Modeling and Evaluation of Cache Coherence Mechanisms for Multicore Processors

نمذجة وتقييم طرق تماسكية الذاكرة المخبئة في المعالجات متعددة الأنوية

by

Malik Amin Mohammad Al-Manasia

Supervisor

Dr. Faruq Al-Omari

Co-Supervisor

Dr. Mohammad Al-Jarrah

Program: Industrial Automation Engineering

April 17, 2011

Modeling and Evaluation of Cache Coherence Mechanisms for Multicore Processors

by

Malik Amin Mohammad Al-Manasia

B.Sc. Computer Engineering, Mutah University, 2008

A thesis submitted in partial fulfillment of the requirements for the degree of Master *of Science in the Department of Computer Engineering*, Yarmouk University, Irbid, Jordan

Approved by:

Faruq A. Al-Omari .....**Chairman**

Associate Professor of Computer Engineering, Yarmouk University

Salem Y. Al-Agtash .....**Member**

Associate Professor of Computer Engineering, Yarmouk University

Hussien R. Al zoubi .....**Member**

Assistant Professor of Computer Engineering, Yarmouk University

Emad M. Shawakfa ..... **Member**

Assistant Professor of Computer Information System, Yarmouk University

April 17, 2011

ACKNOWLEDGMENTS

Thank God that I have received a lot of help and support from many people over the past few months. I cannot imagine making it this far without such support. I am deeply grateful to my supervisors, Dr. Faruq Al-Omari and Dr. Mohammad Al-Jarrah. They have been outstanding mentors who provided me with a lot of support and guidance. I have learned a lot under their supervision. I am thankful that they gave me as much of their time as I asked for, and provided me with support when I needed it the most.

My research required many hours of simulation time, and I would like to acknowledge those who helped made it possible. I thank Virtutech AB and Wind River "the companies that support the Simics simulator which we used in the scope of this thesis" for answering many of my technical questions. I would also like to thank the Condor project from Wisconsin University for their support.

My heartfelt thanks go to my mother, to my father, to my siblings, and to my friends. This work could not have been accomplished without their support.

April 2011

TABLE OF CONTENTS

Table of Contents **Page**

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Almanasia, Malik Amin. Modeling and Evaluation of Cache Coherence Mechanisms for Multicore Processors. Master of Science Thesis, Department of Computer Engineering, Yarmouk University, 2011 (Supervisor: Dr. Faruq Al-Omari and Dr. Mohammad Al-Jarrah).

Multiple core designs have become commonplace in the processor marketplace, and are therefore a major focus in modern computer architecture research. Thus, for both product development and research, multiple core processor performance evaluation is a mandatory step in marketplace. A well-known positive feedback property of computer design is that we use computers of today to design more powerful computers for the future. Thus, with the appearance of Chip Multi-Processors (CMP), it is more natural to take advantage of its efficiency.

Multicore computing have presented many challenges for system designers; one of which is data consistency between a shared cache or memory and the local caches of the chip. This is also known as cache coherency. The cache coherence mechanisms are a key component in the direction of accomplishing the goal of continuing exponential performance growth through widespread thread-level parallelism.

In the scope of this research, we have studied the available efficient methods and protocols used to achieve cache coherence in multicore architectures. These protocols were further modeled and evaluated utilizing simics simulator for multicore architectures. We also investigated the weaknesses and strengths of different protocols and discussed the way of improving them.

Key Words: Cache Coherence Protocols, Multicore, Snooping, Directory, Token.

الملخص

المناسيه، مالك أمين. نمذجة وتقييم طرق تماسكية الذاكرة المخبئه في المعالجات متعددة الأنويه. رسالة ماجستير بجامعة اليرموك. 2011. (المشرف: د.فاروق العمري، المشرف المشارك: د.محمد الجراح)

أصبحت المعالجات متعددة الأنوية منتشرة بشكل واضح في الأسواق، وأصبح هناك تركيز على أبحاث هيكلية الحاسوب الحديثة. ولتطوير هذا المنتج والأبحاث المتعلقة به صار من الضروري إجراء تقييم أداء للمعالجات متعددة الأنوية لنسير خطوة نحو الأمام. ولحسن الحظ انه يمكننا الإستفادة من الحواسيب العالية الآداء حاليا لإنتاج حواسيب أفضل آداء للمستقبل.

قدمت الأنظمة متعددة الأنوية عدد من التحديات لمصممي الأنظمة، وتعتبر تماسكية وترابط البيانات بين الذاكرة المخبئة المشتركة او الذاكرة الرئيسية و الذاكرة المخبئة المحلية احد أهم هذه التحديات. يؤدي تحقيق هذا للوصول الى زيادة كبيرة في الآداء والى الاستفادة من العمليات المتوازية بشكل أفضل.

في نطاق هذه الرسالة قمنا بدراسة الطرق والبروتوكولات الأكثر فاعلية لتحقيق تماسكية الذاكرة المخبئة في المعالجات متعددة الأنوية. قمنا بنمذجة وتقييم هذه البروتوكولات باستخدام أحد أفضل برامج المحاكاة المتوفرة يسمى simics. ايضا قمنا ببحث ودراسة نقاط الضعف والقوة لكل هذه البروتوكولات وكيفية جعلها أفضل.

الكلمات المفتاحية: تماسكية الذاكرة المخبئة، المعالجات متعددة الأنوية.

Chapter 1 Introduction

Four Intel Corporation technologists have established multicore processors to come to market after the turn of the 20th century, who forecasted the future through the lens of Moore's Law. This was in the 1989 issue of IEEE Spectrum, an article entitled "Microprocessors Circa 2000" [1]. After fifteen years their predictions are verifying true and multicore processor capability development have turned out to be one of the top business and product initiatives for Intel and other companies.

1.1 Cache Coherence and Multicore

There is a great correlation between power and processor clock rate. When the clock rate is enhanced, the power will also rise; after which, the temperatures will also increase [2]. Multicore processors take advantage of this relationship by combining multiple cores. Each core is able to run at a lower frequency. By splitting up the power "provided to a single core" normally between all cores [3], the performance will enhance, whereas the power and temperatures are still under control. Figure 1.1 shows this main advantage and the significant performance enhancement over the single core processor [2, 4].

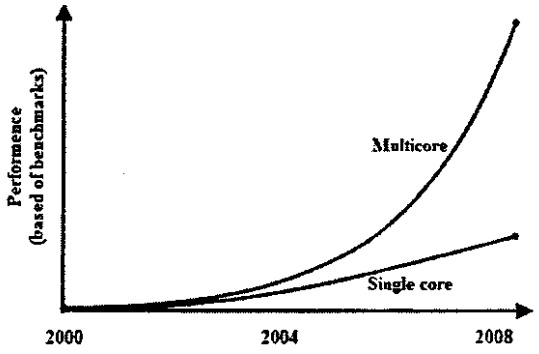


Figure 1.1 Multi core performance compared to single core.[2]

Also, multicore technology can improve system efficiency and application performance for computers running multiple applications and multi-threaded programs simultaneously [5, 6].

Computing has revolutionized society and worked as an engine of the world's economy. Much of this revolution is the result of the advent and incredible progress of the low-cost microprocessor [7].

Enhancement of microprocessors is guided greatly by Moore's Law, which has forecasted that the number of transistors per silicon area will double every eighteen months [8]. Computer architects are embarking on a fundamental shift in how the transistor bounty is used to increase performance, while Moore's Law is expected to continue at least into the next decade [111]. Figure 1.2 illustrates the transistor count has been doubling every two years.

Regarding multicore processors, cache coherency stands for the credibility of data stored in each core's cache. Multicore processors may contain distributed and shared caches on the chip, so we should account for the coherence protocols to assure that when a core reads from memory, it reads the current piece of data and not a value that has been updated by a different core.

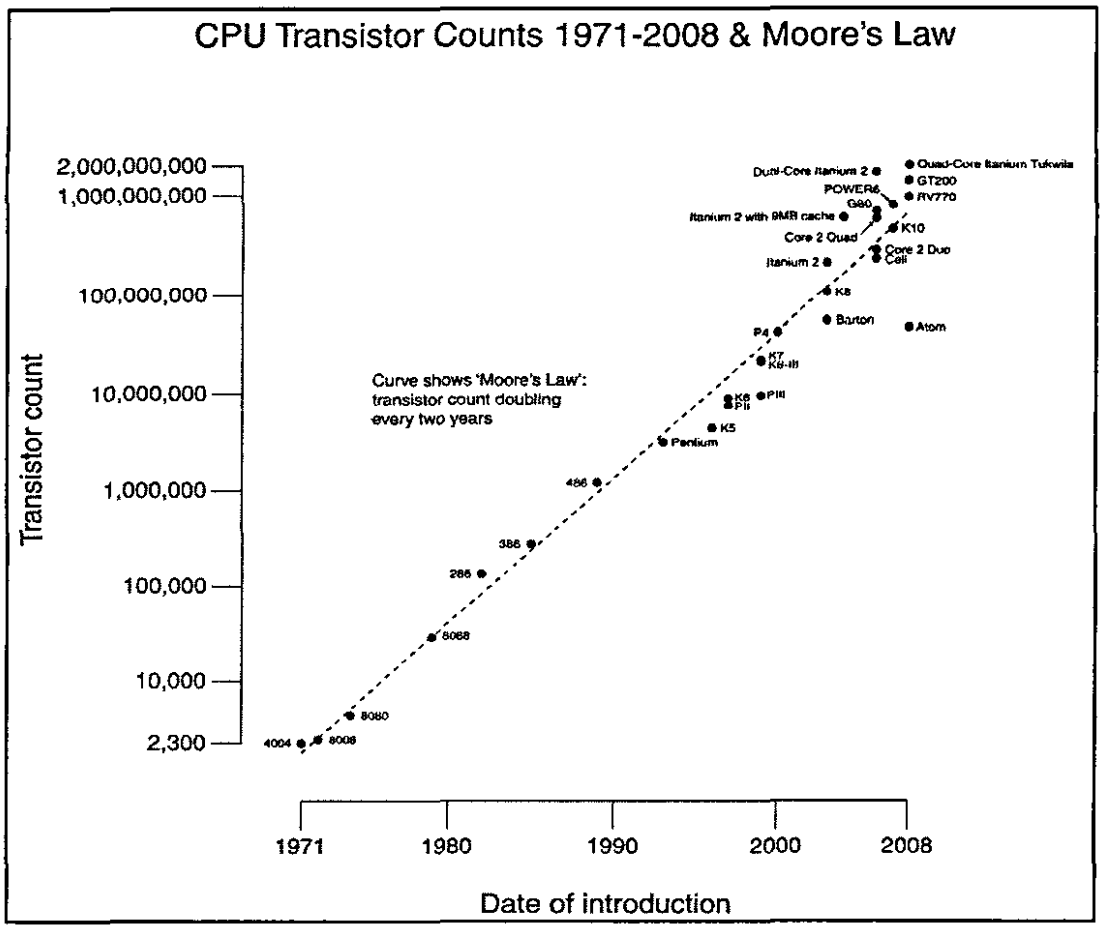


Figure 1.2. Moore's law for a number of IC intensities [8].

Cache coherence is considered to be one of the most important dilemmas in a multicore environment due to distributed Level1(L1) and sometimes Level2 (L2) caches. The copy of the data in any cache may not always be the most recent version, as each core has its own cache. For example, consider a quad-core processor where each core fetched a certain block of memory into its private cache. One core writes a value to a specific location; when the second core tries to read that value from its cache, it might not have the updated copy unless its cache entry is invalidated and a cache miss occurs. This cache miss forces the updating of the other core's cache entry. If this coherence policy wasn't present, garbage data would be read and invalid results would be produced, this may lead to crashing the program or the entire system.

There are two ways to deal with the cache coherency problems.

- Software Cache Coherence Schemes.
- Hardware Cache Coherence Schemes.

In the Software-Based Coherence, which is a straightforward method, shared data are not cached [9]; this can be made by the operating system, the compiler, or the programmer.

Hardware-Based Coherence is one, which is enforced by snoop devices attached to the cores and their caches. In this scheme, because caches are guaranteed to be coherent, shared data can be cached, but a programmer must deal with the synchronization of shared data [9].

In hardware-based cache coherency, there are in general two methods, a snooping protocol and a directory-based protocol. The Snoopy cache-coherence methods require sending information to all of cache controllers. However, if the number of cores increases the cache messages will also increase, then the required bus, "which connect the caches and all messages will pass through it", bandwidth will be bigger than the available one, and then a total saturation of the bus bandwidth would occur. These techniques can be used in small-scale systems due to this limitation.

On the other hand, the Directory-based protocol will scale to larger number of processors or cores than the snoopy-based coherence protocol, since it enables multiple coherence actions to take place at the same time [9].

Over the last 10 years, much work have been done to enhance cache coherency performance, this has resulted in a number of new cache coherency protocols. Token coherence protocol, is one of the most efficient new protocols.

The recently suggested Token coherence protocol [10, 11, 12] can remove the constraint of directory indirection without sacrificing either the decoupling of the

interconnection from the coherence protocol or the decoupling of coherence from consistency. To resolve races without asking for a home node or an ordered interconnection; Token coherence uses token counting. As will be discussed in details in chapter four.

1.2 Desirable Cache Coherence Attributes

There are three critical attributes that have an impact on the performance of any cache coherence protocol:

1. **Low-latency Cache-to-Cache Misses**

Many commercial workloads exhibit abundant thread-level parallelism, and thus, using multiple processors or cores is a desirable method for enhancing their performance [11]. To efficiently support the frequent communication and synchronization in these workloads, systems are required to optimize the latency of cache-to-cache misses [13]. A cache-to-cache miss is a miss frequently caused by accessing shared data that requires another processor's cache to provide that data. To decrease the latency of cache-to-cache misses, a coherence protocol should ideally support direct cache-to-cache misses. For example, snooping protocols support fast cache-to-cache misses by broadcasting all requests to find the responder directly. On the contrary, by placing a directory lookup and a third interconnect traversal on the critical path of cache-to-cache misses, directory protocols indirectly locate remote data.

2. **No Reliance on a Bus or Bus-like Interconnect**

Unfortunately, snooping protocols depend on a bus or bus-like interconnect to enable their fast cache-to-cache transfers. Such interconnects are not as good as with two important technology trends: high-speed point-to-point links and increasing levels of integration. As discussed below shortly, creating a bus-like or "virtual bus"

interconnect demands it to give a total order of requests [11]. An interconnect gives a total order if all messages are delivered to all destinations in some order. A total order demands an ordering between all the messages (even those from different sources or sent to different destinations). For example, if any processor receives message A before message B, then no processor receives message B before A. Unluckily, creating a totally-ordered interconnect that uses both of the two important technology trends described below is infeasible using the traditional techniques. Because of that, protocols depending on a totally-ordered interconnect—such as snooping protocols— are unattractive, and protocols that do not depend on such an interconnect—such as most directory protocols—are more desirable [11].

- High-speed point-to-point links.

Continued scaling of the bandwidth of shared-wire buses is hard due to electrical implementation realities [14]. To overcome this limitation, some multiprocessor systems replace shared-wire buses with high-speed point-to-point links that can provide considerably more bandwidth per pin than shared-wire buses [15]. Many recent snooping protocols use virtual bus switched interconnects that exploit high-speed point-to-point links, even though; many early snooping systems depended on shared-wire buses. These interconnects provide the bus-like ordering properties necessary for snooping, often by ordering all requests at the root switch chip.

- Higher levels of integration.

Moore's Law predicted enhanced number of transistors per chip, this has encouraged, and will continue to encourage, more integrated designs, making "glue" logic (e.g., dedicated switch chips) less desirable. Many systems integrate processor(s), cache(s), coherence logic, switch logic, and memory controller(s) on a single die (e.g.,

AMD's Hammer [16]). Connecting these highly-integrated nodes directly leads to a high bandwidth, low-cost, low-latency "glueless" interconnect [11].

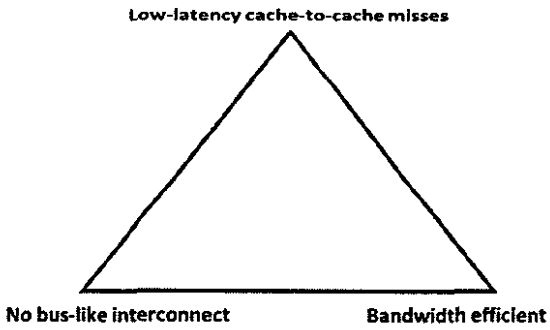
3. Bandwidth Efficiency

Bandwidth efficiency is the third—and almost the least important at the time—required attribute [11]. A cache coherence protocol should conserve bandwidth to decrease the cost and avoid interconnect contention (since contention reduces performance), but a protocol should not sacrifice any of the first two attributes for the sake of obtaining this less-important third attribute. For example, an essay estimated that, less than 10 systems contained 256 or more processors (~ 0.03%) of the 30,000 Origin 200/2000[1] [17] systems shipped, and less than 250 of the systems had 128 processors or more (~ 1%) [11]. No new multicore processors exceeded one hundred cores per chip [18].

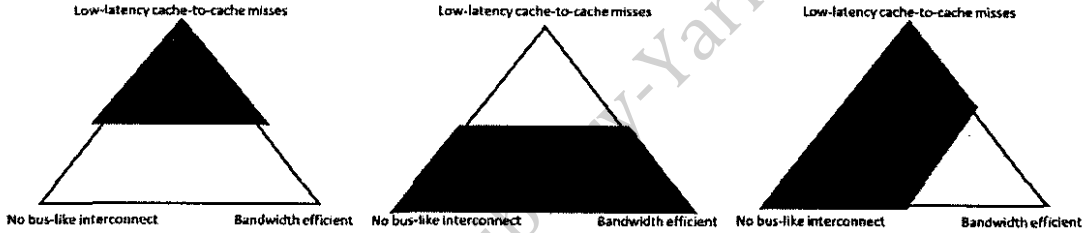
As exhibited in Figure 1.3, neither predominant approach to coherence captures all three of these attributes: two are captured by directory protocols and TokenB while one is captured by snooping.

Figure 1.3 exhibits the three required properties described in this section and which of these attributes are supported by snooping (part a), directory protocols (part b), and TokenB protocol (part c). Each triangle stands for a different protocol, and each vertex accounts for a different attribute. The shaded portions of the triangle exhibits the attributes illustrated by the corresponding protocol. As illustrated in this figure, the set of these desirable attributes captured by snooping, directory, and Token protocols is disjoint, because of that, neither protocol has all the required properties.

[1] **Origin 2000**, code named *Lego*, is a family of mid-range and high-end servers developed and manufactured by SGI (Silicon Graphics, Inc.).



Three desirable Attributes



(a) Snooping Protocols **(b) Directory Protocols** **(c) TokenB**

Figure 1.3 Characterizing Common Protocols in Terms of Three characteristics [11].

Token-using-Broadcast (TOKENB) performance policy aims at capturing 1) Low-latency Cache-to-cache Misses and 2) No Reliance on a Bus or Bus-like Interconnect at the same time (i.e., avoid an ordered interconnect and provide low-latency cache-to-cache misses). These two attributes are the most important of the three attributes, and aiming at both of these attributes results in a low-latency, broadcast-based coherence protocol that is suitable for applications with a glueless, point-to-point interconnect.

1.3 Thesis Structure

This thesis begins with introductory chapter (Chapter 1) and continues with a chapter that describes the fundamentals and background about multicore architecture and memory caching (Chapter 2). We then describe and discusses the tools, methodology, and workloads used for the evaluation and simulation of different systems (Chapter 3) and continues with a chapter that describes the traditional cache coherence protocols (Directory and snooping) in addition to TOKENB protocol (Chapter 4). And then we evaluate each protocol and discuss the results (Chapter 5). The thesis ends with chapter 6 which concludes our work and gives directions for the future work.

Chapter 2 Background and Motivation

2.1 Multicore Architectures

Chip Multiprocessor [19,20,21] refers to the technique of summing the power of more than one core on one chip or die. The process of connecting multiple-cores is done over a packet-based or bus-based network. Chip manufacturers are tending to concentrate on the production of multicore chips, as their architectures allow the computing of a number of tasks at the same time thus, enhancing the performance of the system [22,23,24].

As the world is evolving into a more digitalized pattern and the constant growth of performance, multicore architectures are presenting the computing technology with a key development. Multicore architectures will prevail as the predominant computing model for, they far exceed the performance and productivity benefits of single-core processors.

2.1.1 Multicore Necessity

Numerous new applications are becoming multithreaded and the computer architecture is turning its focus toward parallelism. This is because it is very hard to enhance the performance of single core processors by increasing the clock frequencies, let alone the probable difficulties such as heating or speed of light, if the frequency exceeds certain ranges, the design and verification needs a large team as well.

The usage of packet based on-chip networks decreases the power dispersion so that it provides the ability for other cores to operate (although in a decreasing fashion). If one core fails, for they are completely independent, Multicore Architecture allows for

hardware scalability and escalades software scalability [25]. Multicore Architecture also supports hardware reuse thus, helping in decreasing time-to-market, increasing the productivity and diminishing the cost [26].

The cache coherency circuitry functions at a much higher clock rate performed over multiple CPU cores on the die than when signals travel off-chip.

The cache snoop operations are highly improved by combining counterpart CPUs on a single die. By this, we mean that signals travel a shorter distance between variant CPUs as so the signals will degrade less. The finer quality of signals allow for more data sending in a given period of time as individual signals do not need duplication.

Multicore design makes best usage of the silicon die area; baring in mind that the availability of the silicon substance is insufficient for the demand, thus, by using proven core library designs to come up with minimal design errors compared to devising a new wider core design. Adding additional cache as well produces the disadvantage of diminishing returns [27].

2.1.2 Multicore Challenges

Multicore design is faced with many challenges, like any new design. These challenges require identification and understanding. One of the main challenges facing Chip Multiprocessors (CMPs) is the competition for shared resources. This challenge forms a restriction bottleneck [28,29,30]. Some of the shared resources are: main memory, bandwidth and capacity, cache bandwidth and capacity, memory subsystem interconnection bandwidth, and system power. Listed below are the main challenges:

1. Multicore architecture might escalade the complexity for designers and developers for its parallel architecture of hardware and software. It is easier to manage lower density single-chip designs' conduction of heat compared to multicore chips that on their turn, generate lower returns on production [26].

The architectural perspective concludes that a better usage of the silicon surface area is obtained by single CPU designs, not multiprocessing core, risking the extent of the development dedicated to this particular architecture. Last but not least; raw processing power is the only, but one of, the obstacles facing system performance. The real world performance advantage is constrained by sharing the same system bus and memory bandwidth between the two processing cores. Dual core improvement is limited between 30% and 70% if a single core is close to being memory bandwidth limited, On the other hand, an improvement of 90% is expected if memory bandwidth is not an issue. More than 100% improvement is accounted if an application that used two CPUs operated at a faster level on dual core due to the limiting factors in communication between the CPUs [27].

2. The concurrency principles are not fully accustomed by today's designers, developers, and test engineers. These principles are difficult to absorb and, as a result, the time for product development is increased as the development teams will have to be retrained [31]. Existing software modifications are mandatory as well as operating system support (OS). Moreover, multicore processors' ability to enhance application software performance is bound to the application usage of multiple threads. For example, a present-day PC operates faster on a 3 GHz single-core processor than on a 2 GHz dual core (of the same core architecture), regardless of the theoretical assumption that dual-core processors obtain more processing power, because of their incapability of seizing more than one core at a time in an efficient manner.

3. Many issues need to be considered by developers like Processor synchronization, latency, and speed gap, Bandwidth requirements, while transmitting data over network, and program partitioning issues [32].

4. Throughout a debugging session; deadlocks, livelocks, and data corruption are difficult to trace, as they are discontinuous. Furthermore, the pluralities of the current high-level languages are non-supportive for concurrent programming.

2.1.3 Multicore processor vs. Multiprocessors

Multicore processor derives from the family of multiprocessors with the significant difference of the gathering of all the processors on the same chip. Compared to multiprocessors, multicore processor is a better option for its low design complexity, high clock frequency, and high throughput.

Multicore processors differ for their Multiple Instructions Multiple Data (MIMD) as the various cores execute different threads (Multiple Instructions), operating on multiple parts of memory (Multiple Data). In fact, the same memory is shared among all cores as so, multicore is a shared memory multiprocessor.

While multi processors are multiple chips that are plugged into the motherboard and therefore allocate an unshared cache for each chip, multicore processors have more than one core with the ability to execute processing on one chip, normally by allocating an unshared L1 cache for each core and the sharing of L2 and/or L3 caches.

Communication of data among processors in the current shared-memory multiprocessors may vary from 50 clock cycles (for multicores) to more than 1000 clock cycles (for large-scale multiprocessors) thus, affected by the communication mechanism, the type of interconnection network, and the scale of the multiprocessor. The impact of such delay in long communications is of obvious significance [33].

Multiprocessors contain one large and fast superscalar core that performs greatly on a single thread, although, still exploits instruction level parallelism exclusively. On the other hand multicore obtain many cores each is smaller comparatively less powerful

(although easier to design and manufacture) still, highly capable with thread level parallelism.

The multicore CPU uses a smaller Printed Circuit Board (PCB) than multi-chip SMP designs that, if we hypothetically assumed that the die can be physically placed in the package. Moreover, the power consumption of a dual-core processor is somewhat less than two single-core processors combined; hence, the power needed to drive signals external to the chip, and as cores are able to operate at lower voltages on a less silicon process geometry, as a result to such reduction, the latency is reduced. Furthermore, some circuitry is shared between cores such as: the L2 cache and the interface to the front side bus (FSB) [34,35].

2.1.4 Multicore Commercial examples

The first on-chip multiprocessor for the computing market of the general purpose was presented in the year 2000 by IBM. Followed in the year 2005 by AMD that presented the two processor versions for the server market. In the end of the year 2007 and early 2008, the kickoff quad-core and triple-core processors were introduced, whereas an 8-core chip for computer-farm application was produced by Sun in 2006. On the 20th of August 2007 however, Tilera gave off its 64-core processor.

Intel also released its two-processor versions in 2005 for the server market; its quad core processor was presented on Dec 13, 2006. Within the coming years, Intel is expected to release its 80-core processor prototype, each running at 3.16GHz[36].

2.2 Memory Caching in Multicore Systems

Cache memory is defined as a specific memory subsystem whereas data of persistent usage is stored for fast access. The frequently accessed upper memory level locations and addresses of the data items are stored in the memory cache. The cache

checks to see if the addresses referenced by the processor in memory are held in that address, so that if they are, the data is returned to the processor, if not; an ordinary memory access takes place. A cache is of use when microprocessors' speed exceeds the RAM accesses' speed, for cache memory is ever faster than main RAM memory [32].

Memory cache levels are divided into three types: *1) Level 1 cache* (L1) is a small, fast memory cache contained in each core as it helps increase the access speed to the frequently-used data; *2) Level 2 cache* (L2) is of a bigger size than L1 and is built into the microprocessor chip and occasionally between the microprocessor and the main memory; and *3) Level 3 cache* (L3) is a combination of fast, built-in memory chips located between the microprocessor and the main memory, although not found in all MA designs[37].

2.2.1 Cache necessity and it's work principle

At this point, a question is raised "Why can't we speed up the entire computer's memory to match the speed of the L1 cache, and therefore eliminate the need for cache memory?" The answer to such question would be: speeding up the entire memory is a reasonable solution, but it would be extremely expensive. Caching is used to increase the speed of large amounts of slow, affordable memory by using a small amount of expensive memory [38].

The goal of designing a computer is to permit the microprocessor to run at its most speed with the least cost possible. A 500-MHz chip runs 500 million cycles per second (one cycle every two nanoseconds). The main memory would need 60 nanoseconds, or we would waste 30 cycles in accessing the memory without the usage of L1 and L2 caches [38].

It is somewhat astonishing that such comparatively micro amounts of memory allows for the magnification of much larger amounts of memory. Consider an L2 cache

of 256-kilobytes enabling the caching of 64 megabytes of RAM. If so, 64, 000, 000 bytes are cached efficiently using 256, 000 bytes [38]. This is justified by the computer science principle "locality of reference", Indicating that only small portions of a code is mostly used at the same time, even with great programs of several megabytes of instructions. Programs often spend large periods of time repeatedly working in one small area of the code, mostly repeating the same work many times redundantly with barely any different data, shifting afterwards to a different area. "Loops" are the reason behind this, as they are what programs use in order to work many times in rapid succession [39].

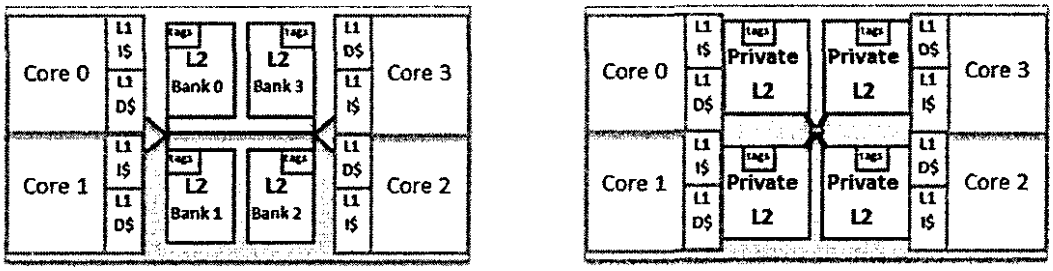
The cache is programmed (in hardware) to save the recently-accessed memory locations in case of later necessity. After being loaded for the first time from the memory, instructions are saved in the cache. Next time the instructions are needed, the processor checks the cache first, to see if the needed instructions are available, and directly loads these instructions from cache and not from the slower system components; RAM. The function of the caches design and size are what determines the number of instructions buffered by this method.

2.2.2 Private vs. Shared caches

Shared L2 caches are used by a group of CMPs in order to magnify the on-chip cache capacity and to minimize off-chip misses. Some use private L2 caches as to replicate data in order to limit the delay caused by global wires and to minimize cache access time. However, recent hybrid proposals balance latency and capacity using selective replication, on the other hand, their static replication rules cause a degradation in performance for some combinations of workloads and system configurations [40].

The occurrence of Chip Multiprocessors in mainstream systems forces them to provide a variety of workloads with a reasonable performance. In order to confront the

interfering requests of reducing off-chip misses (capacity) and handling slow global wires (latency) in particular, Level-2 (L2) cache management brings forward a main challenge. The IBM Power 5 [41] and Dun Niagara [42] of the existing CMP systems use shared L2 caches (Figure 2.1a illustrates shared caches in CMP) to prevent replication in order to maximize the on-chip capacity. Although the cross global wires to reach distant L2 banks, many requests causes them to have higher access latencies. In contrast, private L2 caches (Figure 2.1b illustrates private caches in CMP), reduce average access latency by replicating data close to the requesting core, but sacrifice effective capacity and incur more misses [43,44].



a) Layout of CMP-Shared L2 b) Layout of CMP-Private L2

Figure 2.1 CMP Shared and Private L2 Caches.

2.3 Cache Coherence Protocols

2.3.1 The cache coherence problem

Storing frequently accessed data in faster memory caches provide a more enhanced performance. More than one core is enabled to cache an address (or data item) simultaneously being that the same address space is shared by all cores. Although, many other elements of great importance are to be taken into consideration when using cache memory in a multicore environment, to be precise, preserving the right copies of data in all caches in the system. Inconsistencies can occur and incorrect executions may take

place as a result of updating a data item by a single processor without informing the remaining processors.

More than one copy of the same data may be kept by multiple caches in a multicore system; therefore, inconsistency between caches may take place as well as inconsistency between the main memory or upper cache levels and the private cache. A complication only exists if individual processors alter their copies of data, because shared data copies should be kept identical for correct operation. Maintaining the data in all the caches the same is known as cache coherence. The phrase cache consistency is also used.

The cache coherence problem is illustrated in Figure 2.2, which illustrates a shared L2 cache by four cores with private caches via a bus. The cores access location X sequentially. At the beginning, core1 brings a copy to its cache by reading X from L2 cache. After that, core 4 brings a copy to its cache by reading X from L2 cache. Thereafter core 4 changes X's location value from 8 to 5. With a write-through cache, causing the L2 cache location to be updated, but in (action 4) when core1 reads location X again, it will read the old value 8 from its own cache rather than reading its correct value 5 from L2 cache.

The writeback caches complicates the situation even further. Core4's write would not update L2 cache right away; instead, it would barely set the dirty (or modified) bit concerned with the cache block holding location X. Contents of cache block would be written back to L2 cache solely, when this cache block is subsequently replaced from the cache of core 4. The reading of the old value is not inclusive to core1. Furthermore, core2 and core3 will miss in their caches when reading location X (actions 5 and 6) by reading the old value 8 instead of 5 from L2 cache. Last but not least, if more than one core write different values in their write-back caches to location X, the last value that

will reach the L2 cache will not be related to the sequence in which the writes to X occurred. Instead, it will be determined by the sequence in which the cache blocks that contain X are replaced.

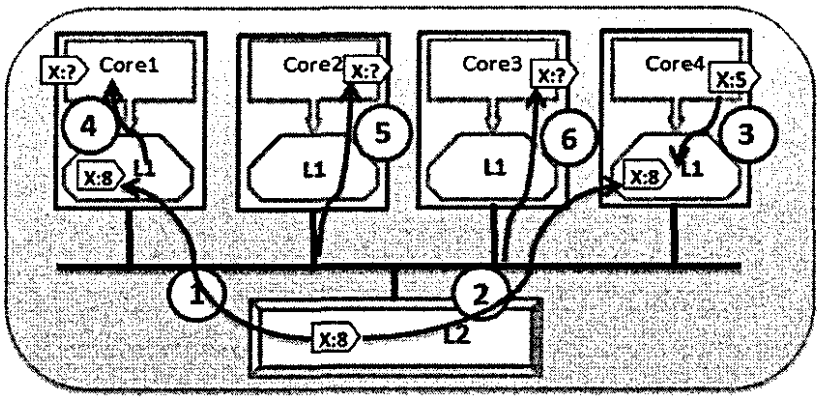


Figure 2.2 cache coherence problem

Coherence between the caches has to be enforced in order for correct execution. This process is affected by two main factors: Performance and implementation costs. In the process of designing a cache coherence mechanism, four main issues must be taken into consideration as well: 1) Coherence detection strategy- How possible incoherent memory access is detected by the system, 2) Coherence enforcement strategy- How to change cache entries in order to assure coherence (i.e. Updating or invalidating), 3) Precision of block sharing information- How sharing information and cache and memory blocks are stored and cache block size, and finally 4) the size of line in the cache and how the performance of the system is affected by it [45].

The cache coherence protocol comprises an absolute design choice for multicore and multiprocessor systems as it directly influences the overall system performance. Many system elements can influence the general performance to various certain levels thus reckon the selection of coherence protocols and the target application workloads, some of which are: the maximum achievable bandwidth (in snoop-based protocols) and network transactions number (in directory based protocols). Thorough evaluations over

the past decades have been made on the coherence of protocols since their commencement.

2.3.2 Basic Operation of Cache Coherence Protocols

In order to assure coherence invariant, coherence protocols track read and write permissions of clocks held by processor cores caches by using the protocol states. The states providing a set of common states in for reasoning about cache coherence protocols are described in this section, most common protocols apply these states such as MSI (Modified, Shared, and Invalid), MESI (Modified, Exclusive, Shared, and Invalid), MOSI (Modified, Owned, Shared and Invalid) and MOESI (Modified, Owned, Exclusive, Shared, and Invalid).

Firstly; the Modified, Shared and Invalid states are taken into consideration all implanting the MSI protocol. When a process is signified as it may neither read nor write, it then has a block in the Invalid or I state. Implicitly, a block is considered to be in the invalid state in the cache when it is not found in the cache. When a processor can read the block, but cannot write it, it's then considered to be in the Shared or S state. The Modified state or M state of a processor indicates that a processor can both read and write the block. To directly implement the coherence invariant, the three states of (Invalid, Shared, and Modified) are used through (a) permit only one processor at a time to be in the Modified state, and (b) forbid all other processors to be in the Shared state when one processor is in the Modified state at a certain point in time [11].

Replacing or victimizing occurs when a processor must evict a block currently held in the cache when demanding a new block. Two factors impact on the effort needed to evict a block; the coherence state of the block and the specific protocol. e.g. while evicting blocks in the Modified state, most protocols demand a data writeback to memory and permit a discrete eviction in the Shared state. If a protocol permits a

processor to evict blocks in the Shared state without sending a message, it is then considered to support silent evictions. If protocols require sending a notification message (although not the entire block) to the upper memory level in the Shared state, then they do not support silent eviction.

Much like the Shared state, the optional Owned or O state in a processor's cache allows read-only access to the block, while indicating that the main memory's value is incoherent or old. Hence; before evicting a block, the processor in the Owned state must update the upper memory level. Only one processor at a time is permitted to be in the Owned state, just like the modified state. Other processors, though, are permitted to be in the Shared state when one processor is in the Owned state unlike the Owned state [11].

Subtending the owned state provides two main advantages:

- First: Reducing system traffic hence, the Owned state does not ordain a processor while transitioning from Modified to Shared when in a read request to update memory. A protocol, not including the Owned state, requires the responder to provide data to the requester and updating memory (illustrated in Table 2.1) when transitioning from Modified to Shared. The addition of the Owned state permits a processor to transit from Modified to Owned state without sending a message to the memory at that time (illustrated in Table 2.1). A reduction of memory traffic occurs if another processor issues a write request for the block before it is evicted from the Owned processor's cache.

- Second: in some protocols (e.g., systems based on IBM's NorthStar/Pulsar processors) a processor can respond more quickly by providing data from its SRAM cache than the home memory controller can respond from its DRAM. In order to enable this enhancement, a suitable mechanism for selecting a single responder is provided by

giving the processor in the Owned state the farther responsibility of responding to requests for data. On the other hand, a response from memory in other protocols (e.g., most directory protocols) is considerably faster than providing data from an Owned copy. The Owned state is not applied in these systems (preferring less latency for additional traffic).

The last state is the Exclusive or E state. The Exclusive state and the Modified state have many similarities, but differ in that the Exclusive state implies the contents of memory match the contents of the exclusive block. As a result to differentiating between the clean Exclusive state and its corresponding Modified dirty state, the need to update the block at the home memory when a block is evicted in Exclusive is eliminated. The memory responds to a read request with a clean-data response as long as there is no other processor caching the block. The duty of updating the memory is removed when the requesting processor transitions to Exclusive, granting it read/write permission to the block. By silently transitioning from Exclusive to Modified (demanding a writeback upon subsequent eviction), the block is then written fast without an external coherence request. Figure 2.3 illustrates the basic operations of MOESI states.

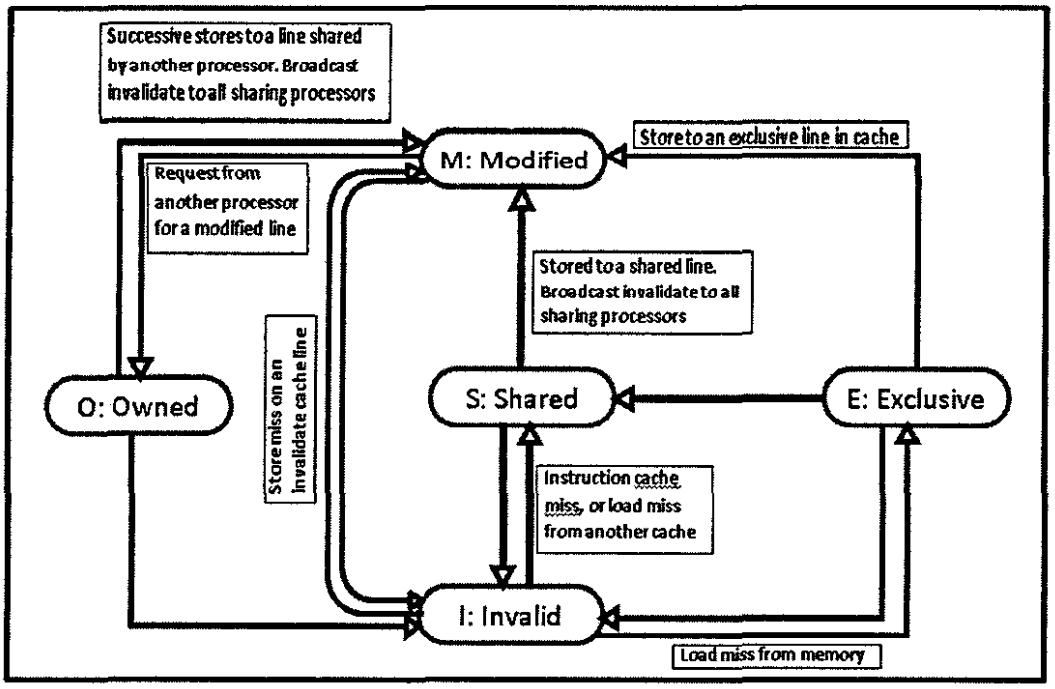


Figure 2.3 The basic operations of MOESI states.

Table 2.1 MOESI State Transitions

State		Modified (M)	Exclusive (E)	Owned (O)	Shared (S)	Invalid (I)
Core Action	Load	Hit	Hit	Hit	Hit	Read request (response: Shared) →Shared OR (response: clean) →Exclusive
	Store	Hit	Hit →Modified	Write request → Modified	Write request → Modified	Write request → Modified
	Eviction	Writeback →Invalid	Silent evict →Invalid	Writeback →Invalid	Silent evict →Invalid	None
Incoming	Read Request	Send data →Owned	Send data →Shared	Send data	None	None
	Write Request	Send data →Invalid	Send data →Invalid	Send data →Invalid	None →Invalid	None

2.3.3 Hardware Protocols

- Hardware protocols are divided into two primary groups: Snoop Bus Mechanism & Directory Based Method. Token protocol is also considered as another efficient protocol implemented by Milo M. K. Martin [11] for multiprocessor. Chapter 4, 5 and 6 describe these methods in further details.

2.3.3.1 Snoop Bus Protocols

Snooping protocols obtain coherence depending on a shared bus between the processors. On a processor write, the write is passed to the main memory via a bus passing through the cache. Updating or invalidating the cache entry appropriately is possible to any processor caching the address. Although, the Snooping protocols' disadvantage is that they are incapable of scaling well beyond 32 processors due to the shared bus.

The transactions in this method are observed and all memory write operations are monitored by a bus watcher unit built-in each processor/cache. (CH4 describe Snooping protocols in More Details).

2.3.3.2 Directory-Based Protocols

Unlike snoopy based protocols Directory based protocols do not exchange coherence information using a shared bus. Directory based protocols are of better scalability (might posses hundreds of cores per chip). According to this method, each core might obtain its own memory and, for efficiency, generally weak consistency is applied.

Systems of point-to-point unordered networks usually implement directory-based cache coherence protocols. Although cache miss latencies may be increased by these protocols as they propose indirection to import coherence information from the directory (usually on chip as a directory cache). Chapter four describes Directory protocols in More Details.

2.3.3.3 Token Protocols

Token based protocols are assumed to combine the best features of both snooping and directory protocols: low-latency cache to cache misses and not depending on totally-ordered interconnects. Token Coherence [12] is a framework suggested in order to facilitate the development of token based cache coherence protocols. Three components comprise the Token Coherence: the ***taken counting mechanism;*** assuring the coherent reading and writing of data. The ***persistent request mechanism;*** solves protocol races and prevents starvation. These two mechanisms form the correctness foundation in order to assure correct operation in the different cases. On the other hand, the third component, which is ***Performance policy,*** is used to make the protocol fast and bandwidth efficient. More details about Token Protocol are discussed in chapter four.

2.3.4 Compiler and Software protocols

Software protocols impose consistency with limited hardware support depending on the compiler or specialized software handlers. They are somewhat comparable to Distributed Shared Memory (DSM) systems but a lower level such as: sharing, usually in blocks not pages, the need for more efficiency to obtain better performance and architecture support for sharing. We just focus on the hardware coherence protocols.

Software protocols can be classified in accordance to many criteria[45]. The most important criteria are:

- **Dynamism**: compile-time or run-time analysis
- **Selectivity**: level of coherence actions
- **Restrictiveness**: conservative or as-needed consistency enforcement
- **Adaptivity**: can protocol adapt to access patterns
- **Granularity**: coherence data size and structure
- **Blocking**: program block on which coherence is enforced
- **Positioning**: position of coherence instructions
- **Updating**: how memory is updated after a write
- **Checking:** how incoherence is detected

2.3.4.1 Software Coherence with Limited Hardware Support

According to this approach, a consistency code must be generated by the compiler for, there is no hardware coherence provided. Time tags are kept in the hardware and are updated on every write. In order to ensure data consistency, the compiler while on a read generates coherence read that screens time tags. The hardware's duty is to preserve tags while it's the compiler's duty to detect inconsistent reads. Using tags, also allows for performing dynamic self-invalidation of blocks. A number of techniques are based on using these time tags.

Petersen and Li have developed a special algorithm if the hardware has no time tag, as it only uses page translation hardware and page statues table [112]. According to this algorithm, a page handler at the page-level is in charge of maintaining the sharing of information. In the case of page access or fault, the sharing of information, updating page table and performing coherence actions is the software handler's duty. Software

handlers are slower than hardware for they are limited by the memory access paths and bound to the OS [45].

2.3.4.2 Enforcing coherence by Restricting Parallelism

Structuring the language, in order to limit parallelism, is another way compilers use to assure coherence. Using this method, enforce coherence are easier to enforce. Moreover; a limit is forced upon the programmer and potential parallelism. However; no hardware support is needed to attain well performance and it simplifies the compiler design. Do all parallel loops, as well as, master/slave processes are contents of parallel language restrictions[45].

2.3.4.3 Optimizing Compilers

Optimizing compilers are designed in order to maintain coherence of limited hardware support, and at the same time, not limiting the programmer too much. Thus, depending on detecting data dependencies, probably using synchronization variables (locks, barriers), might provide hints to the hardware, able to detect the need for coherence, dynamic sharing problems are also probable. Overall; providing good performance, but difficult to design [45].

2.4 Literature Review and Related work

This section presents some related work on existing cache coherence techniques. The scope and the amount of related work are large, so we focus on the aspects most fundamental and related to the research in this thesis.

Snooping coherence on a bus was first described by Goodman [56]. Early bus implementations used electrically shared wires that held the bus for an entire coherence

transaction. But more modern snooping systems implement a logical bus using additional switches, state, and logic rather than shared electrical wires.

Barroso et al. [113] examined snooping on a ring and proposed an approach that Michael Marty [111] generalize and call greedy snooping. The primary commercial systems using ring-based coherence, the IBM Power4/5, also uses a greedy-like snooping protocol for coherence on a ring [114]. A greedy snooping protocol broadcasts coherence requests to all other nodes in the system. While a ring naturally accomplishes the broadcast operation, there is no total ordering or atomicity. Therefore, unlike the bus protocol, a requestor cannot be assured that its coherence request is ordered once the message is transmitted and racing (or conflicting) coherence requests must be handled differently.

A directory protocol contains state about the sharing status of a given block to determine the actions needed when a coherence request is received. A typical directory includes a list of sharers for each block, and a field that points to the current owner. A directory can also take other forms, such as a linked list of sharers [101], or sharing lists at a coarser granularity than single processor-cache nodes [98]. Directory-based cache coherence was first suggested by Tang [85] and Censier et al. [84]. Examples of commercial machines using directories include the SGI Origin [17] and the Alpha 21364 [67].

The previous techniques to coherence, snooping and directory, both require the careful coordination of message exchanges and of state-machine transitions to ensure the coherence invariant. The properties of the interconnect also further complicate the design of the protocol to ensure the invariant. A technique proposed in 2003, token coherence, directly enforces the coherence invariant through a simple technique of counting and exchanging tokens.

Token coherence [12] associates a fixed number of tokens with each block. In order to write a block, a processor must acquire all the tokens. To read a block, only a single token is needed. In this way, the coherence invariant is directly enforced by counting and exchanging tokens. Cache tags and messages encode the number of tokens using Log2N bits, where N is the fixed number of tokens for each block.

Token coherence enables a broadcast protocol on an unordered interconnect as well as others described in Martin's thesis [11]. The TokenB broadcast protocol has some similarities to the greedy snooping approach and a few key differences. In TokenB, coherence requests are broadcast directly from the requesting processor to all other processors like greedy snooping. Unlike greedy snooping, only processors sharing the block must respond with an acknowledgement message. However in TokenB, conflict is not explicitly detected because a snoop response is not received from every processor. Therefore, TokenB uses a per-request timer that is used to issue retries or to invoke a persistent request upon timeout.

Many work have been done through the last years to evaluate and propose an efficient cache coherence protocols; but no one model the Snooping, Directory, and Token protocols on the multicore architecture and evaluate them using meaningful performance metrics as we did in this thesis.

Chapter 3 Evaluation Methodology

The objective of evaluation is to explain the relative behavior of different coherence protocols (traditional protocols and Token Coherence based protocols). We aim not to (1) Generate ultimate execution times or throughput rates for our simulated systems or (2) Evaluating such protocols on all of the future's system configurations. We use an approximation of a chip multiprocessor system in order to accurately obtain relative comparisons and evaluations instead. Full system simulation and modeling of the first-order timing effects for approximating an aggressive multicore system operating commercial loads are the means to reach such an aim. Our goal is to get the first-order effects, although –similar to most architectural simulations- Capturing all system's aspects in precise detail is not what we try to do.

3.1 Simulation Tools

In order to evaluate the demand system; full-system simulation is used. Using full system simulation allows for evaluating the proposed systems when running realistic scientific applications on top of actual operating systems. As well as capturing the subtle timing effect that can't be captured with trace-based evaluation.

The Simics full-system multiprocessor simulator [46] extended with the Wisconsin GEMS simulation environment [47] is used in order to perform the analysis. Simics is a system-level architectural simulator developed by Virtutech AB [46] with the ability to operate unmodified commercial applications and operating systems. Simics only provides an interface equipped by GEMS as to support the memory hierarchy model. GEMS is a set of modules that extends Simics with timing fidelity. Two primary modules make up GEMS: Ruby and Opal. Ruby models memory hierarchies and uses the SLICC domain-specific language to specify protocols. Opal

models the timing of an out-of-order SPARC processor. The relation between the Simics simulator and GEMS toolset is illustrated in Figure 3.1.

3.1.1 Simics Simulator

An overview of the simulation tools used is illustrated in Figure 3.1. Simics is found on the top. Simics is defined as a full functional system simulator enabling the booting of an unmodified operating system thus in order to execute actual applications.

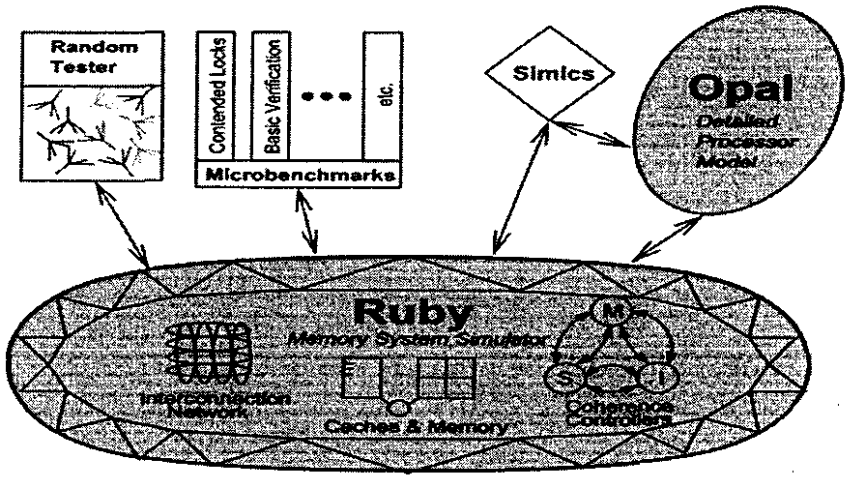


Figure 3.1 A view of the GEMS architecture with simics simulator [47].

A simple in-order processor is modeled in Simics. Simics uses Ruby in order to pass all the load and store instructions and the instruction fetch requests. Determines, if the operation hits or misses, thus by performing the cache access. In the case of a hit, instructions are executed normally by Simics. But in the case of a stall, Ruby simulates a cache miss only after stalling Simics' request from the issuing processor. The request completion is determined by contention, latency of messages, and other factors. Ruby determines the timing-dependent functional simulations in Simics by controlling Simics' advancing time.

3.1.2 Ruby Module

Ruby is found under Simics, as it is a module used to protocol-independent components (cache arrays, memory arrays, message buffers, and assorted glue logic) as well as protocol-dependent components (cache controllers and memory controllers).

To simulate timing, Ruby takes on a queue-driven event model. Message buffers are in charge of communicating cache controllers and memory controllers. The recipient is scheduled to wake up as soon as the next message becomes available to read from the buffer.

Ruby solves the miss in the case of a cache miss by generating the events required by the implemented protocol. When an exchange of messages is implied by an event through the network, the messages are stalled in the message buffers and their transmissions are simulated in details in the network simulator upon their arrival. A wake up is scheduled as soon as the message arrives to the recipient component.

3.2 Performance Metrics

Throughout this research, we demonstrate how the overall performance of our proposals by the runtime i.e. measuring the time necessary to complete certain amount of work. The metric instructions-per-cycle has been used by other works instead of runtime in judging performance improvements. However; system timing effects of multiprocessor workloads may alter the number of instructions executed therefore, Instruction Per Cycle (IPC) is not a suitable metric for evaluating the coherence protocols and systems. Thus the performance of the system is not guaranteed to be reflected by measuring IPC and running the simulator for a specific number of instructions [48]. This is due to the magnificence of the variation in the instruction path of multithreading workloads that run on multiple processors.

Runtime is used to conclude that "protocol A is X % faster than protocol B" using the formula:

$$\mathbf{X} = \left(\frac{\mathbf{runtime(B)}}{\mathbf{runtime(A)}} - \mathbf{1.0}\right) * \mathbf{100}\,. \qquad \text{Equation \#1}$$

The measurement is started at the parallel phase so that we avoid measuring thread forking. Until now, a full system checkpoint (to provide a well-defined starting point) is used to initialize the system state and to simulate the execution until the end of the parallel phase. The number of cycles is recorded and referred to as application time in order to complete the parallel phase.

Endpoint traffic (in messages per miss) and interconnect traffic (in terms of bytes on interconnect links per miss) are other ways besides reporting runtime that measure and report the traffic. The endpoint traffic shows the amount of controller bandwidth needed for handling incoming messages. The amount of link bandwidth used by the messages are indicated in the interconnect traffic as they traverse the interconnection.

The last metric is mostly not bound to the particular interconnect and message size. On the contrary, the interconnect topology, the use of bandwidth-efficient multicast routing, and message size are what influence the last metric.

Due to the computational intensity of detailed architectural timing simulations, we are limited to simulating only a short segment of the workload's entire execution. Two techniques are used to subdue such limitations and partially overcoming this problem. At the beginning, system cold-start effects are avoided by warming up and check pointing all workloads, and restoring the cache contents captured as to assure the warming of caches all as part of our checkpoint creation process. Second, the approach of simulating each design point a number of times as to address the variability in commercial loads, this is done by small, pseudo-random perturbations of request latencies [49, 50]. These perturbations cause alternative operating system scheduling

paths in deterministic simulations. A distribution of runtimes is created by operating many of these pseudo-randomly perturbed systems. The reduction of the effect of the skewed nature of the distribution is possible by eliminating all data points beyond 1.5 standard deviation from the mean. Error bars in our runtime results approximate a 95% confidence interval centered on the arithmetic mean of the remaining data points.[11] Every data point is approximately the aggregate of 5 to 15 data points; a large number of simulations are used for configurations and workloads exhibiting the most variation.

3.3 Workload Descriptions

We used three multi-threaded commercial workloads from the Wisconsin Commercial Workload Suite [49]: an online transaction processing workload (OLTP), a Java middleware workload (SPECjbb), and a static web serving workload (Apache). The previously mentioned workloads operate on a simulated 16-core SPARC processor that runs Solaris 9. The simulated system has 4GBs of main memory.

Online Transaction Processing (OLTP): The OLTP workload is built on a TPC-C v3.0 benchmark with 16 users/processor and no think time. IBM's DB2 v7.2 EEE database management system formalizes the back-end and is responsible for almost all of the activities in this workload. The users query a 5GB database with 25,000 warehouses stored on eight raw fiber-channel disks. The database logic is also stored on the disk. The system is warmed up with 100,000 transactions and the hardware caches are warmed up with additional 500 transactions.

Java Server Workload: SPECjbb. SPECjbb2000 is a server-side Java benchmark modeling a 3-tier system, its primary focus is on the middleware server business logic. Sun's HotSpot 1.4.0 Server JVM drives the benchmark. The experiments use 1.5 threads

and 1.5 warehouses per processor. However; we use over a million transactions to warm up the system and 100,000 transactions to warm up simulated hardware caches.

Static Web Content Serving: Apache. Apache 2.0.43 configured is used as to obtain a hybrid multi-process multi-threaded server model with 64 POSIX threads per server process. The web server is SURGE driven with 3200 simulated clients each with a 25ms think time between requests. Apache logging is disabled to maximize server performance. We use 800,000 requests to warm up the system and 1000 requests to warm up simulated hardware caches.

3.4 Modeling a CMP with Simics/GEMS

This section is dedicated to describing how GEMs are used to model CMP memory systems in this thesis. Like most simulation, some components are realistically modeled and some idealized. Our goal in this evaluation is to validate designs and to give insights into the relative merits of a subsystem studied and not to simulate realistic or absolute runtimes for all future CMPs.

We aim to capture first-order effects of coherence protocols, such as all the messages needed in order to implement the protocol at certain interconnect. Most of the idealized components of the simulator are expected to impact on all protocols in the same way, or else the designer has to compensate the design dependant subsystems in order for them to match the given protocol. e.g., if a protocol requires that a cache snoop X tags/cycle, then the designers would engineer this ability into the implementation. Whenever necessary; these events are measured and their counts are reported even if they do not affect the simulated runtime.

At first, many controllers are connected via networks in a specific topology as to formalize the CMP memory model. L1 Caches interface processor models. After that L1 caches interact with other controllers (i.e., directory/memory controller) and model the timing of an L1 miss by interconnecting links. Timing is usually modeled by a controller that specifies the delay when a message is injected into the network, and delay incurred by modeling the delivery of the message.

Following are details of how the main components of the system are modeled:

3.4.1 Simulated System

A multicore server is simulated while running commercial workloads and using multiple interconnects and coherence protocols thus, evaluate Cache Coherence Protocols. The system we target is a 16-core processor SPARC v9 system with highly integrated nodes each including a dynamically-scheduled processor, split first level instruction and data caches, unified second level cache, coherence protocol controllers, and a memory controller for part of the globally shared memory. Sequential consistency is implemented in the system by using invalidation-based cache coherence and an aggressive, speculative processor implementation [51, 52].

We select a variety of coherence protocols, system interconnects, latencies, bandwidths, cache sizes, and other structure sizes. The parameters for the memory system and the processors are listed in Table 3.1. We limit the bandwidth of memory controllers and cache controllers; which limit the bandwidth that is caused by external requests for the DRAM, cache tag arrays, and cache data arrays indirectly. We also apply simulation both with unbounded interconnect link bandwidth and interconnects with 4GB/sec links. By these two types of simulations, we are able to distinguish between changes in uncontained latency and changes in latency due to interconnect bandwidth constraints.

Table 3.1 Simulation Parameters

Coherent Memory System Parameters	
Private L1 Caches	Split I&D, 64 KB 4-way set associative, 64-byte line, 1ns latency (2 cycles)
L2 unified cache size and latency	4MB, 6ns latency (12 cycles)
main memory size and latency	4GB, 80ns (160 cycles)
interconnect link	4GB/second or unbounded bandwidth 15ns latency (30 cycles)
Calculated Average Miss Latencies	
Interconnect hop	68ns (136 cycles)
memory-to-cache	= L2 miss(6) + 2 hops(2 x 68) + mem(80) = 222ns (444 cycles)
direct cache-to-cache	= L2 miss + 2 hops + cache(6) = 148ns (296 cycles)
indirect cache-to-cache	= L2 miss + 3 hops + directory(DRAM=80,SRAM=6) + cache DRAM directory = 296ns (592 cycles) SRAM directory = 222ns (444 cycles)

After that, we describe the coherence protocols and system interconnection:

3.4.2 Coherence Protocols

Using a few distinct MOESI coherence protocols the simulated systems are compared. The migratory sharing optimization [11], is applied to protocols to improve their performance, the upgrade requests are not supported by any protocols. Coherence is kept at on aligned 64-byte blocks. All request, acknowledgment, invalidation, and dataless token messages are of the size 8-bytes (including the 40+ bit physical address and token count whenever necessary); data messages include the 8-byte header and the

64 bytes of data. These message sizes do not include any extra bits used by the interconnect to detect and correct bit errors. Three main protocols are used: Snooping (an aggressive snooping protocol described in Section 2.3.3.1), Directory (a traditional directory protocol described in Section 2.3.3.2), and Token (an optimized version of a protocol that migrate and improve Snooping and directory protocols described in Section 2.3.3.3).

3.4.3 System Interconnects

The same GEMS' are used in the networking model as to approximate all of the target interconnection networks. For each target CMP, a specific network topology is specified using a configuration file. The links between network switches and the endpoints of the interconnect are determined by the file. Fix latency and bandwidth parameters are the specifications of each link since GEMS does not model the characteristics of links at the lower network levels. The latency specified and other queuing delay resulting from insufficient bandwidth, are always incurred in a message.

Whenever convenient, the following chapters will elaborate each of the previous components and evaluations in specified details.

Chapter 4 Cache Coherence Protocols

In this chapter, we describe the traditional protocols (Snooping based protocols and Directory based protocols) in addition to the Token coherence protocol: how they are working and what the advantages and disadvantages of each of them are and how the improvement of these protocols can be done, and in the last section we describe the output raw simulations results.

4.1 Snooping Protocols

Snooping protocols are mainly focused on observing bus activities and carrying out the appropriate coherency commands [53]. The bus shows all writes and read misses. Global memory is moved in blocks, a state is dedicated to each block thus determining what happens to the entire contents of the block [54]. Figure 4.1 shows the snooping process.

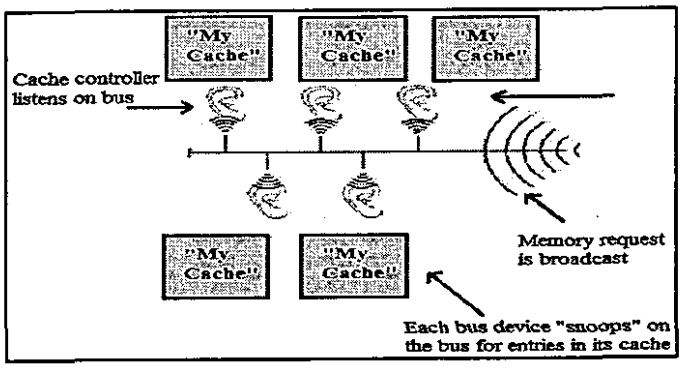


Figure 4.1 Snooping Process.

The reliance of snooping protocols on a "bus" or "virtual bus" interconnect are the primary properties that differentiate snooping protocols from other protocols. Former multichip processors connected all cores and memory modules in the system by a shared-wire, multi-drop bus. Snooping protocols make use of such bus-based

interconnects depending on two bus properties: (1) the visibility of all requests that appear on the bus are to all components connected to the bus (cores and memory modules), and (2) the visibility of all requests to all components by a similar total order (the order in which they gained access to the bus) [55]. Essentially, low-cost atomic broadcast of requests is provided by a bus.

The processor cores snoop every bus transaction and respond with appropriate state changes for the corresponding cache lines depending on two elements: the cache line status and bus transaction type. Two primary policies classify the snoop-based coherence protocols: the **invalidation-based protocols** (e.g., the write-once [56], the Synapse [57], the Berkeley [58] and the Illinois [65]) and the **update-based protocols** (e.g., the Firefly [59] and the Dragon [60]). Usually, the underlying scheme used for the former two policies is a **write-back cache** (the original data is not changed at the same time that changes are made to cached data. Instead, the changed data is marked, and when the cache data is deallocated, the original data is updated. A write-back operates faster than a **write-through cache** whereas changes made to cached data are made to the original copy at the same time, instead of marking it for later updating). These two policies differ in whether to invalidate or update the shared cache lines at the time the processor writes to the same memory block. All the shared cache lines that are held by other processor cores are invalidated in the invalidation-based protocol, on the other hand, in all caches that share the same memory block data is updated in the update-based protocol. Since the performance of each policy depends greatly on the data-sharing patterns exhibited by the application's workload; its controversial to argue with respect to which policy is of a better performance. Most vendors use invalidation-based strategies as the default protocol as they are considered to be more robust generally [55]. Hence invalidation-based coherence has been favored over update-based

coherence protocols in most up-to-date systems (e.g., [17,63,64,65,66,67,68]), this thesis considers only invalidation-based cache coherence protocols.

The atomic nature of a bus is exploited by some snooping protocols by applying the previously described abstract MOESI protocols directly. Processor cores begin coherence transactions by arbitrating for the shared bus in these systems. Once granted access to the bus, the processor puts its request on the bus, and the bus is listened to or snooped (hence the name snooping protocol) by all the other processors. The snooping processors transmit their state and perhaps respond with data (as determined in the abstract protocol operation (section 2.3.2)). The memory specifies the correct response whether to store in the memory the state for each block (the approach used by the Synapse N+1 [57] as described by Archibald and Baer [69]) or to observe the snoop responses generated by the processors.[1] No other processor is allowed to initiate a request until the requesting processor receives its data response (completing its coherence transaction).

Snooping protocols have presented many evolutionary improvements to these atomic-transaction system designs in order to increase effective system bandwidth. Split-transaction designs pipeline allow for a more efficient bus usage thus is done by permitting a bus release by the requesting processor while waiting for its response. In order to reduplicate available bandwidth; systems also take on multiple address-interleaved buses and separate data-response interconnects (buses or unordered point-to-point interconnects). Exploiting point-to-point links, dedicated switch chips, and distributed arbitration allow the avoidance of the electrical limitations of shared-wire buses entirely and implement a virtual bus thus in more aggressive systems. But these

[1] The state in memory or snoop response also determines if a requesting processor should transition to SHARED or EXCLUSIVE on a read request.

virtual-bus systems still depend on totally-ordered broadcasts in the issuance of requests. Numerous snooping systems take on these techniques to create high-bandwidth systems with dozens of processors although each of these enhancements adds considerable complexity (e.g., Sun's UltraEnterprise servers [64,65,70,71]).

A bus connected to all L1 caches is exploited by snoopy based cache coherence protocol. In this mechanism, for every L1 cache misses, a coherence message is allocated in the global state that is in L2 cache that is connected to bus and all other L1 caches keep their cache states and initiate a response to the message if it's theirs. Request messages, invalidation messages, intervention messages, data block transfers, etc are the mainly used messages.[72] Each of the messages relates to communication between on chips, and some messages are in critical section of low latency needs and some are not in critical section such as data transfer which doesn't require low latency .

Wire properties in multicore architecture allow for efficiently improving snoopy based protocols. These wire properties are of constant improvement and gets a tradeoff between latency and bandwidth. By varying properties such as wire/width spacing and repeater size spacing [73] different combination of wires with different latency, bandwidth and power consumptions can be implemented.

L1 caches have low latency in snoopy based protocol because requests are transferred directly to the remaining L1 caches, whereas L2 caches, on the other hand, are of high latencies because requests have to transfer to bus than later to other L1 caches. In snoopy based protocols, total saturation of bus bandwidth occurs when more cores are embedded.

4.1.1 An Example

Maintaining the coherence requirement mentioned in the prior subsection, two methods are used: The first one is to make sure that a processor obtains exclusive access

to a data item before it writes it. Hence, it invalidates other copies on a write; this style of protocol is called a **write invalidate protocol**. It is the most common protocol, for both snooping and directory schemes. Exclusive access makes sure that when the write occurs no other readable or writable copies of an item are available: The remaining cached copies of the item are invalidated.

Figure 4.2 illustrates an example of an operating invalidation protocol for a snooping bus with write-back caches. In order to understand how this protocol ensures coherence, take into consideration a write followed by a read by another processor: Any copy exploited by the reading processor has to be invalidated (hence the protocol name), because the write requires exclusive access. As so, it misses in the cache and is forced to bring in a new copy of the data, when the read occurs. In a write case, it's mandatory that the writing processor obtains exclusive access, forbidding other processors from being able to write simultaneously. When two processors require writing the same data simultaneously, only one of them is enabled (we'll see how we decide who is enabled), causing the invalidation of the other processor's copy. In order for the other processor to finish its write, it must first acquire a new copy of the data, which should contain the updated value at that moment. By thus, this protocol enforces write serialization.

The invalidate protocol alternative is the updating of all the cached copies of a data item when that item is written. This type of protocol is called a (**write update or write broadcast protocol**). A write update consumes considerably more bandwidth hence the protocol is bound to broadcast all writes to shared cache lines. Because of that, all the latest multiprocessors and CMP have opted to apply a write invalidate protocol, and for the rest of the thesis we will focus solely on invalidate protocols.

Processor activity	Bus activity	Contents of Core A's cache	Contents of Core B's cache	Contents of memory location X
				0
Core A reads X	Cache miss for X	0		0
Core B reads X	Cache miss for X	0	0	0
Core A writes a 1 to X	Invalidation for X	1		0
Core B reads X	Cache miss for X	1	1	1

Figure 4.2: An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

In this example we suppose that no cache primarily holds X and that the value of X in memory is 0. The cores and memory contents show the value after the processor and bus activity have both been finished. A blank stands for no activity or no copy cached. Core A responds with the value, thus canceling the response from memory, when the second miss by B occurs. Also, the contents of B's cache and the memory contents of X are both updated. This update of memory, which takes place when a block becomes shared, makes the protocol simple, it is also possible to track the ownership and force the write back only upon block replacement. This calls for presenting another state called "owner," thus explaining that a block is possibly shared, but updating any other processors and memory when it changes the block or replaces it is the owning processor's responsibility.

4.1.2 Advantages of Snooping Protocols

The main present advantage of snoop-based multiprocessors is the low average miss latency, especially for cache-to-cache misses. The responder knows fast that it has to send a response because a request is sent directly to all the other processors and

memory modules in the system. As discussed in Chapter 1, low cache-to-cache miss latency is of great importance for workloads with considerable amounts of data sharing. Replying with data from processor caches when possible can reduce the average miss latency if cache-to-cache misses have lower latency than fetching data from memory (i.e., a memory-to-cache miss). Low-latency memory access is a result of the tightly-coupled nature of these systems [11].

Formerly, snooping has had two additional advantages, but these advantages are of as important to current or future systems. First, shared-wire buses used to be cost-effective interconnects for numerous systems and bus-based coherence offered a complexity-effective approach to applying cache coherence. Unfortunately, shared-wire buses will be used by few future high-performance systems because it is difficult to scale the bandwidth of shared-wire buses. Second, bus-based snooping protocols were comparatively simple. This advantage that was of great importance in the past is now much less importance; New snooping protocols that use virtual buses are often as complex or more complex than alternative approaches to coherence. For example, the need to reason about the protocol operation in logical (ordered) time, rather than in physical time is one potential source of subtle complexity in aggressive snooping protocols.

4.1.3 Disadvantages of Snooping Protocols

The first primary disadvantage of snooping is that—even though system designers have evolved beyond shared-wire buses—snooping designers are still bound to choosing interconnects that can provide virtual-bus behavior (i.e., a total order of requests) when they choose interconnect. These virtual-bus interconnects could be more expensive (e.g., by requiring switch chips), may obtain lower bandwidth (e.g., due to a bottleneck at the root), or might acquire higher latency (since all requests need to reach

the root).[1] On the contrary, an unordered interconnect (such as a directly connected grid or torus of processors) might have more attractive latency, bandwidth and cost attributes [11].

The second primary disadvantage is that snooping protocols are still naturally broadcast-based protocols; i.e., protocols whose bandwidth requirements increase with the number of processors. This broadcast requirement limits system scalability even after eliminating the bottleneck of a shared-wire bus or virtual bus. To control this limitation, recent proposals [74,75,76] aim at reducing the bandwidth requirements of snooping by using destination-set prediction (also known as predictive multicast) instead of broadcasting all requests. These proposals suffer from snooping's other disadvantage: They rely on a totally-ordered interconnect, Even though they reduce request traffic.

4.1.4 Techniques used to improve Snoopy based Cache Coherency

In this section, we will describe some of the best techniques that can be used to improve the Snoopy Cache Coherency protocols:

1. 3 wired OR signals: In this technique, when any other cache has a copy of block besides the requester the first signal is asserted, and when any cache has exclusive copy of block the second signal given. The third signal is asserted when all snoop actions are finished on the bus. [3] When the third signal is asserted, the other two signals are safely examined by the requesting L1 and the L2. Performance can be improved by implementing these signals using low-latency L-Wires since all of them are on the critical path.

[1] A recent proposal [77] attempts to overcome these disadvantages by using timestamps to reorder messages on an arbitrary unordered interconnect to reestablish a total order of requests; however, this proposal adds significant complexity to the interconnect.

2. Voting wires is another technique used to enhance snoopy based protocol with low latencies. Generally cache to cache transfers occur from the data in the modified state, whereas there is a single supplier. [78] Although, a block can be retrieved from other cache rather than memory in MESI protocol. Multiple caches share copy voting mechanism is generally used to provide data therefore voting mechanism works with low latencies and enhances processor performance.

4.1.5 Implementing Snoopy Cache Coherence

In this section, we will describe the basic implementation technique and the proposed Snooping Protocol Implementation.

4.1.5.1 Basic Implementation Techniques

Using the bus is the key to applying an invalidate protocol in a small-scale multichip processor, or another broadcast medium, to carry out invalidates. The processor maintains bus access and broadcasts the address to be invalidated on the bus in order to perform invalidation. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated.

When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors try to write shared blocks simultaneously, when they arbitrate for the bus their attempts to broadcast an invalidate operation will be serialized. The first processor that acquired bus access will cause the writing of all other copies of the block to be invalidated. If the processors tried to write the same block, their writes are also serialized by the bus. One implication of this scheme is that a write to a shared data item cannot be finished until a bus access is acquired. Either by serializing access to the communication medium or another shared

structure; all coherence schemes require some method of serializing accesses to the same cache block.

Allocating a data item when a cache miss occurs is also necessary in addition to invalidating outstanding copies of a cache block that is being written into. Finding the recent value of a data item in a write-through cache is rather simple, because all written data are always sent to the memory, where the most recent value of a data item can always be obtained. Using write through makes the implementation of cache coherence simple in a design with adequate memory bandwidth supporting the write traffic from the processors.

Hence, the latest value of a data item can be in a cache rather than in memory for a write-back cache, the problem of finding the latest data value is even more complex. Fortunately, write-back caches the same snooping scheme can be used both for cache misses and for writes: Each processor snoops every address placed on the bus. If a processor figures that it acquires a dirty copy of the requested cache block, it provides that cache block as a respond to the read request and causes the abortion of memory access. The additional complexity comes from being bound to retrieve the cache block from a processor's cache, which can often take more time than retrieving it from the shared memory if the processors are in separate chips. Since write-back caches provide lower requirements for memory bandwidth, they can facilitate for a larger numbers of faster processors and have been chosen to be the approached used in most multiprocessors, despite the additional complexity of maintaining coherence. Therefore, we will examine the implementation of coherence with write-back caches.

In order to apply the process of snooping, normal cache tags can be used, and the valid bit for each block makes invalidation easy to apply. Whether caused by an invalidation or by some other event, read misses are also straightforward because they

simply depend on the snooping capability. For writes knowing whether any other copies of the block are cached is necessary, because if there are no other cached copies, then the write has to be located on the bus in a write-back cache. Not sending the write decreases both the time taken by the write and the required bandwidth.

The cache-address tags must be checked by every bus transaction, which might interfere with processor cache accesses. Duplicating the tags is a way to reducing this interference. By directing the snoop requests to the L2 cache, the interference can also be decreased in a multilevel cache, which the processor uses solely when it has a miss in the L1 cache. In order for this scheme to operate, each entry in the L1 cache must be present in the L2 cache, a property by the name of inclusion property. If a hit in the L2 cache happens to the snoop, it must then arbitrate for the L1 cache to update the state and possibly retrieve the data, which requires a stall of the processor normally. Duplicating the tags of the secondary cache may even be useful sometimes as to further decrease contention between the processor and the snooping activity.

4.1.5.2 The proposed Snooping Protocol Implementation: SNOOPING

We applied a traditional, but aggressive, MOESI snooping protocol optimized for migratory sharing [11]. This thesis will use the notation SNOOPING to refer to the applying of this specific protocol. Our implementation is based on a modern snooping protocol [64].

A processor injects the request message into the interconnect, in order to issue a request and the requester waits to see its own request return to it on the interconnect. This method of issuing requests avoids an explicit global-arbitration mechanism by permitting the interconnect to determine the exact total order that requests will be delivered (e.g., an indirect interconnect could order requests at its root). All processors

will see the requests in the same order due to the total order of requests, allowing each of them to make a globally consistent decision (much like a shared-wire bus).

The processor logically has the permissions and responsibilities associated with its new coherence state once it has observed its own request, even though in most cases the data response will have not arrived yet. This window of vulnerability leads to many complex race cases.

In order to avoid explicit acknowledgment messages from each processor the total order of requests is also mandatory. Instead of using explicit acknowledgment messages, when a requesting processor observes its write request in the total order, it knows that all other processors have logically invalidated any copies. However, because the delivery of coherence requests is permitted to be unsynchronized, this invalidation guarantee only holds in logical time—not physical time. This approach to cache coherence can provide a sequentially consistent view of the memory system to the software, as long as all processors enforce coherence permissions in logical time and take care not to reorder various types of messages.

Instead of using snooping response combining (as used in many snooping systems), by maintaining two bits per block in memory SNOOPING avoids the complexity and latency of snoop response combining. The first bit determines if responding to requests for the block is the memory's responsibility (i.e., no other processor is in OWNED, EXCLUSIVE, or MODIFIED). The second bit determines if the memory is allowed to respond to a read request for the block with an exclusive-data response (thus permitting the recipient to transition to EXCLUSIVE). Snoop response combining avoidance is especially attractive in this protocol due to its non-synchronous

nature[1]. These two bits can be encoded by the memory controller using the memory's error correction (ECC[2]) bits.

4.2 Directory-based Protocols

Memory is distributed among different processors in directory based protocols and directory is maintained for each such memory. Currently, several Chip Multiprocessors, as Piranha [79] also use directory protocols in order to maintain cache coherency. L1 cache misses are sent to L2 caches and a directory which store the status of block is maintained across each L2 caches. The request goes to home node where the original data is stored to check whether it has. When request comes from requester node from another cache, if it is not available the request goes to remote node by home node and first fetches data from remote node and sends it later on to requester node. Also write-invalidate-direct based protocol is employed in one of the most common chip multiprocessors technology we are using, which is core2duo. In directory based protocol, a coherence message is sent to the directory at the L2 to check the cache line's global state, when a request misses is placed in an L1 cache. If there is a clean copy in the L2 and the request is a READ, it is served by the L2 cache. [55] Otherwise, another L1 must hold an exclusive copy and the READ request are sent to the exclusive owner, which provides the data. Although, if any other L1 caches hold a copy of the cache line in a WRITE request, coherence messages are sent to each of them asking for the

[1] The "non-synchronous" means that a request does not have to arrive at all destinations in the same system clock cycle and that this system clock is not bound to being tightly synchronized. We do not use this term to imply that the implementation of the system will use any sort of asynchronous logic.

[2] Chen and Hsiao [80] and Peterson and Weldon [81] provide an overview of various error correcting codes.

invalidation of their copies and the requesting L1 cache acquires the block in exclusive state only after all invalidation messages have acknowledged.

4.2.1 An Example:

Figure 4.3 shows an example of Directory based protocol where the home directory is the central unit and all requests and permissions done by it.

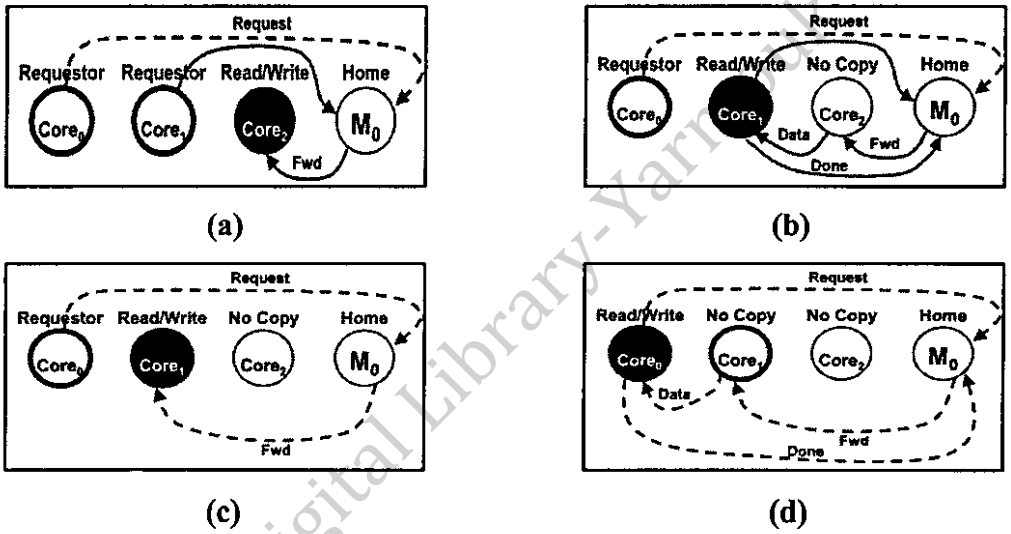


Figure 4.3 Directory based protocol example.

Directory protocols target the avoidance of the scalability and interconnection limitations of snooping protocols. Directory protocols predate snooping protocols for a fact, with Censier and Feautrier [84] and Tang [85] performing early work on directory protocols in the late 1970s. Systems that use these protocols—also known as distributed shared memory (DSM) or cache-coherent non-uniform memory access (CC-NUMA) systems—are preferred when scalability (in the number of processors or cores) is a first-order design constraint. These protocols often sacrifice fast cache-to-cache misses in exchange for this scalability, even though these protocols are significantly more scalable than snooping protocols, Examples of systems that use directory protocols include Stanford's DASH [86,87] and FLASH [88], MIT's Alewife [89], SGI's Origin [17], the

AlphaServer GS320 [90] and GS1280 [91], Sequent's NUMA-Q [92], Cray's X1 [93], and Piranha [79].

The bus-based multiprocessor communicates through a bus and uses a snoop-based protocol, depending on a broadcast mechanism to invalidate or update the data in remote caches. Because of that, this basic mechanism is less scalable when accounting for the number of processors allowed on a single bus because of limited bandwidth and electrical load (capacitance). This lack of scalability is a main dilemma to the construction of a large-scale system with more than hundreds or even thousands of processors.

During the 1990s, the research on distributed shared memory (DSM) machines [17,73,79,86,87,88,89,90;92,94,95] focused mainly on scaling beyond the number of processors that may be kept by a single shared bus. The main memory is distributed across the entire system, and the DSM uses a shared address space. Because of that, the data access latency can be different unlike the bus-based machine, based on the location of the data. The DSM typically employs a directory-based cache coherence protocol [96] for maintaining cache coherence. The DSM explicitly sends requests to appropriate processing nodes after looking up the directory through network transactions Instead of resorting to broadcasting.

DSM's popularity decreased after the emergence of cluster computing. The availability of high-speed networks and increasingly powerful commodity microprocessors is making clusters of computers and networks an appealing solution for cost-effective parallel computing, even though the DSM machine was the dominant form of large-scale multiprocessor systems in the 1990s,. Nevertheless, as the mainstream of future computer architecture research moves towards the many-core (currently implying over eight processors on a chip) architecture, the directory-based

cache coherence is taken as a coherence protocol among distributed caches (e.g., L2 or L3) on a chip.

Avoiding broadcasting requests by only communicating with those processors that might actually be caching data is one goal of directory-based coherence. To avoid broadcast, a processor issues a request by sending it only to the home memory module for the block. The home memory holds a directory that encodes information about the state, and a superset of processors may be caching each of the blocks in that memory module (hence the name directory protocol). Home memory uses the directory information to respond directly with the data and/or forward the request to other processors, when it receives a request. For example, if $core_0$ sends a write request to the home memory, the directory state illustrates that no cores hold copies of the data at that time; the memory responds with the data block and updates the directory state so that it shows that $core_0$ is now in Modified. Further on, when $core_1$ sends a write request for the same to the memory module, it sends the request to $core_0$. $core_0$ supplies the data to $core_1$ as responce to the forwarded request. A bit vector is the simplest encoding for this information (one bit per core) for the sharers and a processor identifier (log2 n bits) for the owner. On the other hand, many researchers have suggested approximate encodings in order to reduce directory state overheads (e.g., [73,97,98,99,100]). These approximate encodings may define a superset of sharers to invalidate, but at the same time must still obtain all processors that could be sharing the block. On the other hand, numerous directory schemes exploit entries at the memory and caches to form a linked list of processors sharing the block [101].

The directory also plays a main role in providing a per-block ordering point to handle conflicting requests or eliminate various protocol races, in addition to tracking sharers and/or owners of a block. Anytime multiple messages for the same block are

active in the system at the same time, a protocol may take place. Since the directory observes all requests for a given block, the order in which requests are processed by the directory unambiguously determines the order in which these requests will occur in the system. In order to delay subsequent requests to the same block by queuing or negatively acknowledging (nacking) requests at the directory while a previous request for the same block is still active in the system; many directory protocols use multiple busy states (also known as pending or blocking states). Subsequent requests are only allowed to proceed past the directory when the first request has complete. When a request reaches the directory, a simple directory protocol might enter a busy state; a more optimized protocol enters busy states less frequently. The directory responds or forwards the request when necessary, and it only clears the busy state at the time the requester sends the directory an acknowledgment message illustrating the "all clear" signal. On the contrary, some directory protocols can avoid all busy states, especially protocols that depend on point-to-point ordering in the interconnect.

The use of explicit invalidation acknowledgment messages to allow requesters to detect completion of write requests is another important aspect of directory-based cache coherence. Protocols that use a total order of requests (for all blocks) to enable implicit acknowledgments unlike snooping, most directory protocols eschew a totally-ordered interconnect, therefore these protocols should depend on explicit invalidation acknowledgments.[1] The directory forwards the request to any potential sharers and/or the owner at the time a requester issues a write request. When each of these processors receives the forwarded requests they must send an explicit message to indicate that they invalidated the block. Having a per-block ordering point (i.e., the directory) is not

[1] The AlphaServer GS320 is a notable exception; it uses a totally ordered interconnect to avoid explicit acknowledgments [73].

convenient to avoid explicit acknowledgments because for implementing a consistency model the requester should be acknowledged when its request has been ordered with all other accesses in the system (not just those for the same block).

The following three characteristics of directory protocols result in both the advantages and disadvantages of directory protocols (described in the following):

- Tracking sharers/owner in a directory.
- Using the directory as a per-block ordering point.
- Explicit acknowledgments—directly.

4.2.2 Advantages of Directory Protocols

Directory protocols' better scalability than snooping protocols and avoidance of snooping's virtual bus interconnect are the two primary advantages of directory protocols. The most discussed and studied advantage is perhaps the significantly improved scalability of directory protocols. By only contacting those processors that might have copies of a cache block (or a small number of additional processors when using an approximate directory implementation), the traffic in the system grows linearly with the number of processors. In contrast, the endpoint traffic of broadcasts used in snooping protocols grows quadratically [11]. Combined with a scalable interconnect (one whose bandwidth grows linearly with the number of processors), Using directory protocol the system is permitted to scale to hundreds or thousands of processors.

Two scalability dilemmas are encountered when using large system sizes:

- First, the amount of directory state required becomes great concern.
- Second, interconnect of reasonable is not truly scalable.

A deep study of these two problems have been applied extensively, and actual systems supporting hundreds of processors exist (e.g., the SGI Origin 2000 [17]).

The ability to exploit arbitrary point-to-point interconnects is the second—and maybe the more important—advantage of directory protocols. The point-to-point interconnects are generally have high-bandwidth and low-latency [11].

4.2.3 Disadvantages of Directory Protocols

Directory protocols are of two main disadvantages. First, the extra interconnect traversal and directory access is on the critical path of cache-to-cache misses. Hence the memory lookup is normally performed simultaneously with the directory lookup memory-to-cache, misses do not incur a penalty. Directory lookup latency is similar to that of main memory DRAM in numerous systems, and thus locating this lookup on the critical path of cache-to-cache misses increases cache-to-cache miss latency considerably. Although the directory latency can be decreased by using fast SRAM in order to hold or cache directory information, the extra latency presented by the additional interconnect traversal is harder to mitigate. A combination of these two latencies enhances cache-to-cache miss latency significantly. With the prevalence of cache-to-cache misses in many important commercial workloads, these higher-latency cache-to-cache misses might have a dramatical impact on system performance.

The storage and manipulation of directory state could be considered the second—and perhaps less important—disadvantage of directory protocols. This disadvantage was present on earlier systems, ones that used dedicated directory storage (SRAM or DRAM) thus adding to the total system cost. On the other hand, to save directory state while eliminiating additional storage capacity overhead; numerous modern directory protocols have used the main system DRAM and reinterpretation of bits used for error correction codes (ECC) (e.g., the S3mp [103], Alpha 21364 [67], UltraSparc III [66],

and Piranha [79,104]). by increasing the number of memory reads and writes [104] storing these bits in main memory enhances the memory traffic, as Token Coherence can use a similiar method to hold token counts in main memory [11].

4.2.4 Techniques used to improve directory based protocol

In this section, we will describe some of the best techniques that can be used to improve the Directory based Cache Coherency protocols:

1: Exclusive Read Request for a block in a shared state

The L2 cache is a clean copy and upon receiving a request from the L1 cache the L2 cache is going to invalidate the every L1 cache in this approach. Before sending the data to the processor, The L1 cache will receive an invalidate acknowledgement from the other L1 caches. Normally the L2 cache requires a hop for the reply and where as it requires 2 hop's for an acknowledgment. So the latencies of the Hop and the latency's of reply and acknowledgement messages should be of the same value. In this approach both the acknowledgment and reply messages are sent at the same time via the corresponding low latency L-wires and low power PW – wires. This approach enhances the performance and lowers the consumption of power

2: Read request for block in exclusive state

The exclusive owner will receive a read request for the L2 cache sends as the requesting L1 receives a copy of data from the L2 cache. The exclusive owner will send a reply message to the requesting L1 cache pointing out that the data sent by the L2 cache is valid if the exclusive copy is a clean one. If the requesting node requests for data the exclusive owner will send a copy of data to the requesting L1 cache while updating the data in the L2 cache. The requesting cache will not go on until it receives a message from the exclusive owner. Simultaneously the data the L2-cache sends the data through slow PW wires.

An acknowledgement message should be sent to the requester from the exclusive owner through low bandwidth L-wires if the owner copy is a clean copy, whereas if the owner copy is a dirty copy then the message should be sent through the B-wires and the write back to the L2 is done via PW-wires. This approach mainly follows the ways of improving the performance by sending the prioritized data through the L-wires and the least prioritized one through PW-wires.

3: The methods of which the "Proximity – Aware coherence" protocols enhances the performance of a multicore thus by lowering the unnecessary access to the off-chip memory are described below.

CMP ARCHITECTURE WITH DIRECTORY-BASED COHERENCE: In this architecture 16 cores were arranged in a 4*4 mesh of tiles and each core holds a private L1 and L2 instruction and data caches, a network switch that connects to the on-chip network, a directory controller and an on-chip memory controller thus accessing either the directory or program memory. A problem of Non-uniform latencies between the nodes is present in this architecture, in which a nearer node will have a low latency comparative to that of a distant node with the increase of more no of core's and because of wire delays.

Baseline Coherence Protocol for read and write operations: In this protocol only if there is a read miss in its L2-cache does a requester send a read request to the home node, and then the home node checks the directory cache to check the coherence state. If the home node acknowledges that it is sharing the data, it will then ask the local L2-cache for request. If the home node sees that the block is being shared then it will send the data from the main memory, alternatively if it is a dirty exclusive read then it will pass the request on to the remote node and the remote node will forward the most updated data to the requester and as the home node similarly. [108] The requester will

send a read-exclusive request to the home node and it will then verify the request by the requester thus if it is in uncached state, all in case of write operation. If it is in shared state then the home node passes on an invalidate signal to all the nodes and waits for their acknowledgement. The request will be forwarded to its owner then the remote node will pass the data directly on to the requester and an ACK to the home node; if the requested block is a dirty block.

4.2.4 The proposed Directory Protocol Implementation: DIRECTORY

We apply a directory protocol that we will refer to as DIRECTORY in order to give a concrete comparison with other coherence protocols. This protocol is a standard full-map directory protocol inspired by the Origin 2000 [17] and Alpha 21364 [67]. The directory state is stored in the main memory DRAM [66,67,103,104] by the base system, but we also asses systems with perfect directory caches by simulating a zero cycle directory access latency.

We designed Directory as a low-latency directory protocol, therefore whenever we have to choose between reducing message count or reducing latency, we choose to reduce latency. No ordering in the interconnect is required in this protocol (not even point-to-point ordering) and no negative acknowledgments or retries are used, but it does line up requests at the directory controller by using busy states sometimes. The directory controller lines up messages on a per-block criterion, allowing the proceeding for non-busy blocks messages. To prohibit starvation the directory uses per-block fair queuing.

Each time the directory receives a request it enters a busy state. The busy state prohibits all subsequent operations on the block if the operation causes a change of ownership; although, multiple read requests to the same block are permitted to proceed in parallel thus by exploiting a special busy state that counts the number of parallel

requests (and unblocks only upon completion of all the parallel requests). Upon completion of any operation, a completion message is sent to the directory by the initiating processor in order to (1) remove the busy state and (2) let the directory know if the recipient received a clean or migratory data response (so the directory is informed if the responding processor invoked the migratory sharing optimization). These additional completion messages create interconnect traffic and increase controller occupancy; on the other hand, they also allow the supporting of all the MOESI states by the protocol and optimize for migratory sharing. [11]

Silent eviction of Shared blocks is supported by Directory, although, Modified, Owned, and Exclusive blocks call for a three-phase eviction process. When the processor sends a message to the directory asking permission to evict the block, the process is then initiated. As a response to this message the directory sends the processor an acknowledgment, and the directory switches to a waiting-for-writeback busy state. The processor sends a writeback message that includes the data (for blocks in Modified and Owned) or a data-less eviction message (for blocks in Exclusive) when it receives the acknowledgment. The directory updates memory and transitions to a non-busy state when it receives this message, thus concluding the process.

Our Directory protocol uses the MOESI states, unlike many directory protocols (e.g., [17,67]) thus being MESI protocols. As illustrated in Section 2.3.2, the Owned state enables decreasing traffic, but more importantly it can also decrease latency by permitting the system to source data from the Owned processor instead of fetching it from memory. when the combination of accessing the directory and one additional interconnect traversal is of a higher speed than a DRAM memory lookup, getting data from the Owner processor is faster than getting data from memory, For example, small systems with a high speed interconnect and high speed SRAM directory may have a

more rapid cache-to-cache misses than memory-to-cache misses. Allowing all protocols to use similar base states is considered another advantage of using an MOESI protocol thus, making them easier to compare quantitatively.

In the next section, we describe token coherence our third and final coherence protocol used for comparisons, which is one of the newest and efficient protocols.

4.3 Token-based Protocols

Even though designers and researcher encourage choosing directory protocols over snooping protocols because of their complexity and design verification, directory protocols obtain an important disadvantage of performance that must be acknowledged: directory protocols add indirection latency to cache-to-cache misses. As a solution to races, all requests are sent to a home node by a directory protocol; the home node then passes on the request (if needed) or responds with the data from memory. Alternatively, indirection is avoided by broadcasting all requests to all nodes by a snooping protocol, thus is done depending on interconnect ordering to facilitate. An extra interconnect traversal stays on the critical path of some cache misses, even though the additional indirection latency of a directory protocol may be partially mitigated by using a directory cache.

The constraint of directory indirection are eremoved without sacrificing either decoupling of the interconnect from the coherence protocol or decoupling of coherence from consistency thus is done by using the recently-proposed token coherence protocol [10,11,12]. Token coherence take on token counting to resolve races without the need to require home node or an ordered interconnect. Token coherence contains even further levels of decoupling by separating the correctness substrate from the system's

performance policy. The correctness substrate is decoupled further on invoking safety and avoiding starvation.

Token coherence's correctness substrate **enforces safety** by counting the tokens of each block of memory in the system. Each block in the system has a determined number of tokens (T). The processor is allowed to read and write the block if its cache has all T tokens for the block, if a processor's cache obtains a minimum of one token, it can read the block (but can't write it). The processor can neither read nor write the block if its cache holds no tokens. These token counting rules assure that if one processor is writing the block, no other processor is reading or writing it. Mainly, it directly enforces the multiple-reader-single-writer coherence invariant that is appropriate for allowing the processor to enforce the requested memory consistency model. These easy rules allow for reasoning about protocol safety in a much more basic way, and naturally, token coherence does not depend on complicated ordering properties of the interconnect or the use of a directory home node in resolving races.

Token counting does not ensure that a request is satisfied in the end even though it ensures safety. Thus, the correctness substrate gives persistent requests to **prevent starvation**. The processor initiates a persistent request when it detects possible starvation. Using a fair arbitration mechanism, the substrate then activates at most one persistent request per block. Each system node remembers all activated persistent requests (for example, in a table at each node) and forwards all tokens for the block—those tokens are present at the time being and received in the future—to the request initiator. Finally, the initiator performs a memory operation (a load or store instruction) and deactivates its persistent request when it has the necessary tokens.

4.3.1 Performance policies

The correctness substrate provides a foundation for implementing many performance policies. These performance policies are mainly with making the system fast and bandwidth-efficient, but are not obliged to any correctness responsibilities, since the substrate is responsible for correctness. This decoupling of responsibility between the correctness substrate and performance policy enables the enhancement of performance policies that hold many of the desirable attributes of snooping and directory protocols. For example, token coherence performance policies have been developed [11] to approximate an unordered broadcast-based protocol (inspired by snooping protocols), a bandwidth-efficient performance policy that emulates a directory protocol, and a predictive hybrid protocol that uses destination-set prediction [75].

4.3.2 Ramifications on design verification

Token coherence perhaps has some additional verification advantages. For example, Marty et al. [105] used a single-level token coherence protocol in order to approximate the performance properties of a two-level hierarchical coherence protocol. Hence, the protocol is flat for correctness but hierarchical for performance. Using token coherence in this way permits performance benefits of a hierarchical protocol in combination with the ease of verification of a single-level protocol. Marty et al. [105] also show that verifying a single-level directory protocol is comparable to the difficulty of verifying the token coherence correctness substrate. Also, the flexibility provided by the performance policy should permit a system using a token coherence protocol to be improved over time without major changes to the correctness substrate.

4.3.3Token Operation and Implementation

Throughout system initialization, the system assigns a fixed number of tokens for each block, T. In most implementations, each block will contain an identical number of tokens. Although, token counting proceeds even if blocks have a different number of tokens as long as all system elements are acknowledged of how many tokens are present in each block. The number of tokens for each block (T) is normally at least as much as the number of processors. Tokens are tracked per block and can be kept in processor caches, memory modules, coherence messages (in-flight or buffered), and input/output devices. A coherence message is any message that is sent as a component of the coherence protocol. We collectively point out to those devices that can hold tokens as system components. Basically, the block's home memory module contains all tokens of a block. Tokens and data are permitted to transfer between system components as long as the substrate follows the following rules:

1. **Conservation of Tokens**: Tokens must not be created or destroyed, once the system is initialized. This rule ensures that tokens are never created or destroyed by the substrate and enforces the invariant that at all times each block in the system have T tokens.

2. **Write Rule**: A system component can only write a block if it holds all T tokens for that block.

3. **Read Rule**: A system component can only read a block if it holds at least one token for that block.

   Second and third rules ensure that a processor cannot write the block while it's being simultaneously read by another processor.

4. **Data Transfer Rule**: A coherence message must contain data if it contains one or more tokens. This rule ensures that processors holding tokens maintain at all times a valid copy of the data block.

In more familiar terms, token possession maps directly to traditional coherence states (described in Section 2.3.2): holding all T tokens is MODIFIED; one to (T – 1) tokens are SHARED; and zero tokens are INVALID.

Unlike most coherence protocols, token coherence does not allow silent evictions (i.e., evicting the block without sending any messages). On the contrary, many traditional protocols allow silent evictions of clean (i.e., SHARED and EXCLUSIVE) blocks. Both token coherence and traditional protocols require writeback messages that contain data when replacing dirty (i.e., OWNED and MODIFIED) blocks. Our implementation is based on a TOKENB protocol invented by milo [11].

TOKENB protocol focuses on both avoiding indirection latency for cache-to-cache misses (like snooping protocols) and not requiring any interconnect ordering (like directory protocols). One seemingly clear approach is to directly send broadcasts on an unordered interconnect. The TOKENB achieved by using the following performance policies to avoid both interconnect ordering and indirection overheads:

- **Issuing Transient Requests:** TOKENB broadcasts all transient requests (*i.e.*, it sends them to all processors and the home memory for the block). This policy works in a good way (1) for moderate-sized systems where interconnect bandwidth is plentiful and (2) when racing requests are rare.

- **Responding to Transient Requests:** Components (processors and the home memory module) react to transient requests as they would in most MOESI

protocols (as previously described in Section 2.3.2). A component with no tokens (INVALID) ignores all requests. A component with only non-owner tokens (SHARED) ignores transient read requests, but responds to a transient write request by sending all of its tokens in a dataless message (like an invalidation acknowledgment in a directory protocol). A component with the owner token but not all other tokens (OWNED) sends the data with one token (usually not the owner token) on a read request, and it sends the data and all of its tokens on an write request. A component with all the tokens (MODIFIED or EXCLUSIVE) responds in the same way as a component in OWNED, with the exception given in the next paragraph.

TOKENB implements a well-known optimization for migratory data to optimize for common migratory sharing patterns (for more details refer to [11]). If a processor with all tokens has written the block as it received the block, it responds to read requests by sending data and all tokens (instead of the data and one token).

- **Reissuing Requests and Invoking Persistent Requests:** the performance policy reissues the transient request if a transient request was not finished after a short timeout interval. If the request was still not finished after an even longer interval, the processor invokes the persistent request mechanism. This approach allows the occasional race to be handled without the overhead of persistent requests, but yet it invokes continuous requests soon enough not to waste bandwidth and time reissuing transient requests frequently.

4.3.4 TokenB Coherence Example

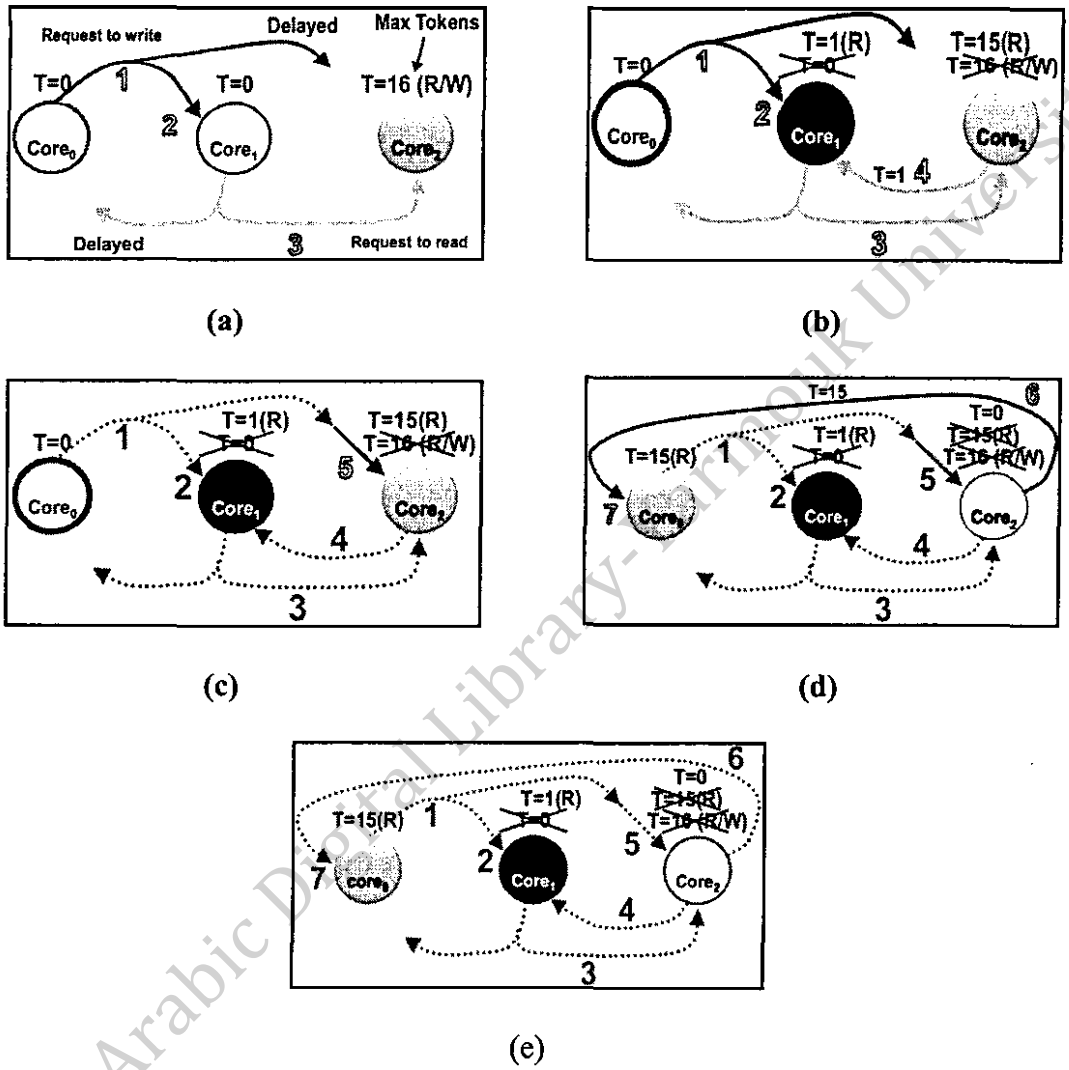


Figure 4.4 TOKENB based protocol example.

Figure 4.4 demonstrate the operation of Token based protocols, $Core_2$ firstly have all tokens, which means $Core_2$ can read and write while $Core_0$ and $Core_1$ each of them have zero tokens, which means they cannot read or write. In the first two steps $Core_0$ request to write by sending two messages to other cores but $Core_1$ in third step request to read and take the token in the fourth step before $Core_0$ requests deliver to $Core_2$, at this time $Core_1$ and $Core_2$ can only read because they have at minimum one but not all

tokens. In the fifth step $Core_0$ requests arrives to $Core_2$ and take all tokens from $Core_2$ but it cannot take write access because $Core_1$ has one token, for this reason $Core_0$ must issue its request another time.

4.4 Simulation Results

Tables 4.1 , 4.2 and 4.3 exhibit the basic simulation data for traditional protocols (Snooping and Directory) and TokenB protocol on GEMS/simics simulator.

We illustrate these simple simulation metrics in order to (1) give a short characterization of our workloads, and (2) give sufficient "raw" data to understand our preceding performance results (chapter5 discusses these data in more details). The system configuration name is exhibited in the first row of these tables. The second row illustrates that the number of cycles per transaction metric is worse (larger) for slower protocols (clearly). The third row shows that the number of misses per transaction is somewhat stable between different protocols and interconnects (also clearly).

Table 4.1 Snooping Coherence Protocol Results. Simulation data for Snooping coherence protocol.

	SPECjbb		Apache		OLTP	
configuration	**Perfect L2**	**Snooping**	**Perfect L2**	**Snooping**	**Perfect L2**	**Snooping**
Cycles / transaction	18,721	43,735	173,718	715,231	1,639,324	7,567,962
(L2 misses) / transaction	NA	179	NA	3,989	NA	44,710
Instructions / transaction	53,739	56,798	271,591	399,744	3,363,773	7,057,801
Cycles / instruction	0.348	0.770	0.640	1.789	0.487	1.072
Misses / (thousand instructions)	NA	3.15	NA	9.98	NA	6.33
(endpoint messages) / miss	NA	19.97	NA	18.93	NA	17.65
(interconnect bytes) / miss	NA	269.73	NA	237.91	NA	213.79

Table 4.2 Directory based Coherence Protocol Results. Simulation data for Directory coherence protocol.

	SPECjbb		Apache		OLTP	
configuration	**Perfect L2**	**Snooping**	**Perfect L2**	**Snooping**	**Perfect L2**	**Snooping**
Cycles / transaction	18,721	45,634	173,718	789,714	1,639,324	12,820,209
(L2 misses) / transaction	NA	178	NA	3,993	NA	49,265
Instructions / transaction	53,739	56,841	271,591	399,507	3,363,773	20,039,730
Cycles / instruction	0.348	0.803	0.640	1.977	0.487	0.640
Misses / (thousand instructions)	NA	3.13	NA	9.99	NA	2.46
(endpoint messages) / miss	NA	5.62	NA	5.02	NA	4.48
(interconnect bytes) / miss	NA	153.82	NA	126.20	NA	106.54

Table 4.3 TOKENB based Coherence Protocol Results. Simulation data for TOKENB coherence protocol.

	SPECjbb		Apache		OLTP	
configuration	**Perfect L2**	**Snooping**	**Perfect L2**	**Snooping**	**Perfect L2**	**Snooping**
Cycles / transaction	18,721	36,017	173,718	533,460	1,639, 324	5,716,102
(L2 misses) / transaction	NA	175	NA	3,792	NA	41,316
Instructions / transaction	53,739	55,840	271,591	337,045	3,363,773	5,781,223
Cycles / instruction	0.348	0.645	0.640	1.642	0.487	0.989
Misses / (thousand instructions)	NA	3.134	NA	11.250	NA	7.147
(endpoint messages) / miss	NA	17.72	NA	17.69	NA	17.91
(interconnect bytes) / miss	NA	371.23	NA	328.07	NA	301.90

Alternatively, some of the data in these tables are unsought for. The number of instructions per transaction is *not* stable between configurations for two of our three workloads. For example, the difference in number of instructions per transaction for OLTP between TOKENB and DIRECTORY is over a factor of three. Even though the source of these extra instructions per transaction is unknown, such a significant increase is related to the more time spent spinning on contended locks that are obtained more slowly or more time spent in the idle loop.

Unbounded interconnect link bandwidth are used in all of these simulated configurations. The "perfect L2" configuration simulates an idealized system in which all references hit in the second-level cache. The number of cycles executed by all cores per transaction (i.e., the total runtime in cycles multiplied by the number of cores) is present in the second row. The four following rows (from up to down) are: second-level cache misses per transaction, instructions per transaction, cycles per instruction, and misses per thousand instructions. The two other rows are metrics of system traffic: (1) endpoint messages per miss and (2) bytes on the interconnect links per miss.

Further details of the results, demonstrations and evaluations are discussed and analyzed in Chapter 5.

Chapter 5 Analysis and Evaluations

Because of the varying number of instructions per transaction in the results shown in tables 4.1, 4.2 and 4.3, normalizing by instruction count can be misleading. As these tables propose, the number of cycles per instruction (CPI) doesn't indicate at all times the actual runtime or throughput. For example, consider OLTP workload: DIRECTORY has a better (smaller) CPI than SNOOPING (the fifth row of results tables). By only examining the CPI, one might think that SNOOPING is slower than DIRECTORY. Whereas, the cycles per transaction row illustrate that SNOOPING is considerably faster than DIRECTORY (by about a factor of two); the misses per thousand instructions row is skewed as well by the non-stable number of misses per transaction. Due to these effects, we use cycles per transaction to be our only metric of performance.

These tables also contain the performance of a perfect second-level cache in addition to these protocols (SNOOPING, DIRECTORY and TOKENB). In these idealized simulations, all references hit in the perfect second-level cache. These tables exhibit a considerable difference between the "perfect L2" configuration and (SNOOPING, DIRECTORY and TOKENB) protocols in the number of cycles per transaction. For example, the perfect L2 configuration is almost twice as fast as (SNOOPING, DIRECTORY and TOKENB) protocols for SPECjbb; for OLTP perfect L2 can be above 5 times faster. Because the majority of the workloads time is spent in the memory system, and many of the cache misses are cache-to-cache misses, there is a considerable performance opportunity for protocols that optimize cache-to-cache misses.

5.1 Memory-to-Cache and Cache-to-Cache Misses

A cache-to-cache miss is a miss—caused mostly by accessing shared data—that requires the data to be supplied by another core's cache. The cache-to-cache misses are often present in commercial workloads and their effect on performance is significant. Figure 5.1 shows the normalized misses per transaction for our three workloads for a range of cache sizes to support this proposal.

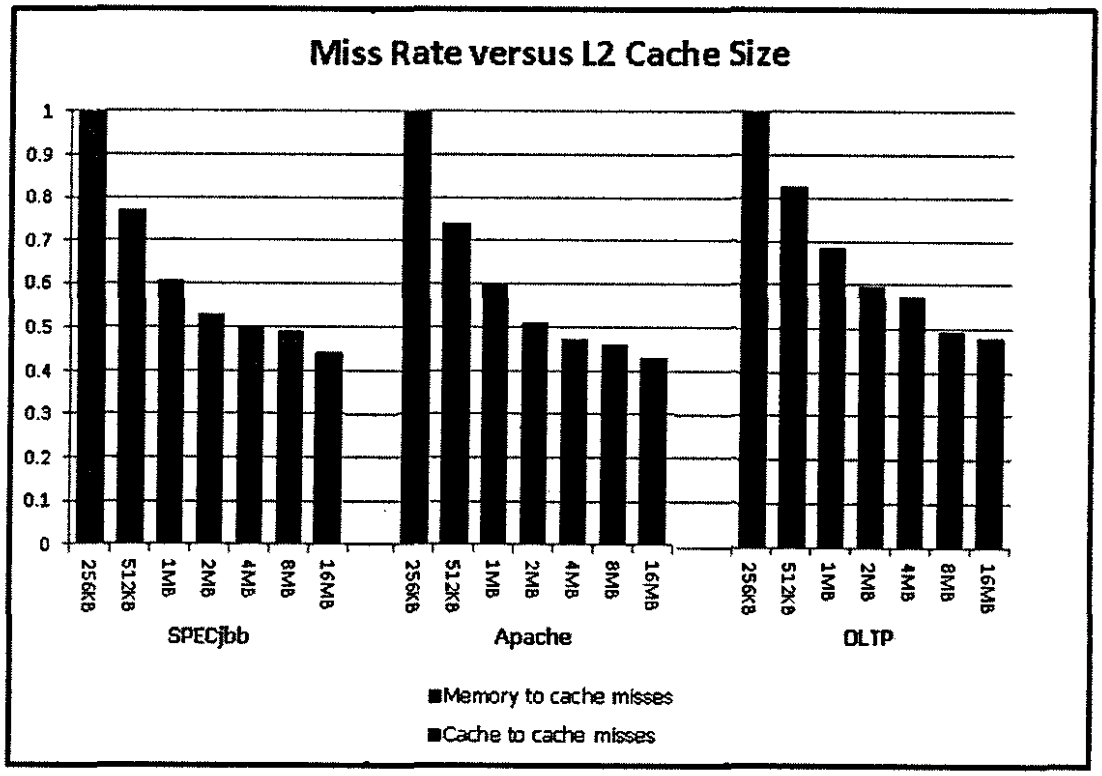


Figure 5.1 Miss Rate vs. Cache Size.

Figure 5.1 shows the miss rate in normalized misses per transaction for a range of second-level cache sizes (256KB to 16MB) is illustrated in this graph. Each bar in this graph is divided into two segments: memory-to-cache misses and cache-to-cache misses. These specific results are generated using the SNOOPING protocol (although the results are mostly independent of a particular protocol and interconnect).

This graph gives numerous different insights into our workloads:

- First, even for systems with large caches, the overall second-level miss rate for these workloads is significant. For example, for the SNOOPING the miss rate is 3.15-9.98 misses per thousand instructions (as illustrated in Table 4.1). This corresponds to the average of 100-317 instructions executed between each second-level miss. These misses have a bad effect on performance; as each second-level miss is hundreds of processor cycles long,

- Second, the graphs in Figure 5.1 exhibits the pre-sought results that both the overall number of misses per transaction (total height of each bar) and the memory-to-cache miss rate decrease as the cache size increases.

- Third, as cache size increases the number of cache-to-cache misses increases for a fact. This increase occurs because (1) larger caches don't eliminate misses caused by sharing, and (2) larger caches increase the probability another processor is caching the requested block (this impact is especially present in systems that support the EXCLUSIVE and OWNED states). For example, in the limit of infinite caches, in a MOESI protocol only the first access by any processor to a block will be a memory-to-cache miss and all other misses to that block (by any core) will be cache-to-cache misses.

If cache-to-cache misses are of less frequency than memory-to-cache misses the increasing cache-to-cache miss rate is a detriment to performance. For such systems, researchers have proposed decreasing cache-to-cache miss rates by using predictive invalidation techniques to proactively evict blocks [106,107]. On the contrary, many snooping protocols and directory protocols with a low-latency directory have faster cache-to-cache misses than memory-to-cache misses. In similar systems, large caches

improve performance by transforming memory-to-cache misses into cache-to-cache misses.

- Fourth, the overall effect of the previous two effects—the increasing number of cache-to-cache misses and decreasing number of memory-to-cache misses—produce a large percentage of cache-to-cache misses. 11-21% using 256KB L2 and 52-85% using 16MB of all second-level misses are cache-to-cache misses.

5.2 Indirection and its Effects on Performance

Directory protocols use indirection to avoid broadcast, but this indirection places a third interconnect traversal and directory access latency on the critical path of cache-to-cache misses. The decreased performance because of these two sources of overhead for a range of second-level cache sizes is exhibited in Figure 5.2.[1] We calculated the contribution of these sources of overhead using three simulations: (1) SNOOPING, (2) DIRECTORY without interconnect traversal and directory lookup latency, and (3) DIRECTORY. This experiment uses unbounded link bandwidth to isolate the impacts that are only due to uncounted miss latency.

In figure 5.2, the orange bar represents runtime of a system with a perfect L2 cache. The blue bar represents the runtime of SNOOPING, which suffers from neither overhead. The red bar represents the runtime of Directory including a third interconnect traversal rather than Snooping on the critical path of cache-to-cache misses. The green bar represents the runtime of Directory, which include a third interconnect traversal and include the impact of directory lookup latency.

[1] Even though cache hit latency increases frequently with cache size, these simulations use the same cache hit latency for all cache sizes to isolate the impact of the higher hit rate of larger caches.

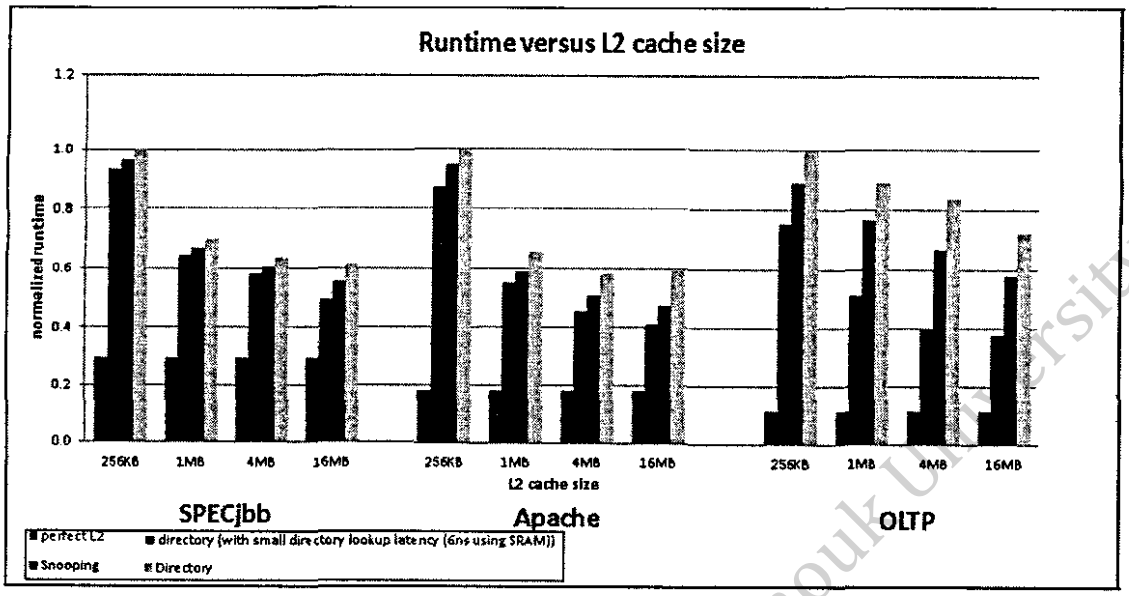


Figure 5.2 Runtime vs. Cache Size. The normalized runtime of three protocols for a range of second-level cache sizes (256KB to 16MB).

As follows from the results in Figure 5.1, both (1) the absolute number of cycles of overhead and (2) the percentage of the runtime because of indirection increase as cache size increases. In this specific set of experiments, the directory latency and interconnect traversal each calculate for approximately half of the runtime overhead. Although, the relative importance of these two factors relies directly on the relative latencies of average interconnect latency and directory access latency.

Placing a directory looking and third interconnect traversal on the critical path of cache-to-cache misses has a magnificent effect on performance. For example, for the 4MB cache configuration, eliminating the DRAM directory lookup using an SRAM directory results in a protocol that is 5–25% faster; removing only the interconnect traversal results in a protocol that is 5–70% faster, and removing both overheads results in a protocol that is 10–110% faster (*i.e.*, SNOOPING is 10–110% faster than DIRECTORY with a DRAM directory).

A simple back-of-an-envelope calculation shows these speeds are reasonable even though these performance differences seem large. For example, consider OLTP. 65% of

OLTP's misses are cache-to-cache misses while the memory-to-cache misses are 35%. For SNOOPING memory-to-cache and cache-to-cache misses are 444 cycles and 296 cycles long, respectively (these latencies were presented in Table 3.1). The average miss latency for OLTP and SNOOPING is (296 x 0.65) + (444 x 0.35) = 348 cycles. For DIRECTORY using a DRAM directory the memory-to-cache miss latency is the same (444 cycles), but the cache-to-cache miss latency increases to 592 cycles. The average miss latency for OLTP and DIRECTORY is (592 x 0.65) + (444 x 0.35) = 540 cycles. In this example, DIRECTORY's average miss latency is 55% more than SNOOPING's miss latency. As the majority of the time is spent in the memory system by these workloads (shown by the large gap between perfect and non-perfect second-level caches), a 55% increase in average cache miss latency will have a considerable impact on runtime.

5.3 TOKENB versus SNOOPING

Figure 5.3 shows normalized runtime (smaller is better) that TOKENB is faster than SNOOPING with unlimited bandwidth links by (21–34%).

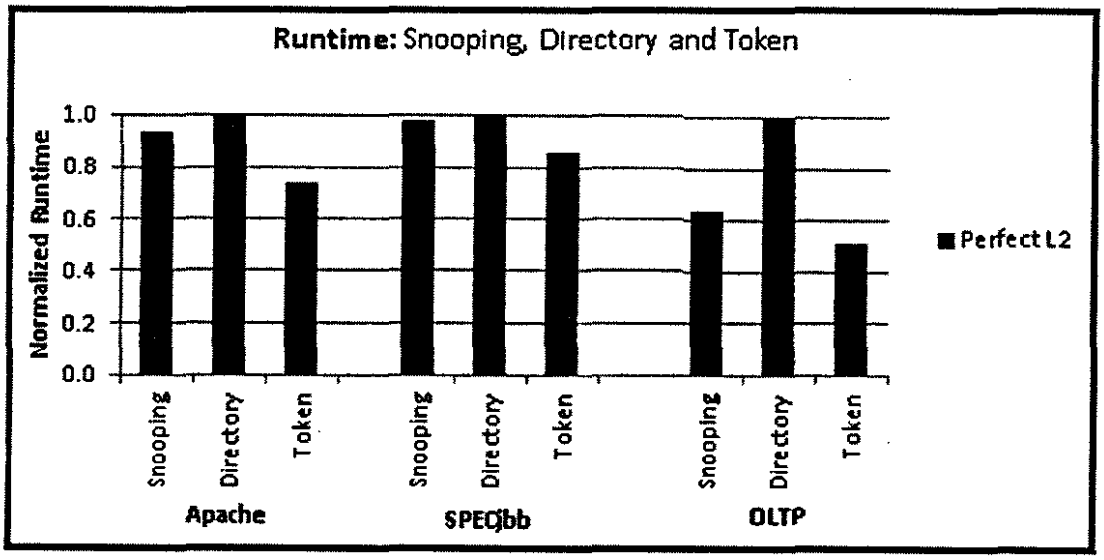


Figure 5.3 Runtime of SNOOPING, DIRECTORY, and TOKENB. The runtime of SNOOPING, DIRECTORY, and TOKENB with unbounded link bandwidth.

TOKENB's endpoint traffic is similar to or less than SNOOPING while has more interconnect traffic than SNOOPING. Figure 5.4 illustrates the endpoint traffic (in normalized messages per miss received at each endpoint coherence controller), and Figure 5.5 exhibits the interconnect traffic (in normalized bytes per miss). When examining only data and non-reissued request traffic, TOKENB and SNOOPING are practically identical. TOKENB adds some additional traffic overhead (comes from reissued and persistent requests), but the overhead is small for all three of our workloads. SNOOPING and TOKENB both use extra traffic for writeback control messages, but because of the detailed implementation decisions in SNOOPING involving writeback acknowledgment messages, SNOOPING uses more traffic for writebacks than TOKENB. SNOOPING sends a writeback request on the ordered interconnect to both the memory and to itself as a marker message. If it is still the owner of the block, it receives the marker message and sends the data back to the memory. Ignoring this precise implementation overhead leads us to conclude that these protocols generate the same amounts of traffic.

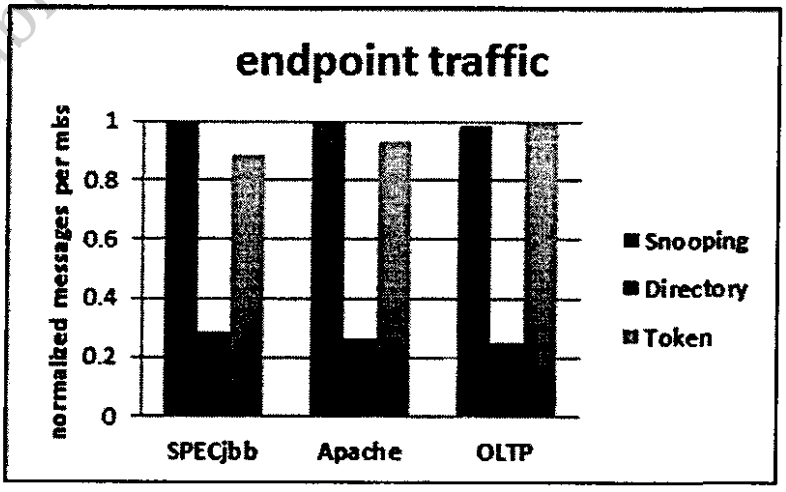


Figure 5.4 Endpoint Traffic of SNOOPING, DIRECTORY, and TOKENB. The endpoint traffic (in normalized messages per miss) of SNOOPING, **DIRECTORY**, and TOKENB.

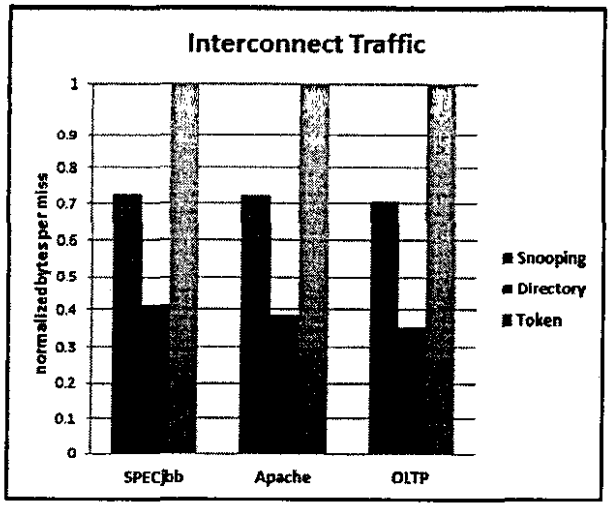


Figure 5.5 Interconnect Traffic of SNOOPING, DIRECTORY, and TOKENB.

The interconnect traffic (in normalized bytes per miss) of SNOOPING, **DIRECTORY**, and TOKENB.

5.4 TOKENB versus DIRECTORY

TOKENB is faster than DIRECTORY by (26–124%) even if we eliminate the directory lookup latency and interconnect traversal from the critical path of cache-to-cache misses the TOKENB will still faster than DIRECTORY by (7-27%). Figure5.3 illustrates the normalized runtime (smaller is better) with unbounded link bandwidth.

TOKENB is faster than DIRECTORY because it (1) avoids the third interconnect traversal for cache-to-cache misses, (2) avoids the directory lookup latency (DIRECTORY only), and (3) eliminates blocking states in the memory controller. TOKENB remains faster than DIRECTORY by 7–27% even if the directory lookup latency is reduced to 6ns (to approximate a fast SRAM directory or directory cache)

TOKENB generates bigger endpoint traffic and interconnect traffic than DIRECTORY. Figure 5.4 shows a traffic breakdown in normalized endpoint messages per miss (smaller is better) for TOKENB, and DIRECTORY. Figure 5.5 illustrates traffic regarding interconnect traffic in bytes per miss (smaller is better) for the same protocols.

These figures show TOKENB generates *more* traffic than DIRECTORY about three times more endpoint traffic and interconnect traffic). Thus results in, incredibly higher bandwidth coherence controllers are required by TOKENB. Luckily, researchers have suggested several techniques for creating high-bandwidth and low-power coherence controllers (*e.g.*, [108,109,110]), and these techniques can be readily performed on Token Coherence.

TOKENB depends on broadcast, which limits its scalability. TOKENB is less scalable than DIRECTORY, hence DIRECTORY avoids broadcast.

However, as the number of processors increases, TOKENB endpoint bandwidth improves linearly. The interconnect traffic difference between TOKENB and DIRECTORY enhances slowly. Thus, TOKENB can operate well for almost up to 64 processors if bandwidth is rich (by using high-bandwidth links and high-throughput coherence controllers). On the other hand, TOKENB is not a good choice for larger or more bandwidth-limited systems.

Chapter 6 Conclusions and Recommendations for Future Work

6.1 Conclusions

The cache coherence mechanisms are a key element aiming at accomplishing the goal of proceeding exponential performance growth through widespread thread-level parallelism. The available efficient methods and protocols were studied in this thesis, which were used to achieve cache coherent in multicore architectures. These protocols (Snooping-based protocols, Directory-based protocols and TOKENB-based protocols) modeled and evaluated on the simics/GEMS. The weaknesses and strengths of each protocol were demonstrated and we discussed how the improvement of them can be done.

TOKENB is both (1) better than SNOOPING and (2) faster than DIRECTORY when bandwidth is plentiful. TOKENB is better than SNOOPING because it uses the same amounts of traffic and can outperform SNOOPING by exploiting a faster, unordered interconnect. As discussed in Chapter 1, such interconnects might as well present high bandwidth more cheaply by avoiding dedicated switch chips. TOKENB is of a higher speed than DIRECTORY in bandwidth-rich situations by avoiding placing directory lookup latency and a third interconnect traversal on the critical path of common cache-to-cache misses. On the other hand, TOKENB uses a moderate amount of additional interconnect traffic and considerable more endpoint message bandwidth than DIRECTORY for small systems. Thus, DIRECTORY performs better than TOKENB in a bandwidth-constrained situation. Although TOKENB is a message-intensive protocol, it is only one of many possible high performance policies.

6.2 Recommendations for Future Work

The choice of coherence protocol is as complicated and subtle today as it has ever been. CMPs will enable even more cost-effective multicore processors by reducing the number of discrete components in the system.

For the future work, we recommend further modeling for each of the snooping, Directory, and Token protocols by testing additional benchmarks rather than the three benchmarks that we use in this thesis, in the same architecture for more accuracy results.

CMPs may change the design tradeoff between large processor cores (for highest uniprocessor performance using a large number of transistors) and more area-efficient cores (for moderate per-core performance using dramatically fewer transistors).

Such multi-level and hierarchical coherence protocols often add complexity to already complex systems. Coherence for Multiple-CMP Systems (hierarchical systems) and Virtual hierarchies needs evaluation and testing their reliability, scalability and cost efficiency.

A Multiple-CMP system combines many CMP chips together to form a larger, shared memory system. These systems will require mechanisms to keep caches coherent both within CMPs and between CMPs.

Unlike prior multiprocessors built using single-core processors, Multiple-CMP systems (or M-CMPs) use a CMP as the basic building block. In the short term, vendors will continue to build modest-sized M-CMPs that continue to support a single, logically shared memory. However the techniques for doing so present different tradeoffs.

REFERENCES

[1] Gelsinger, Dr. Paolo Gargini, Gerhard Parker and Albert Yu, **Microprocessors Circa 2000**, IEEE Spectrum, October 1989.

[2] Multi Processors, their Memory organizations and Implementations by Intel & AMD.http://ece.uic.edu/~wenjing/courses/fa08ECE569/ECE569/w21.pdf, 2009.

[3] D. Geer, **"Chip Makers Turn to Multicore Processors"**, Computer, IEEE Computer Society, May 2005.

[4] Josef Spjut, Andrew Kensler, Daniel Kopta and Erik Brunvand, **"TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing"**, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 28, NO. 12, DECEMBER 2009.

[5] D. Geer, **"For Programmers, Multicore Chips Mean Multiple Challenges"**, ***Computer***, IEEE Computer Society, September 2007.

[6] Wei-Chun Ku, Shu-Hsuan Chou, Jui-Chin Chu, Chi-Lin Liu, Tien-Fu Chen, Jiun-In Guo, and Jinn-Shyan Wang, **"VisoMT: A Collaborative Multithreading Multicore Processor for Multimedia Applications with a Fast Data Switching Mechanism"**, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. 19, NO. 11, NOVEMBER 2009.

[7] Markus Levy and Thomas M. Conte, **"EMBEDDED MULTICORE PROCESSORS AND SYSTEMS"**, IEEE MICRO, MAY/JUNE 2009.

[8] G. E. Moore. **"Cramming More Components onto Integrated Circuits"**. Electronics, pages 114–117, Apr. 1965.

[9] Nagarjun Manchineni, Rupesh Koneru, Venkata Sangana . **"Improving the performance of a Multicore architecture with different Cache coherence protocols and its related work"**, Cleveland State University,

http://academic.csuohio.edu/yuc/ca581/project07/team5.doc

[10] M.M. K.Martin,M. D. Hill, and D. A.Wood. **Token Coherence: A New Framework for Shared-Memory Multi-processors**. IEEE Micro, November-December 2003.

[11] Milo M. K. Martin. **Token Coherence**. PhD thesis, University of Wisconsin, 2003.

[12] Milo M. K. Martin, Mark D. Hill, and David A. Wood. **Token Coherence: Decoupling Performance and Correctness**. In Proceedings of the 30th Annual International Symposium on Computer Architecture, pages 182–193, June 2003.

[13] L. A. Barroso, K. Gharachorloo, and E. Bugnion. **Memory System Characterization of Commercial Workloads**. In Proceedings of the 25th Annual International Symposium on Computer Architecture, pages 3–14, June 1998.

[14] W. J. Dally and J. W. Poulton. **Digital Systems Engineering**. Cambridge University Press, 1998.

[15] M. Horowitz, C.-K. K. Yang, and S. Sidiropoulos. **High-Speed Electrical Signaling: Overview and Limitations**. IEEE Micro, 18(1), Jan/Feb 1998.

Ahmed, P. Conway, B. Hughes, and F. Weber. **AMD Opteron Shared Memory MP Systems**. In Proceedings of the 14th HotChips Symposium, Aug. 2002.

[16] URL http://www.hotchips.org/archive/hc14/program/28_AMD_Hammer_MP HC_v8.pdf.

[17] J. Laudon and D. Lenoski. **The SGI Origin: A ccNUMA Highly Scalable Server**. In Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 241–251, June 1997.

[18] J. R. Mashey. **NUMAflex Modular Design Approach: A Revolution in Evolution**. Posted on comp.arch news group, Aug. 2000.

[19] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu,M. Chen, and K. Olukotun. **"The Stanford Hydra CMP".**IEEE Micro, 20(2), 2000.

[20] L. Hammond, B. Nayfeh, and K. Olukotun, **"A single-chip multiprocessor".** IEEE Computer, 30(9), 1997.

[21] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. **"Piranha: A scalable architecture based on single-chip multiprocessing".** In ACM ISCA, 2000.

[22] www.sun.com/processors/UltraSPARC-T1/, 2009.

[23] Intel shows off 80-core processor. **www.news.com**/2100-1006 3-6158181.html, 2009.

[24] www.tilera.com, 2009.

[25] Hasselbring, Wilhelm. **"Programming Languages and Systems for Prototyping Concurrent Applications. "** ACM Computing Surveys, Vol. 32, No. 1 (March): 43-79. 2000.

[26] Ankit Jain, **"SOFTWARE DECOMPOSITION FOR MULTICORE ARCHITECTURES"**, M.Sc. thesis, Florida Atlantic University, Boca Raton, Florida, May, 2006.

[27] http://en.wikipedia.org/wiki/Multi-core ,2009.

[28] D. Chandra, F. Guo, S. Kim and Y. Solihin, **"Predicting inter-thread cache contention on a chip multiprocessor architecture"**, In Proc. 11th International Symposium on High Performance Computer Architecture (HPCA), 2005.

[29] L. Hsu, S. Reinhardt, R. Iyer and S. Makineni, **"Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource"**, International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006.

[30] R. Iyer, **"CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms"**, 18th Annual International Conference on Supercomputing (ICS'04), July 2004.

[31] Sutter, Herb. 2005. **"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software"**. Dr. Dobb's Journal, 30(3) March. Internet. Available from http://www.gotw.ca/publications/concurrency-ddj.htm.

[32] Forsell, Martti. 2003. **"A Parallel Computer as a NoC Region"**. Edited by Axel Jantsch and Hannu Tenhunen, Networks on Chip. Boston: Kluwer Academic Publishers.

[33] John L. Hennessy and David A. Patterson, **Computer Architecture A Quantitative Approach**, Fourth Edition, 2007.

[34] http://encyclopedia.thefreedictionary.com/multicore+processor.

[35] http://en.wikipedia.org/wiki/Multi-core.

[36] http://techfreep.com/intel-80-cores-by-2011.htm

[37] Microsoft Computer Dictionary, Fourth Edition, 1999.

[38] http://computer.howstuffworks.com/cache.htm , accessed in 2010.

[39] http://www.pcguide.com/ref/mbsys/cache/funcWhy-c.html , accessed in 2010.

[40] Bradford M. Beckmann, Michael R. Marty and David A. Wood, **"ASR: Adaptive Selective Replication for CMP Caches"**, 39th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-39), 2006.

[41] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. **"Power5 System Microarchitecture"**. IBM Journal of Research and Development, 49(4), 2005.

[42] P. Kongetira. **"A 32-way Multithreaded SPARCE Processor"**. In Proceedings of the 16th HotChips Symposium, Aug. 2004.

[43] K. Krewell. **"UltraSPARC IV Mirrors Predecessor"**. Microprocessor Report, pages 1–3, Nov. 2003.

[44] C. McNairy and R. Bhatia. Montecito: **"A Dual-Core Dual-Thread Itanium Processor"**. IEEE Micro, 25(2):10–20, March/April 2005.

[45] Mehmet ŞENVAR, **Cache Coherence Protocols in Shared Memory Multiprocessors**, project report, Boğaziçi University, 2005.

[46] Virtutech AB. http://www.virtutech.com/.

[47] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R.Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. **Multifacet's general execution-driven multiprocessor simulator (gems) toolset.** SIGARCH Computer Architecture News, 33(4):92–99, 2005.

[48] Alaa R. Alameldeen and David A. Wood. **Ipc considered harmful for multiprocessor workloads**. IEEE Micro, 26(4):8–17, 2006.

[49] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A.Wood. **Simulating a $2MCommercial Server on a $2K PC.**

IEEE Computer, 36(2):50–57, Feb. 2003. URL

http://dx.doi.org/10.1109/MC.2003.1178046.

[50] A. R. Alameldeen and D. A. Wood. **Variability in Architectural Simulations of Multithreaded Workloads**. In Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture, pages 7–18, Feb. 2003.

[51] K. Gharachorloo, A. Gupta, and J. Hennessy. **Two Techniques to Enhance the Performance of Memory Consistency Models**. In Proceedings of the International Conference on Parallel Processing, volume I, pages 355–364, Aug. 1991.

[52] K. C. Yeager. **The MIPS R10000 Superscalar Microprocessor**. IEEE Micro, 16(2):28–40, Apr. 1996.

[53] William James Dally and Brian Towels. **"Principles and Practices of Interconnection Networks"**. Morgan Kaufman Publishers, 2004.

[54] Selim Erten, **"Development and Evaluation of a Memory Consistency and Cache Coherence Protocol for the Nocsim NoC Simulator"**, Master of Science Thesis, Stockholm, Sweden 2007.

[55] Culler, D. E., Singh, J. P., and Gupta, A., **"Parallel Computer Architecture: A Hardware/Software Approach"**. Morgan Kaufmann Publishers, Inc., 1999.

[56] Goodman, J. R., **"Using Cache Memory to Reduce Processor-Memory Traffic",** in Proceedings of the 10th Annual International Symposium on Computer Architecture, pp. 124–131, 1983.

[57] Frank, S. J., **"Tightly Coupled Multiprocessor System Speeds Memory–access Times",** Electronics, vol. 57, pp. 164–169, January 1984.

[58] Katz, R. H., Eggers, S. J.,Wood, D. A., Perkins, C. L., and Sheldon, R. G., **"Implementing a Cache Consistency Protocol"**, in Proceedings of the 12th Annual International Symposium on Computer Architecture, pp. 276–283, 1985.

[59] Thacker, C. P., Stewart, L. C., and Satterthwaite, E. H., **"Firefly: A multiprocessor workstation. IEEE Transactions on Computers"**, IEEE Transactions on Computers, vol. 37, pp. 909–920, August 1988.

[60] McCreight, E., **"The Dragon Computer System: An Early Overview"**, tech. rep., Xerox Corporation, Sept. 1984.

[61] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi. **"Scalability Port: A Coherent Interface for Shared Memory Multiprocessors"**. In Proceedings of the 10th Hot Interconnects Symposium, pages 65–70, Aug. 2002. URL http://www.hoti.org/archive/hoti10/program/Kumar_ScalabilityPort.pdf.

[62] J. Borkenhagen and S. Storino. **"4th Generation 64-bit PowerPC-Compatible Commercial Processor Design"**. IBM Server Group Whitepaper, Jan. 1999. URL http://www.ibm.com/servers/eserver/pseries/hardware/whitepapers/nstar.html.

[63] J. M. Borkenhagen, R. D. Hoover, and K. M. Valk. **"EXA Cache/Scalability Controllers"**. In IBM Enterprise X-Architecture Technology: Reaching the Summit , pages 37–50. International Business Machines, 2002.

[64] A. Charlesworth. **"Starfire: Extending the SMP Envelope"**. IEEEMicro, 18(1):39–49, Jan/Feb 1998.

[65] A. Charlesworth. **The Sun Fireplane Interconnect**. In Proceedings of SC2001, Nov. 2001.

[66] T. Horel and G. Lauterbach. **UltraSPARC-III: Designing Third Generation 64-Bit Performance**. IEEE Micro, 19(3):73–85, May/June 1999.

[67] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. **The Alpha 21364 Network Architecture**. In Proceedings of the 9th Hot Interconnects Symposium, Aug. 2001.

[68] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. **POWER4 System Microarchitecture**. IBMServer GroupWhitepaper, Oct. 2001. URL http://www.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.pdf.

[69] J. Archibald and J.-L. Baer. **Cache Coherence Protocols: Evaluation Using aMultiprocessor Simulation Model**. ACM Transactions on Computer Systems , 4(4):273–298, Nov. 1986. URL http://doi.acm.org/10.1145/6513.6514.

[70] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yaun, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvery, E. Hagersten, and B. Liencres. **Gigaplane: A High Performance Bus for Large SMPs.** In Proceedings of the 4th Hot Interconnects Symposium, pages 41–52, Aug. 1996.

[71] A. Charlesworth. **The Sun Fireplane SMP Interconnect in the Sun 6800**. In Proceedings of the 9th Hot Interconnects Symposium, Aug. 2001.

[72]. **"Interconnect-Aware Coherence Protocols for Chip Multiprocessors"**, Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, John B. CarterSchool of Computing, University of Utah{legion,navven,karthikr,Rajeev,retrac} @cs.utah.edu. Proceedings of the 33rd annual international symposium on Computer Architecture ISCA '06.

[73] A. Agarwal, R. Simoni,M. Horowitz, and J. Hennessy. **An Evaluation of Directory Schemes for Cache Coherence**. In Proceedings of the 15th Annual International Symposium on Computer Architecture, pages 280–289, May 1988.

[74] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. **Multicast Snooping: A New Coherence Method Using a Multicast Address Network**. In Proceedings of the 26th Annual International Symposium on Computer Architecture, pages 294–304, May 1999.

[75] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. **Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors**. In Proceedings of the 30th Annual International Symposium on Computer Architecture, pages 206–217, June 2003.

[76] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. K. Martin, and D. A. Wood. **Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol**. IEEE Transactions on Parallel and Distributed Systems , 13(6):556–578, June 2002.

[77] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. **Timestamp Snooping: An Approach for Extending SMPs**. In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems , pages 25–36, Nov. 2000.

[78] M. Galles and E.Williams. **"Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor"**.

[79] Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B., **"Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing,"** in Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 282–293, June 2000.

[80] C. L. Chen and M. Y. Hsiao. **Error-Correcting Codes for Semiconductor Memory Applications**: A State-of-the-Art Review. IBM Journal of Research and Development, 28(2), Mar. 1984.

[81] W. W. Peterson and E. J. Weldon, Jr. **Error-Correcting Codes** . MIT Press, 1972.

[82]. **"TLB and Snoop Energy-Reduction using Virtual Caches in Low-Power Chip-Multiprocessors"** Magnus Ekman, *Fredrik Dahlgren, and Per Stenström Department of Computer Engineering Chalmers University of Technology SE-412 96 Göteborg, SWEDEN {mekman,pers}@ce.chalmers.se. "Proceedings of the 2002 international symposium on Low power electronics and design ISLPED'02".

[83] S. V. Adve, V. S. Pai, and P. Ranganathan. **Recent Advances inMemory ConsistencyModels for Hardware Shared Memory Systems**. Proceedings of the IEEE, 87(3):445–455, Mar. 1999.

[84] L. M. Censier and P. Feautrier. **A New Solution to Coherence Problems in Multicache Systems**. IEEE Transactions on Computers , C-27(12):1112–1118, Dec. 1978.

[85] C. K. Tang. **Cache Design in the Tightly Coupled Multiprocessor System**. In Proceedings of the AFIPS National Computing Conference, pages 749–753, June 1976.

[86] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. **The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor**. In Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 148–159, May 1990.

[87] D. Lenoski, J. Laudon, K. Gharachorloo,W.-D.Weber, A. Gupta, J. Hennessy,M. Horowitz, and M. Lam. **The Stanford DASH Multiprocessor**. IEEE Computer, 25(3): pp. 63–79, Mar. 1992.

[88] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. C. apin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. **The Stanford FLASH Multiprocessor**. In Proceedings of the 21st Annual International Symposium on Computer Architecture, pp. 302–313, Apr. 1994.

[89] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K.Mackenzie, and D. Yeung. **The MIT Alewife machine: Architecture and Performance**. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 2–13, June 1995. URL http://doi.acm.org/10.1145/223982.223985.

[90] Gharachorloo, K., Sharma, M., Steely, S., and Doren, S. V., **"Architecture and Design of AlphaServer GS320,"** in Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 13–24, Nov. 2000.

[91] Z. Cvetanovic. **Performance analysis of the Alpha 21364-based HP GS1280 multiprocessor**. In Proceedings of the 30th Annual International Symposium on Computer Architecture, pages 218–229, June 2003. URL http://doi.acm.org/10.1145/859618.859643.

[92] T. D. Lovett and R.M. Clapp. **STiNG: A CC-NUMA Computer System for the Commercial Marketplace**. In Proceedings of the 23th Annual International Symposium on Computer Architecture, pp. 308–317, May 1996.

[93] D. Abts, D. J. Lilja, and S. Scott. **So Many States, So Little Time: Verifying Memory Coherence in the Cray X1**. In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS), Apr. 2003.

[94] Michael, M. M., Nanda, A. K., Lim, B.-H., and Scott, M. L., **"Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors,"** in Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 219–228, 1997.

[95] Chaiken, D., Kubiatowicz, J., and Agarwal, A., **"LimitLESS Directories: A Scalable Cache Coherence Scheme,"** in Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 224–234, 1991.

[96] Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A., **"Directory-Based Cache Coherence in Large-Scale Multiprocessors,"** IEEE Computer, pp. 49–58, June 1990.

[97] J. Archibald and J.-L. Baer. **An Economical Solution to the Cache Coherence Problem**. In Proceedings of the 11th Annual International Symposium on Computer Architecture, pages 355–362, June 1984. URL http://doi.acm.org/10.1145/800015.808205.

[98] A. Gupta, W.-D. Weber, and T. Mowry. **Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes**. In International Conference on Parallel Processing (ICPP) , volume I, pages 312–321, 1990. URL http://citeseer.nj.nec.com/gupta90reducing.html.

[99] M. Heinrich, V. Soundararajan, J. Hennessy, and A. Gupta. A **Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols**. IEEE Transactions on Computers , 28(2):205–217, Feb. 1999.

[100] B. W. O'Krafka and A. R. Newton. **An Empirical Evaluation of Two Memory-Efficient Directory Methods**. In Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 138–147, May 1990.

[101] D. Gustavson. **The Scalable Coherent Interface and related standards projects**. IEEE Micro, 12(1):10–22, Feb. 1992.

[102]."**Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures**" Jeffery A. Brown Department of Computer Science and Engineering University of California, San DiegoRakesh Science Laboratory University of Illinois Dean Tullsen Department of ComputerScience and Engineering University of California. Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures SPAA '07 .

[103] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, and M. Parkin. **The S3.mp Scalable Shared Memory Multiprocessor**. In Proceedings of the International Conference on Parallel Processing, volume I, pages 1–10, Aug. 1995.

[104] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk. **Efficient ECC-Based Directory Implementations for Scalable Multiprocessors**. In Proceedings of the 12th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD 2000) , Oct. 2000.

[105] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. **Improving Multiple-CMP Systems Using Token Coherence.** In Proceedings of the 11th IEEE Symposium on High-Performance Computer Architecture, Feb. 2005.

[106] A. R. Lebeck and D. A. Wood. **Dynamic Self-Invalidation: Reducing Coherence Over-head in Shared-Memory Multiprocessors**. In Proceedings of the

22nd Annual International Symposium on Computer Architecture , pages 48–59, June 1995.

[107] A.-C. Lai and B. Falsafi. Selective, **Accurate, and Timely Self-Invalidation Using Last-Touch Prediction**. In Proceedings of the 27th Annual International Symposium on Computer Architecture, pages 139–148, June 2000.

[108] I. Pragaspathy and B. Falsafi. **Address Partitioning in DSMClusters with Parallel Coherence Controllers**. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Oct. 2000.

[109] A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph. **High-Throughput Coherence Controllers**. In Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture, Jan. 2000.

[110] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. **JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers**. In Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture , Jan. 2001.

[111] Michael R. Marty, **CACHE COHERENCE TECHNIQUES FOR MULTICORE PROCESSORS**, PhD thesis, University of Wisconsin, 2008.

[112] Karin Petersen and Kai Li, **Multiprocessor Cache Coherence Based on Virtual Memory Support**, Journal of Parallel and Distributed Computing - JPDC , vol. 29, no. 2, pp. 158-178, 1995.

[113] L. A. Barroso and M. Dubois. **Cache Coherence on a Slotted Ring**. In Proceedings of the International Conference on Parallel Processing, pages 230–237, Aug. 1991.

[114] S. Kunkel. **IBM Future Processor Performance**, Server Group. Personal Communication, 2006.

APPENDECES

Appendix 1: ABBREVIATIONS

CC-NUMA	Cache Coherent - Non Uniform Memory Access
CMP	Chip Multiprocessor
CPI	cycles per instruction
CPU	Central Processing Unit
CU	Competitive Update
DRAM	Dynamic Random Access Memory
DSM	Distributed Shared Memory
ECC	Error Correction Codes
FSB	Front Side Bus
GBs	Giga Bytes
GEMS	General Execution-driven Multiprocessor Simulator
GHz	Giga Hertz
IPC	Instruction Per Cycle
KB	Kilo Byte
L1	Level 1 cache
L2	Level 2 cache
L3	Level 3 cache
L-wires	low latency – wires
MA	Multicore Architecture
MB	Mega Byte
MESI	Modified, Exclusive, Shared and Invalid

MHz	Mega Hertz
MIMD	Multiple Instruction Multiple Data
MOESI	Modified, Owned, Exclusive, Shared and Invalid
MOSI	Modified, Owned, Shared and Invalid
MSI	Modified, Shared and Invalid
ns	nano second
OLTP	Online Transaction Processing workload
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
PW – wires	low power – wires
RAM	Random Access Memory
SLICC	Specification Language including Cache Coherence
SMP	Symmetric Multiprocessor
SRAM	Static Random Access Memory
T#	Token number
TLB	Translation Look-aside Buffer
WI	Write Invalidate
WU	Write Update