Cache Coherence in Large-Scale Shared Memory Multiprocessors: Issues and Comparisons

David J. Lilja
Department of Electrical Engineering
University of Minnesota
200 Union Street S.E.
Minneapolis, MN 55455

Phone: (612) 625-5007
FAX: (612) 625-4583
E-mail: *lilja@ee.umn.edu*

ACM Computing Surveys, Vol. 25, No. 3, September 1993, pp. 303-338.

Abstract

Private data caches have not been as effective in reducing the average memory delay in multiprocessors as in uniprocessors due to data spreading among the processors, and due to the cache coherence problem. A wide variety of mechanisms have been proposed for maintaining cache coherence in large-scale shared memory multiprocessors making it difficult to compare their performance and implementation implications. To help the computer architect understand some of the trade-offs involved, this paper surveys current cache coherence mechanisms, and identifies several issues critical to their design. These design issues include: 1) the *coherence detection strategy*, through which possibly incoherent memory accesses are detected either statically at compile-time, or dynamically at run-time; 2) the *coherence enforcement strategy*, such as updating or invalidating, that is used to ensure that stale cache entries are never referenced by a processor; 3) how the *precision of block sharing information* can be changed to trade-off the implementation cost and the performance of the coherence mechanism; and 4) how the *cache block size* affects the performance of the memory system. Trace-driven simulations are used to compare the performance and implementation impacts of these different issues. In addition, *hybrid* strategies are presented that can enhance the performance of the multiprocessor memory system by combining several different coherence mechanisms into a single system.

Categories:
B.3.2 Cache Memories
C.1.2 Multiprocessors — MIMD
C.5.1 Large Computers

Keywords: cache coherence; shared memory; comparison; consistency; memory disambiguation; block size; directory; tagged directory; version control; adaptive.

1. Introduction

The sequence of memory addresses generated by a program typically exhibit the properties of *temporal* and *spatial* locality [Smith1982]. Temporal locality, or locality in time, means that memory addresses recently referenced by a program are likely to be referenced again in the near future. Spatial locality means that the addresses referenced by a program in a short period of time are likely to span a relatively small portion of the entire address space. For example, programs frequently operate on large data structures in which the consecutive elements of the structure are located in sequential memory locations. Thus, the memory addresses generated by a program to access such structures are likely to be clustered into a small range of the address space. Private data caches, which are small, fast memories physically located near a processor, exploit these memory referencing properties to reduce the average time required to access the larger main memory. By temporarily storing in the cache a copy of a value from the main memory that is being actively referenced by a program, caches amortize the time required to copy the memory location from the slower main memory into the faster cache over several references to the same (temporal locality) and nearby (spatial locality) memory locations.

In a shared memory multiprocessor such as that shown in Figure 1, private data caches have been shown to be quite effective in reducing the average delay to access the shared memory [Gottlieb1982, Pfister1985]. Caches have not provided the same level of memory performance improvement in multiprocessors as in uniprocessors, however, since the data referenced by a program

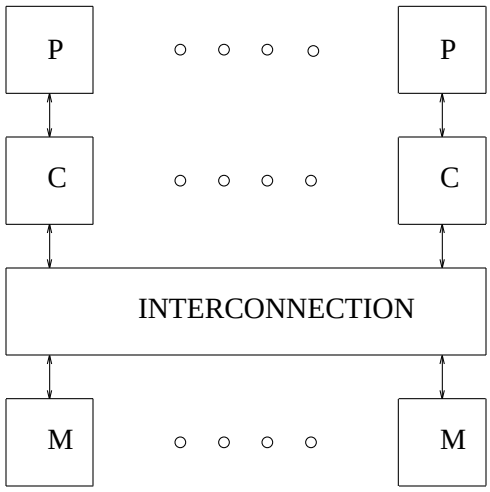


Figure 1: Shared memory multiprocessor architecture.

in a multiprocessor is distributed among the processors. This data spreading reduces the processors' locality of reference, and thus reduces the effectiveness of the caches. In addition, since multiple copies of a shared memory location can be resident in several different caches simultaneously, the private data caches introduce a *coherence* problem in which it is possible for the different cached copies to have different values at the same time. It is the responsibility of the cache coherence mechanism to ensure that whenever a processor reads a memory location, it receives the correct value.

This paper examines mechanisms for maintaining cache coherence in large-scale shared memory multiprocessors, such as the New York University Ultracomputer [Gottlieb1982], the University of Illinois Cedar [Kuck1986], the IBM RP3 [Pfister1985], the Alliant FX-series [Perron1986], and the Stanford DASH [Lenoski1992]. The remainder of this section defines the cache coherence problem, and presents an overview of three different types of mechanisms proposed to solve this problem. In addition, a new framework is presented that identifies the primary factors affecting the implementation cost and the performance of the cache coherence mechanisms. Section 2 describes a trace-driven simulation methodology that is used to illustrate the performance effects of these different factors. Since large-scale parallel machines frequently are used for executing numerical application programs, the simulation comparisons presented in Section 3 use memory traces produced by a multiprocessor emulator executing several different numerical programs. While the use of these applications may bias the simulation results, the issues presented are important to any shared memory multiprocessor system, regardless of the application programs executed by the system.

1.1. Problem Definition

There are two important, related aspects to the cache coherence problem. The first, which is briefly discussed in the next subsection, is the model of the memory system presented to the programmer. The second important aspect, and the primary focus of the remainder of this paper, is the mechanism used by the system to maintain coherence among the caches and the main memory.

1.1.1. Consistency Models

One definition of a system with coherent caches is a system that guarantees that '' ... the value returned on a Load instruction is always the value given by the latest Store instruction with the same address.''[Censier1978] The difficulty with this definition is that the meaning of ''latest'' is not precisely defined when the loads and stores occur on different processors that are running asynchronously with respect to each other. Due to delays and buffering in different portions of the processor-memory interconnection network, and within the processors and memories themselves, each processor and each memory module can observe a different ordering of events. The *consistency model* of a multiprocessor defines the programmer's view of the time-ordering of events that occur on different processors. These events include memory read and write operations, and synchronization operations. As fewer assurances are made by the system to the programmer regarding the order of events, there is a greater potential to overlap operations from different processors with each other, and with other operations within the same processor, and thereby increase the system performance. However, the cost of this greater performance is the added burden on the programmer (or on the compiler) to ensure that any dependences between operations are not violated.

From the programmer's view of the memory system, the *sequential consistency* model defines a strict ordering of the sequence of execution of memory operations allowed within a processor and between processors. Specifically, a multiprocessor system is said to be sequentially consistent if "...

the result of any execution [of the program] is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport1979] With this consistency model, each access to the shared memory must complete before the next shared memory access can begin. In addition, all memory operations are executed in the order defined by the program. This strong ordering of memory accesses imposes a severe performance penalty by greatly limiting the allowable overlap between memory operations issued by an individual processor, and by other processors.

The *weak-ordering* consistency model [Dubois1988] relaxes the guaranteed ordering of events of the sequential consistency model to allow for greater overlap of memory reads and writes. With the weak model, only memory accesses to programmer-defined synchronization variables are guaranteed to occur in a ‘‘sequentially consistent’’ order. All memory references by different processors to shared data variables between accesses to synchronization variables (i.e. between *synchronization points*) can occur in any arbitrary order. Thus, in a system with a weak-ordering model, the programmer can make no assumptions about the ordering of events between synchronization points. To prevent nondeterministic operation, each processor must guarantee that all of its outstanding shared memory accesses are completed before issuing a synchronization operation. Similarly, the synchronization operation must be completed before any subsequent shared memory operations can be issued.

In addition to relaxing the ordering constraints on data references, the *release consistency* model [Gharachorloo1990] weakens the ordering constraints on synchronization variables by splitting the synchronization operation into separate *acquire* and *release* operations. The *acquire* operation is issued by a processor when it wishes to obtain exclusive access to some shared memory object. To prevent interference with another processor that may currently have exclusive access to the shared object, the processor must wait for the *acquire* operation to complete before initiating any references to the shared memory. The *release* operation, on the other hand, is used to give up exclusive access to a shared memory object. To ensure that any changes made by the processor to the shared object are actually performed in the shared memory before exclusive access is surrendered, the processor must wait for all of its shared memory accesses to complete before issuing the release operation. This splitting of the synchronization operation into two separate phases allows this consistency model to achieve a greater overlap of the memory operations issued by all of the processors than either the weak or sequentially consistent models. To quantify the effect of this additional overlap, several studies have examined the performance improvement that can be obtained by using these relaxed consistency models [Torrellas1990, Gharachorloo1991, Gupta1991, Zucker1992].

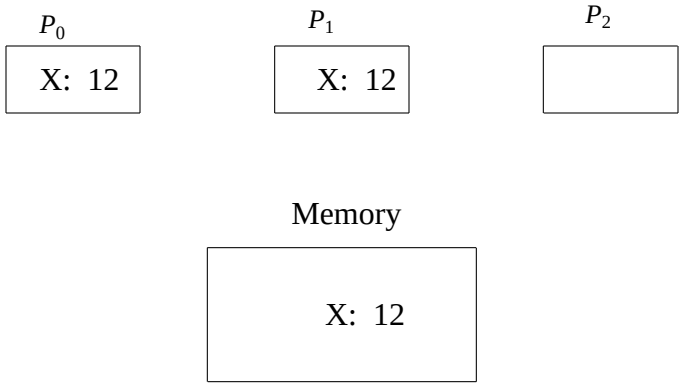
1.1.2. Cache Coherence

A related problem to the memory consistency model, and the primary focus of this paper, is the mechanism used by the system to ensure that processors do not access stale data. In a shared memory multiprocessor, each of the processors can directly access any location in the common memory address space using a single read (load) or write (store) instruction. Since each processor has a private data cache, a copy of the same shared memory location may be present in one or more of the caches at the same time. When a shared memory location is written by any processor, the fact that the value in that location has been changed must be propagated to all of the processors with a cached copy of the location to ensure that none of them use a stale version.

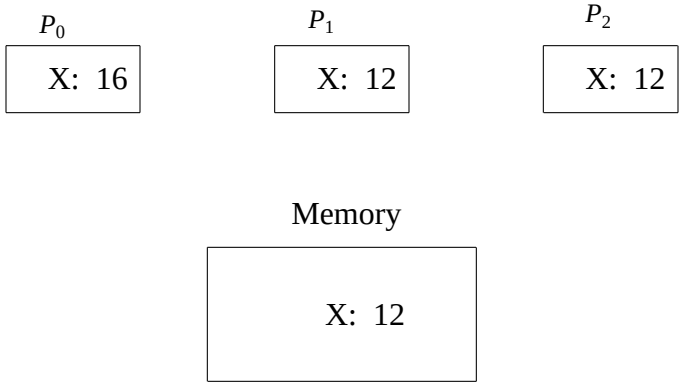
For example, consider a system with three processors, each with a private data cache, in which the sequence of reads and writes shown in Figure 2(a) are performed. After the first two reads have been completed at time t_2, the caches of both processors P_0 and P_1 will contain the value ‘‘12’’ for the variable stored at memory location X, as shown in Figure 2(b). At time t_3, processor P_0 writes to this

Time	P_0	P_1	P_2
t_1	read X		
t_2		read X	
t_3	write 16,X		
t_4	read X	read X	read X

(a) Sequence of reads and writes.



(b) Cache contents after the read at time t_2.



(c) Cache contents after the reads at time t_4.

Figure 2: An example of the cache coherence problem.

memory location changing its value to ‘‘16’’. In a system without a cache coherence mechanism, this value will be updated only in P_0’s cache so that when P_1 rereads X at time t_4, it will read the old value

of ‘‘12’’ from its cache, as shown in Figure 2(c). Similarly, processor P_2 also will read the old value since the main memory has not been updated with the latest value written by P_0. The cache coherence mechanism is necessary to ensure that the stale values of X in the other processors’ caches and in the main memory (i.e. the value ‘‘12’’) will not be propagated to future read operations.

1.1.3. Relationship Between Consistency Models and Coherence

A useful way of viewing the relationship between the memory consistency model and the cache coherence mechanism is that the coherence mechanism ensures that all of the caches see all of the writes to a *specific* block in the same logical order. The consistency model, on the other hand, defines for the programmer the order of writes to *different* blocks as perceived by each of the processors. That is, if the programmer follows the rules of the consistency model for the system being used, the coherence mechanism forces the value returned by any load operation to be the value guaranteed by the consistency model.

In a system that guarantees sequential consistency, for instance, the coherence mechanism ensures that the effects of each write operation to a shared memory location are propagated to all of the caches before the next write to that same location by any processor can proceed. These accesses are said to be *strongly ordered* [Dubois1988]. In a system with a weakly-ordered consistency model, on the other hand, only accesses to predefined synchronization variables are strongly ordered. The ordering of accesses to shared data memory locations by different processors can occur in any order. The processors themselves must then ensure that they do not proceed across a synchronization point until all of the memory accesses they have issued have been acknowledged by the coherence mechanism. Thus, this weak-ordering consistency model ensures that the data values in the caches are coherent only at synchronization points. Examples of the relationship between different consistency models and different coherence mechanisms are presented in the next subsection.

1.2. Overview of Cache Coherence Mechanisms

A variety of mechanisms have been proposed for solving the cache coherence problem. The optimal solution for a given multiprocessor system depends on several factors, such as the size of the system (i.e. the number of processors), the anticipated usage of the system, and the desired system cost. The following subsections present an overview of the operation of the three main types of coherence mechanisms.

1.2.1. Snooping Coherence

Snooping coherence mechanisms rely on a low-latency, shared interconnection between the processors and the memory modules, such as a common bus, that allows each processor to monitor all of the transactions to the shared memory. As a processor ‘‘snoops’’ on the other processors’ memory references, it can detect when a block that it has cached has been changed by another processor. It then invalidates [Goodman1983, Katz1985, Papamarcos1984] its cached copy so that its next reference to the block will force a cache miss, and the current value will be obtained from memory, or from another cache. Alternatively, it can directly update [McCreight1984, Thacker1988] its cached copy with the new value available on the bus. Since the shared bus typically broadcasts the effects of each write operation to a shared memory location to all of the caches in the same cycle as the write itself, these snooping coherence mechanisms typically implement a strongly-ordered consistency model.

While these snooping mechanisms are relatively simple to implement, the shared bus can become a severe performance bottleneck. To reduce the contention on a single bus, the Wisconsin Multicube [Goodman1988] proposed an n-dimensional grid of buses with the processors located at the crosspoints of the buses and the memory modules at the ends. The additional buses provide greater memory bandwidth at the expense of a more complicated coherence protocol. Another approach [Archibald1988, Wilson1987] clusters the processors on smaller, separate buses, and maintains coherence among processors within each cluster. An additional hierarchy of buses is introduced to maintain intercluster coherence. Yet another approach adds a special coherence bus [Bhuyan1989, Marquardt1989] to remove the coherence updating traffic from the normal data read operations on the primary memory bus.

Since all of these schemes use additional buses to increase the bandwidth between the processors and the shared memory, their performance ultimately will be limited by the bus contention when there are too many processors, and by the difficulty of physically constructing these long, high-speed buses. Consequently, it appears that the snooping coherence schemes are limited to use in relatively small-scale multiprocessor systems. Since the focus of this paper is primarily on large-scale multiprocessors, the snooping coherence schemes are not considered further.

Another method of avoiding the bus saturation problem is to replace the bus with an interconnection network, such as a multistage Omega network, a mesh, a fat tree, or a hypercube. These networks provide higher bandwidth between the processors and the memory modules than a shared bus, but they also increase the delay to access memory. This longer delay intensifies the need for the private caches, but, by eliminating the mechanism through which processors monitor the shared memory transactions, the networks compound the coherence problem. Both hardware directory mechanisms and compiler-directed approaches have been suggested for maintaining coherence in these systems.

1.2.2. Directory Coherence

With a directory-based coherence scheme [Tang1976, Yen1985], a processor must communicate with a common directory whenever the processor's action may cause an inconsistency between its cache and the other caches and memory. The directory maintains information about which processors have a copy of which blocks since several processors may have a copy of the same block cached at the same time. Before a processor can write to a block, it must request exclusive access to the block from the directory. Before the directory grants this exclusive access, it sends a message to all processors with a cached copy of the block forcing each processor to invalidate its copy. After receiving acknowledgements from all of these processors, the directory grants exclusive access to the writing processor. When a processor tries to read a block that is exclusive in a different processor, it will send a miss service request to the directory. The directory then will send a message to the processor with the exclusive copy telling it to write the new value back to memory. After receiving this new value, the directory sends a copy of the block to the requesting processor. Directory schemes differ in how much information they maintain about shared blocks, where that information is stored, and whether invalidating or updating is used to ensure coherence, resulting in differences in memory requirements and performance. These trade-offs are discussed further in Section 3.

By waiting for the invalidation and write-back acknowledgements for all writes to a shared memory location before letting a processor proceed with a write, the directory implements a strongly-ordered consistency model. A weakly-ordered consistency model can be implemented with a directory by having the directory delay the writing processor only when it is accessing a synchronization variable. This approach then puts the burden on the processor to ensure that before it

proceeds across a synchronization point it has received acknowledgements from the directory for all of the writes is has issued to shared data memory locations. Since this weakened consistency model delays processor writes only to synchronization variables, it will produce higher performance than the strongly-ordered model.

1.2.3. Compiler-Directed Coherence

Compiler-directed coherence mechanisms determine at compile-time which cache blocks *may* become stale. Special instructions then are inserted into the generated code to be executed by each of the processors to prevent them from using this possibly stale data. One of the simplest of these compiler-directed coherence mechanisms [Veidenbaum1986] uses *indiscriminate invalidation* of the data caches to enforce coherence with a weakly-ordered consistency model. This coherence mechanism assumes a *doall* parallel loop model of execution [Polychronopoulos1988] in which there are no dependences between the iterations of the loop. Thus, all of the iterations can be executed simultaneously on multiple independent processors. The parallel loop terminates when all of the iterations have completed executing. Processors may be reassigned to iterations at the entry and exit points of the parallel loop. These points are called the *loop boundaries.*

At the start of each parallel loop, each processor first executes a *cache-invalidate* instruction to begin the execution of the loop with an empty cache. Each processor also executes a *cache-on* instruction to allow all references to shared-writable variables to be cached during the execution of the loop. The caches are write-through so that all writes to shared memory locations are propagated directly to the global shared memory. At the end of the parallel loop, each processor again invalidates its entire cache to prevent stale entries from propagating into the next section of the program. Incoherent accesses are thereby prevented since the weakly-ordered consistency model used with this coherence mechanism guarantees coherent caches only at loop boundaries. Since the caches are invalidated at the loop boundaries, and since the current value is only in the main memory, coherence is assured. Similar schemes have been suggested for the RP3 machine [Brantley1985] and the Ultracomputer [Edler1985]. These simple approaches tend to invalidate more cache entries than are necessary to maintain coherence, and thus may reduce the memory system performance when compared to a directory mechanism. More sophisticated compiler-directed mechanisms with better performance than this simple mechanism are described in Section 3.1.2.

1.3. Factors Affecting Coherence Mechanisms

The most important consideration in choosing a cache coherence mechanism for a multiprocessor usually is its performance, or how effective it allows the caches to be in reducing the average delay when fetching data from memory. Another important consideration is the implementation cost, typically measured by how much memory is required to store the cache block sharing information, and by the complexity of the control logic. The different coherence schemes have significantly different trade-offs in cost and performance, making it difficult to evaluate the alternatives. The primary issues affecting the cache coherence mechanisms can be summarized as:

1) the *coherence detection strategy*, that is, the strategy by which the coherence mechanism detects a possibly incoherent memory access, which can be done either dynamically at run-time, or statically at compile-time;
2) the *coherence enforcement strategy,* such as updating or invalidating, that is used to ensure that stale cache entries are never referenced by a processor;
3) how the *precision of block sharing information* can be changed to trade-off the implementation cost and the performance of the coherence mechanism; and

4) how the *cache block size* affects the performance of the memory system.

This paper surveys the state-of-the-art in cache coherence mechanisms in the context of the above issues. Trace-driven simulations are used to compare how these different issues affect the performance and implementation costs of the different coherence mechanisms. In addition, hybrid strategies are discussed that combine several different coherence mechanisms into a single system to improve the memory referencing performance. These comparisons should help the computer architect understand some of the trade-offs involved in the various coherence alternatives.

2. Cost and Performance Modeling

The most accurate method of determining the performance of a specific computer design, or for proving the validity of a new architectural approach, is to build it. Unfortunately, actually building a complete computer system is very time-consuming and expensive. It also requires the designer to select specific values for architectural parameters, such as the data cache size and the cache block size, without knowing what reasonable values of the parameters may be for the new system. Therefore, before actually committing an idea to hardware, it is desirable to explore the limits of the design space using mathematical analysis or simulation. A large number of potential design options can be quickly examined by analytically modeling the system and varying the desired parameters. Analytic models are of limited usefulness when comparing cache coherence mechanisms, however, due to the assumptions that must be made concerning memory referencing patterns and data sharing. To provide more realistic results while still maintaining flexibility in choosing system parameters, the performance evaluations presented in this paper use trace-driven simulations.

2.1. Trace-Driven Simulation

An address trace for a multiprocessor is a record of the sequence of memory addresses generated by the processors as they execute a program. There are several different methods of generating these traces [Stunkel1991]. For example, it is possible to instrument an actual computer system to record the memory references as they are generated by the program. This approach has the advantages of being very accurate, very fast, and able to monitor operating system execution as well as a user program. Its main disadvantages are the cost and difficulty of building the hardware monitor, and the complexity of instrumenting all of the processors in a multiprocessor system. It also limits the simulation to using traces from a specific implementation of a computer system, which may be substantially different from the system to be studied.

Another method of generating traces that has many of the advantages of hardware monitoring is to alter the microcode of a processor to generate traces as it executes the instructions. This approach also can trace operating system activity, and it is relatively fast, but it requires a substantial effort to rewrite the microcode. In addition, it is not applicable to processors without microcode, or to those that have their microcode in read-only memory. As more parallel computer systems use hard-wired processors, this technique will become less useful.

Software-based techniques have been suggested to avoid some of the difficulties of the hardware monitoring and microcode-based approaches. In some processors, it is possible to generate an interrupt after the execution of every instruction. The interrupt routine then produces the trace information for the current instruction. Another approach is to modify a program's source code or executable object code to produce a trace as the program executes. Both of these methods can generate traces without significantly slowing down the traced system, but they can introduce significant timing distortions into the trace due to the interrupts and due to the additional trace

generation instructions. They also are limited to the instruction set of the specific processor used to execute the program.

The most flexible method of generating traces, and the one used in this paper, is to simulate the execution of the entire multiprocessor system. The primary disadvantage of this approach is that it is quite slow since the simulator must model all of the operations of all of the hardware elements of the processors. This explicit modeling of all operations, however, produces accurate traces and it allows the simulation of any architectural feature, especially those that may not exist in a real machine.

Starting with application programs written in Fortran (described in Section 2.3), the Alliant compiler [Perron1986] is used to automatically find the parallel loops and to generate parallel assembly code. This assembly code then is executed by a multiprocessor emulator to produce a trace of the memory addresses generated by each of the p processors. This multiprocessor system uses an execution model in which each parallel loop is followed by a sequential section of the program so that execution alternates between p processors executing a parallel section of the program, and a single processor executing a sequential section. The memory traces from the p processors are completely interleaved into a single trace such that during the execution of a parallel section of the program, an address generated by processor i is followed by an address generated by processor i+1, and so on, modulo p. During the execution of sequential sections of the program, processor 0 generates all of the memory references. In an actual system, timing differences between the processors due to cache misses, network and memory contention, and synchronization delays may produce a different ordering of the references, but this interleaving, which represents a valid ordering, highlights the effects of data sharing in the cache coherence mechanisms used in these simulations. Thus, this ordering provides a rigorous test of the different coherence mechanisms.

Since the simulation is very time-consuming, it is limited to executing relatively short programs compared to those that could be executed on actual hardware. In addition, these simulations are for one program running at a time, thus ignoring the effects of multiple programs sharing the system and the effects of the operating system. The simulations also prohibit task migration. In spite of these limitations, this trace-driven methodology provides an adequate means of comparing the performance of the different coherence mechanisms.

2.2. Machine Model

The interleaved memory trace drives a multicache simulator to determine the miss ratio and the cache-memory network traffic. A fully associative data cache with a random replacement policy is used in each processor to eliminate the confounding effects of set associativity conflicts. The long execution time required to perform the simulations limited the size of the application programs' data sets. To maintain a realistic relationship between the size of the data set and the size of the data cache, a data cache of 8 kbytes is used in each of the thirty-two processors, unless otherwise noted.

Since this system assumes that all instructions are only read, they can never cause coherence problems. Consequently, all instruction references are ignored. This multiprocessor architecture uses a separate synchronization bus for distributing the next available iteration values when scheduling loop iterations, and for performing the barrier synchronization at the end of the parallel loops. With this architecture, synchronization variables are never cached with the data and, since this study is concerned primarily with the the effect of the cache coherence mechanism on data references, accesses to synchronization variables are not considered in these simulations. It should be noted, however, that synchronization variables stored in memory can be heavily shared. Heavy sharing of a single memory location by many different processors can cause memory *hot spots* [Pfister1985] which may make it undesirable to cache synchronization variables [Dubois1988]. Special hardware

or software can be added to the system to improve access to synchronization variables [Anderson1990, Goodman1989, Kruskal1986], but the analysis of these techniques is beyond the scope of this study.

The p=32 processors are connected to the shared memory via a packet-switched multistage interconnection network. Network traffic from a processor to the memory, such as a miss service request or write-back data, uses the forward network, while traffic from the memory to a processor, such as an invalidation command or fetched data, uses the separate reverse network. Both the forward and reverse networks use 32-bit data paths. Each packet between the memory modules and the processors requires a minimum of of two words (eight bytes). The first word contains the source and destination module numbers plus a code for the operation type, and the second word contains the actual memory address. Additional words are needed for the actual data values fetched and written. Table 1 details the actions required for each type of memory reference, along with the generated network traffic, when using the p+1-bit full directory [Censier1978]. This directory structure is described more fully in Section 3.3.1.

Table 1: Memory operations and resulting network traffic.
b= number of words per block; word size = 4 bytes.

Memory Operation	Forward traffic (bytes)	Reverse traffic (bytes)
Read hit.		
(none)	-	-
Read miss, block *shared* in one or more caches, or only in memory.		
miss service	8	8+4*b*
Read miss, block *exclusive* in another cache.		
miss service	8	8+4*b*
write-back	8+4*b*	8
Write hit, block *shared* in one or more caches.		
processor requests exclusive access from directory	8	-
directory sends invalidation messages	-	8 * #cached
processors acknowledge invalidations	8 * #cached	-
directory acknowledges writing processor	-	8
Write hit, block *exclusive* in this cache.		
(none)	-	-
Write miss, block only in memory.		
miss service	8	8+4*b*
Write miss, block *shared* in one or more caches.		
processor requests exclusive access from directory	8	-
directory sends invalidation messages	-	8 * #cached
processors acknowledge invalidations	8 * #cached	-
directory acknowledges writing processor	-	8
Write miss, block *exclusive* in another cache.		
miss service	8	8+4*b*
write-back	8+4*b*	8

2.3. Test Programs

Six numerical application programs written in Fortran were used to generate the parallel memory traces for these simulations. *Arc3d* and *flo52* both analyze fluid flows. The *trfd* program uses a series of matrix multiplications to simulate a quantum mechanical two-electron integral transformation. *Simple24* is a hydrodynamics and heat flow problem using a 24-by-24 element grid. The *pic* program uses a particle-in-cell technique to model the movement of charged particles in an electrodynamics application. The *lin125* program is the Linpack benchmark using a 125-by-125 element matrix. The problem sizes and outer loop counts were reduced in these programs so that the entire program could be simulated in a reasonable period of time.

The memory referencing characteristics of the test programs are summarized in Table 2 for p=32 processors, and a cache block size of b=1 word. The blocks were classified by examining the traces and determining how many processors accessed each block. The *private* blocks are those that are referenced by the same processor throughout the program's execution with no references by any other processor. The *shared-writable* blocks are referenced by two or more different processors, at least one of which writes the block. Finally, the *shared read-only* blocks are blocks that are referenced by more than one processor, but are never written. The percentages do not sum to 100 since the table does not show the statistics for the shared read-only blocks.

As shown in this table, fewer than 40 percent of the unique blocks referenced by *arc3d, flo52,* and *pic* are shared-writable, and fewer than half of their total references are made to these blocks. Most of their references are to private and read-only blocks, and thus do not cause any coherence actions. In contrast, more than 78 percent of the blocks referenced by *simple24, lin125,* and *trfd* are shared-writable, although only *trfd* and *lin125* have more than half of their references to these blocks. These different sharing characteristics provide for a broad range of memory performance in the simulations to highlight the strengths and weaknesses of the different coherence mechanisms.

2.4. Performance Metrics

The two most important measures of performance for a memory system are the latency and the bandwidth. The *average memory latency* is the time from when a processor issues a memory read operation until the data requested is available to the processor, averaged over all memory references. If the requested data is resident in the cache, the latency is simply the cache access time, which

Table 2: Memory referencing characteristics of the test programs.
blocks = number of unique single-word data blocks referenced by the program
refs = number of memory references made to the blocks

Prog	Total		Private		Shared-writable	
	blocks	refs	%blks	%refs	%blks	%refs
arc3d	53733	6603772	55.6	48.9	38.1	48.9
pic	100087	8765261	77.0	57.0	22.9	34.8
simple24	10759	4251420	10.7	56.5	88.8	43.1
trfd	1478	5877557	11.2	14.9	88.8	70.5
flo52	115331	10000000	82.3	77.1	17.7	22.5
lin125	21041	10000000	21.7	1.2	78.3	94.4

typically is one cycle. If the data is not in the cache, however, a miss service request is generated and sent to the memory system. The average memory delay can be approximated as $T_{ave}=(1-m)T_{cache}+mT_{miss}$ where m is the cache miss ratio ($0<m\leq 1$), T_{cache} is the time required to access the cache on a hit, and T_{miss} is the time required by the memory system to service the miss. The value of this miss service time is a function of the intrinsic delay in the memory modules, the time required to propagate the request through the network, and the additional time required to perform any necessary coherence operations. The network delay is a function of the total traffic in the network. High network traffic increases the probability that there will be collisions in the network, which then can increase the miss service time. Because of the dependence of the miss service time on specific parameters of the system, such as the memory access delay, this paper uses the miss ratio as a first-order indication of the expected memory performance.

The *bandwidth* of the interconnection network determines how many data bytes per unit time can be transferred between the memories and the processors. To prevent the network from becoming a performance bottleneck, it is important to provide sufficient bandwidth, but it can be expensive to provide the wide data paths and the fast components needed for a high-bandwidth network. Maintaining coherence in this type of system can require many messages for each memory request, which can put a significant load on the network. As a result, the cache coherence protocol should try to minimize the network traffic by maintaining a low miss rate, and by reducing the number of messages required to maintain coherence. The simulations presented in this paper use the average number of bytes transferred per memory request as an indication of the network bandwidth requirements for the different coherence mechanisms.

The total execution time of a program takes into account the tradeoffs between the miss ratio and the network traffic, and it is the performance measure that is most interesting to the user of a multiprocessor system. To understand the impact of architectural decisions on the performance of the different coherence mechanisms, however, it is useful to separate the overall performance into the individual factors that contribute to this performance. As a result, the miss ratio and the average network traffic are the metrics used in this study to compare the different design factors that comprise a cache coherence mechanism. It is felt that examining these two metrics directly provides greater insight into the tradeoffs in the coherence mechanisms than simply comparing total execution times.

3. Performance Impacts

This section uses the trace-driven simulation model described in the previous section to examine the impact on performance and on implementation cost of the primary design factors affecting cache coherence mechanisms. Descriptions of the different coherence mechanisms also are provided.

3.1. Coherence Detection Strategy

There are several interrelated factors that determine the performance of a cache coherence mechanism. One of the most important factors is *when* the mechanism performs *coherence detection* — either dynamically at run-time, or statically at compile-time. The dynamic coherence detection strategies solve the coherence problem by examining the actual memory addresses generated by a program at run-time and dynamically keeping track of which processors have a copy of which blocks. In contrast, the static coherence schemes try to predict which memory addresses *may* become stale by analyzing the program's referencing behavior when it is compiled.

It is important to distinguish the *implementation* of a coherence mechanism from its method of determining when a shared memory location is stale. While the statically detected coherence mechanisms necessarily are software-based since they rely on a compiler, they also need some hardware support to maintain the current state information about the memory locations. Thus, it is not precisely correct to refer to these mechanisms as ‘‘software-only’’ coherence mechanisms. Similarly, mechanisms that dynamically detect the need for coherence actions use hardware to monitor the actual memory addresses, but they also can be augmented with compilers and other software to produce hybrid schemes, as described in Section 4. This paper distinguishes the two major classes of coherence mechanisms as *dynamically detected* and *statically detected* instead of as hardware and software mechanisms.

3.1.1. Memory Disambiguation

The ability to *disambiguate memory* references, that is, the ability to determine if two different memory accesses actually refer to the same physical location in memory, is critical to providing a high-performance cache coherence scheme. The primary advantage of the dynamic detection schemes is that by examining the actual memory addresses being referenced, they are able to perfectly disambiguate these accesses. Statically detected coherence schemes, however, must rely on the imprecise disambiguation performed by the compiler. For example, consider the following sequence of references to the array A():

S_1.	P_1 read:	...	= A(f(.))
		...	
S_2.	P_2 write:	A(g(.)) =	...
		...	
S_3.	P_1 read:	...	= A(h(.))

In this sequence of memory references, the read in statement S_1 loads an element of array A() into processor P_1’s cache. The particular element read is determined by the value of function f(.), which may be anything that produces a valid index into the array. Typically it is some function of the loop count. If functions f(.) and g(.) map into the same memory location, the write in statement S_2 causes the corresponding element in P_1’s cache to become stale since it no longer contains a copy of the current value. If function h(.) in statement S_3 also maps to the same memory location, P_1 will attempt to read this stale value, unless it is first invalidated or updated. Determining whether or not some action is required in this case is the crux of the cache coherence problem [Cheong1988].

For static coherence detection, a data dependence test [Banerjee1988, Lichnewsky1988, Li1990] can be used to determine if the three functions never refer to the same element, in which case no coherence action is necessary, or to determine if the same element is always referenced by the three statements so that some coherence action must be taken. Unfortunately, the data dependence tests often are too imprecise to determine whether the elements are always the same or always different. In this case, static coherence mechanisms must err on the conservative side by assuming that they are the same element, and then inserting the appropriate coherence actions into the generated code.

A related memory disambiguation problem for static coherence detection mechanisms occurs with procedure calls, functions, and subroutines. The name of a variable inside of a procedure most likely will be different than the name of the variable passed to the procedure. To provide precise dependence analysis, the compiler must perform interprocedural analysis to track variable names across procedure boundaries and thereby determine if a particular memory reference may cause a

coherence problem. In many programming languages, this interprocedural analysis can be very difficult to perform, in which case the coherence mechanism may have to take an extremely pessimistic approach and invalidate the entire data cache at the entry and exit points of each procedure [Cheong1988a, Cheong1989]. Procedure calls provide no problem for the dynamic coherence detection schemes since they examine the actual memory addresses at run-time and have no indication that a procedure call has even occurred.

3.1.2. Static (Compile-Time) Coherence Detection Mechanisms

The indiscriminate invalidation schemes discussed in Section 1.2.3 [Veidenbaum1986, Brantley1985, Edler1985] are more conservative than is necessary to ensure coherence in that they invalidate cache entries that are not actually stale. This over-invalidation then produces unnecessarily high miss ratios. More complex schemes determine at compile-time which particular cache blocks may become stale, and when they may be stale, and then invalidate these specific entries before they are accessed. Subject to the memory disambiguation limitations of the compiler, these schemes are able to preserve at least some temporal locality between parallel loops.

For example, the *fast, selective invalidation* scheme [Cheong1988a] associates a *change* bit with each cache block. This bit is set true by the *cache-invalidate* instruction inserted by the compiler at each parallel loop boundary to indicate that the block may have been changed during the current loop. The *memory-read* instruction forces a cache miss when it references a block with its *change* bit set to true. This miss ensures that the current copy of the block is fetched from the main memory. The *change* bit then is reset to false when the data block is loaded into the cache. Subsequent *memory-read* references to the same block will see this reset *change* bit and will generate a cache hit since the cached copy is now assured of being up-to-date.

Another memory referencing instruction, called the *cache-read* instruction, ignores the *change* bit when it accesses a memory location. It is used to reference a shared-writable location that is guaranteed by the compiler to be up-to-date in the cache in the current parallel loop, and therefore can be treated as a cache hit. In addition to the *change* bit, this coherence mechanism requires a *valid* bit for each cache block, but no *dirty* bit is required since it uses a write-through strategy. Because each processor is responsible for maintaining coherence in its own cache, no state information is required in the main memory.

Improvements to this fast, selective invalidation scheme use version numbers [Cheong1989] or time stamps [Min1989] to determine whether or not a cache entry is up-to-date when it is referenced. In the version control mechanism [Cheong1989], for instance, each processor maintains a *current version number (CVN)* in a separate local memory within the processor for each variable used in a program. For each parallel loop, the compiler predetermines which variables may have been written by any processor during the loop. It then generates instructions that are executed by each processor at the end of a parallel loop to increment the *CVN* values for these variables. This change in the *CVN* value indicates to subsequent memory references that a new version of this variable may have been created.

In addition to maintaining one *CVN* entry per program variable, each cache entry has an associated *birth version number (BVN).* The *BVN* value is set equal to the corresponding *CVN* value when the referenced variable is first loaded into the cache from the shared memory. When a variable is written, its *BVN* is set to the new version number, $CVN+1$. Because of this defined relationship between the *BVN* and *CVN* values, a read reference to a memory location will be a cache hit if-and-only-if $BVN \geq CVN$. If $BVN < CVN$, however, the cached copy may be stale and the current value must be loaded from the main memory.

3.1.3. Dynamic (Run-Time) Coherence Detection Mechanisms

With dynamic coherence detection, the memory addresses actually generated by the program are examined at run-time to provide perfect memory disambiguation. An example of a coherence mechanism with dynamic detection is the p+1-bit full directory [Censier1978]. (Other directory configurations are discussed in Section 3.3.) With this directory, two bits per cache block encode one of three states for each of the blocks in the caches. The *invalid* state means that the block is empty and will cause a cache miss when it is referenced. When a block is shared by several processors, it must be in the *shared read-only* state in each processor to prevent any processor from modifying the block without first requesting *exclusive* access from the directory. A processor is free to update a block in the *exclusive* state since it is assured of having the only copy of the block. The directory in each memory module maintains p *valid* bits and a single *exclusive* bit for each block in the module. If the *exclusive* bit is reset, up to p *valid* bits may be set to indicate which processors have a copy of the block in the *shared read-only* state. If the *exclusive* bit is set for a block, a single *valid* bit will be set to point to the processor that has the only copy of the block, which must be in the *exclusive* state.

3.1.4. Performance Comparisons

The trace-driven simulation methodology described in Section 2 is used to quantify the effect of the coherence detection strategy on the memory system performance. Specifically, the performance of the p+1-bit full directory [Censier1978] is compared to the compiler-directed version control coherence mechanism [Cheong1989]. The range of performance of the version control scheme is estimated using three different levels of compiler technology, as summarized in Table 3. The *simple* compiler has imprecise memory disambiguation in that it maintains one version number (i.e. one *CVN* entry) for an entire array. With this compiler, a write to any element of an array creates a new version of the entire array. Furthermore, this compiler cannot track variable names across subroutine boundaries so that the entire data cache is invalidated at the entry and exit points of each subroutine.

The other extreme of compiler performance for the version control mechanism assumes an *ideal* compiler with perfect memory disambiguation and perfect interprocedural analysis. This compiler maintains a unique *CVN* entry for each element of every array, and it never invalidates the caches at subroutine boundaries. It models the best possible performance of the version control scheme, but it is probably impossible to implement this perfect memory disambiguation in an actual compiler. The *realistic* compiler compromises between these two extremes with imprecise memory disambiguation, but perfect interprocedural analysis. It should be pointed out that at the end of every parallel loop, the *CVN* values of every variable that may have had a new version created in that loop must be incremented. The ideal compiler may perform significantly more *CVN* updates at the end of each parallel loop than the other two compilers since it has to update a *CVN* value for every array element that was written, instead of a single *CVN* update per array. The time required to perform this updating adds directly to the average memory delay, which may be significant for large arrays.

Table 3: Compilers used for the version control simulations.

Compiler	Action at subroutine boundaries	Number of *CVN* entries
simple	clear caches	one per array
realistic	ignore subroutine boundaries	one per array
ideal	ignore subroutine boundaries	one per array element

Because of the imprecise nature of compile-time data dependence tests, and because of the information hiding in procedures, coherence mechanisms that rely exclusively on the compiler to disambiguate memory references tend to invalidate more cache entries than are actually necessary to maintain coherence. By tracking the actual memory addresses, dynamic directory mechanisms can invalidate only those blocks that are actually stale, which, as shown in Table 4, can cause the directory mechanism to have a lower overall miss ratio than the ideal compiler-directed version control mechanism for the *arc3d, simple24, trfd,* and *flo52* programs. The directory has slightly higher miss ratios than the ideal implementation of version control for *pic* and *lin125* primarily due to the high number of write misses produced with the directory.

These extra write misses occur because in these two programs, a large, shared array is repeatedly written by different processors during different portions of the programs' execution. When the array is written for the first time, it is marked as exclusive in the writing processor's cache. When another processor tries to overwrite this same location, it misses and must request exclusive access from the directory. These misses can be prevented by allocating a new array so as not to overwrite the same array multiple times, but this approach will require additional memory space. With the version control mechanism, the compiler detects that these write references will not cause coherence violations, and thereby reduces the number of write misses. The performance of the *simple* compiler tends to be poor compared to the other compilers and compared to the directory since it invalidates all of the caches at every subroutine boundary. The *realistic* compiler has slightly better performance than the simple compiler because it can look beyond subroutine boundaries, but its miss ratio generally still is higher than that of the ideal compiler due to its imprecise memory disambiguation.

A major advantage of the static coherence detection mechanisms is that by making each processor responsible for maintaining coherence in its own cache using self-invalidation, interprocessor communication is limited to that required to service the cache misses. The dynamic mechanism, on the other hand, sends many messages between the directory and the processors which increases the congestion in the interconnection network compared to the static mechanism and thereby may increase the memory latency. As shown in Table 5, the total network traffic for the ideal compiler in the version control mechanism is approximately the same as the network traffic required to service only the misses with the directory. These similar traffic requirements are expected since these two approaches have similar miss ratios. However, this table also shows that the network traffic required by the directory for the invalidation messages approximately doubles the total network traffic over that required for servicing only the misses. The simple and realistic compilers in the version control approach produce higher network traffic than the ideal compiler since they have significantly higher miss ratios. Even with these higher miss ratios, though, the network traffic they

Table 4: Miss ratio (percent) for static and dynamic detection strategies.

Program	Directory			Version Control				
				simple	realistic	ideal		
	read	write	overall	overall	overall	read	write	overall
arc3d	15.0	8.0	23.0	41.6	30.2	19.3	8.3	27.6
pic	8.8	8.9	17.7	32.9	25.9	8.0	8.1	16.1
simple24	9.4	3.1	12.5	63.5	56.0	12.6	3.2	15.8
trfd	10.9	1.5	12.4	42.0	18.0	13.9	4.1	18.0
flo52	1.1	0.8	1.9	44.2	44.1	2.7	1.1	3.8
lin125	5.7	4.2	9.9	9.5	9.4	5.9	0.2	6.1

Table 5: Network traffic for static and dynamic detection strategies.
(bytes per reference)

Program	Directory			Version Control		
	miss	invalidate	total	simple	realistic	ideal
arc3d	4.59	5.40	9.98	8.32	6.04	5.52
pic	3.53	4.47	8.00	6.58	5.18	3.23
simple24	2.43	2.44	4.87	12.7	11.2	3.17
trfd	2.48	2.27	4.75	8.40	3.61	3.61
flo52	0.39	0.40	0.79	8.83	8.81	0.76
lin125	1.99	1.99	3.98	1.90	1.87	1.22

produce can be less than the total network traffic produced by the directory since they generate no invalidation messages.

In summary, the primary advantage of dynamic coherence detection is its perfect memory disambiguation. By knowing precisely those addresses being referenced, the dynamic mechanism invalidates only those cache blocks that are actually stale. This exact invalidation generally produces lower miss ratios than a mechanism that statically detects coherence violations. In addition, the dynamic detection mechanism is completely transparent to procedure boundaries, while the static coherence detection mechanism requires good interprocedural analysis to match the performance of the dynamic mechanism. The primary advantage of static detection mechanisms is that they produce lower network traffic than the dynamic mechanisms since they do not require any invalidation messages. Similar results to the simulations presented here have been reported in other studies comparing compiler-directed and directory-based coherence mechanisms [Adve1991, Lilja1991, Min1990].

3.2. Coherence Enforcement Strategy

Another factor affecting the performance of a multiprocessor memory system is the actual method used by the coherence scheme to ensure that no processor accesses a stale memory location. The simplest approach is to make all shared-writable memory locations *non-cacheable* so that there can never be multiple copies [Lilja1989]. However, since references to shared-writable variables can constitute a large fraction of the references made by a program (see Table 2), bypassing the cache for all references to these memory locations can significantly reduce performance. Two other coherence enforcement strategies always allow shared-writable memory locations to be cached, but either *update* or *invalidate* stale cache entries before they are referenced again. With an update approach, the new value of the shared location is distributed to all processors with a copy of the block whenever it is written by any processor. The advantage of this approach is that it prevents an additional miss if the cache block is reused by a processor with a cached copy after it has been written by another processor. A significant disadvantage is the additional network traffic produced by the potentially large number of update messages.

Instead of updating cached copies when they are changed, the invalidation strategy marks all cached copies as *invalid* within the cache to force the processor to miss the next time it references that block. This approach reduces the network traffic compared to the update strategy, but it does introduce the extra delay of another miss if the block is reused. Invalidation schemes can be classified as either *self-invalidation* or *directed-invalidation.* With self-invalidation, the compiler inserts extra instructions into the generated code to force the processor to invalidate some or all of its

data cache before it accesses a stale entry. With directed-invalidation, some outside agent, such as a directory, forces a processor to invalidate a specific block in its cache at a specified time.

3.2.1. Performance Comparisons

In a system in which all of the processors are connected with a shared bus, an update protocol is implemented by having each write to a shared memory location write-through to the bus to broadcast the new value to all of the processors [McCreight1984, Thacker1988]. In the system used in this study, however, the bus is replaced by a multistage interconnection network. Since this type of network does not support broadcasting, coherence updates are implemented using individual messages. For example, when a processor writes to a shared memory location, a message containing the new value is sent to the directory. The directory then sends a message containing the new value to each processor with a cached copy of the block instructing the processors to update their copies. The processors respond with an acknowledgement to the directory, which then acknowledges the processor that performed the initial write. With this approach, updates of written blocks are sent only to processors that actually have a cached copy instead of being broadcast to all of the processors. The invalidation-based directory coherence simulator described in Section 2.2 is modified to use this update-based protocol.

Table 6 compares the miss ratios produced by a directory coherence scheme using either updating or invalidating for three different cache sizes. In the infinite cache, blocks are never replaced due to lack of cache space. Consequently, with an update strategy in an infinite cache, once a block is moved into the cache, it is never removed. The number of misses in this configuration then is simply the number of misses required to bring each block into the cache the first time. That is, the number of misses is the same as the number of unique blocks referenced, and it is the minimum number of misses that can be produced for the given programs. Comparing the invalidation strategy in the infinite cache with the update strategy shows how the invalidations required to maintain coherence increase the miss ratio due to the sharing of cache blocks by different processors. In particular, it demonstrates the performance effect of requiring exclusive access to a block in order to write to the block. Since updating allows writes to blocks that are shared, updating typically produces a lower miss ratio than invalidating. As the cache size is reduced, the miss ratio increases for both updating and invalidating since there is no longer enough space in the caches to store all of the referenced blocks.

The network traffic statistics in Table 7 show that the cost of the lower miss ratio with updating is the considerably higher network traffic it produces compared to the traffic produced by

Table 6: Miss ratio (percent) for updating and invalidating coherence enforcement strategies.

Program	Cache size (bytes)					
	4K		16K		∞	
	inv	up	inv	up	inv	up
arc3d	26.8	16.0	19.9	6.73	18.0	1.7
pic	28.6	26.9	8.4	6.8	7.8	1.4
simple24	13.5	6.4	11.2	3.6	9.3	0.73
trfd	12.4	0.39	12.4	0.38	12.4	0.38
flo52	2.1	1.7	1.9	1.4	1.8	1.4
lin125	10.0	1.7	10.0	1.6	10.0	1.6

Table 7: Network traffic (bytes per reference) due to cache misses and due to coherence enforcement for updating and invalidating.

Program	Strategy	Cache size (bytes)					
		4K		16K		∞	
		miss	coh	miss	coh	miss	coh
arc3d	invalidate	5.36	5.56	3.97	5.17	3.61	5.13
	update	3.21	12.6	1.34	15.0	0.34	16.2
pic	invalidate	5.72	3.35	1.68	2.26	1.56	2.78
	update	5.39	7.69	1.36	8.87	0.29	10.1
simple24	invalidate	2.69	2.39	2.23	2.49	1.86	2.68
	update	1.27	7.37	0.71	7.54	0.15	7.96
trfd	invalidate	2.49	2.27	2.49	2.27	2.49	2.27
	update	0.08	52.3	0.08	52.3	0.08	52.3
flo52	invalidate	0.43	0.45	0.37	0.42	0.36	0.31
	update	0.35	5.23	0.29	5.40	0.27	6.39
lin125	invalidate	1.99	1.88	1.99	1.88	1.99	1.89
	update	0.34	27.1	0.32	27.3	0.32	27.4

invalidating. This table separates the network traffic into that required to move the data into a cache on a miss, and that required to maintain coherence, which is either the update traffic, or the invalidate traffic. The miss traffic for updating is always less than that generated by invalidating since its miss ratio is lower than the miss ratio with invalidating. However, the component of the network traffic due to coherence actions is roughly two to twenty-five times greater for updating than invalidating since updating produces some network traffic on every write to a shared memory location. The invalidation strategy, on the other hand, produces coherence traffic only when a processor first requests exclusive access to a block, or when a write-back is required. Subsequent writes to the same block by the same processor generate no additional traffic.

How the differences in network traffic and miss ratios translate to overall memory delay depend on the implementation details of each individual system. For instance, given a high-bandwidth network, an updating strategy probably will produce lower average memory delays than invalidating since updating has the lowest miss ratio. The high traffic produced by updating may be easily handled by the network without increasing the memory delay. However, if the interconnection network is the system bottleneck, as it is likely to be in many systems, then invalidating may produce the best overall performance in spite of its relatively higher miss ratio since it produces the lowest network traffic.

3.2.2. Adaptive Coherence Enforcement

In addition to the effect these implementation details have on performance, the sharing characteristics of a program also can affect the relative performance of updating and invalidating. In some programs, an invalidation strategy may produce the best performance, while in other programs, an updating strategy may be best [Eggers1989, Karline1986]. For instance, if a shared block tends to be written by only a single processor, but read by many processors, distributing the new values of the block produced by each write using an updating strategy will reduce the miss ratio when compared to an invalidating strategy. If a block is written many times by a single processor between reads by other processors, however, an invalidating strategy will tend to reduce the unnecessary network traffic that would be produced by an updating strategy. Furthermore, the sharing characteristics of a

single block may change over the course of a program's execution making updating the best choice for some references to the block, while other references to the same block may produce better performance using invalidating. To adjust the coherence enforcement strategy to the potentially changing sharing patterns of each block, several *adaptive* coherence schemes have been proposed.

The competitive snoopy cache [Karline1986] initially updates all shared copies of a block by broadcasting writes to these blocks over the shared bus. If a processor has not referenced its copy of the block after a specified number of writes, it invalidates its cached copy so that it no longer requires the block updates. Similarly, the EDWP coherence scheme [Archibald1988] dynamically switches from an updating strategy to an invalidating strategy by keeping track of the number of writes made to each block. After three writes are made to a block by the same processor with no intervening reads by other processors, the block is assumed to be no longer actively shared, and all of the cached copies are invalidated. These two approaches thus attempt to dynamically adjust the coherence enforcement strategy based on the program's run-time behavior.

The Munin system [Bennett1990] implements a coherence mechanism that uses the compiler to categorize each object referenced by the program into a coherence type, and then adjusts the coherence enforcement strategy to each particular type. For example, a data object that is determined to be mostly read is copied to each processor's cache as it is referenced, but an object that is alternately read and written may have a single copy moved among the processors instead of being copied. Another adaptive coherence strategy [Mounes-Toussi1993] examines the program at compile-time to estimate the cost of using updating or invalidating for each write reference to a shared memory block. Each reference then is tagged with the lowest-cost coherence enforcement strategy to be used at run-time. Simulation studies of this approach indicate that by switching enforcement strategies for each shared block it can obtain the low miss ratios of an updating coherence strategy while generating the low network traffic of an invalidating strategy, thereby achieving the best of both enforcement strategies. In addition, since the compiler can look-ahead in a program to predict future memory sharing patterns, this compiler-assisted adaptive scheme tends to produce lower miss ratios and lower network traffic than the adaptive schemes that switch enforcement strategies using only run-time information.

3.3. Precision of Block Sharing Information

Coherence schemes that dynamically determine which memory references need coherence actions have access to the memory addresses only as the program generates them. Since it is impossible for the hardware to predict how the blocks will be shared, the coherence mechanism must track the state and sharing characteristics of every memory block referenced by the program. The number of memory bits needed to store this information can be enormous. *Exact* mechanisms, such as the p+1-bit full directory [Censier1978], maintain enough state information about the sharing of blocks to know exactly which processors have a copy of which blocks. When a block needs to be invalidated, these exact mechanisms send invalidation messages only to those processors that actually have a cached copy of the block. *Imprecise* mechanisms, such as the n-pointer plus broadcast directory [Agarwal1988], reduce the amount of stored information, but occasionally must resort to broadcasting invalidation messages to all processors, even those without a cached copy of the affected block. These broadcasts can significantly increase the memory traffic in the interconnection network. Some recently proposed *tagged* directories further reduce the directory memory requirements by maintaining sharing information only for blocks that are actually cached. The following subsections describe the various directories and present models [Lilja1991] for comparing the number of memory bits needed by each directory to maintain the cache block sharing information.

3.3.1. Traditional Directories

In the traditional directories, memory bits are associated with each block in the memory modules to maintain the current state of the block, and to store information about which processors have a cached copy of each block. The p+1-bit full directory [Censier1978], for example, encodes three states for each cached block using two state bits per cache block. In the *invalid* state, the block is empty or not up-to-date. In the *shared, read-only* state, the block is shared and can only be read by all processors. A processor with a block in the *exclusive* state is assured of having the only copy. Thus, it can both read and write the block. The directory maintains an additional p *valid* bits and a single *exclusive* bit for each block in the memory, where p is the number of processors. The total number of bits dedicated to storing coherence information in this scheme is $p[m(p+1)+2c]$, where m is the total number of blocks in the memory modules, and c is the total number of blocks in the caches.

The broadcast directory [Archibald1984] maintains only the *valid* and *exclusive* state bits for each block in the memories and the caches, for a total of $2p(m+c)$ bits. Because it maintains only this limited information, this directory must broadcast all of its invalidation messages to all of the processors. These broadcasts can be very time-consuming in a system with a complex interconnection network, such as a multistage network, since these networks typically do not support broadcasting. In addition, these broadcasts increase the average memory delay compared to the full directory due to the increased network congestion. The primary advantage of this directory structure is its low memory requirements for storing the block sharing information.

The n-pointers plus broadcast scheme [Agarwal1988] reduces the need for broadcasting by maintaining n pointers with each memory block to point to the first n processors that request a copy of the block. When a block needs to be invalidated, invalidation messages can be sent only to the processors with a cached copy of the block. If more than n processors attempt to simultaneously share the same block, the directory sets a *broadcast* bit to indicate that invalidations must be broadcast to all of the processors. This approach thereby trades-off memory requirements with the need to broadcast. Each of the n pointers in each of the entries in this directory requires $\log_2 p$ bits to point to any processor, plus a bit for each pointer to indicate if it contains a valid processor number. In addition, each entry requires the single *broadcast* bit plus an *exclusive* bit. Finally, each block in the data caches requires two state bits, making a total of $p[2c+m(2+n+n\log_2 p)]$ bits dedicated to maintaining coherence for this directory structure.

The *linked-list* directory [James1990] reduces the size of the directory compared to the full directory structure without requiring broadcasts by maintaining a linked list from the directory to each of the processors having a cached copy of a block. A doubly-linked list typically is used so that normal cache block replacements may be performed within a processor without communicating with other processors. When a block is invalidated due to a coherence operation, the invalidation command is propagated from one end of the list to every processor that has a copy of the block. This single-ended propagation eliminates the potential race condition that exists if invalidations were propagated simultaneously from both ends. The total number of coherence bits required in this linked list directory is $p[3c+2m+2(c+m)\log_2 p]$ since each pointer in each memory block and in each cache block require $\log_2 p$ bits to point to a processor, plus an extra bit to point back to memory. In addition, two state bits are used in each cache block, and an *exclusive* bit is needed for each memory block.

3.3.2. Tagged Directories

In the traditional directories described in the previous section, memory bits for pointing to a processor with a cached copy of a block are statically associated with each block in the main memory. Thus, the total number of coherence bits is proportional to the size of the memory. The *tagged directories* take advantage of the observation that only blocks that are actually cached in one or more processors need to be allocated pointers. In these directories, pointers are dynamically associated with memory blocks using an address tag field only as the blocks are moved from the memory to a cache. With this approach, the number of coherence pointers is proportional to the size of the data caches, which are significantly smaller than the main memory. A variety of different configurations can be used to maintain the coherence pointers themselves, such as those used in the full p+1-bit directory, the n-pointers directory, or the linked-list directory discussed in the previous section. In addition, several other possible tagged directory structures are described below.

The *pointer cache* tagged directory [Lilja1991] maintains one pointer of $\log_2 p$ bits with each address tag of $\log_2 m$ bits. This structure allows multiple entries in the directory to have the same address tag. When n processors share the same block, n distinct pointer entries will be allocated in the directory with the same address tag, but pointing to different processors. The maximum number of processors that can share a block with this scheme is limited by the associativity, a, of the pointer cache itself. When more than a processors try to share a block, or when the entire pointer cache overflows, a free pointer is created by randomly choosing an active pointer and invalidating the selected block in the indicated processor.

The total number of bits needed to store sharing information with this pointer cache is $[r(\log_2 m+\log_2 p+2)+2c]p$, where r is the number of entries in each pointer cache. Typical values of r required for good performance are discussed in the next section. This bit count includes the $\log_2 m$ address tag bits, the $\log_2 p$ processor pointer bits, the pointer valid bit, and the exclusive state bit needed for each pointer, plus the two state bits needed for each block in the data cache. No additional coherence bits are needed in the shared memory since the tagged directories store this sharing information only when a block is actually cached.

The *tag cache* directory [O'Krafka1990] is a variation of the pointer cache idea that uses two levels of caches in the directory. The first level of the tag cache associates n pointers with each address tag. When a block is shared by more than n processors, the corresponding entry in the tag cache is overflowed to the second-level tag cache. This second-level cache uses the p+1-bit structure of the full directory for each address tag. Overflows of this second-level tag cache are handled by invalidating a randomly selected entry to be reused by another block. The number of bits dedicated to maintaining coherence with this directory structure is $p[r_1(log_2 m+n\log_2 p+2)+r_2(log_2 m+p+2)+2c]$ where r_1 and r_2 are the number of entries in the two levels of the tag cache.

The *coarse vector* tagged directory [Gupta1990] incorporates a mode bit into each pointer entry to force the directory controller to interpret the pointer in one of two different ways. If the mode bit is reset, then the v pointer bits are interpreted as n direct pointers to processors. Since $\log_2 p$ bits are required to uniquely identify a processor, each entry can point to $n=\lfloor v/\log_2 p \rfloor$ unique processors. If more than n processors attempt to simultaneously share the same block, the mode bit is set to indicate that the pointer bits should be interpreted as pointing to one of p/g clusters, where there are g processors per cluster. That is, when the mode bit is set, the i^{th} bit of the v pointer bits will be turned on to indicate that at least one of the processors in the i^{th} cluster has a copy of the shared block. Invalidation messages then will be sent to all of the processors in each cluster that has its corresponding bit set in the tag cache entry. The total number of bits needed for coherence with this directory structure is $r_{cv}p[log_2 m+max(p/g,\log_2 p)+3]+2cp$, where r_{cv} is the number of entries in the tag

cache. The *max* function is needed to ensure that at least one complete processor number of $\log_2 p$ bits can be stored in the v bits.

The *LimitLESS* directory, which was proposed as part of the Alewife project [Chaiken1991], uses hardware and software to implement a combination of the n-pointers per address tag structure of the previously discussed tag cache, plus the p+1-bit full directory. Specifically, when more than n processors attempt to share a block, an interrupt service routine is invoked to emulate the complete sharing information of the full directory. Since it is assumed that more than n processors will attempt to share the same block infrequently, the performance of this combined hardware-software approach should be comparable to that of the other tagged directories.

3.3.3. Cost and Performance Comparisons

There are two primary components of the hardware implementation cost of a cache coherence mechanism: 1) the control logic required to implement the mechanism, and 2) the number of memory bits needed to store the cache block sharing information. It is difficult to quantify the control logic cost of the different coherence mechanisms without detailed circuit designs since the complexity of this logic can vary considerably. With detailed designs, the implementation cost can be measured as the VLSI chip area needed to implement the control logic, for instance, but this comparison is beyond the scope of this study. Instead, the amount of memory used to store coherence information is used for an approximate comparison of the implementation cost since it can be a significant portion of the total cost of implementing the mechanism.

To compare the memory requirements of the different coherence mechanisms, the memory overhead is defined to be the ratio of the total number of bits dedicated to coherence functions divided by the total number of data bits in both the main memories and the data caches [Lilja1991]. The total number of data bits in the system is $D=pbw(m+c)$, where p is the number of processors, b is the number of words in each block, w is the number of bits per word, m is the number of blocks in each of the p memory modules, and c is the number of blocks in each of the p caches. If N_x is the number of bits dedicated to coherence functions for a particular coherence scheme, the corresponding overhead is $O_x=N_x/D$.

Table 8 shows the memory overhead for several different directories that maintain different amounts of block sharing information. In this table, the memory overhead is normalized to the number of blocks in the data cache, c. The ratio of the number of pointer cache entries in each memory module, r, to the number of blocks in each data cache is $s=r/c$, and $k=m/c$ is the ratio of the number of blocks in memory, m, to the number of blocks in the data caches.

A 4-way set associative pointer cache is used to provide a fair comparison of a realistic pointer cache implementation. An invalidation on overflow policy is used to create free pointers when the pointer cache overflows. A random replacement policy is used in both the pointer caches and in the fully associative data caches. The word size is w=32 bits with p=32 processors, and the data cache block size is b=1 word. Typical cache memory sizes are in the range of 64K (2^{16}) words to 256K (2^{18}) words, and a typical memory module may contain from 2M (2^{21}) words to 16M (2^{24}) words. Thus, typical values of $k=m/c$, which is the ratio of the number of blocks in each memory module to the number of blocks in each data cache, are in the range of 8 to 256. The following simulations use k=256. The data cache again is c=8 kbytes in each of the p=32 processors.

The network traffic generated by the different directories is shown in Figure 3(a-f) plotted against their respective memory overheads. The number of pointer entries available in the pointer cache tagged directory, r, relative to the number of blocks in the data cache, c, is varied from

Table 8: Normalized memory overhead for the directory mechanisms.

Scheme	Overhead, O_x
1. $(p+1)$-bit full directory	$\frac{k(p+1)+2}{bw(k+1)}$
2. 2-bit broadcast directory	$\frac{2}{bw}$
3. n-pointer + broadcast directory	$\frac{k(2+n+n\log_2 p)+2}{bw(k+1)}$
4. Linked list directory	$\frac{2(k+1)\log_2 p+2k+3}{bw(k+1)}$
5. Pointer cache directory	$\frac{s[\log_2(kc)+\log_2 p+3]+2}{bw(k+1)}$

$s=r/c=1/32$ to $s=2/1$, doubling with each data point. When s is small, there are not enough pointers available to point to all of the processors that try to share cache blocks. As a result, pointers frequently must be reused by randomly choosing an active pointer, and invalidating the block in the processor to which it points. These frequent pointer invalidations produce a large number of invalidation messages, which then generate a large amount of network traffic. In all of the programs tested, a pointer is usually available when one is needed when the number of pointers available in the pointer cache is the same as the number of blocks in the data caches (i.e. $s=1$). This one-to-one ratio usually is adequate because the memory references tend to be uniformly distributed among all of the memory modules. Thus, requests for pointers also tend to be uniformly distributed.

Even with this pointer cache size of $s=1$, the memory overhead of the pointer cache directory is significantly smaller than the overhead of the other directories. The 2-bit broadcast directory has the next lowest memory overhead since it stores only 2 bits for each block in the memory. However, it does not maintain precise information about which processors have cached copies of blocks, forcing it to broadcast all of its invalidation messages. These broadcasts produce extremely high network traffic compared to the other directories. The overhead of the n-pointer directory increases in direct proportion to n, the number of pointers it has available per block. It produces very high network traffic when $n=1$ since it must resort to broadcasting whenever more than one processor attempts to share the same block. Since fewer than four processors typically attempt to share the same block at the same time in most of these traces (and in many other programs [Agarwal1988, Eggers1988, Weber1989]), $n=4$ pointers often is sufficient to reduce the network traffic of this mechanism to be approximately the same as that of the full directory. The network traffic of the linked list directory is the same as that for the full directory since both send invalidations only to those processors that actually have a cached copy of the block. Its memory overhead is less than that of the full directory,

however, since it maintains fewer total pointers.

The miss ratio of the pointer cache tagged directory follows a curve similar to its network traffic. With a small pointer cache, many active blocks are invalidated to obtain free pointers. When these active blocks are again referenced, they force the processor to miss. When the size of the pointer cache increases to s=1, the data cache miss ratio improves to be the same as the full directory. The other directories all produce identical data cache miss ratios since they all allow up to p processors to simultaneously cache the same block. The precision of the block sharing information each maintains (i.e. the memory overhead) affects only the number of invalidation messages they need to generate, and thus affects only the total network traffic and not the miss ratio.

While the network traffic and the miss ratio produced by the linked list scheme is the same as that produced by the full directory, the average memory latency of the linked list scheme is expected to be higher than that of the directory. This longer delay occurs because the entire linked list for a shared block must be traversed when the block is invalidated. This list traversal time adds directly to the delay for the write that triggered the invalidation when a strongly-ordered consistency model is used. With a weakly-ordered model, however, much of this delay may be hidden. With a full directory scheme, on the other hand, the generation and sending of all of the invalidation messages can be pipelined to further reduce the memory delay.

These simulations demonstrate that the memory overhead of the directory mechanisms is directly related to the precision of the block sharing information they maintain, and inversely related to the corresponding memory traffic. That is, more information must be stored in order to reduce the network traffic. However, a tagged cache directory can provide the low network traffic of a full directory while using very little memory since it maintains the sharing information only for blocks that are actually cached. The additional cost of the tagged directory compared to a traditional directory is the relatively more complex control logic it requires.

3.4. Cache Block Size

The cache *block size*, also called the *line size*, is the number of consecutive memory words updated or invalidated as a single unit. The *fetch size*, on the other hand, is the number of words moved from the main memory to the cache on a miss. While these two parameters do not have to be the same, the following discussion assumes that a single block is fetched per miss. Increasing the number of words in a cache block can reduce the miss ratio because of the high probability that memory locations physically near recently referenced locations will be referenced in the near future (i.e. spatial locality). When the block size becomes too large, the miss ratio increases since the probability of using the additional fetched data becomes smaller than the probability of reusing the data replaced. The block size that minimizes the average memory delay generally is smaller than the block size that minimizes the miss ratio because the additional time required to transfer the larger blocks can overwhelm the latency to receive the first word [Przybylski1988, Smith1987]

In addition to allowing a cache to exploit spatial locality, another advantage of blocks larger than a single word is that they reduce the memory overhead of the directory coherence mechanisms. Since pointer information is maintained only for blocks and not for individual words, Table 8 shows that the cache coherence memory overhead is inversely related to the block size. For example, doubling the block size will cut the overhead in half. This relationship is not true for the compiler-directed coherence mechanisms, such as version control, however, since they still need a dirty bit per word, independent of the block size.

Unfortunately, cache blocks larger than a single word can introduce *false sharing* in which two nonshared words end up occupying the same block. For instance, when a loop scans through an

array, the *stride* is the array subscript increment from one iteration to the next. If the stride is one, consecutive elements of the array will be accessed by consecutive iterations of the loop. If the iterations are distributed sequentially across the processors, consecutive array elements will be referenced by different processors. When the cache block size is greater than one array element, and the array elements are arranged linearly in memory, many processors will need a copy of the same block, causing a large amount of sharing. This type of sharing is referred to as false sharing since the processors are not actually sharing data, but are sharing memory blocks due to the placement of the array elements in memory. As long as the processors only read the array this sharing is not harmful, but when a processor attempts to write to an element when using an invalidation protocol, all the copies of the written block will be invalidated, even though not all of the elements are changed. In the worst case, every write to the shared block will cause an invalidation, and every read will be a cache miss, so that blocks will *ping-pong* between caches. As a result, the processor miss ratios and the memory network traffic increase compared to a system with a block size of one word, thereby increasing the average memory delay.

To eliminate the false sharing problem, many dynamic coherence schemes use small blocks, in which case they lose the potential benefits of exploiting spatial locality [Agarwal1988a, Eggers1989a, Goodman1983, Lee1987]. The statically detected coherence schemes also tend to favor small block sizes. With block sizes larger than one word, the compiler must know the block size and it must control the placement of the data in the memory. If the compiler ignores the block size, false sharing can introduce dependences between otherwise independent program statements. These hidden dependences then can cause incorrect program execution since coherence will not be correctly maintained. The solution to this problem is to use one word blocks, or to restrict data placement so that each block contains only one unique variable name. For arrays, this restriction has little effect beyond some fragmentation in the last block allocated to the array, but if large blocks are used, a substantial amount of memory space may be wasted on scalar variables since only one variable can be assigned to a block.

3.4.1. Performance Effects

Figure 4 demonstrates how the cache block size affects the network traffic and the miss ratio for the p+1-bit full directory. The other directory schemes are not shown since they have similar behavior, and the version control scheme is not simulated with block sizes larger than one word due to compiler limitations. Each word is four bytes, and the block size is varied from 1 to 16 words (4 to 64 bytes). The fetch size is set to one block, so that one complete block is fetched on a miss. The parallel loop iterations are scheduled with iteration 1 executing on processor 0, iteration 2 on processor 1, and so on. (The effects of different scheduling strategies have been discussed elsewhere [Lilja1992].)

The lowest miss ratios for *arc3d* and *simple24* occur with a block size of four words, indicating that there is some spatial locality that can be exploited in these programs when using this scheduling strategy. As the block size is increased, however, the larger blocks begin to evict blocks that are still in use, which then increases the miss ratio. For the other programs tested, the lowest miss ratios are produced with single-word blocks due to significant amounts of false sharing with blocks larger than a single word.

Figure 4 also shows that the total network traffic increases as the block size increases. Figure 5 separates this network traffic into the component required to move the blocks into the caches on a miss, and the component required to send the invalidation messages from the directory to the individual processors. For the *arc3d* and *simple24* programs, the network traffic due to misses is relatively flat as the block size increases from 4 to 16 bytes. Since in these two programs the miss

ratio decreases as the block size increases to 16 bytes, there are fewer blocks fetched, but each block is larger. The result is that the miss traffic remains approximately constant until the miss ratio begins to increase when the block size is greater than 16 bytes. The invalidation traffic produced by these two programs decreases slightly as the block size increases from 4 to 16 bytes indicating that there is little false sharing until the block size is larger than 16 bytes. This reduction in invalidation traffic shows that the caches are exploiting the available spatial locality, which also is reflected in the reduced miss ratios.

In the other programs, the miss traffic increases significantly as the block size is increased due to the combination of higher miss ratios, and the fetching of larger blocks. The increases in the invalidation traffic with the larger blocks for these programs shows that the increases in the miss ratios are due, at least in part, to the false sharing effect. That is, as the block size is increased there is more false sharing, which then requires more invalidations to maintain cache coherence. It is interesting to note that the invalidation traffic generally contributes about half as much to the total traffic as does the miss traffic. Thus, the larger blocks tend to cause more network traffic than the traffic produced by the additional invalidation messages from the false sharing effect.

4. Hybrid Techniques

The use of the different cache coherence mechanisms are not mutually exclusive in that several of the different mechanisms can be combined into a single system. This section presents several such hybrid mechanisms.

4.1. Compiler Assistance for Reducing the Directory Size

By allocating pointers to blocks only as they are referenced, the tagged directories can significantly reduce the memory requirements of a directory-based cache coherence scheme. They still waste some directory resources, however, by allocating pointers to blocks that cannot cause coherence problems, such as blocks that are never written or are never shared. To reduce the number of pointers allocated, it is possible to use the compiler to mark all private and read-only blocks as not needing coherence enforcement. Several studies [Agarwal1988a, Eggers1988, Lilja1989, Weber1989] have shown that a substantial fraction of all blocks referenced by a program may be private or read-only, and thus could be marked as not needing coherence enforcement.

When used with a tagged directory, this compiler marking can significantly reduce the number of pointers needed in a given program, and can thereby substantially reduce the required directory size [Lilja1991a]. More complex compile-time analysis techniques can mark each individual memory reference as needing a pointer allocated or not [Nguyen1993]. This more precise marking can reduce the time a pointer is needed for a specific memory shared location, thereby allowing pointers to be reused more frequently than with no marking. This frequent reuse further reduces the size of the directory needed to maintain a given level of memory performance.

4.2. Combining Multiple Coherence Mechanisms

The DASH distributed shared memory multiprocessor prototype developed at Stanford University [Lenoski1992, Lenoski1990] incorporates two different dynamic coherence mechanisms, a snooping bus and a directory, and two different coherence enforcement mechanisms, invalidating and updating, into a single system. The processors in this system are divided into groups, or *clusters*, with four high-performance MIPS R3000 processors in each cluster. Cache coherence within each

cluster is maintained using a bus-based snooping protocol [Papamarcos1984]. Coherence among clusters, in contrast, is maintained using a directory-based invalidation protocol where the directory appears to be another processor on the snooping bus in each cluster.

An interesting feature of this directory is that it in addition to the standard invalidation protocol, it also supports two different update mechanisms. The first is an *update-write* operation in which the new data produced by the write is directly distributed to all processors with a cached copy of the block being written. The sharing information stored in the directory is used to determine which processors need to be updated. The second update mechanism is called the *deliver* operation. With this operation, the processor writing to a block writes into the cache using the invalidate protocol. When it has completed its sequence of writes, it issues a *deliver* instruction specifying which clusters should receive a copy of the block. The directory then sends a copy to each of the specified clusters and the directory is updated appropriately. This write mechanism is useful when the desired destination clusters are unlikely to have a copy of the block already cached, thereby making the *update-write* inadequate.

4.3. Compiler-Plus-Directory Coherence Mechanism

While the version control [Cheong1989] and time-stamp [Min1989] coherence mechanisms keep extra state information in each cache to help preserve temporal locality between parallel tasks, another mechanism that combines static and dynamic coherence detection [Chen1991] keeps this extra state information in a directory in the memory modules. The directory monitors the memory references generated by the program and dynamically updates its state to precisely determine which caches contain which memory blocks, and whether the blocks have been modified. At the parallel task boundary, each processor sequentially scans through its cache and invalidates the cache entries that the stored directory information specifies should be invalidated. Of course, this sequential scan could significantly increase the execution time of the program, but this coherence mechanism may be able to reduce the network traffic compared to a conventional directory. Unlike a conventional directory-based coherence mechanism, this approach uses the directory only to ensure that all cached blocks are updated with the correct state at the parallel task boundary, and not to perform dynamic invalidations. Consequently, it implicitly implements a weakly-ordered consistency model.

4.4. Extending the Memory Hierarchy into the Network

Instead of using the interconnection network for only moving data between the memory modules and the caches, it is possible to extend the memory hierarchy into the network itself. It may be possible to simplify the cache coherence mechanism, and to simultaneously improve performance, by caching data within the switches of the network. For example, the Memory Hierarchy Network [Mizrahi1989] adds a local memory to each switch in the network to cache the data being referenced by the processors that are connected to the switch. In addition, the switches maintain a distributed directory of where data is stored in the system. To simplify the coherence mechanism, only a single copy of a block is allowed in the system. This single copy then migrates through the network as it is referenced by the different processors.

One of the critical parameters in this type of system is the block migration policy. This policy determines when a shared block should migrate, and how far up the network it should move. Simulations of this network with different migration policies have indicated that distributing the directory throughout the network can significantly improve the performance of the memory system, while storing data at intermediate levels of the network has much less of an impact on performance. Additional research is needed to fully evaluate this idea of extending the memory system into the

interconnection network, but early results suggest that it is an approach that may be able to significantly improve multiprocessor memory performance.

5. Conclusions

Using private data caches in a shared memory multiprocessor can significantly reduce the average time required to access memory, but these private caches introduce the complexity of the cache coherence problem. This paper has identified several important architectural issues that affect the performance and implementation cost of a cache coherence mechanism. Trace-driven simulations have been used to quantify the performance impact of these different issues. These architectural issues affecting the cache coherence mechanism are:

5.1. Coherence detection strategy

The coherence detection strategy determines when and how memory references are disambiguated to detect that a possible incoherence exists among the data caches and the main memory. The dynamic coherence detection mechanisms examine the actual memory addresses generated at run-time. The resulting perfect memory disambiguation produces low miss ratios, but the dynamic mechanisms tend to have relatively high network traffic due to the messages required to maintain coherence. The static coherence detection schemes, in contrast, examine memory references at compile-time. Since these techniques rely on imprecise compiler-based data dependence tests to disambiguate memory references, they tend to invalidate more cache entries than are necessary to maintain coherence, and thus produce miss ratios that are higher than the dynamic mechanisms. The self-invalidation used by the static mechanisms tends to compensate for their lower miss ratios by reducing the network traffic compared to that produced by dynamic coherence detection strategies.

5.2. Coherence enforcement strategy

After detecting a possibly incoherent memory access, the cache coherence mechanism must prevent the stale data value from being referenced by a processor. The invalidation coherence enforcement strategy forces processors to invalidate blocks within their caches. If the block is referenced again, a miss will be generated which will cause the processor to fetch the current value of the block either from the main memory, or from another processor. With an update enforcement strategy, the new value of a block created by a write operation is automatically distributed to all processors with a cached copy of the block. When these processors reference the block again, they do not generate another miss service request. As a result, the update strategy tends to produce lower miss ratios than the invalidate strategy. The lower miss ratio of updating comes at the expense of its significantly higher network traffic when compared to invalidating, however.

5.3. Precision of block sharing information

The amount of block sharing information that is maintained by the coherence mechanism has a direct impact on the implementation cost of the mechanism, as measured by the number of memory bits required to store the sharing information, and a direct impact on the performance of the memory system. To reduce the memory requirements, the coherence mechanism, such as the n-pointer plus broadcast directory, can store a relatively small amount of information about which processors have a copy of a cached block. The mechanism then must resort to broadcasting of the invalidation messages when the number of processors sharing a block overflows the available resources. This

approach trades-off lower memory overhead with higher network traffic when compared to a directory that stores complete sharing information. However, recently proposed tagged directory schemes can achieve very low memory overhead by storing sharing information only for those blocks that are actually cached. These directories still can maintain low network traffic since they are able to store sufficient sharing information for each cached block.

5.4. Cache block size

An important factor affecting the performance of the memory system is the cache block size, which is the number of words stored in the cache as a single unit. The use of cache blocks larger than a single word may allow the processors to exploit the spatial locality typical of memory referencing behavior. However, memory references in a multiprocessor system tend to be spread out among the processors which reduces the available spatial locality compared to a uniprocessor system. In addition, blocks larger than a single word introduce the false sharing problem which tends to make multiprocessor systems favor small cache block sizes. In some application programs, it may be possible to reduce the miss ratio by using multiword blocks, but simulation studies suggest that single word blocks minimize the network traffic by reducing both the miss service traffic and the invalidation traffic.

5.5. Summary

Finally, it is important to point out that it is possible to incorporate several different cache coherence mechanisms into a single system. For instance, the DASH prototype has demonstrated a coherence mechanism that incorporates both a bus-based snooping coherence mechanism and a directory-based coherence mechanism, and it gives the programmer a choice of both updating and invalidating coherence enforcement strategies. In addition, it is possible to use compile-time information to augment the performance of a coherence mechanism; for instance, to reduce the size of the directory by reducing the number of coherence pointers that need to be allocated, and by reducing the time they need to be active. Since each of the factors affecting the cache coherence mechanism produces different trade-offs in terms of miss ratios and network traffic, it is likely that these hybrid approaches will provide the best opportunity for increasing the performance and reducing the implementation cost of the cache coherence mechanism in large-scale shared memory multiprocessors.

Acknowledgements

Thanks to Farnaz Mounes-Toussi for generating the data used to compare the updating and invalidating coherence enforcement strategies, and to Hector Garcia-Molina, Dick Muntz, and the anonymous referees for their considerable efforts in reviewing the early drafts of this paper. Their insightful comments and suggestions helped to significantly improve the focus of the paper and the clarity of the presentation. This work was supported in part by the National Science Foundation under grant CCR-9209458, and by the research funds of the Graduate School of the University of Minnesota.

References

Adve1991.
Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon, ‘‘Comparison of Hardware and Software Cache Coherence Schemes,’’ *International Symposium on Computer Architecture*, pp. 298-308, 1991.

Agarwal1988.
Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz, ‘‘An Evaluation of Directory Schemes for Cache Coherence,’’ *International Symposium on Computer Architecture*, pp. 280-289, 1988.

Agarwal1988a.
Anant Agarwal and Anoop Gupta, ‘‘Memory-Reference Characteristics of Multiprocessor Applications Under MACH,’’ *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 215-225, 1988.

Anderson1990.
Thomas E. Anderson, ‘‘The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors,’’ *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 6-16, January 1990.

Archibald1984.
James Archibald and Jean-Loup Baer, ‘‘An Economical Solution to the Cache Coherence Problem,’’ *International Symposium on Computer Architecture*, pp. 355-362, 1984.

Archibald1988.
James K. Archibald, ‘‘A Cache Coherence Approach for Large Multiprocessor Systems,’’ *ACM International Conference on Supercomputing*, pp. 337-345, 1988.

Banerjee1988.
U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

Bennett1990.
John K. Bennett, John B. Carter, and Willy Zwaenepoel, ‘‘Adaptive Software Cache Management for Distributed Shared Memory Architectures,’’ *International Symposium on Computer Architecture*, pp. 125-134, 1990.

Bhuyan1989.
Laxmi N. Bhuyan, Bao-Chyn Liu, and Irshad Ahmed, ‘‘Analysis of MIN-Based Multiprocessors with Private Cache Memories,’’ *International Conference on Parallel Processing, Vol. I: Architecture*, pp. 51-58, 1989.

Brantley1985.
W. C. Brantley, K. P. McAuliffe, and J. Weiss, ‘‘RP3 Processor-Memory Element,’’ *International Conference on Parallel Processing*, pp. 782-789, 1985.

Censier1978.
Lucien M. Censier and Paul Feautrier, ‘‘A New Solution to Coherence Problems in Multicache Systems,’’ *IEEE Transactions on Computers*, Vol. C-27, No. 12, pp. 1112-1118, December 1978.

Chaiken1991.
David Chaiken, John Kubiatowicz, and Anant Agarwal, ‘‘LimitLESS Directories: A Scalable Cache Coherence Scheme,’’ *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 224-234, 1991.

Chen1991.
Yung-Chin Chen and Alexander V. Veidenbaum, ‘‘A Software Coherence Scheme with the Assistance of Directories,’’ *ACM International Conference on Supercomputing*, 1991.

Cheong1988.
Hoichi Cheong and Alexander V. Veidenbaum, ‘‘Stale Data Detection and Coherence Enforcement Using Flow Analysis,’’ *International Conference on Parallel Processing, Vol. I: Architecture*, pp. 138-145, 1988.

Cheong1988a.
Hoichi Cheong and Alexander V. Veidenbaum, ‘‘A Cache Coherence Scheme with Fast Selective Invalidation,’’ *International Symposium on Computer Architecture*, pp. 299-307, 1988.

Cheong1989.
Hoichi Cheong and Alexander Veidenbaum, ‘‘A Version Control Approach to Cache Coherence,’’ *ACM International Conference on Supercomputing*, pp. 322-330, 1989.

Dubois1988.
Michel Dubois, Christoph Scheurich, and Faye A. Briggs, ‘‘Synchronization, Coherence, and Event Ordering in Multiprocessors,’’ *Computer*, Vol. 21, No. 2, pp. 9-21, February 1988.

Edler1985.
Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, Marc Snir, Patricia J. Teller, and James Wilson, ‘‘Issues Related to MIMD Shared-Memory Computers: the NYU Ultracomputer Approach,’’ *International Symposium on Computer Architecture*, pp. 126-135, 1985.

Eggers1988.
Susan J. Eggers and Randy H. Katz, ‘‘A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation,’’ *International Symposium on Computer Architecture*, pp. 373-382, 1988.

Eggers1989.
Susan J. Eggers and Randy H. Katz, ‘‘Evaluating the Performance of Four Snooping Cache Coherency Protocols,’’ *International Symposium on Computer Architecture*, pp. 2-15, 1989.

Eggers1989a.
Susan J. Eggers and Randy H. Katz, ‘‘The Effect of Sharing on the Cache and Bus Performance of Parallel Programs,’’ *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257-270, 1989.

Gharachorloo1990.
K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, ‘‘Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors,’’ *International Symposium on Computer Architecture*, pp. 15-26, 1990.

Gharachorloo1991.
K. Gharachorloo, A. Gupta, and J. Hennessy, ‘‘Performance Evaluation of Memory Consistency

Models for Shared-Memory Multiprocessors,'' *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 245-257, 1991.

Goodman1989.
J. Goodman, M. Vernon, and P. Woest, ''A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor,'' *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64-73, 1989.

Goodman1983.
James R. Goodman, ''Using Cache Memory to Reduce Processor-Memory Traffic,'' *International Symposium on Computer Architecture*, pp. 124-131, 1983.

Goodman1988.
James R. Goodman and Philip J. Woest, ''The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor,'' *International Symposium on Computer Architecture*, pp. 422-431, 1988.

Gottlieb1982.
Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir, ''The NYU Ultracomputer -- Designing a MIMD, Shared-Memory Parallel Machine,'' *International Symposium on Computer Architecture*, pp. 27-42, 1982.

Gupta1990.
Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry, ''Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes,'' *International Conference on Parallel Processing, Vol. I: Architecture*, pp. 312-321, 1990.

Gupta1991.
A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, ''Comparative Evaluation of Latency Reducing and Tolerating Techniques,'' *International Symposium on Computer Architecture*, pp. 254-263, 1991.

James1990.
David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi, ''Scalable Coherent Interface,'' *Computer*, Vol. 23, No. 6, pp. 74-77, June 1990.

Karline1986.
A. R. Karline, M. S. Manass, L. Rudolph, and D. D. Sleator, ''Competitive Snoopy cacheing,'' *Symposium on Foundations of Computer Science*, pp. 244-254, October 1986.

Katz1985.
R. Katz, S. Eggers, D. A. Wood, C. Perkins, and R. G. Sheldon, ''Implementing a Cache Consistency Protocol,'' *International Symposium on Computer Architecture*, pp. 276-283, 1985.

Kruskal1986.
Clyde P. Kruskal, Larry Rudolph, and Marc Snir, ''Efficient Synchronization on Multiprocessors with Shared Memory,'' *ACM Symposium on Principles of Distributed Computing*, pp. 218-228, 1986.

Kuck1986.
David J. Kuck, Edward S. Davidson, Duncan J. Lawrie, and Ahmed H. Sameh, ''Parallel Supercomputing Today and the Cedar Approach,'' *Science*, Vol. 231, pp. 967-974, 28 February 1986.

Lamport1979.
Leslie Lamport, ‘‘How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,’’ *IEEE Transactions on Computers,* Vol. C-28, No. 9, pp. 690-691, September 1979.

Lee1987.
Roland L. Lee, Pen-Chung Yew, and Duncan J. Lawrie, ‘‘Multiprocessor Cache Design Considerations,’’ *International Symposium on Computer Architecture,* pp. 253-262, 1987.

Lenoski1990.
D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, ‘‘The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor,’’ *International Symposium on Computer Architecture*, pp. 148-159, 1990.

Lenoski1992.
D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, ‘‘The Stanford DASH Multiprocessor,’’ *Computer*, Vol. 25, No. 3, pp. 63-79, March 1992.

Li1990.
Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu, ‘‘An Efficient Data Dependence Analysis for Parallelizing Compilers,’’ *IEEE Transactions on Parallel and Distributed Systems,* Vol. 1, No. 1, pp. 26-34, January 1990.

Lichnewsky1988.
A. Lichnewsky and F. Thomasset, ‘‘Introducing Symbolic Problem Solving Techniques in the Dependence Testing Phase of a Vectorizer,’’ *International Conference on Supercomputing*, 1988.

Lilja1989.
David J. Lilja, David M. Marcovitz, and Pen-Chung Yew, ‘‘Memory Referencing Behavior and a Cache Performance Metric in a Shared Memory Multiprocessor,’’ *Center for Supercomputing Research and Development Report No. 836, University of Illinois*, Urbana, IL, 1989.

Lilja1991.
David J. Lilja, ‘‘Processor Parallelism Considerations and Memory Latency Reduction in Shared Memory Multiprocessors,’’ *Center for Supercomputing Research and Development Report No. 1136, University of Illinois (Ph.D. Thesis)*, Urbana, 1991.

Lilja1991a.
David J. Lilja and Pen-Chung Yew, ‘‘Combining Hardware and Software Cache Coherence Strategies,’’ *ACM International Conference on Supercomputing,* pp. 274-283, 1991.

Lilja1992.
David J. Lilja, ‘‘Prefetching and Scheduling Interactions in Shared Memory Multiprocessors,’’ *Midwest Electrotechnology Conference,* pp. 84-87, April 1992.

Marquardt1989.
Douglas E. Marquardt and Hasan S. AlKhatib, ‘‘C2MP: A Cache-Coherent, Distributed-Memory Multiprocessor System,’’ *Proceedings Supercomputing ’89,* pp. 466-475, 1989.

McCreight1984.
E. M. McCreight, ‘‘The Dragon Computer System, an Early Overview,’’ *NATO Advanced Study Institute on Microarchitecture VLSI Computers,* pp. 83-101, July 1984.

Min1989.
Sang Lyul Min and Jean-Loup Baer, ‘‘A Timestamp-Based Cache Coherence Scheme,’’ *International Conference on Parallel Processing, Vol. I: Architecture*, pp. 23-32, 1989.

Min1990.
Sang Lyul Min and Jean-Loup Baer, ‘‘A Performance Comparison of Directory-based and Timestamp-based Cache Coherence Schemes,’’ *International Conference on Parallel Processing, Vol I: Architecture*, pp. 305-311, 1990.

Mizrahi1989.
H. E. Mizrahi, J.-L. Baer, E. D. Lazowska, and J. Zahorjan, ‘‘Extending the Memory Hierarchy into Multiprocessor Interconnection Networks: A Performance Analysis,’’ *International Conference on Parallel Processing, Vol. I: Architecture*, pp. 41-50, 1989.

Mounes-Toussi1993.
Farnaz Mounes-Toussi, ‘‘An Adaptive Cache Coherence Enforcement Strategy with Compiler Assistance,’’ *Department of Electrical Engineering, University of Minnesota (M.S. Thesis)*, Minneapolis, 1993.

Nguyen1993.
Trung N. Nguyen, Zhiyuan Li, and David J. Lilja, ‘‘Efficient Use of Dynamically Tagged Directories Through Compiler Analysis,’’ *International Conference on Parallel Processing*, August 1993.

O’Krafka1990.
Brian W. O’Krafka and A. Richard Newton, ‘‘An Empirical Evaluation of Two Memory-Efficient Directory Methods,’’ *International Symposium on Computer Architecture*, pp. 138-147, 1990.

Papamarcos1984.
Mark S. Papamarcos and Janak H. Patel, ‘‘A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories,’’ *International Symposium on Computer Architecture*, pp. 348-354, 1984.

Perron1986.
Robert Perron and Craig Mundie, ‘‘The Architecture of the Alliant FX/8 Computer,’’ *IEEE COMPCON*, pp. 390-393, 1986.

Pfister1985.
G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, ‘‘The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture,’’ *International Conference on Parallel Processing*, pp. 764-771, 1985.

Polychronopoulos1988.
Constantine D. Polychronopoulos, ‘‘Toward Auto-scheduling Compilers,’’ *Journal of Supercomputing*, No. 2, pp. 297-330, 1988.

Przybylski1988.
Steven Przybylski, Mark Horowitz, and John Hennessy, ‘‘Performance Tradeoffs in Cache Design,’’ *International Symposium on Computer Architecture*, pp. 290-296, 1988.

Smith1982.
Alan Jay Smith, ‘‘Cache Memories,’’ *ACM Computing Surveys*, Vol. 14, No. 3, pp. 473-530,

September 1982.

Smith1987.
Alan Jay Smith, ‘‘Line (Block) Size Choice for CPU Cache Memories,’’ *IEEE Transactions on Computers*, Vol. C-36, No. 9, pp. 1063-1075, September 1987.

Stunkel1991.
Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs, ‘‘Address Tracing for Parallel Machines,’’ *Computer*, Vol. 24, No. 1, pp. 31-38, January 1991.

Tang1976.
C. K. Tang, ‘‘Cache Design in the Tightly Coupled Multiprocessor System,’’ *AFIPS Conference Proceedings, National Computer Conference*, pp. 749-753, 1976.

Thacker1988.
Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, ‘‘Firefly: A Multiprocessor Workstation,’’ *IEEE Transactions on Computers*, Vol. 37, No. 8, pp. 909-920, August 1988.

Torrellas1990.
Josep Torrellas and John Hennessy, ‘‘Estimating the Performance Advantages of Relaxing Consistency in a Shared-Memory Multiprocessor,’’ *International Conference on Parallel Processing, Vol I: Architecture*, pp. 26-33, 1990.

Veidenbaum1986.
Alexander V. Veidenbaum, ‘‘A Compiler-Assisted Cache Coherence Solution for Multiprocessors,’’ *International Conference on Parallel Processing*, pp. 1029-1036, 1986.

Weber1989.
Wolf-Dietrich Weber and Anoop Gupta, ‘‘Analysis of Cache Invalidation Patterns in Multiprocessors,’’ *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 243-256, 1989.

Wilson1987.
A. W. Wilson, ‘‘Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors,’’ *International Symposium on Computer Architecture*, pp. 244-252, 1987.

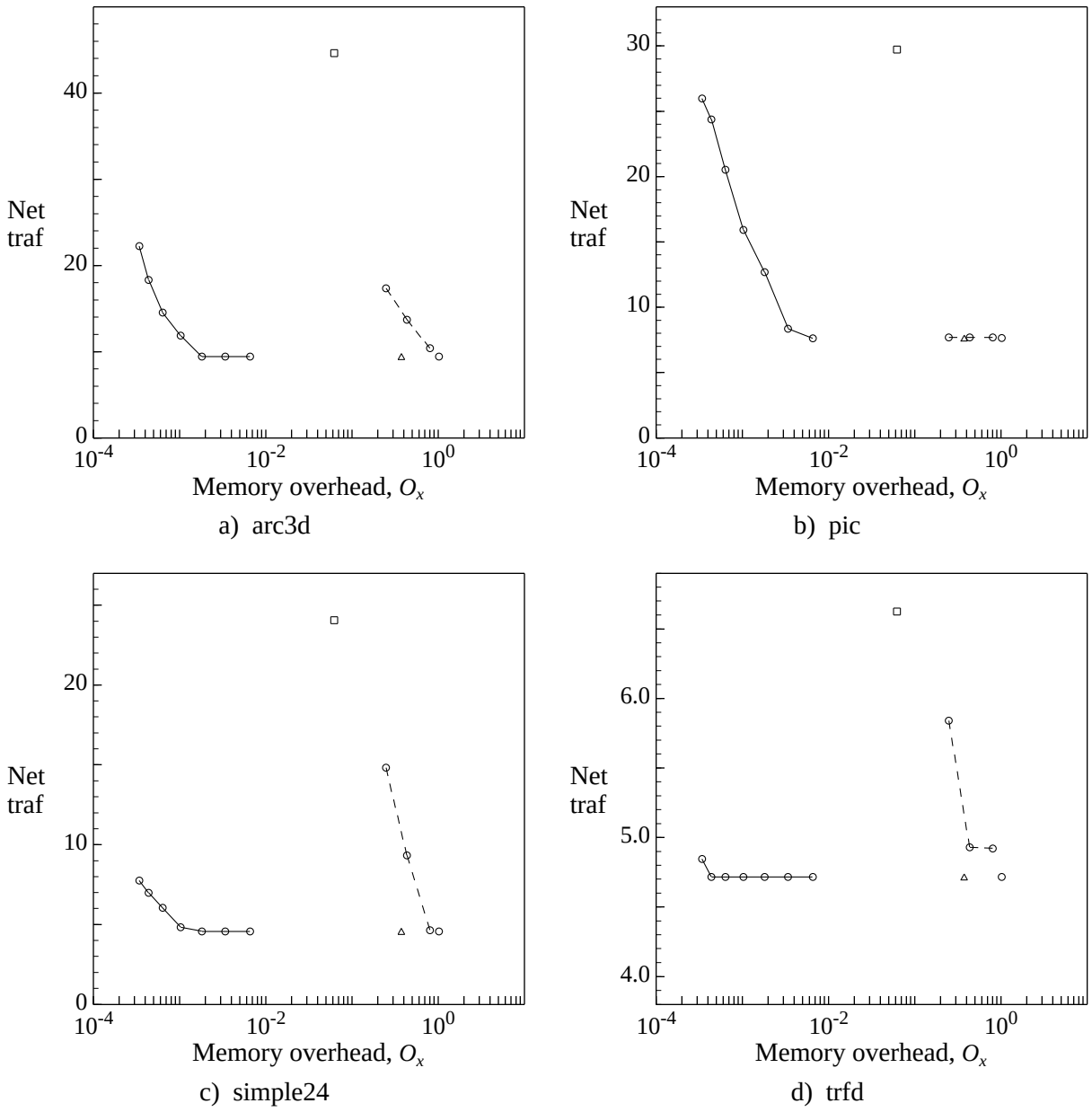
Yen1985.
Wei C. Yen, David W. L. Yen, and King-Sun Fu, ‘‘Data Coherence Problem in a Multicache System,’’ *IEEE Transactions on Computers*, Vol. C-34, No. 1, pp. 56-65, January 1985.

Zucker1992.
Richard N. Zucker and Jean-Loup Baer, ‘‘A Performance Study of Memory Consistency Models,’’ *International Symposium on Computer Architecture*, pp. 2-12, 1992.

CONTENTS

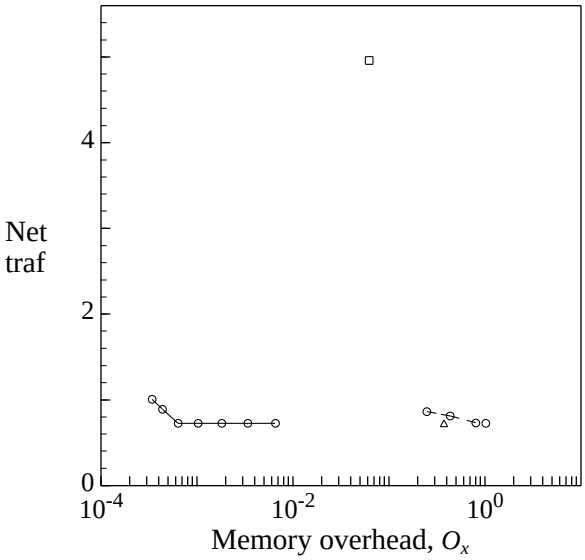
1. Introduction 1
    1.1. Problem Definition 2
        1.1.1. Consistency Models 2
        1.1.2. Cache Coherence 3
        1.1.3. Relationship Between Consistency Models and Coherence 5
    1.2. Overview of Cache Coherence Mechanisms 5
        1.2.1. Snooping Coherence 5
        1.2.2. Directory Coherence 6
        1.2.3. Compiler-Directed Coherence 7
    1.3. Factors Affecting Coherence Mechanisms 7
2. Cost and Performance Modeling 8
    2.1. Trace-Driven Simulation 8
    2.2. Machine Model 9
    2.3. Test Programs 11
    2.4. Performance Metrics 11
3. Performance Impacts 12
    3.1. Coherence Detection Strategy 12
        3.1.1. Memory Disambiguation 13
        3.1.2. Static (Compile-Time) Coherence Detection Mechanisms 14
        3.1.3. Dynamic (Run-Time) Coherence Detection Mechanisms 15
        3.1.4. Performance Comparisons 15
    3.2. Coherence Enforcement Strategy 17
        3.2.1. Performance Comparisons 18
        3.2.2. Adaptive Coherence Enforcement 19
    3.3. Precision of Block Sharing Information 20
        3.3.1. Traditional Directories 21
        3.3.2. Tagged Directories 22
        3.3.3. Cost and Performance Comparisons 23
    3.4. Cache Block Size 25
        3.4.1. Performance Effects 26
4. Hybrid Techniques 27
    4.1. Compiler Assistance for Reducing the Directory Size 27
    4.2. Combining Multiple Coherence Mechanisms 27
    4.3. Compiler-Plus-Directory Coherence Mechanism 28
    4.4. Extending the Memory Hierarchy into the Network 28
5. Conclusions 29
    5.1. Coherence detection strategy 29
    5.2. Coherence enforcement strategy 29
    5.3. Precision of block sharing information 29
    5.4. Cache block size 30

5.5. Summary 30
Acknowledgements 30
References 31

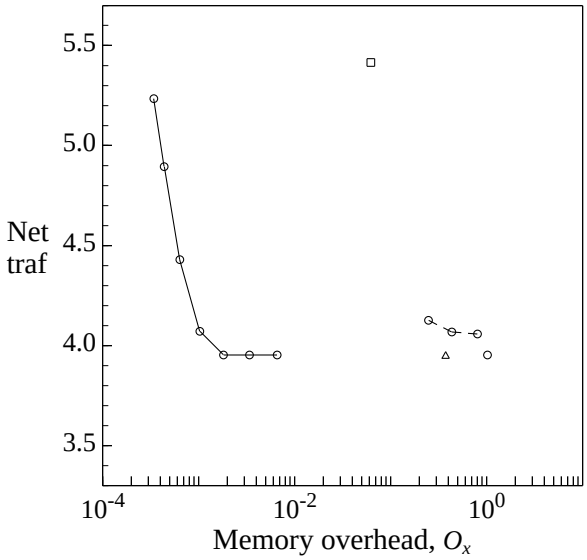


solid line = pointer cache tagged directory, s=1/32,1/16,1/8,1/4,1/2,1/1,2/1
square point = 2-bit broadcast
triangle point = linked list
dashed line = n-pointer plus broadcast, n=1,2,4
circle point = full directory

Figure 3: The effect of the precision of block sharing information (memory overhead) on network traffic.

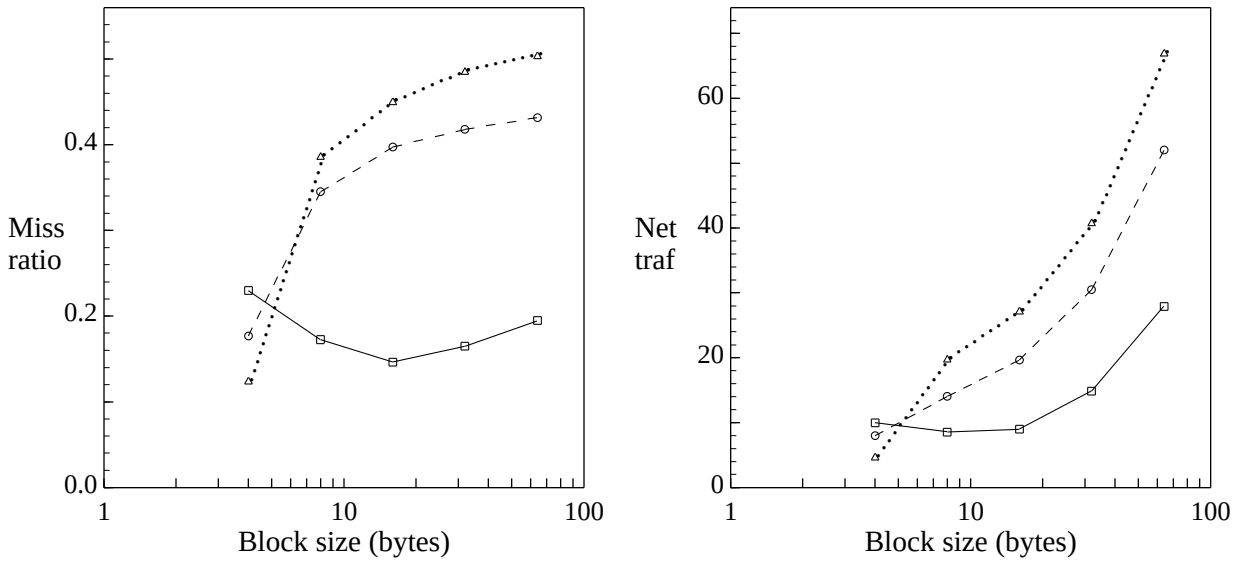


e) flo52



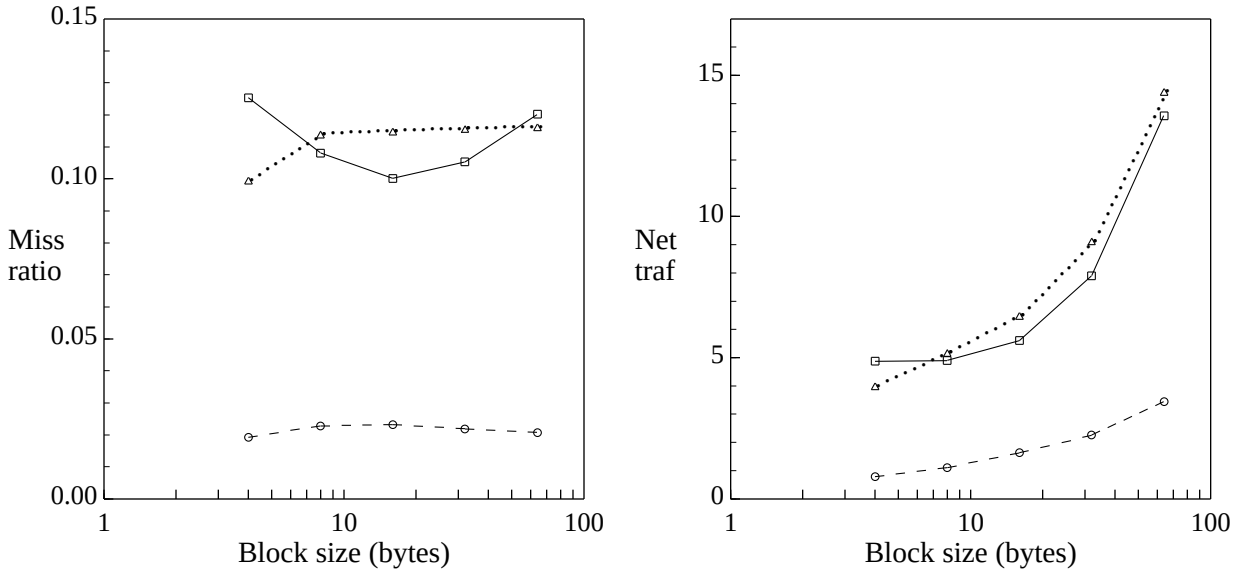
f) lin125

Figure 3: (cont.)



Miss ratio Network traffic

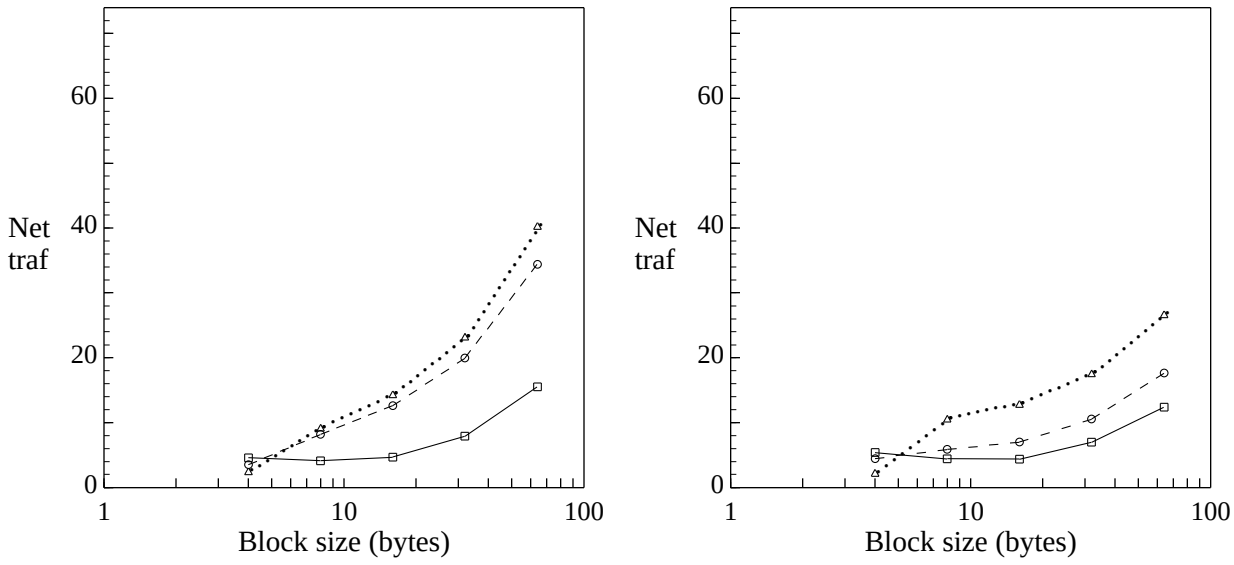
Solid = *arc3d*; Dash = *pic;* Dot = *trfd*



Miss ratio Network traffic

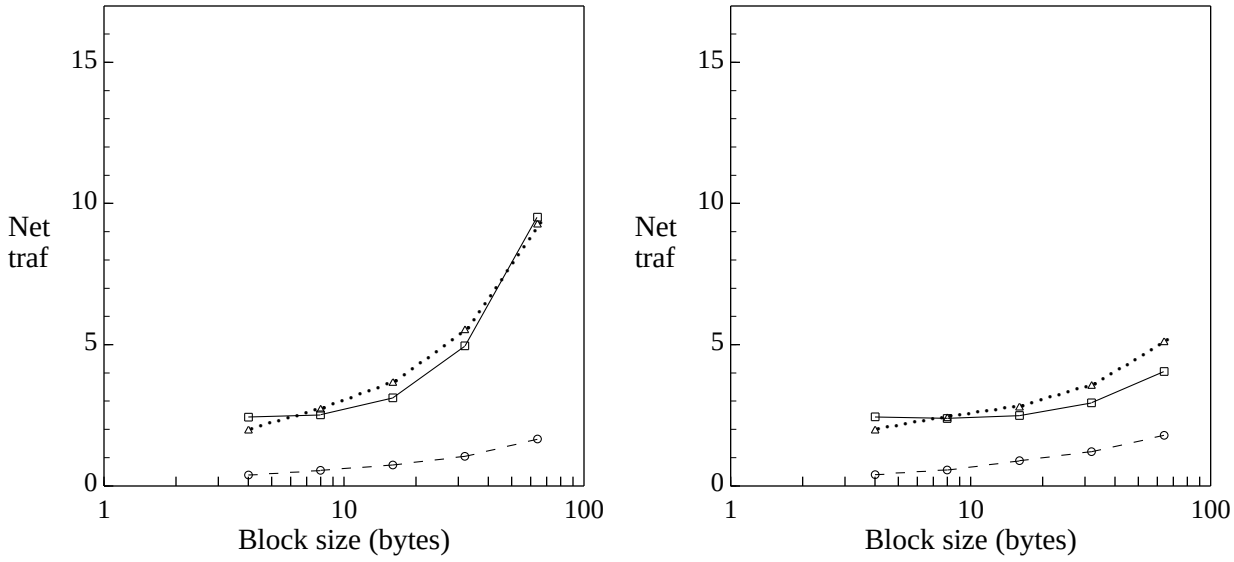
Solid = *simple24*; Dash = *flo52*; Dot = *lin125*

Figure 4: Effect of cache block size on miss ratio and network traffic (bytes/reference).



Miss traffic Invalidation traffic

Solid = *arc3d*; Dash = *pic;* Dot = *trfd*



Miss traffic Invalidation traffic

Solid = *simple24*; Dash = *flo52*; Dot = *lin125*

Figure 5: Components of total network traffic due to cache misses and due to invalidations from data sharing.