A Survey of Cache Coherence Mechanisms in Shared Memory Multiprocessors

Ramon Lawrence
Department of Computer Science
University of Manitoba
umlawren@cs.umanitoba.ca

May 14, 1998

Abstract

This paper is a survey of cache coherence mechanisms in shared memory multiprocessors. Cache coherence is important to insure consistency and performance in scalable multiprocessors. A variety of hardware and software protocols have been proposed and investigated in simulation studies. Our contribution in this work is an in-depth survey of hardware and software protocols including recent work. It is our conclusion that hardware protocols are currently better than software protocols but are more costly to implement. As compiler technology improves and more focus is placed on developing efficient software protocols, the performance of software protocols can only improve as they show great promise for future work.

1 Introduction

The speed of processors has been increasing at a near exponential pace. Although the processor is the heart of the computer, it can not function without other computer components such as high-speed buses and memory. Unfortunately, the bandwidth of high-speed buses and the speed of memory have not increased as rapidly as the processor clock speed. Consequently, accesses to memory slow down the processor and degrade performance. Caches were developed as a bridge between a high-speed processor and a relatively slow memory. By storing subsets of the memory in a fast access region, the performance of the system is improved because most memory accesses can be satisfied by the cache instead of main memory due to the principle of locality of reference.

The disparity between processor and memory speed is especially important in parallel and shared memory multiprocessors. These machines are designed to execute high-performance

programs which require extensive computational resources. Thus, these programs are highly optimized for performance and cannot tolerate the long memory delays associated with accessing main memory. By adding caches to these machines, the memory delay is reduced, but the problem of cache coherence is added. As the processors access a shared memory, it is possible that two or more processors may store the same address in their cache. It then becomes necessary to insure that these values are consistent, otherwise program errors will occur.

The two basic methods for insuring consistency are hardware implementations and software protocols. In a hardware implementation, special hardware is added to the machine to detect cache accesses and implement a suitable consistency protocol which is transparent to the programmer and compiler. In a software approach, the hardware provides a cache mechanism and some instructions for accessing the cache, but relies on the compiler to generate consistent code. Parallel code may be made consistent by the compiler by inserting additional consistency instructions in the code. However, the static compiler analysis can not detect run-time consistency conflicts. Performance of hardware protocols is typically better because they can be implemented efficiently and can detect dynamic sharing sequences.

This survey provides an in-depth overview of cache coherence protocols for shared memory multiprocessors. Both hardware and software protocols are studied and compared. A background of cache coherence problems and simple cache management protocols is presented in Section 2. There are several design considerations, including choice of a consistency model, when constructing a cache which are also explained in that section. Section 3 covers hardware protocols. Hardware protocols include snooping protocols and directory-based protocols. Compiler-directed software protocols are discussed in Section 4. These protocols may rely on some or no hardware support. In Section 5, the differences and performance issues between hardware and software protocols are discussed. Finally, concluding remarks are given in Section 6.

2 Background material

This work surveys hardware and software protocols for cache coherence in shared memory multiprocessors. There have been other surveys on the cache coherence problem which compare protocols using simulation studies. A simulation study may be done using program traces or detailed simulation of address access patterns. Thus, the results obtained from work done in the cache coherence field are based on results extracted from realistic simulations of parallel programs from standardized test suites on realistic machine architectures.

In 1993, Gee *et al.* compared invalidate, update, and adaptive protocols for cache coherence in [6]. They showed that invalidate protocols were best for vector data, update protocols were best for scalar data, and adaptive protocols were the best on average. In this work, they proposed an adaptive protocol called Update Once protocol. In this protocol, the first write to an address is updated in all caches, while following updates invalidate the cache item in other caches holding the data item.

Also in 1993, Lilja[13] studied cache coherence in shared memory multiprocessors. In this extensive work, snooping coherence, directory coherence, and compiler-directed coherence were compared. As well, the effect of consistency models and cache implementation strategies was discussed.

This paper extends previous work by including more recent cache coherence protocols. In addition to the work previously surveyed, this paper also compares lock consistency, and newly defined hybrid consistency protocols to earlier work. The advances made in the areas of hybrid protocols and complier-directed protocols justify a new survey paper.

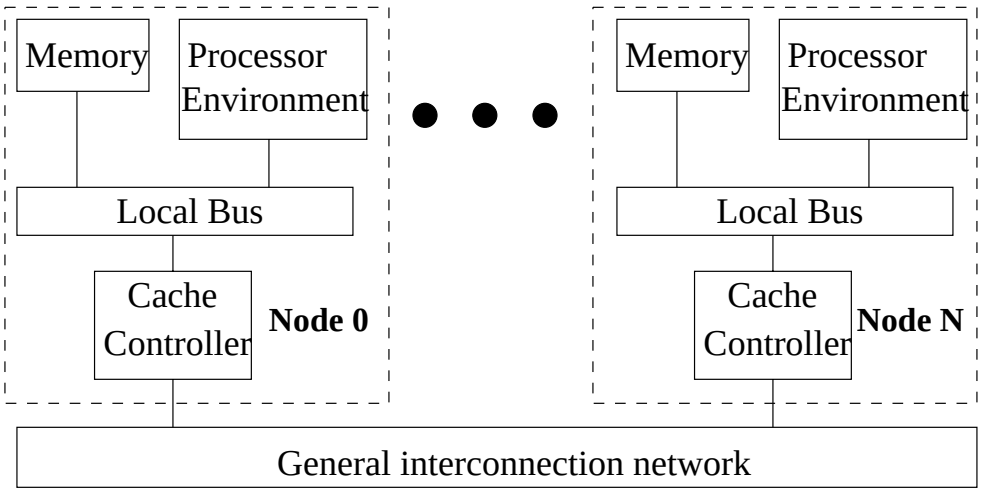
2.1 The Shared Memory Multiprocessor Architecture

A shared memory multiprocessor can be considered as a compromise between a parallel machine and a network of workstations. Although the distinction between these machines is blurring, a parallel machine typically has high-speed interconnects and a physically shared memory. That is, the processors often access the same physical memory or memory banks

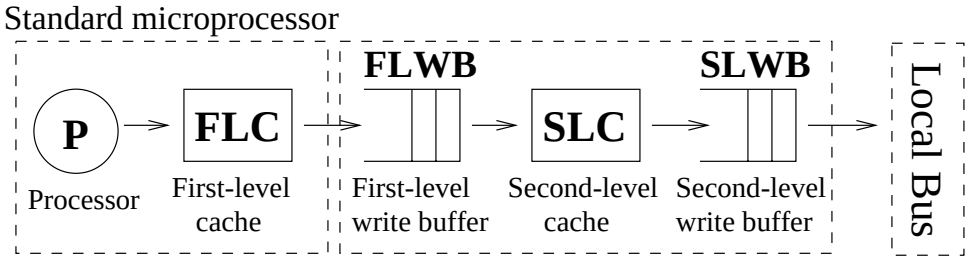
and are not distributed in space. A network of workstations (NOW) is on the other end of the spectrum. A NOW is a parallel machine constructed by combining off-the-shelf components and workstations using suitable software. There is typically limited consistency support in hardware as the machines were not specifically built for this architecture. However, consistency is maintained in these systems at the software level using traditional page-based hardware. A NOW provides a shared address space using the distributed shared memory (DSM) paradigm, but the machines were not specifically designed to share a common address space.

A shared memory multiprocessor (MP), shown in Figure 1, consists of processor nodes and memories combined into a scalable configuration. Each node has one or more processors and its local memory. Optionally, each node has a cache and a cache controller for accessing main memory efficiently and enforcing consistency. However, a shared memory MP differs from a network of workstations because although the physical memory is distributed across the nodes, all nodes share the same global address space. Hence, software techniques for mapping the global address space into local addresses are typically not needed in a shared memory MP. A shared memory MP also has fast interconnection networks which are used to access the distributed memory and pass consistency information. Since the physical memory is distributed, these machines are non-uniform memory access (NUMA) machines.

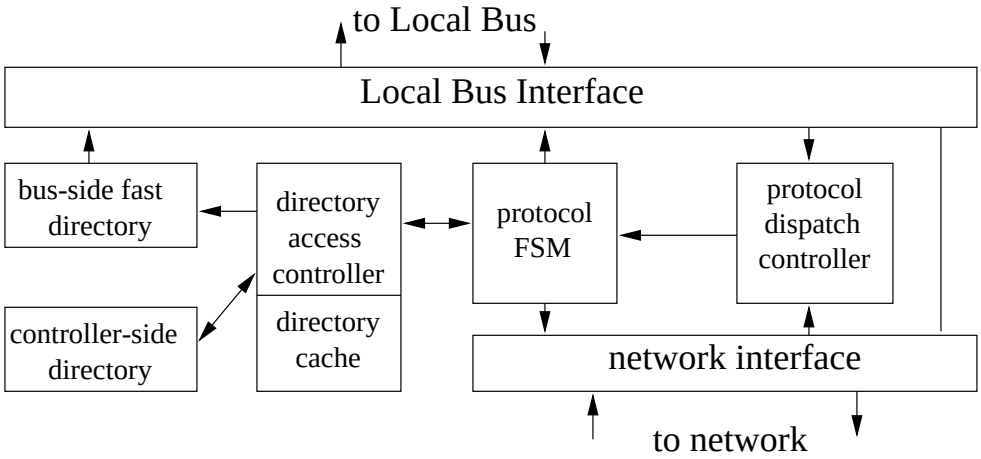
Each processor in a node generally has a write-through first-level cache, and a write-back second-level cache. If there is more than one processor per node, cache coherence between processors must be maintained within a node in addition to between nodes. Also included in this figure is a diagram of a directory-based, hardwired cache controller. The cache controller has interfaces to the local bus, the interconnection network, and a dispatch controller to move cache blocks between the networks. The protocol FSM is the hardwired logic implementing the cache coherence protocol. The directory information is stored and maintained by the directory access controller and the directory cache. Finally, the bus-side and controller-side directories are mini-caches storing frequently used cache blocks.



A Shared Memory Multiprocessor



The Processor Environment



The Cache Controller (hardwired)

Figure 1: The Shared Memory Multiprocessor Architecture

2.2 Cache Coherence in Shared Memory Multiprocessors

As access to main memory is slow compared to the processor speed, hardware caches are necessary for acceptable performance. However, since all processors (and caches) share the same global access space, it is possible that two different caches may cache the same data line (address) at the same time. If one processor updates the data in its cache without informing the other processor in some manner, an inconsistency results, and it becomes possible that the other processor will use a stale data value. The goal of cache coherence is to enforce consistency to insure proper execution of programs in this parallel environment.

There are two major factors affecting cache mechanisms: performance and implementation cost. The need for greater performance is obvious. The programs designed for shared memory multiprocessors have very long execution times so any performance increase would be beneficial. If the time to access main memory is too slow, performance degrades significantly, and potential parallelism is lost. Implementation cost is also an issue because the performance must be obtained at a reasonable cost. Implementation costs occur by adding additional coherence hardware, or by programming consistency enforcing compilers. In addition to these two major factors, there are four primary issues to consider when designing a cache coherence mechanism[13]:

- **coherence detection strategy** - how the system detects possibly incoherent memory accesses
- **coherence enforcement strategy** - how cache entries are changed to guarantee coherence (ie. updating or invalidating)
- **precision of block-sharing information** - how sharing information for cache and memory blocks are stored
- **cache block size** - the size of a line in the cache and how it effects system performance

There are also some complicated issues when developing an efficient coherence mechanism. First, dynamic data is often hard to handle because you can not anticipate sharing patterns in advance. This is especially troublesome for static, compiler-based mechanisms which must

insert code to handle dynamic sharing conflicts. In addition, some special dynamic data, called migratory data, may have to be handled as a special case to increase performance. Migratory data, as the name suggests, moves from one processor to another as it is used, and does not reside at any one processor for long periods of time. This type of data is frequently shared and often includes locks, semaphores, or control-type data. Also for efficiency reasons, data which is read-only or write-only may be handled differently than typical read-write data.

The various cache coherence protocols discussed in this paper have two common characteristics. First, a coherence protocol must update a cache line with a given mechanism (cache enforcement strategy). The two common cache enforcement strategies, write-update and write-invalidate, are explained in the next section. Second, a coherence protocol must enforce a given consistency model such as sequential consistency or weak consistency. Consistency models and their effect on cache coherence protocols are discussed in a following section.

2.3 Cache Enforcement Strategies

There are two basic enforcement strategies for insuring that cache lines are consistent. They are write-update (WU) and write-invalidate (WI). In the write-update protocol, after every write of a data item by a processor, the copy of the data item in other processor caches is updated to the new value. For write-invalidate, a write by a processor on a data item causes a message to be sent to other processor's caching the data item to invalidate the cache line containing the data item. If the processor requires the data item that was invalidated, it will need to be reloaded into the cache.

The choice on when to use WU or WI depends on the types of accesses performed by the program. WI is the better protocol over many applications, but there are applications where WU outperforms WI. WU performs better than WI when the number of writes in a write run is less than or equal to the number of processors that read a block between write runs[20]. A write-run is the number of consecutive writes by the same processor. The performance of write-update can be improved by maintaining a write cache so that multiple processor writes

are combined into only one write (pass one update to other processors for all writes in a write run instead of multiple updates).

As the performance of WU and WI vary depending on the application, it is reasonable that hybrid protocols have been proposed. In a hybrid protocol, one must decide when to change between WU and WI. Typically, a data item starts off in WU-mode. Counters are used to count the number of accesses to the data item. When the number of writes increases to a given threshold, the system changes into WI-mode. The goal of hybrid protocols is to take advantage of WU for cases where it is good, while using WI in cases where it is bad.

Hybrid protocols proposed for bus-based multiprocessors "perform almost as well as, but do not offer any advantages over corresponding variants of the invalidation-based protocols."[20]. However, they provide good performance over a wider-range of applications, and write caches can still be used to reduce the performance liabilities of multiple writes in WU.

Hybrid protocols, also referred to as competitive-update (CU) protocols, are still suboptimal for migratory data. Migratory data is data that is read, modified, and written by many processors in turn. WI by itself is better for migratory data, as updates are wasted for migratory data. To better handle migratory data in CU protocols, migratory data was dynamically detected in [8]. With WI, a read request is sent to the home site of the data, and then an invalidate message is sent when the data is updated. Since the processor knows that it will both read and write migratory data, these two messages can be combined into a read-exclusive request.

To implement migratory data detection, the home node looks for the sequence W_i, R_j, W_j where i and j are different processors. To detect this sequence, the hardware only has to remember the last writer of a block and store a bit indicating if the data block is migratory. When the sequence is detected, the home node polls all caches with a copy of the block to determine if the block is migratory (by asking if they have used it in a given time). If all processors agree that the block is migratory, it is made migratory by the home node and read-exclusive requests can be generated on it. Although false sharing may be a problem, this protocol generates no more traffic than WI and reduces the cache miss-rate and bandwidth

required.

2.4 Consistency models

A consistency model defines how the consistency of the data values in the system is maintained. There are three general consistency models:

- **sequential consistency** - consistency is maintained in a parallel system exactly as would be maintained in a sequential system (ordering)
- **weak consistency** - data is only consistent after synchronization operations
- **release consistency** - data is only consistent after lock release

The choice of consistency model has a great effect on the performance of cache coherence protocols. If the cache coherence protocol implements sequential or strong consistency, every write by a processor must be immediately visible by all other processors caching the data item. Thus, implementing strong consistency is very costly in terms of messaging and is only practical for small, bus-based multiprocessors.

Weaker consistency models such as weak or release consistency only guarantee consistency using synchronization operations. Cache coherence protocols implementing weak consistency are more latency tolerant as updates from one processor do not have to be immediately visible at all other processors. This allows weak consistency protocols to be implemented more efficiently. Since programmers tend to access critical sections or shared data using synchronization operations, weak consistency models fit well with current programming practices and can be implemented more efficiently than sequential consistency models given proper support for synchronization operations.

3 Hardware Protocols

Implementing cache coherence protocols in hardware has been the route taken by most commercial manufacturers. Once a suitable cache coherence protocol has been defined and implemented in digital logic, it can be included at every node to manage cache operations transparently from

the programmer and compiler. Although the hardware costs may be substantial, cache coherence can be provided in hardware without compiler support and with very good performance.

A variety of hardware methods were developed depending on the size of the multiprocessor. Snooping protocols work well for small numbers of processors, but do not scale well when the number of processors increases beyond 32. Directory-based protocols can support hundreds or thousands of processors at very good performance, but may also reach scalability barriers beyond that point. Current commercial systems use directory-based protocols with very good performance. Some hardware protocols are presented in the following sections.

3.1 Snooping protocols

Snooping protocols were designed based on a shared bus connecting the processors which is used to exchange coherence information. The shared bus is also the processors' path to main memory. The idea behind snooping protocols is that every write to the cache is passed through onto the bus (write-through cache) to main memory. When another processor that is caching the same data item detects the write on the bus, it can either update or invalidate its cache entry as appropriate. A processor effectively "snoops" memory references by other processors to maintain coherence.

Since an update is immediately visible to all processors, a snooping protocol generally implements strong consistency. Snooping protocols are simple, but the shared bus becomes a bottleneck for a large number of processors. Although inventive bus schemes have been proposed to increase the bus bandwidth, they often resulted in a greater memory delay. Adding more than one bus or different connection buses is limited by the fact that all processors share the relatively slow memory and bus resources. Thus, snooping protocols are limited to small-scale multiprocessors of typically less than 32 processors.

As the bus is an important commodity in these systems, various approaches were taken to reduce bus traffic. The choice between write-update and write-invalidate protocols is especially important in these systems due to the large number of messages broadcast to maintain strong

consistency. Hybrid protocols between WU and WI were developed in [20] and [3] to reduce consistency traffic. These protocols use write caches to reduce traffic in WU, and allow a cache to dynamically determine whether to invalidate or update a data item based on its access pattern. Read snarfing was also used to take an invalidated cache block from the bus even if the processor did not request it. Bus traffic was reduced by 36-60% in [3] by implementing these optimization techniques.

3.2 Directory-based Protocols

Directory-based coherence protocols eliminate the need for a shared bus by maintaining information on the data stored in caches in directories. By examining a directory, a processor can determine which other processors are sharing the data item that it wishes to access and send update or invalidate messages to these processors as required. Thus, directory-based protocols scale better than bus-based protocols because they do not rely on a shared bus to exchange coherence information. Rather, coherence information can be sent to particular processors using point-to-point connections.

A processor communicates with the directory if its actions may violate or affect consistency. Typically, this occurs when the processor attempts to write a shared data value. The directory maintains information about which processors cache what data blocks. Before a processor is allowed to write a data value, it must get exclusive access to the block from the directory. The directory sends messages to all other processors caching the block to invalidate it, and waits for the acknowledgements of the invalidation requests. Similarly, if the processor tries to read a block that is held exclusively by another processor P, a cache miss occurs and the directory will send a message to P to write back its results to memory and obtain the most current value of the block. Depending on how long the directory waits before granting block access to the requesting processor determines which consistency model is supported. If the system waits for invalidation and write-back acknowledgments before letting a processor proceed, the system implements strong consistency. Better performance can be obtained by delaying the processor

only when accessing a synchronization variable, as in weak consistency.

By using directory information and a cache controller which accesses and maintains the directory, these multiprocessor systems are no longer limited by the performance of a shared bus and/or memory. Now, the processors and their memories can be distributed in space, and connected with point-to-point networks which have much better scalability characteristics. Multiprocessor systems (CC-NUMA machines) which use directory-based protocols can scale to hundreds or even thousands of processors. Many commercial multiprocessor systems implement directory-based coherence including the new SGI Origin[11] which can have 1,024 processors in a maximal configuration. With the appropriate directory and controller hardware and some commands for manipulating cache lines, a high performance machine based on directory protocols is possible as demonstrated by this Silicon Graphics product.

Implementing a directory-based protocol involves several design decisions including:

- **Cache block granularity** - cache line (CC-NUMA) vs. pages (S-COMA)
- **Cache controller design** - the cache controller can be implemented by customized hardware or using a programmable protocol processor
- **Directory structure** - how the directory stores sharing information and how directory information is distributed

These issues will be covered in the following sections.

3.2.1 Cache block granularity

The granularity of sharing in the system is an important design decision in terms of performance and implementation requirements. A small block size reduces false sharing and transmission costs by allowing finer granularity sharing. However, large blocks reduce the directory size and allow for higher cache hit rates, but cause more false sharing. It may also be harder to optimize using software techniques on cache blocks with more than one word. Hence, the choice of cache block size is difficult.

There are two main architectures with different cache sizes and cache block sizes. In CC-NUMA machines, each processor has a dedicated cache in addition to its main memory. The

size of the cache is independent of the size of the memory and is divided into blocks or lines. For example, in the SGI Origin[11] the secondary cache is 4 MB, and each cache line is 128 bytes. In COMA machines, the entire memory of a processor is considered the cache. The granularity of sharing is at the page-level, so some traditional page translation hardware can be used. A descendant of COMA, S-COMA, only allocates a part of a node's local memory to act as a cache for remote pages.

A cache-coherent non-uniform memory access (CC-NUMA) machine tends to be the architecture used most in industry. In a CC-NUMA machine, each processor has its own cache, and each node has a network cache. There may be more than one processor per node. References not satisfied by these caches must be sent to the referenced page's home node to retrieve the data. The global address space is divided up into pages and allocated to the distributed physical memories of the processors. Each page P in the global address space is mapped to a page in physical memory at some node N. Node N is considered the home node for page P, and satisfies requests by other processors for it.

Even though the global address space is divided into pages, the node cache handles data in block sizes which are typically smaller than a page size. Consider the following sequence of events when a processor requests remote data:

- The remote page access causes a page fault at the node.
- The OS maps the page's address to a global physical address and updates the node's page table.
- References to this page by any processor in a node are handled by the cache controller.
- If a block of the page is not in the node cache, a cache miss occurs and the block is retrieved from the home node.

The major difference between S-COMA and CC-NUMA is the size of their respective caches. Although both S-COMA and CC-NUMA use a directory for maintaining cache information, the cache is stored differently in the two architectures. In CC-NUMA, a separate high-speed cache is maintained distinct from the memory of a processor. This cache is generally relatively small and can only hold a small subset of the program's working data. In S-COMA, part of a

node's local memory is allocated as a cache for remote pages. This allows a larger cache size and presumably fewer cache misses. Unfortunately, S-COMA suffers from the overhead of operating system intervention when creating pages in the cache and performing the necessary translation. Since the remote data is mapped at the page-level in S-COMA, standard virtual-address hardware can be used. The only additional hardware needed are access control tags on each block and a translation table with one entry per page to convert between local and global physical addresses.

As S-COMA requires only slightly more hardware than CC-NUMA, some systems have proposed implementing both protocols. In [5], an algorithm called R-NUMA (Reactive-NUMA) is proposed to dynamically react to program behavior to switch between CC-NUMA and S-COMA. In this work, remote pages are classified into two groups:

- **reuse pages** - contain data structures accessed many times within a node and exhibit a lot of remote traffic due to capacity and conflict misses
- **communication pages** - exchange data between nodes and exhibit coherence misses

Their idea was to store reuse pages using S-COMA, and communication pages using CC-NUMA and dynamically detect the two types of pages. The difference between the types of pages is the number of remote capacity and conflict misses a page incurs in the block cache. To detect the difference between the two types of pages, page counters are maintained on each page and an interrupt is generated when they exceed a given threshold. In their algorithm, the first reference to a remote page is mapped using CC-NUMA. The R-NUMA RAD (remote access device) then handles future references to the page either through the cache or by fetching them and updating the per-page counters. When the number of refetches exceeds the threshold as indicated by the counters, the OS relocates the page from CC-NUMA to S-COMA.

The worst-case performance of R-NUMA depends on the cost of relocation relative to the cost of page allocation and replacement. The worst-case performance is approximately 2 times worse than the best possible protocol with an aggressive implementation, although in simulation studies R-NUMA performs better on average over a wide range of applications. In the best case,

CC-NUMA was 17% worse and S-COMA 315% worse than R-NUMA in certain applications. Also, R-NUMA outperforms CC-NUMA when an application has a large number of capacity and conflict misses.

3.2.2 Cache controller design

The cache controller provides cache coherent accesses to memory that is distributed among the nodes of the multiprocessor. A cache controller can be implemented using hardwiring (customized hardware) or by using programmable protocol processors. Custom hardware (CH) is faster but harder to design and more costly than programmable protocol processors (PP). Simulation studies were performed in [14] to determine which implementation method is preferable. In these studies, the following cache controller architectures were simulated:

- custom hardware (based on DASH system)
- protocol processor (based on Typhoon system)
- two hardware controllers - one controller handles local memory access, another handles remote access
- two protocol processors - one for local memory access, another for remote access

They found that the penalty for using a PP over CH was as high as 93% for Ocean and as low as 4% for Lu. The protocol processor was the bottleneck for communication intensive programs. Two protocol engines improves performance for up to 18% for CH and 30% for PP. Also, smaller cache lines worsen performance for both, but especially for protocol processors. Protocol processor performance also suffers more than custom hardware when there is a slower network or more processors per node. In total, custom hardware offers significant performance improvement over protocol processors and utilizing two processors for local and remote access also improves performance.

3.2.3 Directory structure

The directory maintains information on the sharing of cache blocks in the system. Each cache block (associated with a virtual memory page) has a home location. A directory at a given

node may maintain information on the cache blocks stored in its own memory or on all memory blocks. The choice of the structure of the directory is important to reduce the amount of hardware memory needed to store its contents.

There are several basic types of directories:

- **p+1-bit full directory** - each processor has a bit associated with each memory block indicating if they cache the block, plus each block has an exclusive flag (bit) for a total of p+1 bits per block for p processors
- **broadcast directory** - maintains only a valid and exclusive state for each block and forces processors to broadcast invalidation messages to all processors
- **n-pointers plus broadcast schema** - n pointers for the first n processors caching the block. If more than n processors share a block, either invalidates must be broadcast or a processor must be selected to uncache the block.
- **linked-list or chained directory** - maintains a linked-list to each processor caching a block and invalidation requests are passed down the list
- **tagged directories** - pointers are dynamically associated with memory blocks using an address tag field
  - pointer cache TD - one pointer of $log_2 p$ bits with each address tag of $log_2 m$ bits. (m = size of memory) If n processors share a block, n distinct pointer entries in the directory
  - tag cache directory - uses 2 levels of cache. The first level associates n pointers with each address tag, and the second level uses p+1-bit structure of full directory for each address tag.
  - coarse vector TD
  - LimitLESS directory

Fully mapped directories consume too much hardware space to be practical and do not scale well as the number of processors or size of memory increases. A broadcast directory is very inefficient for a large number of processors, and maintaining a limited number of sharing pointers limits potential sharing. Linked-list directories are good, but it takes time to traverse the list. In industry, a variant of the tagged directory is used where each page has a home address

which maintains its sharing information. Requests for the page must go through the home node, and the directory which maintains its sharing information. Thus, sharing information is reduced as each node maintains information only for its local memory and information on data stored in its cache. The directory scales well because its size is not dependent on the size of memory or the number of processors, but only on the data stored in the cache.

3.3 Lock-based protocols

A recent improvement on directory-based protocols was presented in [9]. This work promises to be more scalable than directory-based coherence by implementing scope consistency. Scope consistency is a compromise between lazy release consistency and entry consistency. When processor P acquires lock L from processor Q, it does not pick up all modifications that have been visible to Q as in LRC. Rather, it picks up only those modifications written by Q in the critical section protected by L.

In this scheme, there is no directory and all coherence actions are taken through reading and writing notices to and from the lock which protects the shared memory. The lock release sends all write notices to the home of the lock and all modified memory lines. On lock acquire, the processor knows from the lock's home which lines have been modified and can retrieve modifications. A processor is stalled until all previous acquire events have been performed.

Simulation studies show good performance, and the algorithm has less overhead than CVM. Also, the local address of an object is identical to its global address so no address translation is required. This method is more scalable because directory hardware is not needed, although lock release may be slow as a processor must wait for all writes to be transmitted and receive the acknowledgements. Currently the system is implemented in software similar to distributed shared memory systems, hence the comparison to CVM, but may migrate to hardware when its potential performance improvement can be substantiated in real systems.

3.4 Summary

The "best" coherence protocol depends on the size of the multiprocessor. Snooping protocols work well with shared-bus multiprocessors of 32 or less processors. Directory-based protocols implementing weak consistency and using tagged directories require minimal computer hardware and can be implemented efficiently. However, they may not scale well beyond 1000 processors which makes systems like lock-based coherence look promising. Simulation studies have been performed to evaluate the effects of cache line size, network speed, and application sharing patterns which provide designers with good information when deciding how to construct these systems. In addition, theoretical analysis has been performed to evaluate essential misses in cache traffic[4] and model cache behavior[1]. The performance of COMA and CC-NUMA was also theoretically compared in [22]. This work has lead to great insights on the performance characteristics of these systems and very efficient commercial and prototype implementations. Unfortunately, the cost of the hardware is significant which motivates research in software coherence to improve performance and reduce hardware costs.

4 Compiler-Directed Software Protocols

Although enforcing coherence through hardware is simple and efficient, the extra hardware can be very costly and may not be very scalable. In addition, compilers can gather good data dependence information during compilation which may simplify the coherence task. Optimizing compilers can then be used to reduce coherence overhead, or in combination with operating system or limited hardware support, provide the necessary coherence at a lower cost.

4.1 A Classification for Software-Based Coherence Protocols

A good and fairly recent classification of software-based coherence protocols appeared in [19]. In this work, the authors proposed 10 criteria for classifying software solutions, and mapped many current protocols to their criteria. The criteria is duplicated here, as it represents a good classification of the protocols. In the following section, a few representative software protocols

will be discussed. Unfortunately due to space limitations, the list of software protocols is incomplete, but the protocols presented were chosen to represent a good subset of the protocols and their fundamental ideas.

The ten criteria proposed in [19] are:

- **dynamism** - coherence can be enforced at compile-time (static) or at run-time (dynamic)
- **selectivity** - the level at which coherence actions (eg. invalidations) are performed (ie. whole cache vs. selected cache lines)
- **restrictiveness** - coherence can be maintained preventively (conservative) or only when necessary (restrictive)
- **adaptivity** - the protocol's ability to adapt to data access patterns
- **granularity** - the size and structure of coherence objects
- **blocking** - the basic program block on which coherence is enforced (eg. program, procedure, critical section, etc.)
- **positioning** - the position of instructions to implement the coherence protocol (eg. after critical section, before synchronization, etc.)
- **updating** - an update to a cache line can be delayed before it is sent to main memory (write-through vs. write-back)
- **checking** - the technique used to check conditions of incoherence

4.2 Example Software Protocols

Most software-based protocols rely on the compiler to insert coherence instructions or take advantage of special machine instructions to give the hardware hints on how to enforce coherence. Depending on the target architecture, the amount of hardware support assumed by the compiler can range from none to quite extensive. Even when the hardware enforces coherence, it may be possible to increase performance by giving the hardware hints and structuring the code appropriately. The biggest performance gain usually arises by intelligent code restructuring, which often must be performed by the user.

4.2.1 Enforcing Coherence with Limited Hardware Support

At one end of the software protocol spectrum are the protocols who are solely responsible for enforcing coherence because the architectures they are targeted for do not provide coherence mechanisms in hardware. These architectures typically only have a cache at each processor but no hardware coherence mechanism. They may also have instructions for manipulating the cache. An example architecture of this type is the Cray T3D. The Polaris parallelizing compiler proposed in [2] was implemented for this architecture. As the Cray T3D provides some hardware support, like time tags on each cache line, this work is considered a hardware-supported compiler directed (HSCD) scheme.

The proposed scheme is called Two-Phase Invalidation (TPI). In this scheme, every epoch is assigned a unique number, and every item in the cache has a n-bit time tag which records the epoch in which it was created. A time-read instruction then uses both counters to read potentially stale data. The hardware is responsible for updating the cache time-tag with the epoch number after every update, and the compiler must identify potentially stale data and use time-read instructions to access it. This method gives good performance and a comparable number of misses as a hardware method. However, the additional logic for the epoch numbers and the time-read instruction must be added to the second level cache. Many other software protocols enforce coherence using epoch numbers, version numbers, or timestamps to detect potentially stale data.

Dynamic self-invalidation proposed in [12] is a compiler-based technique where blocks which may be invalidated are detected before they actually need to be invalidated. These blocks are marked as self-invalidation blocks by the compiler, and the blocks' sharing history is maintained using version numbers. The version number is incremented for each exclusive copy of a block. On a cache miss, the version number is sent in the request. The returned block is marked for self-invalidation if the current version number is different than the version number in the request. A block marked for self-invalidation is invalidated during synchronization operations. DSI eliminates invalidation messages by having processors automatically invalidate

local copies of cache blocks before another processor accesses the block. Thus, DSI can implement sequential consistency with performance comparable to weak consistency.

The basis work targeting NCC-NUMA (non-cache coherent) machines was proposed by Petersen and Li. Their protocol was designed for small-scale multiprocessors with non-coherent caches. Using address translation hardware and page mapping, the protocol tracks inconsistent copies in main memory and pages with multiple accessors. This information is stored in a centralized weak list which manages consistency. Their protocol was extended in [10] to distribute the weak list (sharing information) and thus improve scalability. These algorithms give adequate performance for a small number of processors, but since the unit of coherence is a page and sharing is done in software, the performance is less than hardware and not as scalable as directory-based mechanisms.

Cache coherence mechanisms based on virtual memory (VM) use traditional virtual memory hardware to detect memory accesses and maintain coherence through VM-level software. The major factor in these systems is reducing the software handler latency because it is often on the critical memory access path. An example architecture of this type was proposed in [16]. The information in page status tables are used to coordinate sharing, and page status is updated at the page-table level when servicing a read or write request. Cache lines are invalidated at a page-level basis because that is the lowest granularity of sharing in the system. Software handlers maintain information on which sites have the pages cached so that page status changes or flush requests can be sent as needed. This method uses a form of weak consistency. On lock acquire, the pages at other processors are invalidated and their corresponding cache lines are flushed. On lock release, the write-buffer and cache is flushed. This protocol is up to 42% slower than a hardware snoop protocol due to the overhead of software control and the time for memory loads. However, it is a good use of existing hardware and may be applicable in small bus-based multiprocessors.

Another VM-based system designed for a larger number of processors was presented in [7]. In this system, all incoming coherence requests are carried out by software handlers. Using

optimization techniques such as fast state-memory lookup, reducing software latency of dirty misses, and hiding software latency of clean and dirty misses, performance can be only 16-68% worse than hardware methods without the high cost.

4.2.2 Enforcing Coherence by Restricting Parallelism

Coherence can be enforced by the compiler by structuring the parallel language to reduce potential conflicts and data dependencies. Some simple compiler protocols rely on the program being structured in doall parallel loops. In a doall loop, there are no dependencies among loop iterations, so they can be executed in parallel. At the start of the loop, the entire cache is invalidated to guarantee coherent caches at loop boundaries. Unfortunately, this model is very simple, results in many cache misses, and the execution model is restrictive. Thus, variations use bits or timestamps to detect only the variables that have changed during the loop iteration. Along the same lines, Wolski in [21] enforced coherence using a functional language by restricting parallelism to master/slave processors. The implementation architecture was the IBM Power/4 which has private caches, but no hardware coherence and support for post, invalidate, or flush cache actions. However, by restricting the parallelism using master/slave processes, decent performance could be achieved.

4.2.3 Optimizing Compilers Enforcing Coherence

Efforts to optimize the compiler to detect and resolve coherence conflicts have been numerous. In [18], they propose compiler algorithms that use data-flow analysis techniques to detect migratory sharing and load-store sequences to reduce traffic and stall times. The compiler may also divide shared data into regions bound to a synchronization variable as in [17]. Then, the compiler can forward regions to where they are needed, similar to prefetching. Although dynamic structures and algorithms may pose problems, this method may outperform hardware in certain cases because of the coarse granularity sharing and its ability to optimize read/write accesses and reduce cold misses. Forwarding blocks before they are needed gives better performance and higher hit rates but requires large caches.

Compilers can also provide hints to hardware to increase performance even if the hardware maintains coherence. For example, a compiler can identify writes which have coherence conflicts and those that do not. In [15], program writes were divided into three instructions: write-update, write-invalidate, and write-only. Write-update enforces the update coherence strategy, write-invalidate enforces the invalidate coherence strategy, and write-only signals no coherence. Thus, the compiler can decide how to enforce coherence on every write. Trace simulations of this idea resulted in cache miss ratios 0.1-13% less than hybrid protocols, and network traffic 0-92% less than hybrid protocols. An important discovery made is that 78% of the writes do not need coherence enforcement.

4.2.4 Summary of Software Coherence Protocols

In summary, there are various compiler-based techniques for implementing coherence which require various levels of support from the hardware. Many techniques require the hardware to maintain timestamps or counters, and their performance can be improved if the architecture has explicit cache manipulation instructions. By identifying cache coherence conflicts and data dependencies at run-time, the compiler can provide hints to the hardware or enforce coherence by itself. Unfortunately, many algorithms which rely on limited hardware support are inefficient because they rely on software handlers or structures and have restrictive modes of parallel computation (doall loops). However, research into software techniques could result in significant performance improvements comparable to hardware at less cost.

5 Hardware vs. Software protocols

A parallel machine designer must choose between hardware coherence, software coherence, or a combination of the two techniques. Currently, most commercial systems use hardware coherence because it is relatively well-understood and straightforward to implement using a directory-based scheme. As well, a hardware implementation offers superior performance. Although the cost of the hardware may be quite high, this cost is reduced with mass production

and more efficient production techniques.

Software protocols show promise because they do not require the investment in complicated cache controller hardware. However, building an optimizing parallel compiler is still a more expensive procedure due to its complexity and cost of programmer time. Software protocols also suffer from slightly lower performance than hardware protocols.

The most likely approach will be a combination of the two techniques. Hardware support is required for software protocols to identify conflicting accesses using timestamps, or a similar mechanism, and to provide cache access instructions. The hardware is very good at detecting dynamic coherence conflicts which the compiler cannot. In addition, an optimizing compiler can be used to detect static coherence conflicts and optimize accordingly. The compiler could identify non-conflicting writes and blocks where coherence does not need to be enforced. It could also use static information to forward blocks as needed and perform self-invalidation of blocks. This would reduce the coherence traffic and burden placed on the hardware and offer superior performance by using compile-time information.

6 Future Research Directions

Cache coherence implemented in hardware is a mature research area. Although algorithmic improvements are possible to the directory-based approach, there are relatively few major changes that are visible on the horizon. The greatest potential improvement in the hardware area will be manufacturing low-cost coherence controller chips.

On the software-side, there is a lot of research work to be done on optimizing compilers. Currently, compilers have problems performing a complete static analysis on programs to detect potential conflicts. Also, many languages do not have adequate support for parallel computation and coherence maintenance. Research could be done on adding language extensions to allow the programmer to give hints to the compiler on coherence enforcement. Finally, the biggest performance win in parallel systems will be designing algorithms specifically for the parallel environment. Additional data structures or constructs may be needed so that the programmer

and compiler can work together to optimize code for this environment.

7 Conclusion

This paper presented a survey of hardware and software cache coherence protocols for shared memory multiprocessors. Hardware protocols offer the best performance and are implemented in commercial machines. Software protocols show great promise because they can optimize static references, but further work needs to be done to create more advanced compilers. Finally, cache coherence is a difficult problem whose performance greatly affects program performance. Thus, future research should focus on constructing better parallel languages and compilers so that programs can be optimized for the parallel environment and lessen cache coherence overhead.

References

[1] Duane Buck and Mukesh Singhal. An analytic study of caching in computer systems. *Journal of Parallel and Distributed Computing*, 32(2):205–214, February 1996.

[2] Lynn Choi and Pen-Chung Yew. Compiler and hardware support for cache coherence in large-scale multiprocessors: Design considerations and performance study. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 283–294, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society TCCA.

[3] Fredrik Dahlgren. Boosting the performance of hybrid snooping cache protocols. In *Proceedings of the 22th Annual International Symposim on Computer Architecture*, pages 60–69, 1995.

[4] Michel Dubois, Jonas Skeppstedt, and Per Strenstrom. Essential misses and data traffic in coherence protocols. *Journal of Parallel and Distributed Computing*, 29(2):108–125, September 1995.

[5] Babak Falsafi and David A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposim on Computer Architecture*, pages 229–240, 1997.

[6] Jeffrey G. Gee and Alan Jay Smith. Absolute and comparative performance of cache consistency algorithms. Technical Report CSD-93-753, University of California, Berkeley, 1993.

[7] Hakan Grahn and Per Stenstrom. Efficient strategies for software-only directory protocols in shared-memory multiprocessors. In *Proceedings of the 22th Annual International Symposim on Computer Architecture*, pages 38–47, 1995.

[8] Hakan Grahn and Per Stenstrom. Evaluation of a competitive-update cache coherence protocol with migratory data detection. *Journal of Parallel and Distributed Computing*, 39(2):168–180, December 1996.

[9] W. Hu, W. Shi, and Z. Tang. A lock-based cache control protocol for scope consistency. *Journal of Computer Science and Technology*, 13(2), March 1998.

[10] Leonidas I. Kontothanassis and Michael L. Scott. Software cache coherence for large scale multiprocessors. Technical Report TR513, University of Rochester, Computer Science Department, July 1994.

[11] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposim on Computer Architecture*, pages 241–251, 1997.

[12] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22th Annual International Symposim on Computer Architecture*, pages 48–59, 1995.

[13] David J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[14] Maged M. Michael, Ashwini K. Nanda, Beng-Hong Lim, and Michael L. Scott. Coherence controller architectures for SMP-based CC-NUMA multiprocessors. In *Proceedings of the 24th Annual International Symposim on Computer Architecture*, pages 219–228, 1997.

[15] Farnaz Mounes-Toussi and David J. Lilja. The potential of compile-time analysis to adapt the cache coherence enforcement strategy to the data sharing characteristics. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):470–481, May 1995.

[16] K. Petersen and K. Li. An evaluation of multiprocessor cache coherence based on virtual memory support. In *Proc. of the 8th Int'l Parallel Processing Symp. (IPPS'94)*, pages 158–164, April 1994.

[17] Harjinder S. Sandhu. Algorithms for dynamic software cache coherence. *Journal of Parallel and Distributed Computing*, 29(2):142–157, September 1995.

[18] Jonas Skeppstedt and Per Stenström. Using dataflow analysis techniques to reduce ownership overhead in cache coherence protocols. *ACM Transactions on Programming Languages and Systems*, 18(6), November 1996.

[19] Igor Tartalia and Veliko Milutinovic. Classifying software-based cache coherence solutions. *IEEE Software*, 14(3):90–101, May 1997.

[20] Jack E. Veenstra and Robert J. Fowler. The prospects for on-line hybrid coherency protocols on bus-based multiprocessors. Technical Report TR490, University of Rochester, Computer Science Department, March 1994.

[21] Richard Wolski and David Cann. Compiler enforced cache coherence using a functional language. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 173–181, April 1995.

[22] Xiaodong Zhang and Yong Yan. Comparative modeling and evaluation of CC-NUMA and COMA on hierarchical ring architectures. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1316–1331, December 1995.