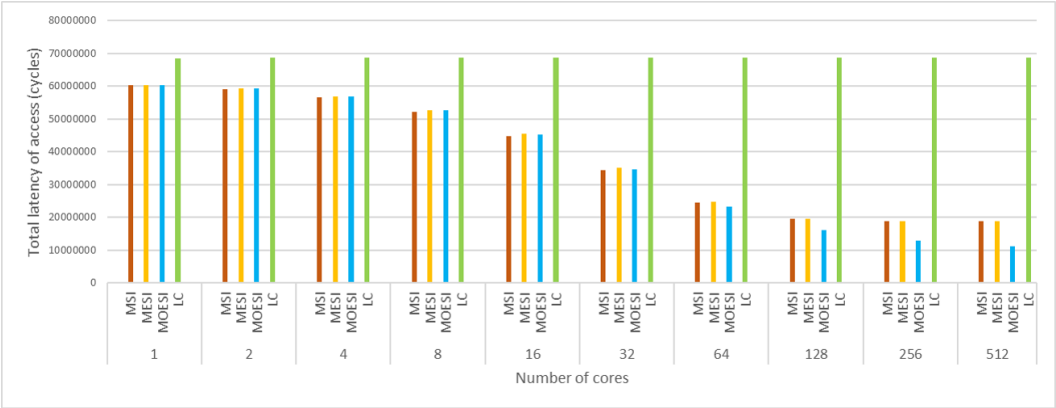


Figure 6.15: Synchronizing shared accesses - Total access latency over number of cores

With respect to the total traffic on chip presented in figure 6.16, cache coherence protocols have poor scalability for synchronizing shared accesses compared to non-synchronizing accesses. Preforming atomic operations under coherence protocols requires flushing write buffers on all other processor which leads to notable increase in on-chip traffic.

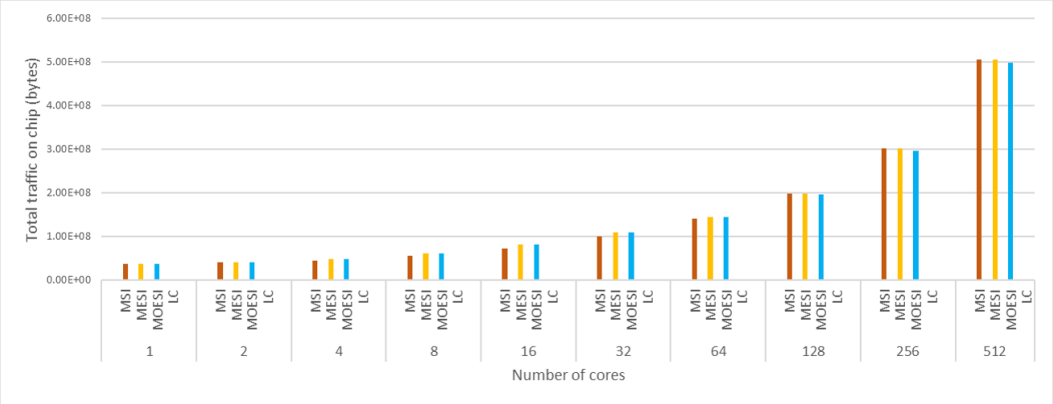


Figure 6.16: Synchronizing shared accesses - Traffic on-chip over number of cores

As shown in figure 6.17 and 6.18, the impact of write operations on cache protocols for synchronizing shared accesses is similar to non-synchronizing accesses. However, in contrast to the case of non-synchronizing accesses, LC cache protocol and cache coherence protocols would have similar sensitivity to write operation since the orderings maintained by each protocol is identical to one another.

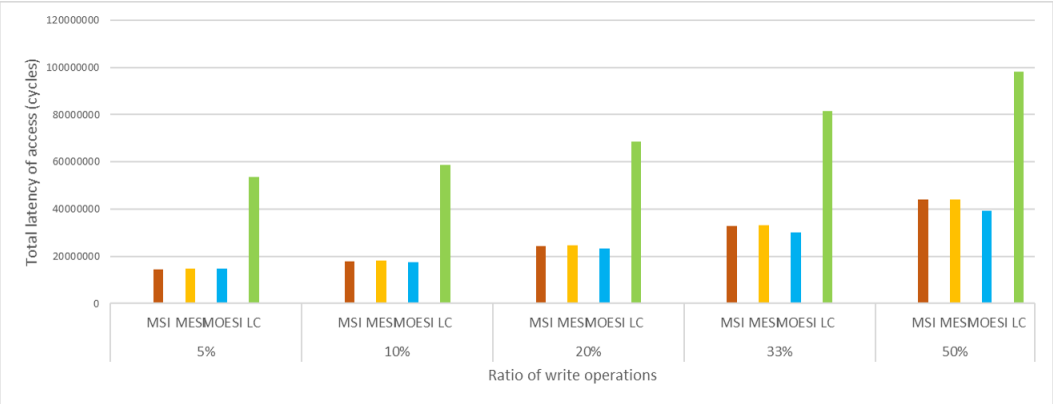


Figure 6.17: Synchronizing shared accesses - Total access latency over ratio of write operations

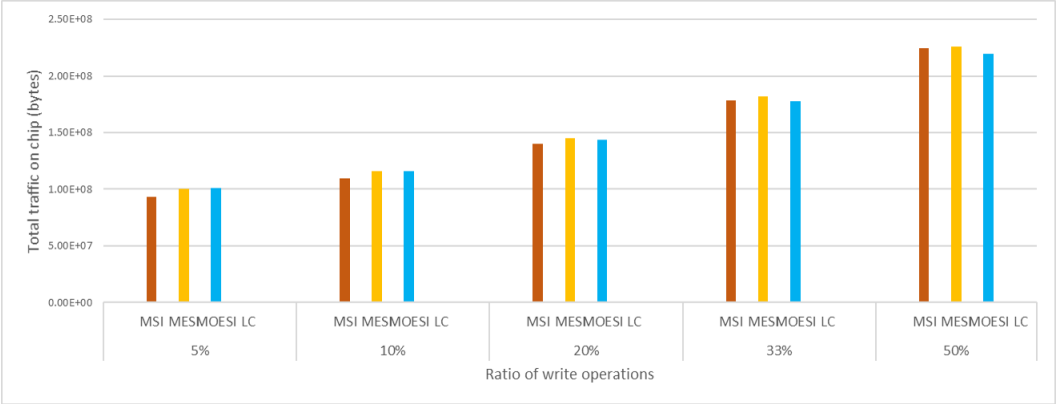


Figure 6.18: Synchronizing shared accesses - Traffic on-chip over ratio of write operations

Scaling over number of locations for synchronizing accesses is shown in figure 6.19 and 6.20. As number of location being accessed increases, the latency of accesses for LC cache protocol significantly increases since LC cache protocol does not use cache to cache forwarding.

In case of 256 locations, where all locations fit within one processor's cache, LC cache protocol outperforms MSI and MESI protocols in terms of latency of accesses and offers similar latency to MOESI protocol. In addition, LC cache protocol does not generated any traffic of chip which makes it a good candidate for exa-scale systems.

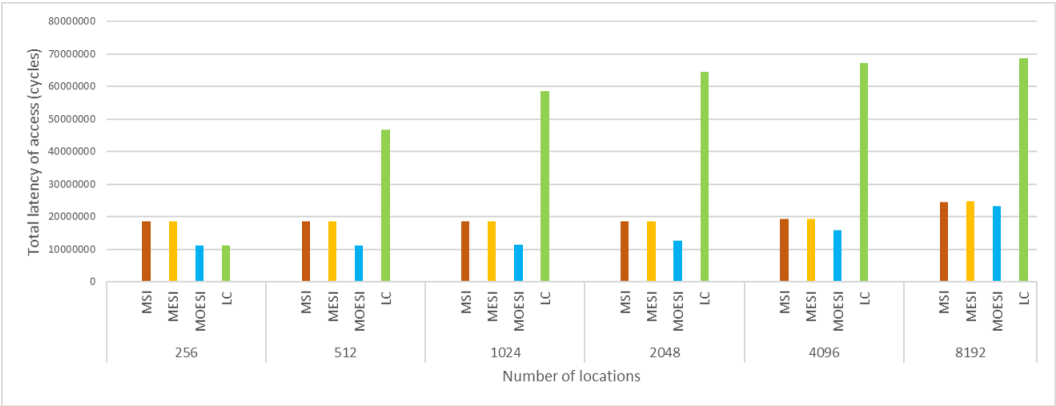


Figure 6.19: Synchronizing shared accesses - Total access latency over number of locations

To summarize, LC cache protocol shown to have higher latency of accesses in

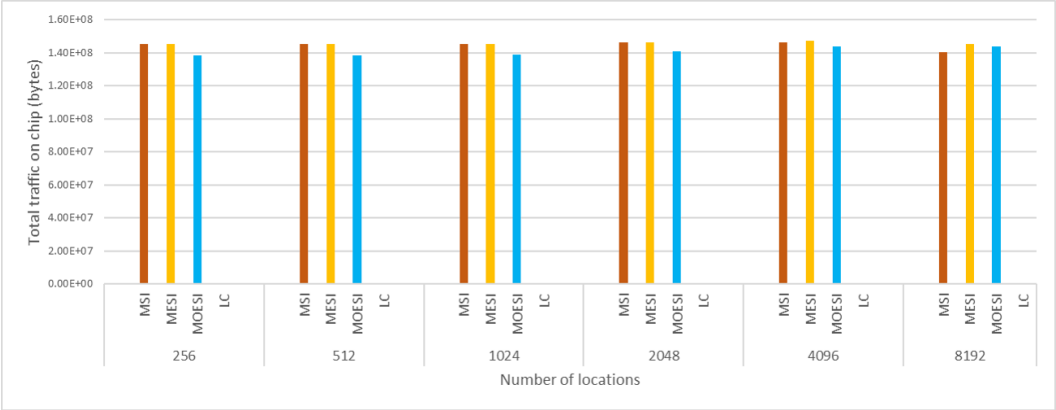


Figure 6.20: Synchronizing shared accesses - Traffic on-chip over number of locations

case of synchronizing accesses compared to cache coherence protocols. However, LC cache protocol is shown to have the potential for scaling specially if the protocol coupled with a forwarding mechanism.

6.4 Combined accesses

Previous scenarios consist of one single type of access for each, intentionally designed to reveal extreme cases. In this section, a more realistic scenario is used by combining different types of accesses. That is, %80 of accesses to private location and %20 of accesses to shared locations. To achieve a fair comparison, all accesses to shared locations are considered as synchronizing operations since LC cache benefits in case of non-synchronizing accesses by avoiding any orderings.

Figure 6.21 illustrates the scaling of total access latency as number of cores increases. Similar to previous cases, LC cache protocol has higher total latency of accesses for a single core system since it does not rely on the copies in the directory. Moreover, in contrast to cache coherence protocols, the latency of accesses does not decreases as number of cores grows since it does not take the advantage of cache to cache data transfers. However, the difference between the latency of accesses for LC cache protocol and cache coherence protocols is smaller (compared to previous scenarios) in case of combined accesses.

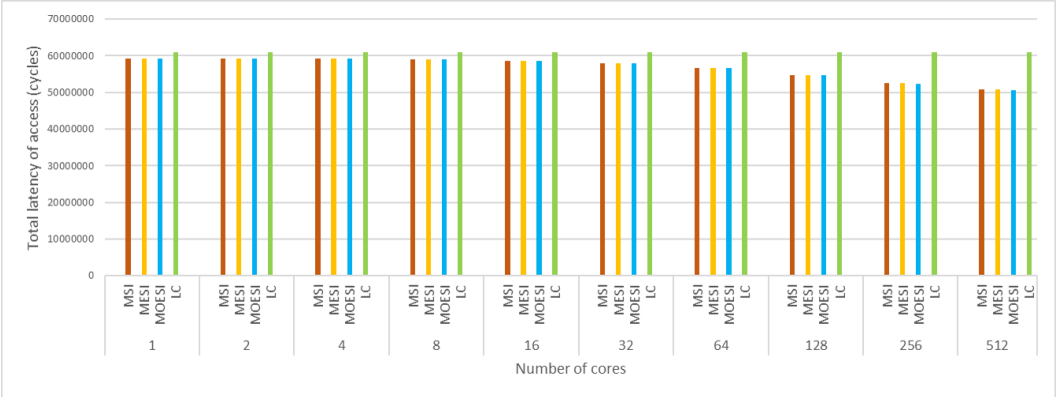


Figure 6.21: Combined accesses - Total access latency over number of cores

With respect to the total traffic on chip presented in figure 6.22, cache coherence protocols have poor scalability for combined accesses similar to synchronizing shared accesses. Preforming atomic operations under coherence protocols requires flushing write buffers on all other processor which leads to notable increase in on-chip traffic.

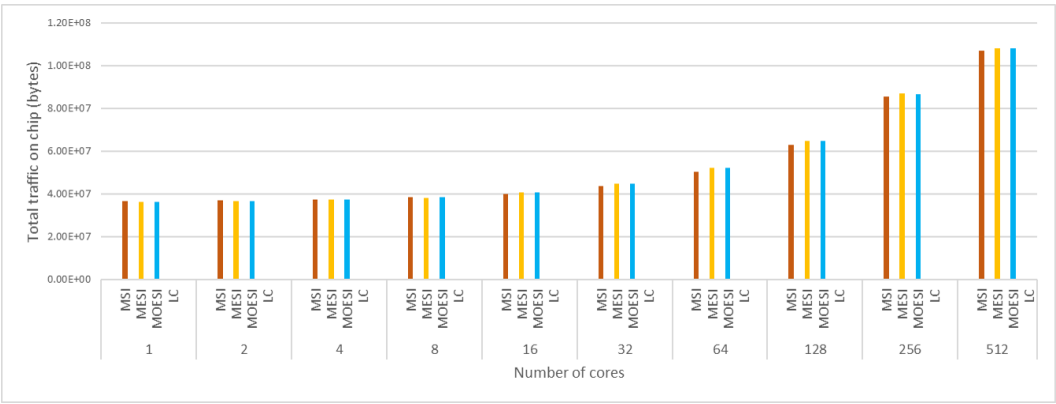


Figure 6.22: Combined accesses - Traffic on-chip over number of cores

As shown in figure 6.23 and 6.24, the impact of write operations on cache protocols for combined accesses is similar to previous shared accesses. However, in contrast to the case of non-synchronizing accesses, LC cache protocol and cache coherence protocols would have similar sensitivity to write operation since the orderings maintained by each protocol is identical to one another.

To summarize, the need for use of cache to cache forwarding in an efficient

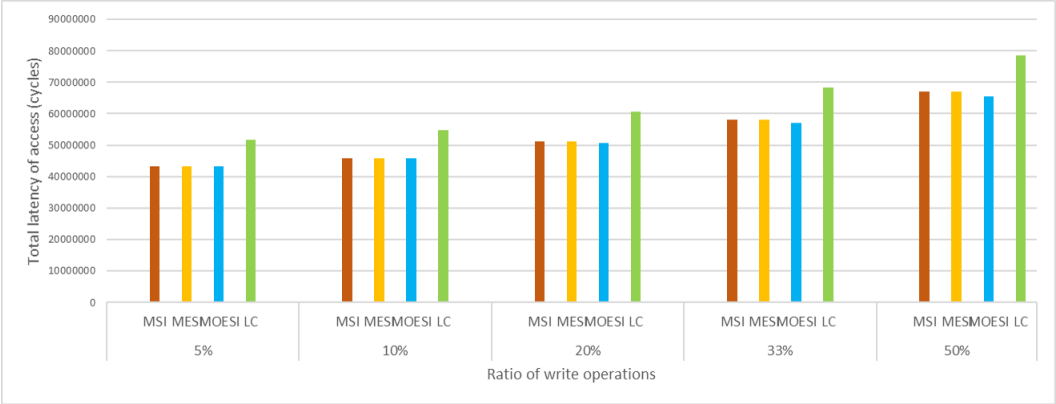


Figure 6.23: Combined accesses - Total access latency over ratio of write operations

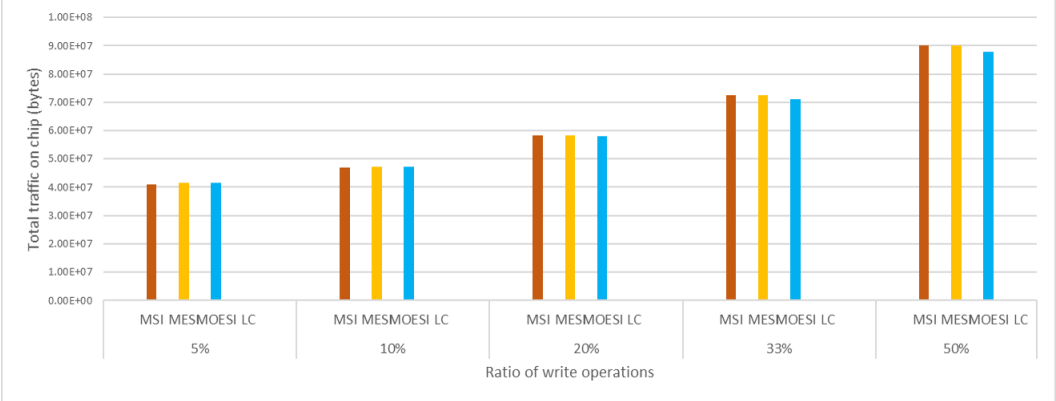


Figure 6.24: Combined accesses - Traffic on-chip over ratio of write operations

implementation of LC cache is undeniable. In addition, such protocol should carefully follow the semantics of the memory model to avoid non-deterministic behavior of memory system.

Chapter 7

RELATED WORK

The idea of using relaxed memory models along with their cache implementations is not new. Keleher et al. proposed a technique for implementing RC called Lazy Release Consistency (LRC) [20]. This model aims reducing both the number of messages and amount of data exchanged between processors. In RC, according to the definition, all ordinary Load/Store accesses must be performed before a release operation is allowed to be performed with respect to any other processor. Therefore, the processor issuing the release operation should send an update to all other processors acquiring the same memory location later in the program order upon every write.

In LRC, both the number of messages and data movements can be reduced by merging all the updates into one message. Also, instead of broadcasting the updates to all processor, LRC only sends updates to the next processor acquiring the location. In other words, a processor acquiring a location should observe all the modifications preceding the acquire operation. First improvement requires buffering the writes in order to merge them all into a single message at the time of release. The second proposed technique, requires tracking of the processor performing the preceding release operation.

Bershad and Zekauskas proposed a framework for writing and executing shared memory parallel programs on distributed shared memory multiprocessors called Midway along with a memory model called Entry Consistency (EC) [5]. EC is similar to LRC as updates in EC arrive at the acquiring processor like LRC. In contrast with LRC, EC requires shared data to be associated with a synchronization variable. The orderings are defined based on the accesses type(*e.g.* exclusive or non-exclusive). Adve

et al. compared implementations of LRC and EC for various workloads and concluded that neither EC nor LRC consistently outperforms the other [2].

Several works aimed at determining the impact of cache coherence protocols on computer systems. Schweizer et al. evaluated the effect of atomic operations on many modern architectures [30]. They have illustrated that hardware implementations of atomic operations prevents instructions-level parallelism even in the absence of dependencies between two instructions.

Martin et al. claim that on-chip hardware coherence scales well as number of cores increases. They considered the effect of coherence on traffic, storage, latency, inclusion property, and energy consumption of the system. For the analysis on traffic, only the traffic generated in case of misses has been discussed while coherence incur on-chip traffic in other cases. With respect to storage analysis, authors assumed a hierarchal organization for processors with equal dimensions in depth and width. In other words, they assumed that a hierarchal organization with N cores would have at most $log(N)$ cores at the lowest level. With this assumption, and with use of distributed directories, $O(log(N))$ storage space is required at directory for each cacheline. Authors concluded that the storage overhead for coherence protocols would not exceed %5 of the total storage required by caches. While this conclusion is valid under the assumption explained, the assumption may not be applicable to all future architectures.

Das et al. proposed use of dynamic directories in multi-core systems to reduce power consumed by on-chip interconnect [11]. Schuchhardt et al evaluated the impact of dynamic directories over the energy consumption of system as well performance [29].

Aiming at designing a scalable coherence protocol, Tardis was proposed by Yu and Devadas [35]. Tardis relies on logical timestamps for memory orderings. It provides better scalability by requiring $O(log(N))$ storage to keep track of N sharers. But it implies overhead for storing the timestamps and incurs communication overhead in some cases.

Chapter 8

CONCLUSION AND FUTURE WORK

Due to physical constrains in fabrication process, future architectures deemed to be even more parallel. Traditional cache coherence protocols face many challenges to scale to these architectures, and cache coherence soon becomes a luxury to afford. As an alternative, LC cache protocol was investigated in this thesis.

The LC cache protocol guarantees that atomicity can be preserved when there is a need for synchronized access to memory, while proposing a much weaker memory consistency model than cache coherence and its most popular implementations (e.g., MESI, MOESI). However, LC cache protocol has several shortcomings. It does not take the advantage of cache to cache forwarding, which has been used in most cache protocols nowadays. Additionally, it cannot be considered a true implementation base on LC since it does not behave as LC expects. Thu, the possibility of using such protocol is eliminated since the cache protocol does not follow the memory model of the system.

Future work includes designing a new cache protocol based on LC cache protocol. Such protocol should provide support for cache to cache transfers, and also behave according to what the memory model defines. The model should be added to LCCSim in order to be compared with other protocols.

REFERENCES

[1] Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, August 2010.

[2] Sarita V. Adve, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Proceedings of the 2Nd IEEE Symposium on High-Performance Computer Architecture*, HPCA '96, pages 26–, Washington, DC, USA, 1996. IEEE Computer Society.

[3] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[4] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. *SIGARCH Comput. Archit. News*, 18(2SI):2–14, May 1990.

[5] Brian N Bershad and Matthew J Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. 1991.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[8] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, June 2008.

[9] Zoran Budimlić, Vincent Cavé, Raghavan Raman, Jun Shirako, Saugnak Tacsirlar, Jisheng Zhao, and Vivek Sarkar. The design and implementation of the habanero-java parallel programming language. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 185–186, New York, NY, USA, 2011. ACM.

[10] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[11] Abhishek Das, Matt Schuchhardt, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 479–484, San Jose, CA, USA, 2012. EDA Consortium.

[12] J. Diaz, C. Muoz-Caro, and A. Nio. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, Aug 2012.

[13] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *SIGARCH Comput. Archit. News*, 14(2):434–442, May 1986.

[14] Michel Dubois, Murali Annavaram, and Per Stenstrm. *Parallel Computer Organization and Design*. Cambridge University Press, New York, NY, USA, 2012.

[15] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[16] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. 1993.

[17] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Trans. Comput.*, 49(8):798–813, August 2000.

[18] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(2SI):15–26, May 1990.

[19] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[20] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Comput. Archit. News*, 20(2):13–21, April 1992.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[22] Charles E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[24] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, January 2005.

[25] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[26] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, January 1993.

[27] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984.

[28] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[29] Matthew Schuchhardt, Abhishek Das, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. The impact of dynamic directories on multicore interconnects. *Computer*, 46(10):32–39, October 2013.

[30] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 445–456, Washington, DC, USA, 2015. IEEE Computer Society.

[31] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[32] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[33] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[34] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[35] Xiangyao Yu and Srinivas Devadas. Tardis: Time traveling coherence algorithm for distributed shared memory. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 227–240, Washington, DC, USA, 2015. IEEE Computer Society.