Chapter 1
Introductory Readings

This chapter includes the papers intended to introduce the reader inexperienced in the field of cache coherence maintenance in shared-memory multiprocessors to the topic. Also, the chapter contains the main contributions to the surveying and the classifying efforts within the field.

The first short paper, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," written by Lamport and widely cited in the literature, introduces the notion of "sequential consistency"—a conservative model of memory access ordering that guarantees the correctness of a multiprocess program on a multiprocessor. This paper defines the requirements that must be satisfied by a multiprocessor to be sequentially consistent.

The second paper, "Synchronization, Coherence, and Event Ordering in Multiprocessors," written by Dubois, Scheurich, and Briggs, presents three closely related topics in the multiprocessor world. This paper gives a broad overview of hardware-level and software-level synchronization mechanisms, presents a number of techniques for coherence maintenance, and discusses two major categories of logical behavior of shared-memory multiprocessors based on the strong or weak ordering of events.

The third paper, "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons," written by Lilja, addresses some issues of interest for deeper understanding of the cache coherence maintenance problem. Such issues as coherence detection strategy or coherence enforcement strategy are crucial for any coherence maintenance mechanism (snooping, directory-based, compiler-directed) and may significantly affect the performance of a multiprocessor.

The fourth paper, originally developed by the editors of this book, "A Survey of Software Solutions for Maintenance of Cache Consistency in Shared-Memory Multiprocessors," concentrates exclusively on software schemes. Although the survey is, in our opinion, broad enough to include most relevant software solutions, we expect that the presentation is sufficiently detailed for proper understanding of the mechanisms and characteristics of the approaches surveyed. We also propose a flexible classification of software solutions to the problem, apply the classification criteria on the existing software schemes, and attempt to generalize the classification.

Suggestions for Further Reading

Adve, S.V., and Hill, M.D., "Weak Ordering—A New Definition," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 2–14.

Adve, S.V., and Hill, M.D., "A Unified Formalization of Four Shared-Memory Models," *IEEE Trans. Parallel and Distributed Systems,* Vol. 4, No. 6, June 1993, pp. 613–624.

Chaiken, D., et al., "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *Computer*, Vol. 23, No. 6, June 1990, pp. 49–58.

Gharachorloo, K., et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 15–26.

Protić, J., Tomašević, M., and Milutinović, V., "A Survey of Distributed Share -Memory Systems," *Proc. Hawaii Int'l Conf. System Sciences*, Vol. 1, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 74–84.

Smith, A.J., "Cache Memories," *ACM Computing Surveys,* Vol. 14, No. 3, Sept. 1982, pp. 473–530.

Stenström, P., "Reducing Contention in Shared-Memory Multiprocessors," *Computer,* Vol. 21, No. 11, Nov. 1988, pp. 26–37.

Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer,* Vol. 23, No. 6, June 1990, pp. 12–24.

Sweazey, P., and Smith, A.J., "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus," *Proc. 13th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1986, pp. 414–423.

Teller, P., "Translation-Lookaside Buffer Consistency," *Computer,* Vol. 23, No. 6, June 1990, pp. 26–36.

Tomašević, M., and Milutinović, V., *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions*, IEEE CS Press, Los Alamitos, Calif., 1993.

Tomašević, M., and Milutinović, V., "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors: Part 1," *IEEE Micro*, Vol. 14, No. 5, Oct. 1994, pp. 52–59.

Tomašević, M., and Milutinović, V., "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors: Part 2," *IEEE Micro*, Vol. 14, No. 6, Dec. 1994, pp. 61–66.

Yen, W.C., Yen, D.W.L., and Fu, K.-S., "Data Coherence Problem in a Multicache System," *IEEE Trans. Computers,* Vol. C-34, No. 1, Jan. 1985, pp. 56–65.

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

Leslie Lamport

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

Index Terms—**Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.**

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for such a computer assumes that the following condition is satisfied: the result of any execution 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. A multiprocessor satisfying this condition will be called *sequentially consistent*. The sequentiality of each individual processor does not guarantee that the multiprocessor computer is sequentially consistent. In this brief note, we describe a method of interconnecting sequential processors with memory modules that insures the sequential consistency of the resulting multiprocessor.

Manuscript received September 28, 1977; revised May 8, 1979.

The author is with the Computer Science Laboratory, SRI International, Merls Park, CA 94025.

We assume that the computer consists of a collection of processors and memory modules, and that the processors communicate with one another only through the memory modules. (Any special communication registers may be regarded as separate memory modules.) The only processor operations that concern us are the operations of sending fetch and store requests to memory modules. We assume that each processor issues a sequence of such requests. (It must sometimes wait for requests to be executed, but that does not concern us.)

We illustrate the problem by considering a simple two-process mutual exclusion protocol. Each process contains a *critical section*, and the purpose of the protocol is to insure that only one process may be executing its critical section at any time. The protocol is as follows.

```
process 1
    a := 1;
    if b = 0 then critical section;
                  a := 0
             else  ···   fi
```

process 2

b := 1;

if a = 0 then critical section;

b := 0

else ··· fi

The else clauses contain some mechanism for guaranteeing eventual access to the critical section, but that is irrelevant to the discussion. It is easy to prove that this protocol guarantees mutually exclusive access to the critical sections. (Devising a proof provides a nice exercise in using the assertional techniques of [2] and [3], and is left to the reader.) Hence, when this two-process program is executed by a sequentially consistent multiprocessor computer, the two processors cannot both be executing their critical sections at the same time.

We first observe that a sequential processor could execute the "b := 1" and "fetch b" operations of process 1 in either order. (When process 1's program is considered by itself, it does not matter in which order these two operations are performed.) However, it is easy to see that executing the "fetch b" operation first can lead to an error—both processes could then execute their critical sections at the same time. This immediately suggests our first requirement for a multiprocessor computer.

Requirement R1: Each processor issues memory requests in the order specified by its program.

Satisfying Requirement R1 is complicated by the fact that storing a value is possible only after the value has been computed. A processor will often be ready to issue a memory fetch request before it knows the value to be stored by a preceding store request. To minimize waiting, the processor can issue the store request to the memory module without specifying the value to be stored. Of course, the store request cannot actually be executed by the memory module until it receives the value to be stored.

Requirement R1 is not sufficient to guarantee correct execution. To see this, suppose that each memory module has several ports, and each port services one processor (or I/O channel). Let the values of "a" and "b" be stored in separate memory modules, and consider the following sequence of events.

1) Processor 1 sends the "a := 1" request to its port in memory module 1. The module is currently busy executing an operation for some other processor (or I/O channel).
2) Processor 1 sends the "fetch b" request to its port in memory module 2. The module is free, and execution is begun.
3) Processor 2 sends its "b := 1" request to memory module 2. This request will be executed after processor 1's "fetch b" request is completed.
4) Processor 2 sends its "fetch a" request to its port in memory module 1. The module is still busy.

There are now two operations waiting to be performed by memory module 1. If processor 2's "fetch a" operation is performed first, then both processes can enter their critical sections at the same time, and the protocol fails. This could happen if the memory module uses a round robin scheduling discipline in servicing its ports.

In this situation, an error occurs only if the two requests to memory module 1 are not executed in the same order in which they were received. This suggests the following requirement.

Requirement R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

Condition R1 implies that a processor may not issue any further memory requests until after its current request has been entered on the queue. Hence, it must wait if the queue is full. If two or more processors are trying to enter requests in the queue at the same time, then it does not matter in which order they are serviced.

Note. If a fetch requests the contents of a memory location for which there is already a write request on the queue, then the fetch need not be entered on the queue. It may simply return the value from the last such write request on the queue. □

Requirements R1 and R2 insure that if the individual processors are sequential, then the entire multiprocessor computer is sequentially consistent. To demonstrate this, one first introduces a relation $\rightarrow$ on memory requests as follows. Define A $\rightarrow$ B if and only if 1) A and B are issued by the same processor and A is issued before B, or 2) A and B are issued to the same memory module, and A is entered in the queue before B (and is thus executed before B). It is easy to see that R1 and R2 imply that $\rightarrow$ is a partial ordering on the set of memory requests. Using the sequentiality of each processor, one can then prove the following result: each fetch and store operation fetches or stores the same value as if all the operations were executed sequentially in any order such that A $\rightarrow$ B implies that A is executed before B. This in turn proves the sequential consistency of the multiprocessor computer.

Requirement R2 states that a memory module's request queue must be serviced in a FIFO order. This implies that the memory module must remain idle if the request at the head of its queue is a store request for which the value to be stored has not yet been received. Condition R2 can be weakened to allow the memory module to service other requests in this situation. We need only require that all requests *to the same memory cell* be serviced in the order that they appear in the queue. Requests to different memory cells may be serviced out of order. Sequential consistency is preserved because such a service policy is logically equivalent to considering each memory cell to be a separate memory module with its own request queue. (The fact that these modules may share some hardware affects the rate at which they service requests and the capacity of their queues, but it does not affect the logical property of sequential consistency.)

The requirements needed to guarantee sequential consistency rule out some techniques which can be used to speed up individual sequential processors. For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task.

References

[1] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, pp. 115-138, 1971.

[2] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 125-143, Mar. 1977.

[3] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Commun. Assoc. Comput. Mach.*, vol. 19, pp. 279-285, May 1976.

SURVEY & TUTORIAL SERIES

Synchronization, Coherence, and Event Ordering in Multiprocessors

Michel Dubois and Christoph Scheurich
Computer Research Institute, University of Southern California

Fayé A. Briggs
Sun Microsystems

Multiprocessors, especially those constructed of relatively low-cost microprocessors, offer a cost-effective solution to the continually increasing need for computing power and speed. These systems can be designed either to maximize the throughput of many jobs or to speed up the execution of a single job; they are respectively called *throughput-oriented* and *speedup-oriented multiprocessors*. In the first type of system, jobs are distinct from each other and execute as if they were running on different uniprocessors. In the second type an application is partitioned into a set of cooperating processes, and these processes interact while executing concurrently on different processors. The partitioning of a job into cooperating processes is called *multitasking*[1]* or *multithreading*. In both systems global resources must be managed correctly and efficiently by the operating system. The problems addressed in this article apply to both throughput- and speedup-oriented multiprocessor systems, either at the user level or the operating-system level.

Efficient multiprocessing depends on a harmonious blend of synchronization, coherence, and event ordering in a system that provides the user with a simple logical model of concurrency behavior.

Multitasked multiprocessors are capable of efficiently executing the many cooperating numerical or nonnumerical tasks that comprise a large application. In general, the speedup provided by multitasking reduces the turnaround time of a job and therefore ultimately improves the user's productivity. For applications such as real-time processing, CAD/CAM, and simulations, multitasking is crucial because the multiprocessor structure improves the execution speed of a given algorithm within a time constraint that is ordinarily impossible to meet on a single processor employing available technology.

Designing and programming multiprocessor systems correctly and efficiently pose complex problems. Synchronizing processes, maintaining data coherence, and ordering events in a multiprocessor are issues that must be addressed from the hardware design level up to the programming language level. The goal of this article is not only to review these problems in some depth but also to show that in the design of multiprocessors these problems are intricately related. The definitions and concepts presented here provide a solid foundation on which to reason about the logical properties of a specific multiproces-

*Multitasking is not restricted to multiprocessor systems; in this article, however, we confine our discussion, with no loss of generality, to multitasking multiprocessors.

Reprinted from *Computer*, Vol. 21, No. 2, Feb. 1988, pp. 9–21.

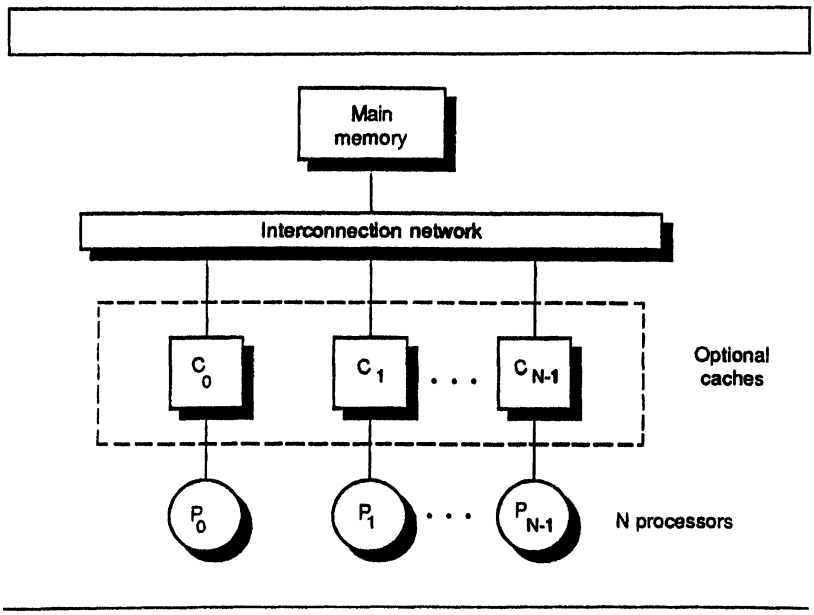


Figure 1. A shared-memory multiprocessor with optional private caches. The interconnection network may be either a simple bus or a complex network.

sor and to demonstrate that the hardware adheres to the logical model expected by the programmer. This foundation aids in understanding complex but useful architectures such as multiprocessors with private caches or with recombining interconnection networks (Figure 1).[2] Other important issues, such as scheduling and partitioning, have been addressed in a previous survey article.[3] Readers who are not familiar with the concept of cache memory should consult the survey by Smith.[4]

Basic definitions

The instruction set of a multiprocessor usually contains basic instructions that are used to implement synchronization and communication between cooperating processes. These instructions are usually supported by special-purpose hardware. Some primary hardware functions are necessary to guarantee correct interprocess communication and synchronization, while other, secondary hardware functions simplify the design of parallel applications and operating systems. The notions of synchronization and communication are difficult to separate because communication primitives can be used to implement synchronization protocols, and vice versa. In general, *communication* refers to the exchange of data between different processes. Usually, one or several sender processes transmit data to one or several receiver processes. Interprocess communication is mostly the result of explicit directives in the program. For example, parameters passed to a coroutine and results returned by such a coroutine constitute interprocess communications. *Synchronization* is a special form of communication, in which the data are control information. Synchronization serves the dual purpose of enforcing the correct sequencing of processes and ensuring the mutually exclusive access to certain shared writable data. For example, synchronization primitives can be used to

(1) Control a producer process and a consumer process such that the consumer process never reads stale data and the producer process never overwrites data that have not yet been read by the consumer process.

(2) Protect the data in a database such that concurrent write accesses to the same record in the database are not allowed. (Such accesses can lead to the loss of one or more updates if two processes first read the data in sequence and then write the updated data back to memory in sequence.)

In shared-memory multiprocessor systems, communication and synchronization are usually implemented through the controlled sharing of data in memory.

A second issue addressed in this article is *memory coherence*, a system's ability to execute memory operations correctly. Censier and Feautrier define a coherent memory scheme as follows: "A memory scheme is coherent if the value returned on a Load instruction is always the value given by the latest Store instruction with the same address."[5] This definition has been useful in the design of cache coherence mechanisms.[4] As it stands, however, the definition is difficult to interpret in the context of a multiprocessor, in which data accesses may be buffered and may not be atomic. Accesses are buffered if multiple accesses can be queued before reaching their destination, such as main memory or caches. An access by processor i on a variable X is atomic if no other processor is allowed to access any copy of X while the access by processor i is in progress. It has been shown that memory accesses need not be atomic at the hardware level for correct execution of concurrent programs.[6,7] Correctness of execution depends on the expected behavior of the machine. Two major classes of logical machine behavior have been identified because they are common in existing multiprocessor systems: the *strongly ordered* and the *weakly ordered* models of behavior.[7] The hardware of the machine must enforce these models by proper ordering of storage accesses and execution of synchronization and communication primitives. This leads to the third issue, the *ordering of events*.

The strictest logical model for the ordering of events is called *sequential consistency*, defined by Lamport. In a multiprocessor *sequential consistency* refers to the allowable sequence of execution of instructions within the same process and among different concurrent processes. Lamport defines the term more rigorously: "[A system is sequentially consistent if] the result of any execution 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."[8]

Since the only way that two concurrent processors can affect each other's execution is through the sharing of writable data and the sending of interrupt signals, it is

the order of these events that really matters. In systems that are sequentially consistent we say that events are strongly ordered.

However, if we look at many systems (transaction systems, for example), it becomes clear that sequential consistency is often violated in favor of a weaker condition. In many machines it is often implicitly assumed that the programmer should make no assumption about the order in which the events that a process generates are observed by other processes between two explicit synchronization points. Accesses to shared writable data should be executed in a mutually exclusive manner, controlled by synchronizing variables. Accesses to synchronizing variables can be detected by the machine hardware at execution time. Strong ordering of accesses to these synchronizing variables and restoration of coherence at synchronization points are therefore the only restrictions that must be upheld. In such systems we say that events are weakly ordered. Weak ordering may result in more efficient systems, but the implementation problems remain the same as for strong ordering: strong ordering must still be enforced for synchronizing variables (rather than for all shared writable data).

We can infer from this discussion that synchronization, coherence, and ordering of events are closely related issues in the design of multiprocessors.

```
          {Processor 1:}
          A:=0
          .
          .
          A:=1                        /* event S1(A) */
LAB1:     If (B=1) goto LAB1          /* event L1(B) */
          <critical section>
          A:=0

          {Processor 2:}
          B:=0
          .
          .
          B:=1                        /* event S2(B) */
LAB2:     If (A=1) goto LAB2          /* event L2(A) */
          <critical section>
          B:=0
```

Figure 2. Synchronization protocol using two shared variables, *A* and *B*.

Communication and synchronization

Communication and synchronization are two facets of the same basic problem: how to design concurrent software that is correct and reliable, especially when the processes interact by exchanging control and data information. Multiprocessor systems usually include various mechanisms to deal with the various granules of synchronizable resources. Usually, low-level and simple primitives are implemented directly by the hardware. These primitives are the basic mechanisms that enforce mutual exclusion for more complex mechanisms implemented in microcode or software.

Hardware-level synchronization mechanisms. All multiprocessors include hardware mechanisms to enforce atomic operations. The most primitive memory operations in a machine are Loads and Stores. With atomic Loads and Stores complex synchronization protocols can be built. Figure 2 depicts a simple protocol. Before a processor can enter its critical section, it sets its control variable (*A* for processor 1 and *B* for processor 2) to 1. Hence, for both processors to be in their critical sections concurrently, both *A* and *B* must equal 1. But this is not possible, since a processor cannot enter its critical section if the other processor's control variable equals 1. Therefore, the two processors cannot execute their respective critical sections concurrently. This simple protocol can be deadlocked, but the problem can be remedied.[8] Such protocols are hard to design, understand, and prove correct, and in many cases they are inefficient.

More sophisticated synchronization primitives are usually implemented in hardware. If the primitive is simple enough, the controller of the memory bank can execute the primitive at the memory in the same way it executes a Load or a Store, at the added cost of a more complex memory controller. This is typically the case for the Test&Set and the Full/Empty bit primitives described below. Interprocessor interrupts are also possible hardware mechanisms for synchronization and communication. To send a message to another process currently running on a different processor, a process can send an interrupt to that processor to notify the destination process.

A common set of synchronization primitives consists of Test&Set(lock) and Reset(lock). The semantics of Test&Set and Reset are

```
TEST&SET(lock)
  { temp ← lock; lock ← 1;
    return temp; }
RESET(lock)
  { lock ← 0; }
```

The microcode or software will usually repeat the Test&Set until the returned value is 0. Synchronization at this level implies some form of busy waiting, which ties up a processor in an idle loop and increases the memory bus traffic and contention. The type of lock that relies on busy waiting is called a *spin-lock*.

To avoid spinning, interprocessor interrupts are used. A lock that relies on interrupts instead of spinning is called a *suspend-lock* (also called *sleep-lock* in the C.mmp[1]). This lock is similar to the spin-lock in the sense that a process does not relinquish the processor while it is waiting on a suspend-lock. However, whenever it fails to obtain the lock, it records its status in one field of the lock and disables all interrupts except interprocessor inter-

rupts. When a process frees the lock, it signals all waiting processors through an interprocessor interrupt. This mechanism prevents the excessive interconnection traffic caused by busy waiting but still consumes processor cycles. Spin-locks and suspend-locks can be based on primitives similar to Test&Set, such as Compare&Swap.

The Compare&Swap(r1,r2,*w*) primitive is a synchronization primitive in the IBM 370 architecture; r1 and r2 are two machine registers, and *w* points to a memory location. The success of the Compare&Swap is indicated by the flag *z*. The semantics of the Compare&Swap instruction are

```
COMPARE&SWAP(r1,r2,w)
  { temp ← w; if (temp = r1)
    then {w ← r2; z ← 1;}
    else {r1 ← temp; z ← 0;}
  }
```

Test&Set and Compare&Swap are also called *read-modify-write* (RMW) primitives. A common performance problem associated with these basic synchronization primitives is the complexity of locking protocols. If *N* processes attempt to access a critical section at the same time, the memory system must execute *N* basic lock operations, one after the other, even if at most one process is successful. The NYU Ultracomputer[2] and the RP3 multiprocessor[9] use the Fetch&Add(*x*,*a*) primitive, where *x* is a shared-memory word and *a* is an increment. When a single processor executes the Fetch&Add on *x*, the semantics are

```
FETCH&ADD(x,a)
  { temp ← x; x ← temp + a;
    return temp; }
```

The implementation of the Fetch&Add primitive on the Ultracomputer is such that the complexity of an *N*-way synchronization on the same memory word is independent of *N*. The execution of this primitive is distributed in the interconnection network between the processors and the memory module. If *N* processes attempt to Fetch&Add the same memory word simultaneously, the memory is updated only once, by adding the sum of the *N* increments, and a unique value is returned to each of the *N* processes. The returned values correspond to an arbitrary serialization of the *N* requests. From the processor and memory point of view, the result is similar to a sequential execution of *N* Fetch&Adds, but it is performed in one operation. Consequently, the Fetch&Add primitive is extremely effective in accessing sequentially allocated queue structures and in the forking of processes with identical code that operate on different data segments. For example, the following high-level parallel Fortran statement[10] can be executed in parallel by *P* processors if there is no dependency between iterations of the loop:

```
DOALL N=1 to 100
  <code using N>
ENDDO
```

Each processor executes a Fetch&Add on *N* before working on a specific iteration of the loop. Each processor will return a unique value of *N*, which can be used in the code segment. The code for each processor is as follows (*N* is initially loaded with the value 1):

```
n ← FETCH&ADD (N,1)
while (n ≤ 100) do
  { <code using N>
    n ← FETCH&ADD (N,1);
  }
```

In the HEP (Heterogeneous Element Processor) system, shared-memory words are tagged as *empty* or *full*. Loads of such words succeed only after the word is updated and tagged as full. After a successful Load, the tag can be reset to empty. Similarly, the Store on a full memory word can be prevented until the word has been read and the tag cleared. These mechanisms can be used to synchronize processes, since a process can be made to wait on an empty memory word until some other process fills it. This system also relies on busy waiting, and memory cycles are wasted on each trial. Each processor in the HEP is a multistream pipeline, and several process contexts are present in each processor at any time. A different process can immediately be activated when an attempt to synchronize fails. Very few processor cycles are wasted on synchronization. However, the burden of managing the tags is left to the programmer or the compiler. A more complex tagging scheme is advocated for the Cedar machine.[3]

Software-level synchronization mechanisms. Two approaches to synchronization are popular in multiprocessor operating systems: semaphores and message passing. We will discuss message passing in the next section. Operations on semaphores are *P* and *V*. A binary semaphore has the values 0 or 1, which signal acquisition and blocking, respectively. A counting semaphore can take any integer value greater than or equal to 0. The semantics of the *P* and *V* operations are

```
P(s)
  { if (s > 0) then
    s ← (s − 1);
        else
    { Block the process and append it
      to the waiting list for s;
      Resume the highest priority pro-
      cess in the READY LIST;}
  }

V(s)
  { if (waiting list for s empty) then
    s ← (s + 1);
        else
    { Remove the highest priority pro-
      cess blocked for s;
      Append it to the READY LIST;}
  }
```

In these two algorithms shared lists are consulted and modified (namely, the Ready List* and the waiting list for *s*). These accesses as well as the test and modification of *s* have to be protected by spin-locks, suspend-locks, or Fetch&Adds associated with semaphore *s* and with the lists. In practice, *P* and *V* are processor instructions or microcoded routines, or they are operating system calls to the process manager. The process manager is the part of the system kernel controlling process creation, activation, and deletion, as well as management of the locks. Because the process manager can be called from different processors at the same time, its associated data structures must be protected. Semaphores are particularly well adapted for synchronization. Unlike spin-locks and suspend-locks, semaphores are not wasteful of processor cycles while a process is waiting, but their invocations require more overhead. Note that locks are still necessary to implement semaphores.

Another synchronization primitive implemented in software or microcode is Barrier, used to "join" a number of parallel processes. All processes synchronizing at a barrier must reach the barrier before any one of them can continue. Barriers can be defined as follows after the task counter Count has been initialized to zero:

```
BARRIER(N)
  { count := count + 1;
    if (count ≥ N) then
    { Resume all processes on barrier
      queue;
```

*The Ready List is a data structure containing the descriptors of processes that are runable.

Table 1. Synchronization, communication, and coherence in various multiprocessors.

Multiprocessor	Number of processors	CPU architecture	Hardware primitives	Cache	Coherence scheme
IBM 3081	≤4	IBM 370	Compare&Swap (CS, CDS), Test&Set (TS)	Write-back	Central table
Synapse N + 1*	≤32	Motorola 68000	Compare&Swap (CAS), Test&Set (TAS)	Write-back	Distributed table/ bus watching
Denelcor HEP*	100s	Custom	Full/empty bit	No cache	
IBM RP3†	100s	IBM ROMP	Fetch&Op (e.g., Fetch&Add)	Write-back	No shared writable data in cache
NYU Ultracomputer†	100s		Fetch&Add	Write-back	No shared writable data in cache
Encore Multimax	≤20	National Semiconductor 32032	Test&Set ("interlocked" instructions)	Write-through (two processors share each cache)	Bus watching
Sequent Balance 8000	≤12	National Semiconductor 32032	Test&Set (spin-lock using lock cache and bus watching)	Write-through	Bus watching

*Commercial machines no longer in production.
†Experimental prototype.

```
    Reset count; }
      else
    Block task and place in barrier
    queue;
}
```

The first N-1 tasks to execute Barrier would be blocked. Upon execution of Barrier by the Nth task, all N tasks are ready to resume. In the HEP each task that is blocked spin-locks on a Full/Empty bit. The Nth task that crosses the barrier writes into the tagged memory location and thereby wakes up all the blocked tasks. This technique is very efficient for executing parallel, iterative algorithms common in numerical applications.

Interprocess communication. In a shared-memory multiprocessor, interprocess communication can be as simple as one processor writing to a particular memory location and another processor reading that memory location. However, since these activities occur asynchronously, communication is in most cases implemented by synchronization mechanisms. The reading process must be informed at what time the message to be read is valid, and the writing process must know at what time it is allowed to write to a particular memory location without destroying a message yet to be read by another process. Therefore, communication is often implemented by mutually exclusive accesses to mailboxes. Mailboxes are configured and maintained in shared memory by software or microcode.

Message-based communication can be synchronous or asynchronous. In a synchronous system the sender transmits a message to a receiving process and waits until the receiving process responds with an acknowledgment that the message has been received. Symmetrically, the receiver waits for a message and then sends an acknowledgment. The sender resumes execution only when it is confirmed that the message has been received. In asynchronous systems the sending process does not wait for the receiving process to receive the message. If the receiver is not ready to receive the message at its time of arrival, the message may be buffered or simply lost. Buffering can be provided in hardware or, more appropriately, in mailboxes in shared memory.

A summary of synchronization and communication primitives for different processors is given in Table 1.

Coherence in multiprocessors

Coherence problems exist at various levels of multiprocessors. Inconsistencies (i.e., contradictory information) can occur between adjacent levels or within the same level of a memory hierarchy. For example, in a cache-based system with write-back caches, cache and main memory may contain inconsistent copies of data.[4] Multiple caches conceivably could possess different copies of the same memory block because one of the processors has modified its copy. Generally, this condition is not allowable.

In some cases data inconsistencies do not affect the correct execution of a program (for example, inconsistencies between memory and write-back caches may be tolerated). In the following paragraphs we identify the cases for which data

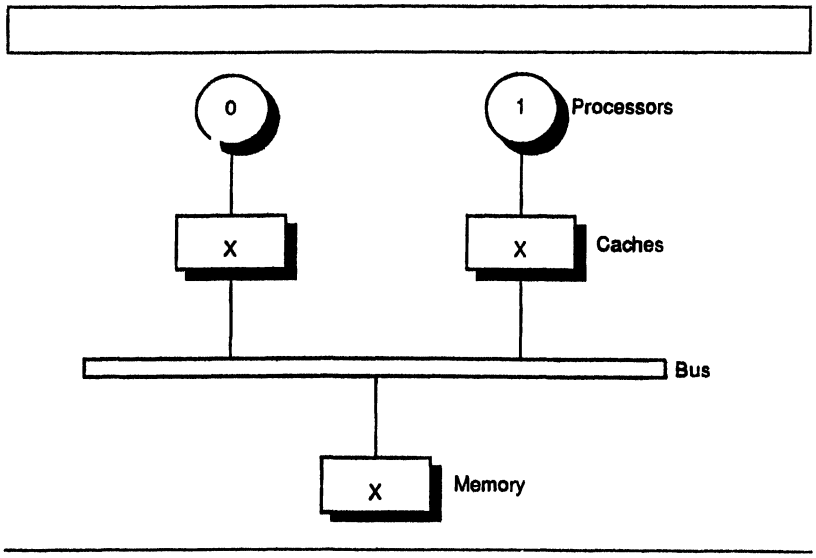


Figure 3. Cache configuration after a Load on X by processors 0 and 1. Copies in both caches are consistent.

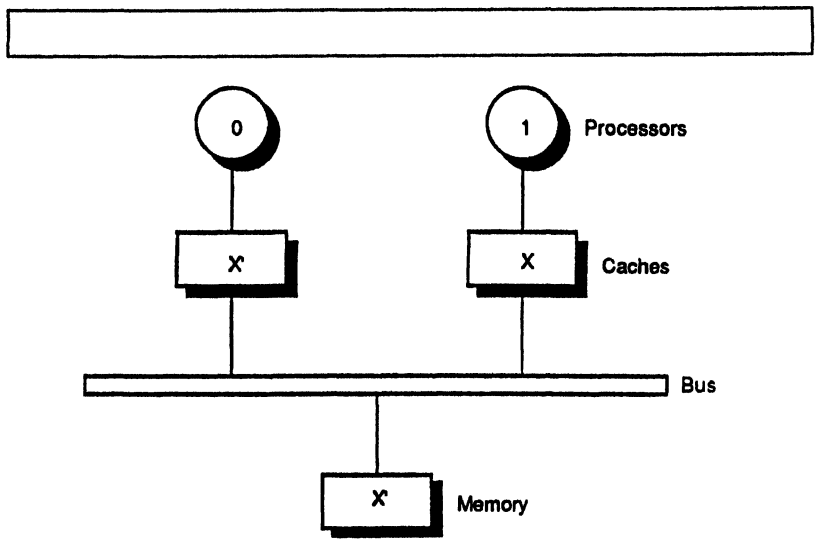


Figure 4. Cache configuration after a Store on X by processor 0 (write-through cache). The copies are inconsistent.

inconsistencies pose a problem and discuss various solutions.

Conditions for coherence. Data coherence problems do not exist in multiprocessors that maintain only a single copy of data. For example, consider a shared-memory multiprocessor in which each CPU does not have a private memory or cache (Figure 1 without optional caches). If Loads, Stores, and RMW cycles are atomic, then data elements are accessed and modified in indivisible operations. Each access to an element applies to the latest copy. Simultaneous accesses to the same element of data are serialized by the hardware.

Cache coherence problems exist in multiprocessors with private caches (Figure 1 with optional caches) and are caused by three factors: sharing of writable data, process migration, and I/O activity. To illustrate the effects of these three factors, we use a two-processor architecture with private caches (Figures 3-5). We assume that an element X is referenced by the CPUs. Let $L_j(X)$ and $S_j(X)$ denote a Load and a Store by processor j for element X in memory, respectively. If the caches do not contain copies of X initially, a Load of X by the two CPUs results in consistent copies of X, as shown in Figure 3. Next, if one of the processors performs a Store to X, then the copies of X in the caches become inconsistent. A Load by the other processor will not return the latest value. Depending on the memory update policy used in the cache, the cache level may also be inconsistent with respect to main memory. A write-through policy maintains consistency between main memory and cache. However, a write-back policy does not maintain such consistency at the time of the Store; memory is updated eventually when the modified data in the cache are replaced or invalidated. Figures 4 and 5 depict the states of the caches and memory for write-through and write-back policies, respectively.

Consistency problems also occur because of the I/O configuration in a system with caches. In Figure 6 the I/O processor (IOP) is attached to the bus, as is most commonly done. If the current state of the system is reached by an $L_0(X)$ and $S_0(X)$ sequence, a modified copy of X in cache 0 and main memory will not have been updated in the case of write-back caches. A subsequent I/O Load of X by the IOP returns a "stale" value of X as contained in memory. To solve the consistency problem in this configuration, the I/O processor must participate in the cache coherence protocol on the bus. The configuration in Figure 7 shows the IOPs sharing the caches with the CPUs. In this case I/O consistency is maintained if cache-to-cache consistency is also maintained; an obvious disadvantage of this scheme is the likely increase of cache perturbations and poor locality of I/O data, which will result in high miss ratios.

Some systems allow processes to migrate—i.e., to be scheduled in different processors during their lifetime—in order to balance the work load among the

processors. If this feature is used in con junction with private caches, data incon sistencies can result. For example, process A, which runs on CPU_0, may alter data contained in its cache by executing $S_0(X)$ before it is suspended. If process A migrates to CPU_1 before memory has been updated with the most recent value of X, process A may subsequently Load the stale value of X contained in memory.

It is obvious that a mere write-through policy will not maintain consistency in the system, since the write does not automatically update the possible copies of the data contained in the other caches. In fact, write-through is neither necessary nor sufficient for coherence.

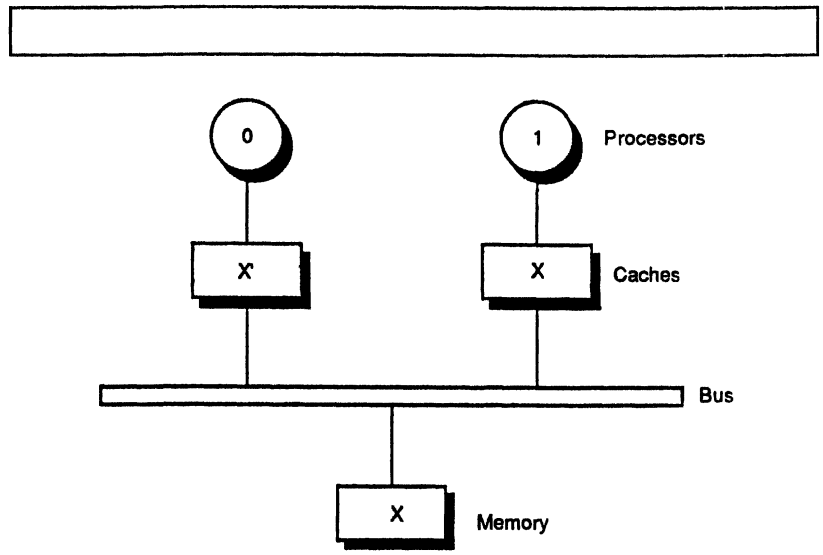


Figure 5. Same as Figure 4 but with write-back cache. The copies are inconsistent.

Solutions to the cache coherence problem. Approaches to maintaining coherence in multiprocessors range from simple architectural principles that make incoherence impossible to complex memory coherence schemes that maintain coherence "on the fly" only when necessary. Here we list these approaches from least to most complex:

(1) A simple architectural technique is to disallow private caches and have only shared caches that are associated with the main memory modules. Every data access is made to the shared cache. A network interconnects the processors to the shared cache modules.

(2) For performance considerations it is desirable to attach a private cache to each CPU. Data inconsistency can be prevented by not caching shared writable data; such data are called *noncachable*. Examples of shared writable data are locks, shared data structures such as process queues, and any other data protected by critical sections. Instructions and other data can be copied into caches as usual. Such items are referred to as *cachable*. The compiler must tag data as either cachable or noncachable. The hardware must adhere to the meaning of the tags. This technique, apparently

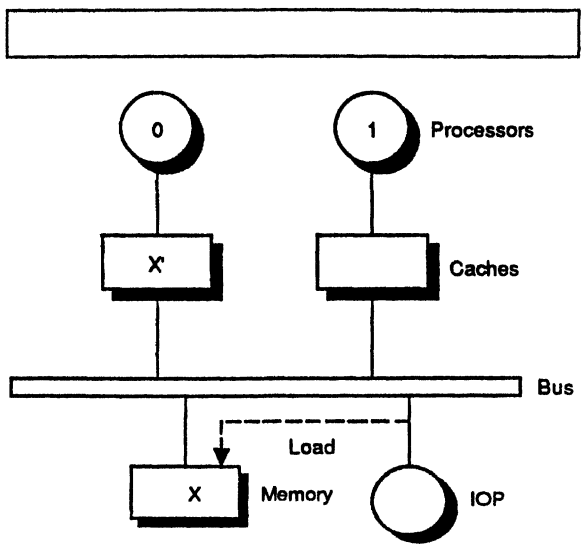


Figure 6. IOPs are attached to the bus and bypass the cache.

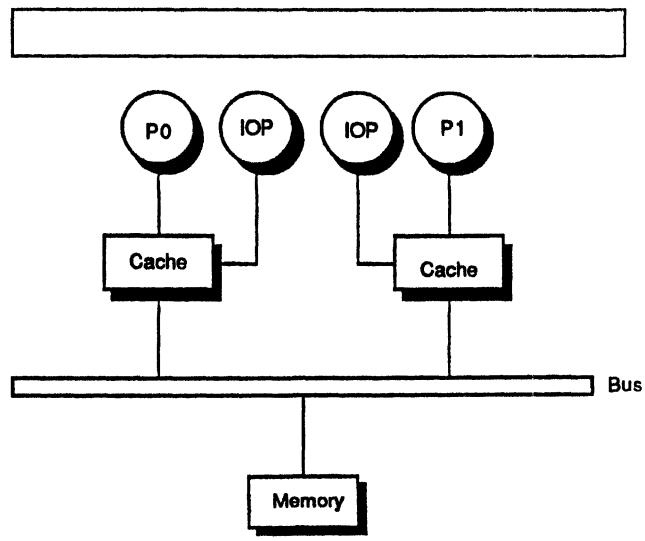


Figure 7. IOPs are attached to the caches.

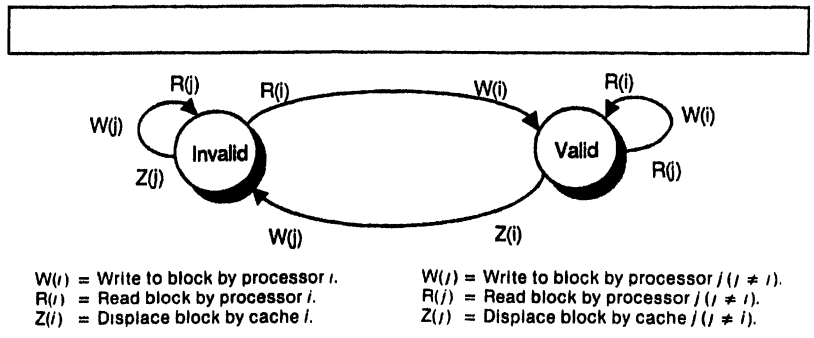


Figure 8. State diagram for a given block in cache *i* for a write-through coherence protocol.

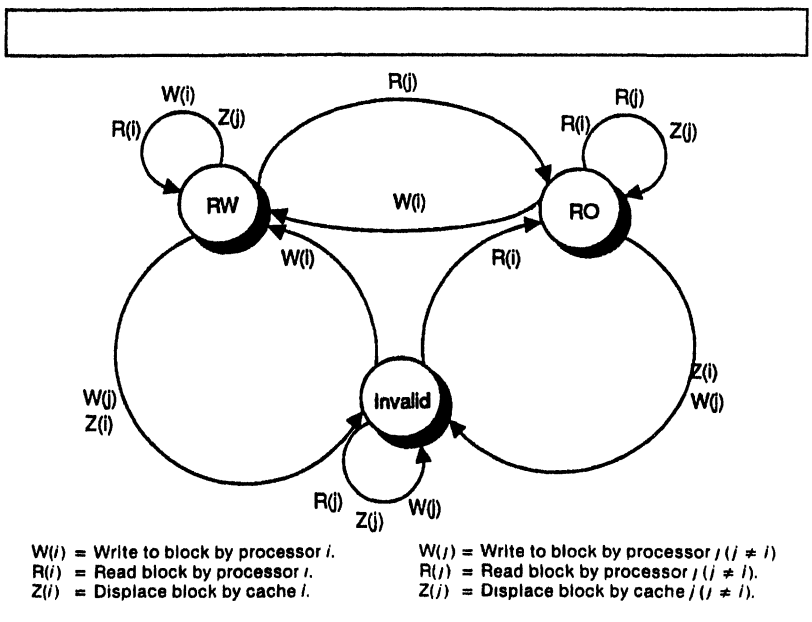


Figure 9. State diagram for a given block in cache *i* for a write-back coherence protocol.

simple in principle, must rely on the detection within each CPU that a block is cachable or not. Such a detection can be made in a virtual memory environment by tagging each page. The tag is stored in entries in the CPU's translation buffers. Translation buffers (TBs) are similar to caches, but they store virtual-to-physical address translations.

(3) If all shared writable data are declared noncachable, the performance may be degraded appreciably. If accesses to shared writable data always occur in critical sections, then such data can be cached. Only the locks that protect the critical sections must remain noncachable. However, to maintain data consistency, all data modified in the critical section must be invalidated in the cache when the critical section is exited. This operation is often referred to as a *cache flush*. The flushing operation ensures that no stale data remain in the cache at the next access to the critical section. If another cache accesses the data via the acquisition of the lock, consistency is maintained. This scheme is adequate for transaction-processing systems in which a shared record is acquired, updated in a critical section, and subsequently released. It works for write-through caches; for write-back caches, the design is more complex.

(4) A scheme allowing shared writable data to exist in multiple caches employs a centralized global table[5] and is used in many mainframe multiprocessor systems, such as the IBM 308x. The table stores the status of memory blocks so that coherence enforcement signals, called *cache cross-interrogates* (XI), can be generated on the basis of the block status. To maintain consistency, XI signals with the associated block address are propagated to the other caches either to invalidate or to change the state of the copies of the referenced block. An arbitrary number of caches can contain a copy of a block, provided that all the copies are identical. We refer to such a copy as a *read-only copy* (RO). To modify a block present in its cache, the processor must own the block with read and write access. When a block is copied from memory into cache, the block is tagged as exclusive (EX) if the cache is the only cache that has a copy of the block. A block is owned exclusively with read and write (RW) access when it has been modified. Only one processor can own an RW copy of a block at any time. The state IN (invalid) signals that the block has been invalidated.

The centralized table is usually located in the storage control element, which may also incorporate a crossbar switch that connects the CPUs to the main memory. To limit the accesses to the global table, local status flags can be provided in the cache directories for the blocks that reside in the cache. Depending on the status of the local flags and the type of request, the processor is allowed to proceed or is required to consult the global table.

(5) In bus-oriented multiprocessors the table that records the status of each block can be efficiently distributed among processors. The distributed-table scheme takes advantage of the broadcasting capability of the bus. Typically, consistency between the caches is maintained by a bus-watching mechanism, often called a *snoopy cache controller*, which implements a cache coherence protocol on the bus. In a simple scheme for write-through caches, all the snoopy controllers watch the bus for Stores. If a Store is made to a location cached in remote caches, then the copies of the block in remote caches are either invalidated or updated. This scheme also maintains coherence with I/O activity. Figure 8 depicts a state diagram of the block state changes depending on

the access type and the previous state of the block. A similar scheme was applied in the Sequent Balance 8000 multiprocessor, which can be configured with up to 12 processors.

The efficiency of the hardware that maintains coherence on the fly is vital. Recognizing that the Store traffic may contribute to bus congestion in a write-through system, Goodman proposed a scheme called *write-once*, in which the initial Store to a block copy in the cache also updates memory.[11] This Store also invalidates matching entries in remote caches, thereby ensuring that the writing processor has the only cached copy. Furthermore, Stores can be performed in the cache at the cache speed. Subsequent updates of the modified block are made in the cache only. A CPU or IOP Load is serviced by the unit (a cache or the memory) that has the latest copy of the block.

Multiprocessors with write-back caches rely on an ownership protocol. When the memory owns a block, caches can contain only RO copies of the block. Before a block is modified, ownership for exclusive access must first be obtained by a read-private bus transaction, which is broadcast to the caches and memory. If a modified block copy exists in another cache, memory must first be updated, the copy invalidated, and ownership transferred to the requesting cache. Figure 9 diagrams memory block state transitions brought about by processor actions. The first commercial multiprocessor with write-back caches was the Synapse N+1.

Variants of the cache coherence bus protocols have been proposed. One scheme, proposed for the Spur project at the University of California, Berkeley, combines compile-time tagging of shared and private data and the ownership protocol. In another system, the Xerox Dragon multiprocessor, a write is always broadcast to other caches and main memory is updated only on replacement. These bus protocols are described and their performances compared in an article by Archibald and Baer.[12]

Advantages and disadvantages. Although scheme 1 provides coherence while being transparent to the user and the operating system, it does not reduce memory conflicts but only the memory access latency. Shared caches, by necessity, contradict the rule that processors and caches should be as close together as possible. I/O accesses must be serviced via the shared caches to maintain coherence.

Tagging shared writable data fails to alleviate the coherence problem caused by I/O accesses.

There are a number of disadvantages associated with scheme 2, which tags data as cachable or noncachable. The major one is the nontransparency of the multiprocessor architecture to the user or the compiler. The user must declare data elements as shared or nonshared if a concurrent language such as Ada, Modula-2, or Concurrent Pascal is used.[13] Alternatively, a multiprocessing compiler, such as Parafrase,[10] can classify data as shared or nonshared automatically. The efficiency of these approaches depends respectively on the ability of the language to specify data structures (or parts thereof) that are shared and writable and of the compiler to detect the subset of shared writable data. Since in practical implementations a whole page must be declared as cachable or not, internal fragmentation may result, or more data than the shared writable data may become noncachable.

Tagging shared writable data also fails to alleviate the coherence problem caused by I/O accesses. Either caches must be flushed before I/O is allowed to proceed, or all data subject to I/O must be tagged as noncachable as well. Depending on the frequency of I/O operations, both approaches reduce the overall hit rate of the caches and hence the speedup obtained by using caches.

Another common drawback of tagging shared writable data rather than maintaining coherence on the fly is the inefficiency caused by process migration. Caches must be flushed before each migration or process migration must be disallowed at the cost of limiting scheduling flexibility.

Scheme 3—flushing caches only when synchronization variables are accessed—has performance problems. In practice the whole cache has to be flushed, or else the data accessed in a critical section must be tagged in the cache. I/O must also be preceded by cache flushing. Note that the programmer must be aware that coherence is restored only at synchronization points. Scheme 3 appears to be attractive only for small caches.

Scheme 4 solves the problems caused by I/O accesses and process migration. However, a global table that must be accessed by all cache controllers can become a bottleneck, even when XIs are filtered by hardware. But the main problem of this coherence scheme is the distance between the processors and the global table. As processors become faster, the access latency of the table becomes a limiting factor of system performance; in particular, when cache access times are very fast, the time penalty for a miss (*miss penalty*) must be minimized.

By distributing the table among the caches, the last scheme partly solves the problems of table access contention and latency. However, the complexity of the bus interface unit is increased because it has to "watch" the bus. Furthermore, since the scheme relies on a broadcast bus, the number of processors that can be interconnected is limited by the bus bandwidth.

Ping-pong effect. In systems with caches employing scheme 4 or 5, the execution of synchronization primitives, such as atomic read-modify-write memory cycles, can create additional access penalties. If two or more processors are spinning on a lock, RMW cycles that cause the lock variable to bounce repeatedly from one cache to another are generated. This can be aggravated by clustering different locks into a given block of memory. However, if RMW operations are implemented carefully, spin-locks can be efficient.

Let us illustrate the ping-pong problem by an example and discuss techniques for reducing system performance degradations. In this example we will assume the use of the Test&Set(lock) instruction; however, the problem can occur with other primitives. The traditional segment of code executed to acquire access to a critical section via a spin-lock is the following:

```
while (TEST&SET(lock) = 1) do nothing;
  /* spin-lock with RMW cycles */
    <execute critical section>
RESET(lock);
  /* exit critical section */
```

Assume that each processor has a private write-back cache and that three or more processors attempt to access the critical section concurrently. If processor P_0 succeeds in acquiring the lock, the other processors (P_1 and P_2) will spin-lock and cause the modified lock variable to be invalidated in the other processors' caches

for each access to the lock. As a result of the invalidation of the modified lock variable, the block is transferred to the requesting cache—a significant penalty. The modification is a result of the writing in the last part of the RMW memory cycle.

One technique for avoiding the ping-pong effect is to use the following segment of code in place of the while statement in the previous code segment:

```
repeat
  while (LOAD(lock) = 1) do nothing;
    /* spin without modification */
until (TEST&SET(lock) = 0);
```

In this segment of code the lock is first loaded to test its status. If available, a Test&Set is used to attempt acquisition. However, while a processor is attempting to acquire the lock, it "spins" locally in its cache, repeating the execution of a tight loop made of a Load followed by a Test. This spinning causes no invalidation traffic on the bus. On a subsequent release of the lock, the processors contend for the lock, and only one of them will succeed. The ping-pong problem is solved; spin-locks can therefore be implemented efficiently in cache-based systems.

Ping-ponging also occurs for shared writable variables. A typical example is the index N in the Doall loop described earlier, in the section on hardware-level synchronization mechanisms. Unless the implementation of Fetch&Add is carefully designed, accesses to the index N create a "hot spot,"[9] which in a cache-based system results in intense ping-ponging between the caches. The careful implementation of synchronization primitives and the creation of hot spots in cache-based systems are research topics that deserve more attention.

Strong and weak ordering of events

The mapping of an algorithm as conceived and understood by a human programmer into a list of machine instructions that correctly implement that algorithm is a complex process. Once the translation has been accomplished, however, it is relatively easy in the case of a uniprocessor to understand what modifications of the machine code can be made without altering the outcome of the execution. A compiler, for example, can resequence instructions to boost performance, or the processor itself can execute instructions out of order if it is pipelined. This is allowable in uniprocessors, provided that hardware mechanisms (interlocks) exist to check data and control dependencies between instructions to be executed concurrently or out of program order.

Local dependency checking is necessary, but it may not preserve the intended outcome of a concurrent execution.

If a processor is a part of a multiprocessor that executes a concurrent program, then such local dependency checking is still necessary but may not be sufficient to preserve the intended outcome of a concurrent execution. Maintaining correctness and predictability of the execution of concurrent programs is more complex for three reasons:

(1) The order in which instructions belonging to different instruction streams are executed is not fixed in a concurrent program. If no synchronization among instruction streams exists, then a very large number of different instruction interleavings is possible.

(2) If for performance reasons the order of execution of instructions belonging to the same instruction stream is different from the order implied by the program, then an even larger number of instruction interleavings is possible.

(3) If accesses are not atomic (for example, if multiple copies of the same data exist), as is the case in a cache-based system, and if not all copies are updated at the same time, then different processors can individually observe different interleavings during the same execution. In this case the total number of possible execution instantiations of a program becomes still larger.

To illustrate the possible types of interleavings, we examine the following three program segments to be executed concurrently by three processors (initially $A = B = C = 0$, and we assume that a Print statement reads both variables indivisibly during the same cycle):

P1	P2	P3
a: $A \leftarrow 1$	c: $B \leftarrow 1$	e: $C \leftarrow 1$
b: Print BC	d: Print AC	f: Print AB

If the outputs of the processors are concatenated in the order P1, P2, and P3, then the output forms a six-tuple. There are 64 possible output combinations. For example, if processors execute instructions in program order, then the execution interleaving a,b,c,d,e,f is possible and would yield the output 001011. Likewise, the interleaving a,c,e,b,d,f is possible and would yield the output 111111. If processors are allowed to execute instructions out of program order, assuming that no data dependencies exist among reordered instructions, then the interleaving b,d,f,e,a,c is possible and would yield the output 000000. Note that this outcome is not possible if processors execute instructions in program order only.

Of the 720 (6!) possible execution interleavings, 90 preserve the individual program order. We have already pointed out that of the 90 program-order interleavings not all six-tuple combinations can result (i.e., 000000 is not possible). The question remains whether out of the 720 non-program-order interleavings all six-tuple combinations can result. So far we have assumed that the memory system of the example multiprocessor is access atomic; this means that memory updates affect all processors at the same time. In a cache-based system such as depicted in Figure 1, this may not be the case; such a system can be nonatomic if an invalidation does not reach all caches at the same time.

In an atomic system it is easy to show that, indeed, not all six-tuple combinations are possible, even if processors need not adhere to program order. For example, the outcome 011001 implies the following: Processor P1 observes that C has been updated and B has not been updated yet. This implies that P3 must have executed statement e before P2 executed statement c. Processor P2 observes that A has been updated before C has been updated. This implies that P1 must have executed statement a before P3 executed statement e. Processor P3 observes that B has been updated but A has not been updated. This implies that P2 must have executed statement c before P1 executed statement a. Hence, e occurred before c, a occurred before e, and c occurred before a. Since this ordering is plainly impossible, we can conclude that in an atomic system, the outcome 011001 cannot occur.

The above conclusion does not hold true

in a nonatomic multiprocessor. Let us assume that the actual execution interleaving of instructions is *a,c,e,b,d,f*. Let us further assume the following sequence of events: When P1 executes *b*, P1's own copy of *B* has not been updated, but P1's own copy of *C* has been updated. Hence, P1 prints the tuple 01. When P2 executes *d*, P2's own copy of *A* has been updated, but P2's own copy of *C* has not been updated. Hence, P2 prints the tuple 10. When P3 executes *f*, P3's own copy of *A* has not been updated, but P3's own copy of *B* has been updated. Hence, P3 prints the tuple 01. The resulting six-tuple is indeed 011001. Note that all instructions were *executed* in program order, but other processors did not *observe* them in program order.

Processor 1	Processor 2	Processor 3
.	.	.
.	.	.
$S_1(A)$	$L_2(A)$	$L_3(B)$
.	$S_2(B)$	$L_3(A)$
.	.	.
.	.	.

Figure 10. Concurrent program for three processors accessing shared variables.

We might ask ourselves whether a multiprocessor functions incorrectly if it is capable of generating any or all of the above-mentioned six-tuple outputs. This question does not have a definitive answer; rather the answer depends on the expectations of the programmer. A programmer who expects a system to behave in a sequentially consistent manner will perceive the system to behave incorrectly if it allows its processors to execute accesses out of program order. The programmer will likely find that synchronization protocols using shared variables will not function. The difficulty of concurrent programming and parallel architectures stems from the effort to disallow all interleavings that will result in incorrect outcomes while not being overly restrictive.

Systems with atomic accesses. We have shown in an earlier work[14] that a necessary and sufficient condition for a system with atomic memory accesses to be sequentially consistent (the strongest condition for logical behavior) is that memory accesses must be performed in the order intended by the programmer—i.e., in program order. (A Load is considered performed at a point in time when the issuing of a Store from any processor to the same address cannot affect the value returned by the Load. Similarly, a Store on *X* by processor *i* is considered performed at a point in time when an issued Load from any processor to the same address cannot return a value of *X* preceding the Store.) In a system where such a condition holds, we say that storage accesses are *strongly ordered*.

In a system without caches a memory access is performed when it reaches the memory system or at any point in time when the order of all preceding accesses to memory has become fixed. For example, if accesses are queued in a FIFO (first in, first out) buffer at the memory, then an access is performed once it is latched in the buffer. When a private cache is added to each processor, Stores can also be atomic in the case of a bus system because of the simultaneous broadcast capability of the buses; in such systems the invalidations generated by a Store and the Load requests broadcast by a processor are latched simultaneously by all the controllers (including possibly the memory controllers). As soon as each controller has taken the proper action on the invalidation, the access can be considered performed.

Buffering of access requests and invalidations also become possible if the rules governing sequential consistency are carefully observed. With extensive buffering at all levels, and provided that the interconnection and the memory system have sufficient bandwidth, the efficiency of all processors may be very high, even if the memory access latency is large compared to the processor cycle time. Two articles present more detailed discussions of the buffering of accesses and invalidations in cache-based multiprocessors.[7,15]

In a weakly ordered system the condition of strong ordering is relaxed to include accesses to synchronization variables only. Synchronization variables must be hardware-recognizable to enforce the specific conditions of strong ordering on them. Moreover, before a lock access can proceed, all previous accesses to nonsynchronization data must be allowed to "settle." This means that all shared memory accesses made before the lock operation was encountered must be completed before the lock operation can proceed. In systems that synchronize very infrequently, the relaxation of strong ordering to weak ordering of data accesses can result in greater efficiency. For example, if the interconnection network is buffered and packet-switched, the interface between the processor and the network can send global memory requests only one at a time to the memory if strong ordering is to be enforced. The reason for this is that in such a network the access time is variable and unpredictable because of conflicts; in many cases waiting for an acknowledgment from the memory controller is the only way to ensure that global accesses are performed in program order. In the case of weak ordering the interface can send the next global access directly after the current global access has been latched in the first stage of the interconnection network, resulting in better processor efficiency. However, the frequency of lock operations will be higher in a program designed for a weakly ordered system.

Systems with nonatomic accesses. In a multiprocessor system with nonatomic accesses, it has been shown that the previous condition for strong ordering of storage accesses (and sequential consistency) is not sufficient.[14]

Example 1. In a system with a recombining network[2] the network can provide for access short-circuiting, which combines Loads and Stores to the same address within the network, before the Store reaches its destination memory module. For the parallel program in Figure 10—$S_i(X)$ and $L_i(X)$ represent global accesses "Store into location *X* by processor *i*" and "Load from location *X* by processor *i*," respectively—such short-circuiting can result in the following sequence of events:

(1) Processor 1 issues a command to store a value at memory location *A*.

(2) Processor 2 reads the value written by processor 1 "on the fly" before A is updated.

(3) Because of the successful read of A in step 2, processor 2 issues a command to write a value at memory location B.

(4) Processor 3 reads the value written by processor 2; it reflects the updated B.

(5) Processor 3 reads memory location A and an old value for A is returned because the write to A by processor 1 has not propagated to A yet.

Each processor performs instructions in the order specified by the programmer, but sequential consistency is violated. Processor 2 implies that step 1 has been completed by processor 1 when it initiates step 3. In step 4 processor 3 recognizes that implication by successfully reading B. But when processor 3 then reads A, it does not find the implied new value but rather the old value. Consequently, processor 3 observes an effect of step 1 before it is capable of observing step 1 itself.

Example 2. In a cache-based system where memory accesses and invalidations are propagated one by one through a packet-switched (but not recombining) network, the same problem as in the previous example may occur. Initially, all processors have an RO copy of A in their cache.

(1) Processor 1 issues a command to store a value at memory location A. Invalidations are sent to each processor with a copy of A in its cache. (For simplicity we assume that the size of a cache block is one word.)

(2) Processor 2 reads the value of A as updated by processor 1, because the invalidation has reached its cache; processor 1 writes the data back to main memory and forwards a copy to processor 2.

(3) Because of the successful read of A in step 2, processor 2 issues a command to write a value at memory location B, sending invalidations for copies of B.

(4) Processor 3 reads the value written by processor 2; it reflects the updated B, because the associated invalidations have propagated to processor 3.

(5) Processor 3 reads memory location A and an old value for A is returned because the invalidation for A caused by processor 1 has not yet propagated to the third processor's cache.

Again each processor executes all instructions in program order. Furthermore, a processor does not proceed to issue memory accesses before all previous invalidations broadcast by the processor have been acknowledged. Yet the same problem occurs as in the previous example; sequential consistency is violated. This is the case because invalidations are essentially memory accesses. Because invalidations are not atomic, the system is not strongly ordered.

User interface

The discussion in this article shows that the issues of synchronization, communication, coherence, and ordering of events in multiprocessors are intricately related and that design decisions must be based on the environment for which the machine is destined. Coherence depends on synchronization in some coherence protocols because the user has to be aware that synchronization points are the only points in time at which coherence is restored. Strict ordering of events may be enforced all the time (strong ordering) or at synchronization points only (weak ordering).

At the user level most features of the physical (hardware) architecture are not visible. The instruction set of each processor and the virtual memory are the most important system features visible to the programs. Depending on the features of the physical architecture that are visible to the programmer, the task of programming the machine may be more difficult, and it may be more difficult to share the machine among different users.

Nontransparent coherence or ordering schemes. A sophisticated compiler may succeed in efficiently detecting and tagging the shared writable data to avoid the coherence problem. Such a compiler may also be able to make efficient use of synchronization primitives provided at different levels. The compiler may be aware of access ordering on a specific machine and generate code accordingly. It is not clear that compiler technology will improve to a point where efficient code can be generated for these different options.

If a program is written in a high-level concurrent language, the facility to specify shared writable data may not exist in the language, in which case we must still rely on the compiler for detecting the minimum set of data to tag as noncachable. It should be emphasized that perfectly legal programs in concurrent languages that allow the sharing of data, generally will not execute correctly in a system where events are weakly ordered.

User access to synchronization primitives. Programmers of concurrent applications may have in their repertoires different hardware- or software-controlled synchronization primitives. For performance reasons it may be advisable to let basic hardware-level synchronization instructions be directly accessible to users, who know their applications and can tailor the synchronization algorithm to their own needs. The basic drawback of such a policy is the increased possibility of deadlocks, resulting from programming errors or processor failure. Spin-locks and suspend-locks consume processor cycles and bus cycles. Therefore, such locks should never be held for a long time. Ideally, a processor should not be interruptible during the time that it owns a lock; for example, one or several processors may spin forever on a lock if the process that "owns" the lock has to be aborted because of an exception. In a virtual-machine environment the user process does not have any control over the interruptibility of the processor, and thus a process can be preempted while it is owning a lock. This will result in unnecessary, resource-consuming spinning from all other processes attempting to obtain the lock.

A solution to this problem is the task-force scheduling strategy, in which all active processes of a multitask are always scheduled and preempted together. Another solution is the implementation of some kind of time-out on spinning. The drastic solution to all these problems is to involve the operating system in every synchronization or communication, so that it can include these mechanisms in its scheduling policy to maximize performance.

Making a multiprocessor function correctly can be a simple or an extremely difficult task. Basic synchronization mechanisms can be primitive or complex, wasteful of processor cycles or highly efficient. In any case the underlying hardware must support the basic assumptions of the logical model expected by the user. In a strongly ordered system such an assumption usually is that the system behaves in a sequentially consistent manner.

Increased transparency comes at the cost of efficiency and increased hardware complexity. But traditional and significant advantages such as the ability to protect users against themselves and other users, ease of programming, portability of programs, and efficient management of

shared resources by multiple users are strong arguments for the designers of general-purpose computers to accept the hardware complexity and the negative effect on performance. The designers of general-purpose machines will probably prefer coherence enforcement on the fly in hardware, strong ordering of memory accesses, and restricted access to synchronization primitives by the user.

On the other hand, for machines with limited access by sophisticated users, such as supercomputers and experimental multiprocessor systems, the performance of each individual task may be of prime importance, and the increased cost of transparency may not be justified.

The challenge of the future lies in the ability to control interprocess communication and synchronization in systems without rigid structures. Efficient multiprocessing will be provided by systems in which synchronization, coherence, and logical ordering of events are carefully analyzed and blended together harmoniously in the context of efficient hardware implementations. It is necessary, however, to provide the programmer with a simple logical model of concurrency behavior. When multiprocessors do not conform to the concept of a single logical model, but rather must be viewed as a dynamic pool of processing, storage, and connection resources, the control in software over communication and synchronization becomes a truly formidable task. The concepts of strong and weak ordering as defined in this article correspond to two widely accepted models of multiprocessor behavior, and we believe that future designs will conform to one of the two models.□

Acknowledgment

Through many technical discussions, William Collier of IBM Poughkeepsie helped shape the content of this article.

References

1. A.K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems—A Status Report," *Computing Surveys*, June 1980, pp. 121-165.
2. A. Gottlieb et al., "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers*, Feb. 1983, pp. 175-189.
3. D. Gajski and J.-K. Peir, "Essential Issues in Multiprocessor Systems," *Computer*, June 1985, pp. 9-27.
4. A.J. Smith, "Cache Memories," *Computing Surveys*, Sept. 1982, pp.473-530.
5. L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, Dec. 1978, pp.1112-1118.
6. W.W. Collier, "Architectures for Systems of Parallel Processes," IBM Technical Report TR 00.3253, Poughkeepsie, N.Y., Jan. 1984.
7. M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering In Multiprocessors," *Proc. 13th Int'l Symp. Computer Architecture*, June 1986, pp. 434-442.
8. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, Sept. 1979, pp. 690-691.
9. G.F. Pfister et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 Parallel Processing Conf.*, pp. 764-771.
10. D.A. Padua, D.J. Kuck, and D.H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. Computers*, Sept. 1980, pp. 763-776.
11. J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Int'l Symp. Computer Architecture*, June 1983, Stockholm, Sweden, pp. 124-131.
12. J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems*, Nov. 1986, pp. 273-298.
13. G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Mar. 1983, pp. 3-43.
14. M. Dubois and C. Scheurich, "Dependency and Hazard Resolution in Multiprocessors," Univ. of Southern Calif. Technical Report CRI 86-20.
15. C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-Based Multiprocessors," *Proc. 14th Int'l Symp. Computer Architecture*, June 1987, pp. 234-243.

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

DAVID J. LILJA

Department of Electrical Engineering, University of Minnesota, Minneapolis, Minnesota 55455

Due to data spreading among processors and due to the cache coherence problem, private data caches have not been as effective in reducing the average memory delay in multiprocessors as in uniprocessors. 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, 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 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. Additionally, *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 and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache Memories*; C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures: *MIMD*; C.5.1 [**Computer System Implementation**]: Large and Medium Computers

Additional Key Words and Phrases: Adaptive, block size, cache coherence, comparison, consistency, directory, memory disambiguation, shared memory, tagged directory, version control

1. INTRODUCTION

The sequence of memory addresses generated by a program typically exhibits the properties of *temporal* and *spatial* locality [Smith 1982]. 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, some programs frequently operate on large data structures in which the consecutive elements of the structure are located in sequential memory locations. Thus, the

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.

CONTENTS

1. INTRODUCTION
   1.1 Problem Definition
   1.2 Overview of Cache Coherence Mechanisms
   1.3 Factors Affecting Coherence Mechanisms
2. COST AND PERFORMANCE MODELING
   2.1 Trace-Driven Simulation
   2.2 Machine Model
   2.3 Test Programs
   2.4 Performance Metrics
3. PERFORMANCE IMPACTS
   3.1 Coherence Detection Strategy
   3.2 Coherence Enforcement Strategy
   3.3 Precision of Block-Sharing Information
   3.4 Cache Block Size
4. HYBRID TECHNIQUES
   4.1 Compiler Assistance for Reducing the Directory Size
   4.2 Combining Multiple Coherence Mechanisms
   4.3 Compiler-Plus-Directory Coherence Mechanism
   4.4 Extending the Memory Hierarchy into the Network
5. CONCLUSIONS
   5.1 Coherence Detection Strategy
   5.2 Coherence Enforcement Strategy
   5.3 Precision of Block-Sharing Information
   5.4 Cache Block Size
   5.5 Summary
ACKNOWLEDGMENTS
REFERENCES

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 [Gottlieb et al. 1982; Pfister et al. 1985]. Caches have not provided the same level of memory performance improvement in multiprocessors as in uniprocessors, however, since the data referenced by a program in a multiprocessor is distributed among the processors. This data spreading reduces the processors' locality of reference, thus reducing the effectiveness of the caches. Additionally, 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.

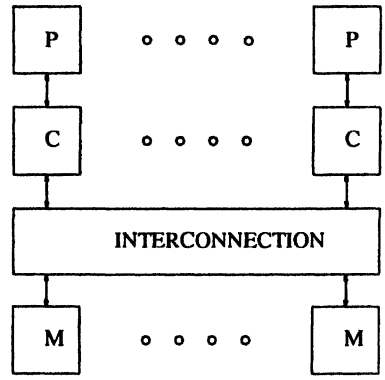


Figure 1. Shared-memory multiprocessor architecture.

This paper examines mechanisms for maintaining cache coherence in large-scale shared-memory multiprocessors, such as the New York University Ultracomputer [Gottlieb et al. 1982], the University of Illinois Cedar [Kuck et al. 1986], the IBM RP3 [Pfister et al. 1985], the Alliant FX-series [Perron and Mundie 1986], and the Stanford DASH [Lenoski et al. 1992]. 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. Additionally, a new frame-

work 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 to execute 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 next, is the model of the memory system presented to the programmer. The second important aspect, and the primary focus of the remainder of this survey, 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" [Censier and Feautrier 1978]. 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 execution sequence of memory operations allowed within a processor and among 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" [Lamport 1979]. With this consistency model, each access to the shared memory must complete before the next shared-memory access can begin. Also, 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 [Dubois et al. 1988] 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 [Gharachorloo et al. 1990] 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 than 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 [Gharachorloo et al. 1991; Gupta et al. 1991; Torrellas and Hennessy 1990; Zucker and Baer 1992].

1.1.2 Cache Coherence

A related problem to the memory consistency model, and the primary focus of this survey, 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 uses a stale version.

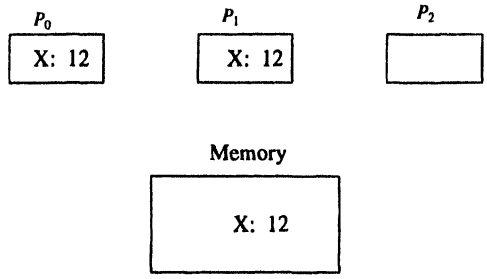
For example, consider a system with three processors, each with a private data cache, in which the sequence of reads and writes shown in Figure 2(a) are performed. After the first two reads have been completed at t_2, the caches of both processors P_0 and P_1 will contain the value of 12 for the variable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

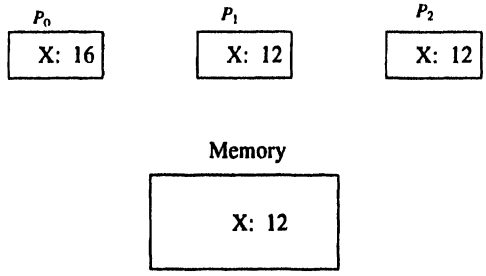
A useful way of viewing the relationship between the memory consistency model and the cache coherence mechanism is

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.

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* [Dubois et al. 1988]. 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 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 next.

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 three 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 among the processors and the memory modules, such as a common bus, that allows each processor to monitor all 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 [Goodman 1983; Katz et al.

1985; Papamarcos and Patel 1984] its cached copy so that its next reference to the block will force a cache miss, and thus the current value will be obtained from memory, or from another cache. Alternatively, it can directly update [McCreight 1984; Thacker et al. 1988] 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 [Goodman and Woest 1988] 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 [Archibald 1988; Wilson 1987] 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 [Bhuyan et al. 1989; Marquardt and Al Khatib 1989] to remove the coherence-updating data 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 survey 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 [Tang 1976; Yen et al. 1985], 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 acknowledgments 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 infor-

mation 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 acknowledgments 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 acknowledgments from the directory for all of the writes it 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 [Veidenbaum 1986] 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 [Polychronopoulos 1988] in which there are no dependences among 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 [Brantley et al. 1985] and the Ultracomputer [Edler et al. 1985]. These simple approaches tend to invalidate more cache entries than are necessary to maintain coherence, and thus they 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 runtime, 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 tradeoff the implementation cost and the performance of the coherence mechanism.
(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. Additionally, 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 survey 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 [Stunkel et al. 1991]. 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 that has many of the advantages of hardware monitoring to generate traces 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. Also, 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 hardwired 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 survey, 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 [Perron and Mundie 1986] 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. Additionally, 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 KB is used in each of the 32 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 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* [Pfister et al. 1985], which may make it undesirable to cache synchronization variables [Dubois et al. 1988]. Special hardware or software can be added to the system to improve access to synchronization variables [Anderson 1990; Goodman et al. 1989; Kruskal et al. 1986], but the analysis of these techniques is beyond the scope of this survey.

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 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 [Censier and Feautrier 1978]. This directory structure is described more fully in Section 3.3.1.

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% of the unique blocks referenced by *arc3d*,

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

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

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% 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 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 survey 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 survey 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 trade-offs 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 survey to compare the different design factors that comprise a cache coherence mechanism. This author feels that examining these two metrics directly provides greater insight into the trade-offs in the coherence mechanisms than by 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 are also 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 by 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 survey 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 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 [Cheong 1988b].

For static coherence detection, a data dependence test [Banerjee 1988; Lichnewsky 1988; Li 1990] 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 [Cheong and Veidenbaum 1988a; 1989]. 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 [Veidenbaum 1986; Brantley et al. 1985, Edler et al. 1985] are more conservative than is necessary to ensure coherence in that they invalidate cache entries that are not actually stale. This overinvalidation 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 [Cheong and Veidenbaum 1988a] 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 [Cheong and Veidenbaum 1989] or timestamps [Min and Baer 1989] to determine whether or not a cache entry is up to date when it is referenced. In the version control mechanism [Cheong and Veidenbaum 1989], 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 (Censier and Feautrier 1978]. (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 [Censier and Feautrier 1978] is compared to the compiler-directed version control coherence mechanism [Cheong and Veidenbaum 1989]. 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.

Because of the imprecise nature of

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

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

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

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

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 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. Additionally, 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 [Adve et al. 1991; Lilja 1991; Min and Baer 1990].

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 *noncacheable* so that there can never by multiple copies [Lilja et al. 1989]. 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 [McCreight 1984; Thacker et al. 1988]. In the system used in this survey, 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 acknowledgment 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 invalidating. This table separates the network traffic into that required to move the data into a cache on a miss, and into that required to maintain coherence, which is either the update traffic or the invalidate

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

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 2 to 25 times greater for updating that 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 depends 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 [Eggers and Katz 1989b; Karline et al. 1986]. 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 [Karline et al. 1986] 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 [Archibald 1988] 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 [Bennett et al. 1990] implements a coherence mechanism that uses the compiler to categorize each object referenced by the program into a coherence type, and it 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 processors' 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-Toussi 1993] 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. Additionally, 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 [Censier and Feautrier 1978], 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 [Agarwal et al. 1988], 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 current models [Lilja 1991] 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 [Censier and Feautrier 1978], 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 [Archibald and Baer 1984] 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. Additionally, 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 [Agarwal et al. 1988] 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. Additionally, 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 [James et al. 1990] 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 requires $\log_2 p$ bits to point to a processor, plus an extra bit to point back to memory. Additionally, 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 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. Additionally, several other possible tagged-directory structures are described below.

The *pointer cache* tagged directory [Lilja 1991] 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'Krafka and Newton 1990] 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 [Gupta et al. 1990] 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 the p/g clusters, where there are g processors per cluster. That is, when the mode bit is set, the ith bit of the v pointer bits will be turned on to indicate that at least one of the processors in the ith 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 $r_{cc}p[\log_2 m + max(p/g, \log_2 p) + 3] + 2cp$, where r_{cc} 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 [Chaiken et al. 1991], 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 survey. 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 [Lilja 1991]. 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 associate 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

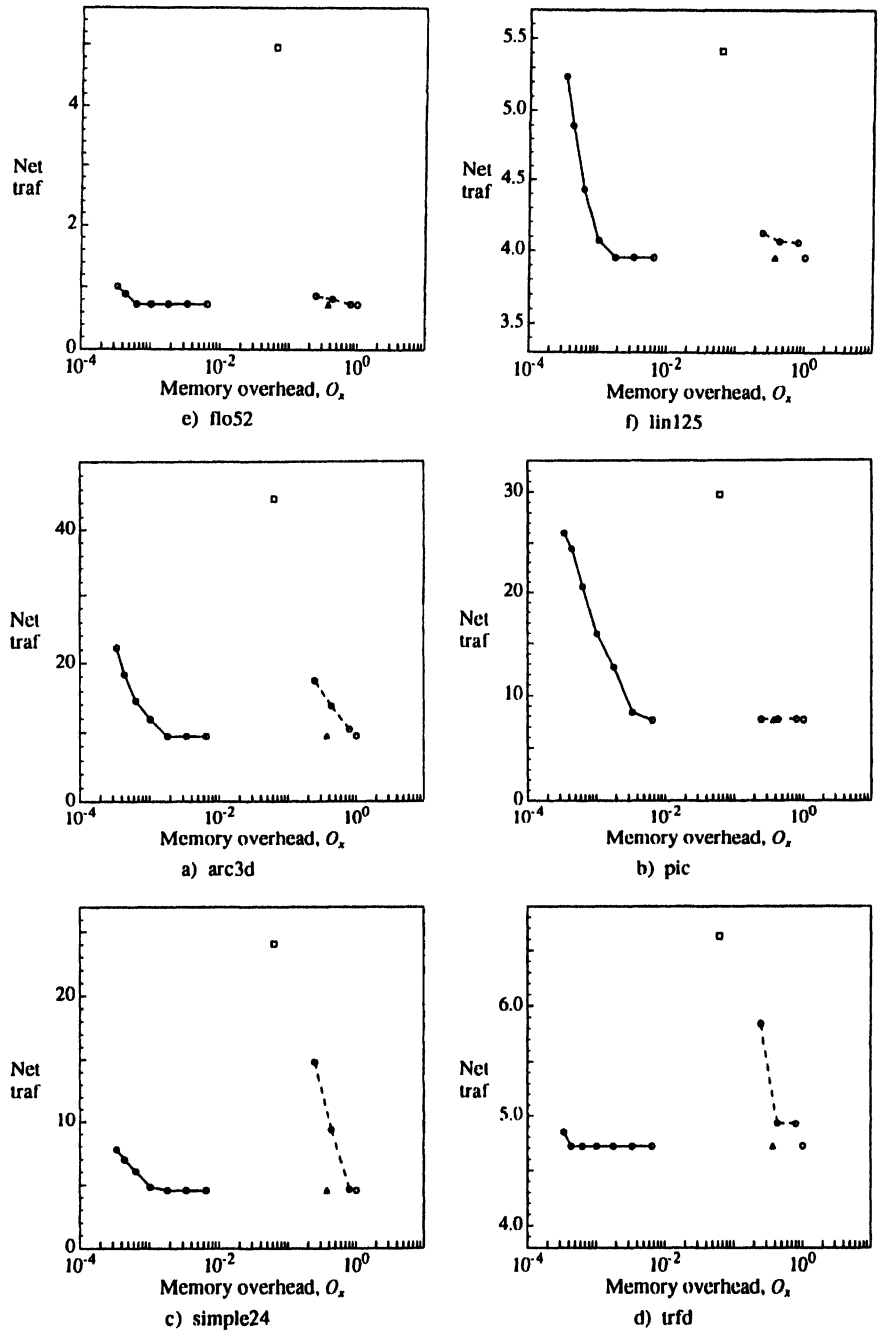
Table 8. Normalized Memory Overhead for the Directory Mechanisms

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	$\frac{s[\log_2(kc)+\log_2 p+3]+2}{bw(k+1)}$

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 64 KB (2^{16}) words to 256 KB (2^{18}) words, and a typical memory module may contain from 2 MB (2^{21}) words to 16 MB (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 $c = 8$ KB 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 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 $s = r/c = 1/32$ 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



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.

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 [Agarwal et al. 1988; Eggers and Katz 1988; Weber and Gupta 1989]), $n = 4$ 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 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 is 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 [Przybylski et al. 1988; Smith 1987].

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 worse 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 [Agarwal and Gupta 1988; Eggers and Katz 1989a; Goodman 1983; Lee et al. 1987]. 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 containsonly 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 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

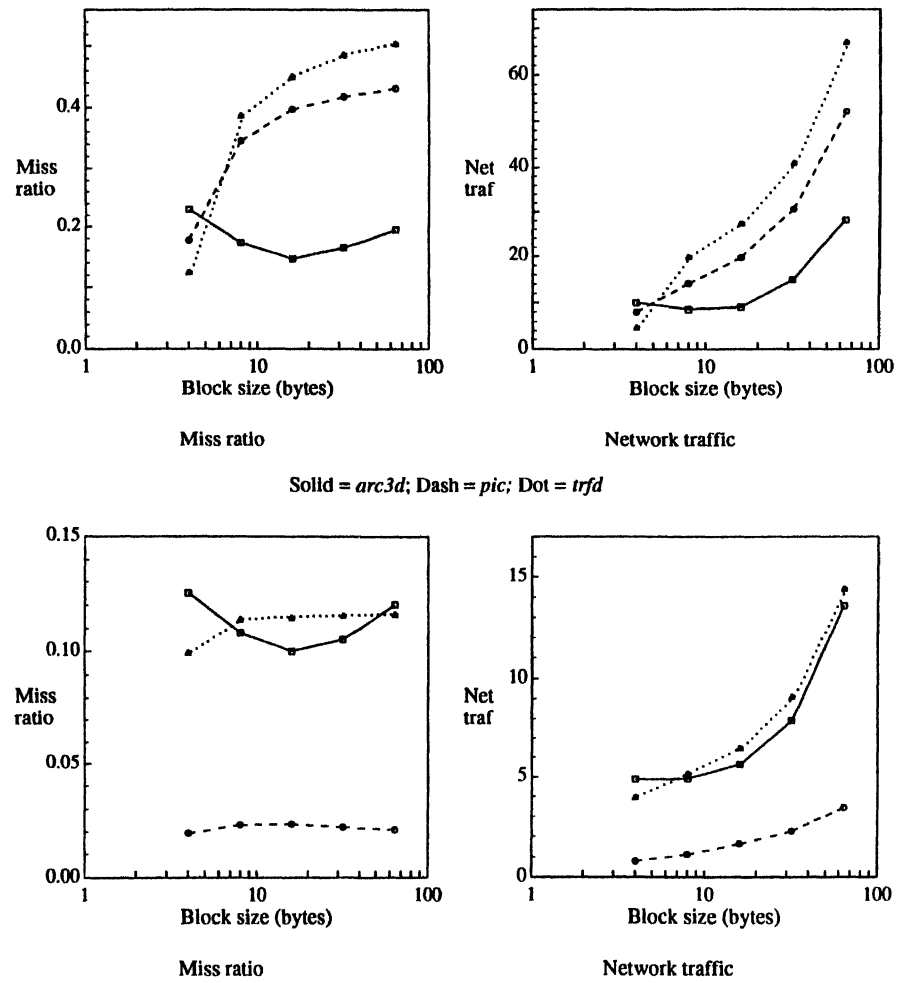


Figure 4. Effect of cache block size on miss ratio and network traffic (bytes/reference).

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 [Lilja 1992].)

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 into the component required to

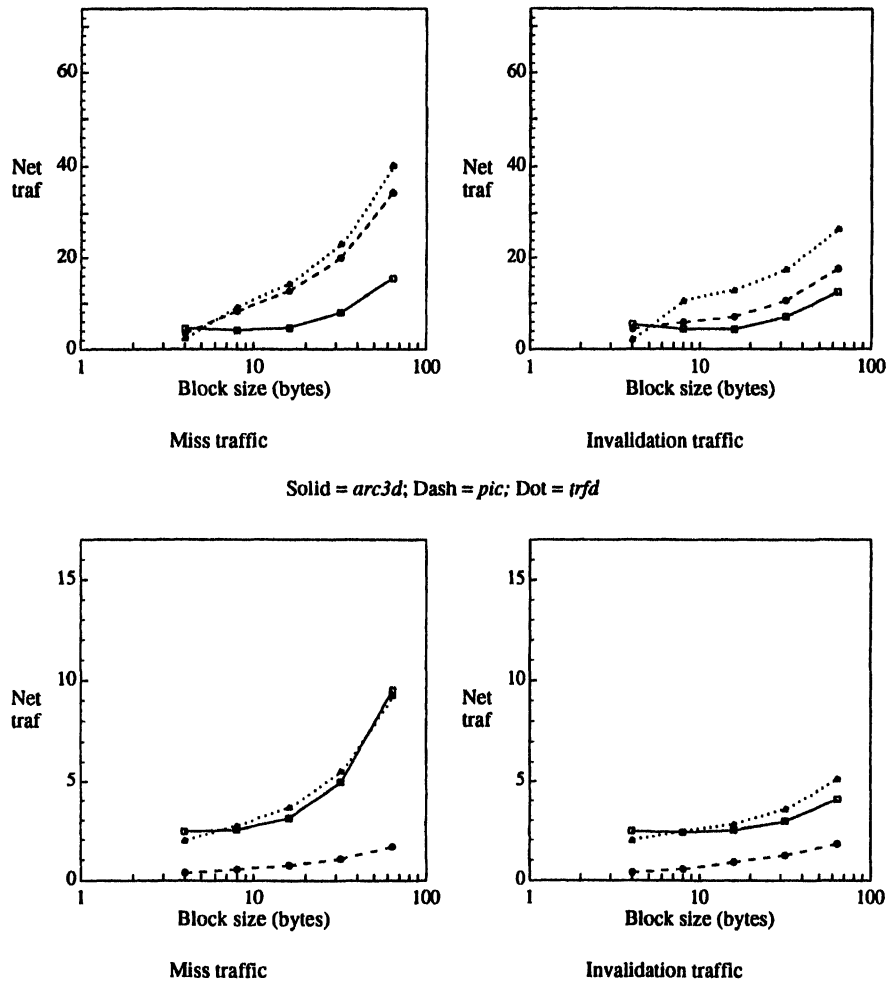


Figure 5. Components of total network traffic due to cache misses and due to invalidations from data sharing.

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 is 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 [Agarwal and Gupta 1988; Eggers and Katz 1988; Lilja et al. 1989; Weber and Gupta 1989] have shown that a substantial fraction of all blocks referenced by a program may be private or read-only, and thus they 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 [Lilja and Yew 1991]. More complex compile-time analysis techniques can mark each individual memory reference as needing a pointer allocated or not [Nguyen et al. 1993]. This more precise marking can reduce the time a pointer is needed for a specific shared-memory 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 [Lenoski et al. 1990; 1992] 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 [Papamarcos and Patel 1984]. 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, 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 [Cheong and Veidenbaum 1989] and timestamp [Min and Baer 1989] 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 [Chen and Veidenbaum 1991] 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 (Mizrahi et al. 1989] 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. Additionally, 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 survey has identified several important architectural issues that affect the performance and im-

plementation 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 the coherence detection strategy, the coherence enforcement strategy, the precision of block-sharing information, and the cache block size.

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. Additionally, 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. Additionally, 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 produce 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.

ACKNOWLEDGMENTS

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 survey. Their insightful comments and suggestions helped to significantly improve the focus and clarity.

REFERENCES

ADVE, S. V., ADVE, V. S., HILL, M. D., AND VERNON, M. K. 1991. Comparison of hardware and software cache coherence schemes. In the *International Symposium on Computer Architecture*, 298–308.

AGARWAL, A., AND GUPTA, A. 1988. Memory-reference characteristics of multiprocessor applications under MACH. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM, New York, 215–225.

AGARWAL, A., SIMONI, R., HENNESSY, J., AND HOROWITZ, M. 1988. An evaluation of directory schemes for cache coherence. In the *International Symposium on Computer Architecture*, 280–289.

ANDERSON, T. E. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parall. Distrib. Syst. 1*, 1 (Jan.), 6–16.

ARCHIBALD, J. K. 1988. A cache coherence approach for large multiprocessor systems. In the *ACM International Conference on Supercomputing*. ACM, New York, 337–345.

ARCHIBALD, J., AND BAER, J.-L. 1984. An economical solution to the cache coherence problem. In the *International Symposium on Computer Architecture*, 355–362.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, Mass.

BENNETT, J. K., CARTER, J. B., AND ZWAENEPOEL, W. 1990. Adaptive software cache management for distributed shared memory architectures. In the *International Symposium on Computer Architecture*, 125–134.

BHUYAN, L. N., LIU, B.-C., AND AHMED, I. 1989. Analysis of MIN-based multiprocessors with private cache memories. In the *International Conference on Parallel Processing*. Vol. 1, *Architecture*, 51–58.

BRANTLEY, W. C., MCAULIFFE, K. P., AND WEISS, J. 1985. RP3 processor-memory element. In the

International Conference on Parallel Processing, 782–789.

CENSIER, L. M., AND FEAUTRIER, P. 1978. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput. C-27*, (Dec.), 1112–1118.

CHAIKEN, D., KUBIATOWICZ, J., AND AGARWAL, A. 1991. LimitLESS Directories: A scalable cache coherence scheme. In the *International Conference on Architectural Support for Programming Languages and Operating Systems*, 224–234.

CHEN, Y.-C., AND VEIDENBAUM, A. V. 1991. A software coherence scheme with the assistance of directories. In the *ACM International Conference on Supercomputing*. ACM, New York.

CHEONG, H., AND VEIDENBAUM, A. 1989. A version control approach to cache coherence. In the *ACM International Conference on Supercomputing*. ACM, New York, 322–330.

CHEONG, H., AND VEIDENBAUM, A. V. 1988a. A cache coherence scheme with fast selective invalidation. In the *International Symposium on Computer Architecture*, 299–307.

CHEONG, H., AND VEIDENBAUM, A. V. 1988b. Stale data detection and coherence enforcement using flow analysis. In the *International Conference on Parallel Processing*. Vol. I, *Architecture*, 138–145.

DUBOIS, M., SCHEURICH, C., AND BRIGGS, F. A. 1988. Synchronization, coherence, and event ordering in multiprocessors. *Computer 21*, 2 (Feb.), 9–21.

EDLER, J., GOTTLIEB, A., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., SNIR, M., TELLER, P. J., AND WILSON, J. 1985. Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach. In the *International Symposium on Computer Architecture*, 126–135.

EGGERS, S. J., AND KATZ, R. H. 1989a. The effect of sharing on the cache and bus performance of parallel programs. In the *International Conference on Architectural Support for Programming Languages and Operating Systems*, 257–270.

EGGERS, S. J., AND KATZ, R. H. 1989b. Evaluating the performance of four snooping cache coherency protocols. In the *International Symposium on Computer Architecture*, 2–15.

EGGERS, S. J., AND KATZ, R. H. 1988. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In the *International Symposium on Computer Architecture*, 373–382.

GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. 1991. Performance evaluation of memory consistency models for shared-memory multiprocessors. In the *International Conference on Architectural Support for Programming Languages and Operating Systems*, 245–257.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In the *International Symposium on Computer Architecture*, 15–26.

GOODMAN, J. R. 1983. Using cache memory to reduce processor-memory traffic. In the *International Symposium on Computer Architecture*, 124–131.

GOODMAN, J. R., AND WOEST, P. J. 1988. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In the *International Symposium on Computer Architecture*, 422–431.

GOODMAN, J., VERNON, M., AND WOEST, P. 1989. A set of efficient synchronization primitives for a large-scale shared-memory multiprocessor. In the *International Conference on Architectural Support of Programming Languages and Operating Systems*, 64–73.

GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. 1982. The NYU Ultracomputer—Designing a MIMD, shared-memory parallel machine. In the *International Symposium on Computer Architecture*, 27–42.

GUPTA, A., HENNESSY, J., GHARACHORLOO, K., MOWRY, T., AND WEBER, W.-D. 1991. Comparative evaluation of latency reducing and tolerating techniques. In the *International Symposium on Computer Architecture*, 254–263.

GUPTA, A., WEBER, W.-D., AND MOWRY, T. 1990. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In the *International Conference on Parallel Processing* Vol. I, *Architecture*, 312–321.

JAMES, D. V., LAUNDRIE, A. T., GJESSING, S., AND SOHI, G. S. 1990. Scalable coherent interface. *Computer 23*, 6, (June), 74–77.

KARLINE, A. R., MANASS, M. S., RUDOLPH, L., AND SLEATOR, D. D. 1986. Competitive snoopy cacheing. In the *Symposium on Foundations of Computer Science*, 244–254.

KATZ, R., EGGERS, S., WOOD, D. A., PERKINS, C., AND SHELDON, R. G. 1985. Implementing a cache consistency protocol. In the *International Symposium on Computer Architecture*, 276–283.

KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. 1986. Efficient synchronization on multiprocessors with shared memory. In the *ACM Symposium on Principles of Distributed Computing*. ACM, New York, 218–228.

KUCK, D. J., DAVIDSON, E. S., LAWRIE, D. J., AND SAMEH, A. H. 1986. Parallel supercomputing today and the Cedar approach. *Science 231* (Feb. 28), 967–974.

LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28*, 9, (Sept.), 690–691.

LEE, R. L., YEW, P.-C., AND LAWRIE, D. J. 1987. Multiprocessor cache design considerations. In

the *International Symposium on Computer Architecture*, 253–262.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A. HENNESSY, J., HOROWITZ, M., AND LAM, M. S. 1992. The Stanford DASH Multiprocessor. *Computer 25*, 3 (Mar.), 63–79.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. 1990. The directory-based cache coherence protocol for the DASH multiprocessor. In the *International Symposium on Computer Architecture*, 148–159.

LI, Z., YEW, P.-C., AND ZHU, C.-Q. 1990. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. Parall. Distrib. Syst. 1*, 1 (Jan.), 26–34.

LICHNEWSKY, A., AND THOMASSET, F. 1988. Introducing symbolic problem solving techniques in the dependence testing phase of a vectorizer. In the *International Conference on Supercomputing*.

LILJA, D. J. 1992. Prefetching and scheduling interactions in shared memory multiprocessors. In the *Midwest Electrotechnology Conference*, 84–87.

LILJA, D. J. 1991. Processor parallelism considerations and memory latency reduction in shared memory multiprocessors. Rep. No. 1136, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana.

LILJA, D. J., AND YEW, P.-C. 1991. Combining hardware and software cache coherence strategies. In the *ACM International Conference of Supercomputing*. ACM, New York 274–283.

LILJA, D. J., MARCOVITZ, D. M., AND YEW, P.-C. 1989. Memory referencing behavior and a cache performance metric in a shared memory multiprocessor. Rep. No. 836, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana.

MARQUARDT, D. E., AND ALKHATIB, H. S. 1989. C2MP: A cache-coherent, distributed-memory multiprocessor system. In *Proceedings of Supercomputing '89*, 466–475.

MCCREIGHT, E. M. 1984. The Dragon Computer System, an early overview. In the *NATO Advanced Study Institute on Microarchitecture VLSI Computers*, 83–101.

MIN, S. L., AND BAER, J.-L. 1990. A performance comparison of directory-based and timestamp-based cache coherence schemes. In the *International Conference on Parallel Processing*. Vol. I, *Architecture*, 305–311.

MIN. S. L., AND BAER J.-L. 1989. A timestamp-based cache coherence scheme. In the *International Conference on Parallel Processing*. Vol. I, *Architecture*, 23–32.

MIZRAHI, H. E., BAER, J.-L., LAZOWSKA, E. D., AND ZAHORJAN, J. 1989. Extending the memory hierarchy into multiprocessor interconnection networks: A performance analysis. In the *International Conference on Parallel Processing*. Vol. I, *Architecture*, 41–50.

MOUNES-TOUSSI, F. 1993. An adaptive cache coherence enforcement strategy with compiler assistance. M.S. Thesis, Dept. of Electrical Engineering, Univ. of Minnesota, Minneapolis.

NGUYEN, T. N., LI, Z., AND LILJA, D. J. 1993. Efficient use of dynamically tagged directories through compiler analysis. In the *International Conference on Parallel Processing*.

O'KRAFKA, B. W., AND NEWTON, A. R. 1990. An empirical evaluation of two memory-efficient directory methods. In the *International Symposium on Computer Architecture*, 138–147.

PAPAMARCOS, M. S., AND PATEL, J. H. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In the *International Symposium on Computer Architecture*, 348–354.

PERRON, R., AND MUNDIE, C. 1986. The architecture of the Alliant FX/8 Computer. In *IEEE COMPCON*. IEEE, New York, 390–393.

PFISTER, G. F., BRANTLEY, W. C., GEORGE, D. A., HARVEY, S. L., KLEINFELDER, W. J., MCAULIFFE, K. P., MELTON, E. A., NORTON, V. A., AND WEISS, J. 1985. The IBM research parallel processor prototype (RP3): Introduction and Architecture. In the *International Conference on Parallel Processing*, 764–771.

POLYCHRONOPOULOS, C. D. 1988. Toward auto-scheduling compilers. *J. Supercomput.*, 2, 297–330.

PRZYBYLSKI, S., HOROWITZ, M., AND HENNESSY, J. 1988. Performance tradeoffs in cache design. In the *International Symposium on Computer Architecture*, 290–296.

SMITH, A. J. 1987. Line (block) size choice for CPU cache memories. *IEEE Trans. Comput. C-36*, 9 (Sept.), 1063–1075.

SMITH, A. J. 1982. Cache memories. *ACM Comput. Surv. 14*, 3 (Sept.), 473–530.

STUNKEL, C. B., JANSSENS, B., AND FUCHS, W. K. 1991. Address tracing for parallel machines. *Computer 24*, 1 (Jan.), 31–38.

TANG, C. K. 1976. Cache design in the tightly coupled multiprocessor system. In the *AFIPS Conference Proceedings, National Computer Conference*. AFIPS, Arlington, Va., 749–753.

THACKER, C. P., STEWART, L. C., AND SATTERTHWAITE, E. H. 1988. Firefly: A multiprocessor workstation. *IEEE Trans. Comput. 37*, 8 (Aug.), 909–920.

TORRELLAS, J., AND HENNESSY, J. 1990. Estimating the performance advantages of relaxing consistency in a shared-memory multiprocessor. In the *International Conference on Parallel Processing*. Vol. I, *Architecture*, 26–33.

VEIDENBAUM, A. V. 1986. A compiler-assisted cache coherence solution for multiprocessors.

In the *International Conference on Parallel Processing*, 1029–1036.

WEBER, W.-D., AND GUPTA, A. 1989. Analysis of cache invalidation patterns in multiprocessors. In the *International Conference on Architectural Support for Programming Languages and Operating Systems*, 243–256.

WILSON, A. W. 1987. Hierarchical cache/bus architecture for shared memory multiprocessors. In the *International Symposium on Computer Architecture*, 244–252.

YEN, W. C., YEN, D. W. L., AND FU, K.-S. 1985. Data coherence problem in a multicache system. *IEEE Trans. Comput. C-34*, 1 (Jan.), 56–65.

ZUCKER, R. N., AND BAER, J.-L. 1992. A performance study of memory consistency models. In the *International Symposium on Computer Architecture*, 2–12.

Received September 1991; final revision accepted March 1993.

Software Cache Consistency in Shared-Memory Multiprocessors

A Survey of Approaches and Performance Evaluation Studies

Igor Tartalja and Veljko Milutinović
Department of Computer Engineering
School of Electrical Engineering
University of Belgrade
POB 816
11000 Belgrade, Yugoslavia

E-mail:{etartalj, emilutiv}@ubbg.etf.bg.ac.yu

Abstract

This paper represents a comprehensive survey of software solutions for maintenance of cache consistency in shared-memory multiprocessor systems. The lack of widely known, acceptably systematic, and flexible classification in this research field has been our basic motivation for this work. We have proposed here a classification based on a set of ten carefully selected criteria that we considered most relevant. Existing solutions have been described and decomposed on the basis of this classification. Different solutions correspond to various points of an abstract multidimensional criterion-space. Such a generalized approach enables the points corresponding to nonexistent, but potentially useful solutions to be noticed and selected for exploration. Finally, an overview of papers dealing with performance evaluation of software solutions has been given. Different evaluation techniques have been presented and compared, using representative examples.

1. Introduction

Shared-memory multiprocessor systems represent an efficient architectural support for applications characterized by significant data sharing among parallel processes. The efficiency of these systems can be significantly improved if cache memories are used. The presence of cache memories is important for two reasons: first, because of the speed difference between the processor and the main memory; second, because of the access contention for both the interconnection network and the memory modules. A shared cache [YEH83] overcomes the speed

gap, but not the access contention. Private caches enable most of the memory references to be satisfied locally, thereby reducing the contention both on the interconnection network and the memory modules.

One problem inherent to the use of private cache memories is potential inconsistency of the memory system. If several processes on several processors access the shared data, several copies of shared data will exist in private cache memories. If these data are changeable, a change made by one processor renders the other copies out of date. These stale copies should either be invalidated or modified before some other processor uses them. This problem has been studied for at least two decades. Consequently, a variety of possible solutions to the problem have been proposed. Two major approaches are the hardware and the software maintenance of the consistency of private caches.

Hardware approaches make consistency maintenance fully transparent for all levels of software, thereby simplifying the programming model of the multiprocessor system with private cache memories but increasing the hardware complexity of the system. One recent survey of hardware approaches is given in [TOMA94a, TOMA94b]; most of the related papers are classified and reprinted in the IEEE Computer Society Press book {TOMAŠ93}.

Software approaches [WULF72, GOTTL82, PFIST85, BRANT85, EDLER85, SMITH85, CHERI86, VEIDE86, MCAUL86, LEE87b, CHEON88, CYTRO88, CHEON89, MIN89, BENN90b, TARTA92, CHEON92, LOURI92, CHIUE93, DARNE93] lift the transparency of the problem above the operating system level or the compiler level. This complicates the programming model to some extent, but decreases the complexity of the hardware support. Also, software approaches tend to be more efficient than hardware approaches in a class of nonnumeric applications characterized by predominantly migratory data [ADVE91]. This survey of software approaches can be treated as the continuation of the effort that started with the above-mentioned survey of hardware approaches [TOMA94a, TOMA94b].

The motivation to do the classification and the survey presented in this paper comes from the lack of a wide, systematic, and flexible classification and a corresponding survey of software solutions. Existing survey papers in the field of cache consistency focus mostly on hardware solutions. For example, papers [ARCHI86] and [EGGER88] present and compare (using simulation methodologies) hardware schemes based on snooping. Paper [AGARW88] gives a survey and an evaluation of hardware schemes based on directories. Papers [TOMA94a, TOMA94b] contain the most comprehensive survey of hardware solutions. Paper [STENS90] contains both hardware and software solutions, but the solutions included are not detailed and many recent software solutions are missing. A similar statement applies to paper [LILJA93]. Paper [CHEON90] gives a comparative analysis of three software schemes based on compiler assistance—all three were developed by Cheong and Veidenbaum. Paper [MIN92] contains an overview of software solutions and proposes one relatively narrow classification of software solutions.

In this paper we propose a set of criteria for the classification of software solutions. Existing software solutions are presented in the context of these criteria.

After that, the proposed classification is generalized through the introduction of an abstract space of relevant criteria. Each specific software solution is associated with one specific point (specified with multiple coordinates) of the multidimensional criterion space. This type of generalization marks the points with no known associated solution. Some of these points correspond to solutions that are potentially useful under the circumstances.

In addition, this paper contains a survey of representative efforts in the domain of performance evaluation of software solutions [MIN90, TARTA92, OWICK89, ADVE91]. The efforts presented include both analytical (using mathematical models) and empirical (using simulation models) analyses, as well as comparisons of software and hardware solutions.

The paper is organized as follows. The second section defines the cache consistency problem in the context of software solutions. The third part proposes a set of classification criteria, defines the basic classes of software solutions, and mentions typical examples for each class. The fourth part contains a broad survey of existing solutions based on the above mentioned classification. The fifth part introduces a generalization of the above classification and points to some potentially useful solutions not yet found in the open literature. The sixth part contains a survey of efforts related to performance evaluation and comparison of software solutions.

2. Approaches to Classification of Cache Consistency Schemes

One of the most widely cited definitions of memory system consistency (coherency) is the Censier–Feautrier definition from [CENSI78]. It states that "a memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address." This definition allows copies of shared data to contain different values temporarily. It is only important that the value is updated before it is read. The ability to delay consistency maintenance activities to the moment when they are necessary decreases the frequency of consistency maintenance activities and consequently the system overhead. Most software solutions exploit this possibility.

The most common criterion used for classification is the placement of the consistency maintenance mechanism—either in the hardware or in the software. Hardware solutions make the consistency problem completely transparent to the user and to all software levels. Software solutions make the consistency problem visible either to the operating system or the compiler. The virtual machine accessible to the programmer that uses the compiler or the operating system primitive operations must be consistent. This classification has a drawback because some solutions are hybrid by nature, and not always easy to classify according to the hardware/software division line. The consistency maintenance algorithms in software are implemented more or less with hardware support. On the other hand, some authors [SKEPP95, DAHLG95] offer compiler-level optimizations for hardware solutions. Still, this paper starts from the hardware/software division, primarily in conformance with the widely adopted basic classification of [STENS90] and [TOMA94a, TOMA94b].

Instead of the hardware/software criterion, other criteria could serve as well for the classification of cache consistency maintenance solutions in shared-memory multiprocessor systems. These include, but are not limited to, the following:

1. **Dynamic versus static.** If consistency maintenance activities are planned at the execution time, the solution is dynamic. All hardware solutions, as well as the solutions predominantly implemented on the operating system level, belong to this group. If the consistency maintenance activities are planned at the compile time, the solution is static. All compiler solutions belong to this group.
2. **Centralized versus distributed.** If the consistency maintenance data (such as directories in some hardware solutions) are kept at a centralized site, the solution is centralized. If the consistency maintenance data (such as descriptions of cache line states in some solutions with snooping) are kept in a distributed fashion, the solution is distributed.
3. **With communications versus without communications.** If the consistency maintenance activities imply communications among processors, as is typically the case with hardware solutions, the solution is "with communications." If the consistency maintenance activities are performed without any communications among processors, as is typically the case with software solutions, the solution is "without communications." On a related issue, Min and Baer [MIN90] classify the existing schemes to those (a) without invalidation (such as the write-update snoopy schemes), (b) with induced invalidation (such as write-invalidate snoopy schemes or directory-based schemes), and (c) with self-invalidation (such as software schemes based on compiler assistance).
4. **Scalable versus nonscalable.** All software schemes that we are aware of, as well as some directory-based hardware schemes, are characterized by good scalability. Snoopy schemes are characterized by poor scalability and are applicable only in multiprocessors with a relatively small number of processors.

3. A Proposal for the Classification of Software Schemes

Of the papers that treat, among other issues, the classification of software solutions, the most complete is that by Min and Baer [MIN90]. Their classification is based on several criteria (some of which we have accepted and reproduced in this paper) and encompasses a relatively small number of existing solutions. It will be referred to here as the Min–Baer classification. This paper proposes a set of 10 criteria which are believed to be broad enough to create clear distinctions among most of the existing software solutions. Given criteria are not fully orthogonal, that is, some criteria are derived from another ones. The classification method is flexible enough to permit widening of the criterion set by adding new criteria if and when so required by a newly generated solution. For each criterion, we define a set of corresponding classes. Each class is illustrated with examples from the open literature (a reference to the original paper is included). Also, a "wild card" class is introduced, referred to as (*), which can be associated

with any criterion. If a given criterion is irrelevant for some solution, it will be assumed that the solution is marked as (*) in relation to that criterion.

Criterion #1: **Dynamism (D).** Decisions about what consistency maintenance activities to take can be made at compile time (statically) or at run time (dynamically).

Classes: Static schemes (s), as in [WULF72, VEIDE86, CHEON89], or dynamic schemes (d), as in [SMITH85, BENN90b, TARTA92].

Comment: In this work, only the schemes that require absolutely no compile time analysis are designated as dynamic schemes. Consequently, some of the schemes that make invalidation-related decisions at run time, but also perform some compile time analysis are classified as static schemes.

Criterion #2: **Selectivity (S).** Explicit consistency maintenance activities (such as invalidation) can be done on the entire cache memory (that is, without any spatial discrimination) or on only a portion of the cache memory (that is, selectively).

Classes: Indiscriminative schemes (i), as in [VEIDE86], or selective schemes (s), as in [CHEON88, CYTRO88, TARTA92].

Comment: The schemes referred to as selective are characterized by a wide range of granularity of data which can be the subject of the consistency maintenance activities. For example, in the case of the IBM RP3 [BRANT85], a wide plethora of objects, all of them of different sizes, can be the subject of the invalidation actions. The granularity differences are not reflected through this criterion; however, another criterion (to be introduced later) covers the granularity issue.

Criterion #3: **Restrictiveness (R).** The consistency maintenance activities can be performed for preventive purposes (conservatively) or only when absolutely necessary (restrictively).

Classes: Conservative schemes (c), as in [VEIDE86, SMITH85], or restrictive schemes (r), as in [CHEON89, TARTA92].

Comment: This criterion could result in a fine division having more than two classes. One could introduce a measure called the "level of restrictiveness," defined as the ratio of absolutely necessary consistency maintenance activities to the total number of consistency maintenance activities performed, averaged over a number of benchmark programs. In an absolutely restrictive scheme, this ratio would be equal to 1.

Criterion #4: **Adaptivity (A).** The consistency maintenance algorithm can be fixed or adaptable to the characteristics of the access patterns to data objects.

Classes: Fixed schemes (f), as in [SMITH85, CYTRO88, CHEON89, TARTA92], or adaptive schemes (a), as in [BENN90b].

Comment: So far, only one adaptive scheme appears in the open literature [BENN90b]; thus no need exists to introduce a finer measure

called the "level of adaptivity." However, as time goes by and other adaptive schemes are introduced, a need may arise for such a measure.

Criterion #5: **Locality (L).** The consistency maintenance algorithm may reduce misses strictly locally (interblock reduction) or globally (intrablock reduction).

Classes: Local schemes (l), as in [SMITH85, VEIDE86], or global schemes (g), as in [CHEON89, TARTA92].

Comment: Obviously, this criterion is an extension of the restrictiveness criterion and is irrelevant for the absolutely conservative schemes (those without any reduction of misses).

Criterion #6: **Granularity (G).** The size and structure of the unit object varies from one consistency maintenance algorithm to the other.

Classes: Line (l), also referred to as "cache line" or "cache block" as in [CHEON89]; page (p), as in [WULF72, SMITH85]; segment (s), as in [TARTA92]; or flexible object (f), as in [BRANT85, BENN90b].

Comment: To some extent, this criterion extends the selectivity criterion. In the case of fully indiscriminative schemes, this criterion is irrelevant.

Criterion #7: **Blocking (B).** The basic program block in the consistency maintenance process varies in size and structure from one algorithm to the other.

Classes: Critical region (c), as in [SMITH85, TARTA92]; epoch (e), as in [MCAUL86, VEIDE86, LEE87, CHEON89, MIN89]; subroutine (s), also referred to as "procedure" as in [CYTRO88]; or the entire program (p), as in [WULF72].

Comment: Actually, terms such as "computational unit" [MCAUL86], "loop" [VEIDE86], "epoch" [LEE87, MIN89], "task level" [CHEON89], and the like, refer to the issue which is essentially the basic consistency block. Consequently, one entire class was introduced to encompass all these cases.

Criterion #8: **Positioning (P).** Instructions to implement the consistency maintenance algorithm can be positioned at various places in the program.

Classes: Entry/exit into/from the critical region (e), as in [SMITH85, TARTA92]; loop boundary (b), as in [VEIDE86, CHEON89]; source/sink of the data dependency (d), as in [CYTRO88]; or interrupt procedure (i), as in [CHERI86].

Comment: Some of the schemes do comparisons at each reference (using specialized hardware support), which can also be treated as a consistency maintenance activity. However, this criterion considers only the special instructions for consistency maintenance (invalidate, flush, post, and the like).

Criterion #9: **Updating (U).** The main memory can be updated either at the time of cache write or after some delay.

Classes: Write-through (t), as in [CHEON89]; write-back (copy-back) (b), as in [CYTRO88, BENN90b]; hybrid (h), which implies both write-through AND write-back, as in [SMITH85]; or alternative (a), which implies either write-through OR write-back, as in [TARTA92].

Comment: One should notice the difference between the hybrid schemes which (as in [SMITH85]) offer write-through for private data and write-back for shared data, on one hand, and alternative schemes which (as in [TARTA92]) propose that one or the other approach be used exclusively.

Criterion #10: **Checking (C).** Conditions of inconsistency can be checked using several techniques.

Classes: Checking the data type or the reference type (r), as in [WULF72, LEE87a, MIN89]; program structure analysis (s), as in [BRANT85, VEIDE86]; data dependency analysis (d), as in [CYTRO88]; run-time information comparison (c), as in [CHEON89]; or monitoring of the traffic on the interconnection network (m), as in [CHERI86].

Comment: The classification based on this criterion should be treated conditionally. Some schemes include more than one technique for detection of inconsistency. For example, this is the case with the scheme in [CHEON88]. In such cases, the schemes are classified according to the technique that dominates. However, for completeness, the presence of other techniques will be indicated accordingly (using slashes, as in c/+s+d+r/).

4. An Overview of Software Schemes

This section contains a relatively broad overview of existing software schemes for maintenance of consistency among private cache memories. Using the criteria and the classes introduced in the previous section, a (10-element) set of attributes is associated with each scheme presented. These attributes determine the place of a given scheme in the proposed classification. In accordance with the (first) dynamism criterion, this section is divided into two subsections. One deals with static schemes and the other with dynamic schemes.

For consistency of presentation, each scheme description of this section includes the following elements:

(a) Information about the authors and the home institution of their research, the project of which that research was a part, and the major references to the original work.
(b) The essence of the contribution, or the major characteristic of the research, in succinct form.
(c) Description of the scheme, in which the level of detail is related to the impact of the scheme, its relevance to the class under consideration, and the amount of data available through the open literature.
(d) Final discussion of the scheme (advantages, disadvantages, and possibly a discussion of some specific issues).

(e) At the end, the 10-element descriptor vector is given, thus defining the place of the scheme within the overall classification used in this work.

4.1. Static Schemes

Static schemes predominantly rely on program analysis at compile time. Analysis points to potential causes of inconsistency, and additional information is added to the program to avoid fatal inconsistency errors during the program execution. This additional information includes, but is not limited to, issues such as marking of data, marking of references, and insertion of special instructions. This analysis (and related actions) eliminates the need for processors to communicate during the execution of the critical parts of the code, thereby making static schemes well scalable. On the other hand, static solutions degrade system performance because the precise prediction of memory conflicts due to cache inconsistency is not possible (actions are performed on the basis of some kind of "worst-case" analysis, not on the basis of "deterministic-optimum" analysis).

4.1.1. The C.mmp page marking from Carnegie-Mellon University

Wulf and Bell [WULF72] from Carnegie-Mellon University, working on the C.mmp project, were among the first to notice the problem of multiple values of shared variables existing concurrently in private cache memories of different processors in a multiprocessor system. The essence of the proposed method for the maintenance of cache consistency is in keeping the read-write shared data out of the cache at all times.

Only the read-only shared pages can be cached in the private cache memories. This is particularly the case with pages that contain shared instructions. Pages are marked as "cachable" using a bit reserved for this purpose in the relocation registers.

This scheme is extremely conservative, thereby permitting relatively easy realization but decreasing the processing power considerably [OWICK89].

Classification: (D:s,S:*,R:c,A:f,L:*,G:p,B:p,P:*,U:*,C:r).

4.1.2. The Ultracomputer program structure analysis from New York University

Gottlieb and his coauthors [GOTTL82, GOTTL83], Edler with his coauthors [EDLER85], and McAuliffe in his PhD thesis [MCAUL86] proposed the approach for software maintenance of cache consistency in the Ultracomputer multiprocessor developed at New York University. The essence of the approach is that the caching of read-write shared data is permitted in the intervals of the program execution when the read-write shared data are used exclusively by one processor.

The Ultracomputer supports two instructions for software managing of cache memories. These instructions are inserted into the user code at compile time. The Release instruction frees one cache memory entry without copying the data (from that entry) into the main memory. This means that the Release instruction

prevents memory traffic. According to [GOTTL83], the main memory is updated according to the write-back approach. In a later paper [EDLER85], a combined approach is used: write-back for private data and write-through for shared data. Independently of cache consistency maintenance, the Release instruction can be used, at the exit from a begin-end block, to free the space in cache that was used by the local variables. In the context of cache consistency maintenance, the read-write shared variables are kept in cache only during the time intervals when it is guaranteed that they will be accessed only for read. At the end of these intervals, the variable has to be removed from the cache, using the Release instruction, and marked as noncachable. The second of the two cache consistency maintenance instructions is the Flush instruction which copies data from the cache memory entry into the main memory without erasing the entry. For private data, the Flush instruction must be executed when blocking a task, because the execution of the task may continue on another processor. For data shared by a parent-task and its child-tasks, instructions Flush and Release are executed before the spawn operation. The variable that was private before a new task was created now becomes shared and must be marked as noncachable; however, its value in the main memory is up to date. After the child-tasks are completed, the parent task can continue to work with the same variable, treating it as a private, and therefore cachable, variable. The first next reference to that variable will place its current value into the cache. All activities described here can be performed on the segment level or on the cache level.

This scheme is less conservative than the Wulf–Bell scheme. Still, this caching scheme has no restrictiveness in the invalidation of shared data. The scheme supports only the locality of references within "safe" intervals. The consistency maintenance related instructions can be made selective with the segment-level granularity; however, something like that would slow them down due to the required scan through cache directory.

Classification: (D:s,S:s,R:c,A;f,L:l,G:s,B:e,P:b,U:h,C:s).

4.1.3. The RP3 flexible invalidation from the IBM T.J. Watson Research Center

Brantley, McAuliffe, and Weiss [BRANT85], as well as Pfister and coauthors [PFIST85], proposed a cache consistency maintenance scheme similar (but more flexible with respect to granularity of data) to the one from the NYU Ultracomputer. The research was done at the IBM T.J. Watson Research Center and was intended for the IBM's RP3 multiprocessor.

At compile time, shared data are marked as cachable if they are used exclusively in the given program block. Data marked as noncachable in one program block may become cachable in the next program block, and vice versa. Also at compile time, data are marked as volatile or not, which helps in the efficient management of the temporary cachable data. In short, all data exhibit two attributes: cachability and volatility. These attributes are also specified on the level of segment and page descriptors for specific segments and specific pages. They are also applied to segment tables and page tables. Moreover, the RP3 includes a number of different data invalidation instructions: line invalidation, page invalidation, segment invalidation, user space invalidation, supervisor space invalidation, and

invalidation of all volatile data (which can exist in many different segments or pages). The main memory is updated using the write-through approach.

In general, the cache consistency maintenance strategy of the IBM RP3 scheme is still very conservative; however, its selective invalidation is more flexible in the sense that the granularity of the invalidation can be varied. Consequently, in the case of the IBM RP3 scheme, the software designer can use more powerful "tools" in an effort to minimize the probability that a data item is "expelled" from the cache, although it may be needed later.

Classification: (D:s,S:s,R:c,A:f,L:l,G:f,B:e,P:b,U:t,C:s).

4.1.4. Cache on/off control from the University of Illinois

Veidenbaum [VEIDE86] proposed a relatively simple static software scheme for cache consistency maintenance in the Cedar multiprocessor at the University of Illinois. The scheme assumes a program structure based on parallel/serial loops, which are widely used to express parallelism in numeric applications. The essence of the scheme is that compiler inserts special-purpose instructions for cache consistency maintenance only at loop boundaries and at subroutine call points.

Assuming that the main memory value of the shared variable is current, which is achieved here using the write-through approach, incoherence occurs when a value fetched from the cache is different from the value in the main memory. According to this definition, Veidenbaum gives a formal proof of the lemma about the necessary conditions for inconsistency. When processor Pj (j=1,2,...) fetches the variable X, the inconsistency will happen if (1) the value of the variable X is present in the cache memory of the processor Pj, and (2) the new value of X is placed into the main memory by the processor Pk (k≠j, k=1,2,...) after the processor Pj had accessed the variable X last time. Detection of the necessary conditions of inconsistency can be based on the data dependency graph. As indicated in Figure 1, the inconsistency will happen if the following is satisfied: (1) Pj executes instruction S1 which writes into or reads from X; (2) another processor Pk (k≠j) executes instruction S2 which writes a new value into X; and (3) Pj executes instruction S3 which reads from X again.

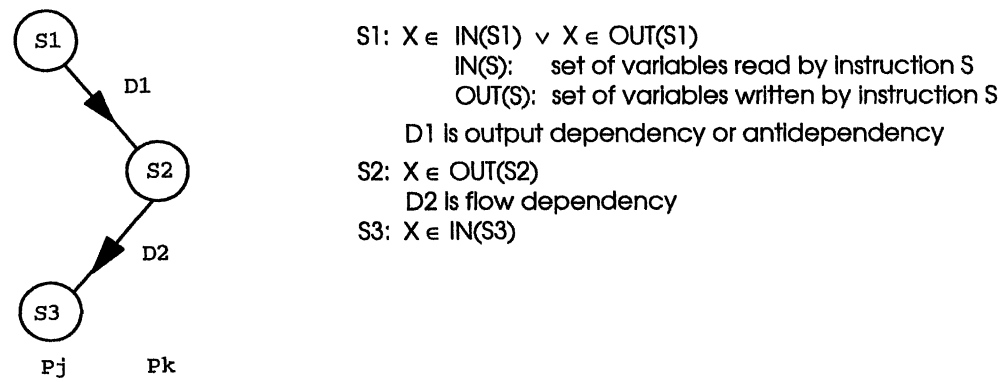


Figure 1. Data dependency graph containing the conditions of inconsistency.

Instructions for cache consistency maintenance are as follows: (1) Flush (or Invalidate according to [CHEON90]), which deletes the contents of the entire cache (indiscriminative invalidation); (2) Cache-on, which enables the cache, that is, enables all references to go through the cache; (3) Cache-off, which disables the cache, that is, forces all references to bypass the cache. The proposed cache coherence maintenance algorithm is applied at compile time—each instruction in the program is examined, and where necessary, the above-mentioned cache consistency maintenance instructions are inserted before, inside, or after the loops as well as before/after procedure calls, as follows:

1. DoAll: There is no data dependency between different iterations of the loop, thus there is no inconsistency hazard. Consequently, cache memory can be enabled. The loop is expanded with Cache-on and Flush instructions, so each processor, at the beginning of the iteration which is assigned to that particular processor, enables its own cache and deletes the old contents.
2. DoAcross: Parallelization is possible with additional synchronization required to cope with the data dependency between different iterations. Consequently, conditions of inconsistency can arise, and the Cache-off instruction should be inserted in the loop. However, the cache could be enabled. In that case, the Flush instruction should be inserted after the synchronization-related wait-on-semaphore operation in the appropriate iteration on the processor which contains the sink of data dependency. With respect to other details, the treatment of the DoAcross loop is the same as the treatment of the DoAll loop.
3. Serial loop: Cache memory can be enabled because the loop is entirely executed on one processor. This is the consequence of the fact that the structure of data dependencies between iterations disables any parallelization. Instructions Cache-on and Flush have to be inserted before the loop.
4. DoEnd: If the cache was enabled before the loop and disabled in the loop, the Cache-on instruction is inserted after the loop. If the cache was enabled before the loop and enabled (fully or partially) in the loop (because the loop was of the type DoAll or DoAcross), the Flush instruction is inserted after the exit from the loop.
5. Call: Instruction Cache-off is inserted before the call; if the cache is enabled before the call, instructions Cache-on and Flush are inserted after the call.

As can be seen, the algorithm includes no analysis of individual data dependencies for each particular instruction. It only does the preventive disabling and flushing of the cache at the boundaries of those loop types in which the nonconsistency condition can be created. Consequently, this algorithm is very conservative, its restrictiveness of invalidation is very low, and the invalidation is indiscriminative.

Classification: (D:s,S:i,R:c,A:f,L:l,G:*,B:e,P:b,U:t,C:s).

4.1.5. Program analysis and reference marking from the University of Illinois

Lee, Yew, and Lawrie [LEE87a, LEE87b] proposed a scheme based on the static

analysis of program structure and marking of individual references. The research was done at the University of Illinois.

The approach starts from the notion of epoch, which refers either to a parallel loop or to a piece of sequential code between two parallel loops. First, the program is analyzed and segmented into a sequence of epochs. Second, for each data item, its cachability status is marked for each epoch separately (the cachability status of a data item may change from one epoch to the other). The marking is done at compile time, using Paraphrase, a parallelizing Fortran compiler. Within an epoch, all references to a variable accessed by a number of processors, but for at least by one processor for write, are marked as noncachable (references to other variables are marked as cachable). References marked as noncachable bypass the cache at execution time and access the main memory directly. At the end of each epoch, the invalidation of the cache contents is done, so that the processor obtains up-to-date values from the main memory at the next reference to the variable. The main memory is updated using the write-back approach.

This algorithm efficiently supports the intraepoch localities. However, the cache is flushed at the end of each epoch, which means that the interepoch localities are not supported and the performance of the scheme is degraded.

Classification: (D:s,S:i,R:c,A:f,L:l,G:*,B:e,P:b,U:b,C:r/+s/).

4.1.6. Fast selective invalidation from the University of Illinois

Cheong and Veidenbaum [CHEON88] proposed another software scheme for consistency maintenance which also originated from research done at the University of Illinois. The speed of invalidation in this scheme is close to that of indiscriminative invalidation, but its selectivity is such that it results in a better hit ratio.

It is assumed that the value of a variable in shared memory is always valid because the write-through approach is used. Also, it is assumed that the synchronization variables are not cached. Each word in cache has a change bit associated with it. When set, this bit indicates that the word is potentially invalid (in another paper, which originated from the same research [CHEON90], this bit is defined in the inverse way). When a new line is entered into the cache, the corresponding change bits are set to 0 (zero). Also, a cache write sets this bit to 0. The Invalidate instruction only sets the change bits to 1—it does not perform the actual invalidation. The setting of change bits is done in one cycle for all change bits in the cache. That is why the invalidation is as fast as in the case of indiscriminative invalidation. At compile time, the Invalidate instructions are inserted at the critical places in the program (loop boundaries), in the same way as in a previously described scheme [VEIDE86]. Also, each word in cache has a clear bit associated with it. This bit, when set, indicates that nothing was read/written from/into the corresponding word. The Clear-cache instruction sets the clear bits of the entire cache memory to 1 (one), which brings the cache into the initial clear state. Marking of references is done at compile time, as follows. All references that can be guaranteed not to fetch an invalid data copy are marked as cache-read. Other references are marked as memory-read. The marking algorithm

is applied to each procedure separately. It is assumed that the cache is always cleared at the beginning of a procedure (using the Clear-cache instruction). References to data that are only read within a procedure are marked as cache-read; also, all references to read-write data coming chronologically prior to the first write to the same data are marked as cache-read. Other references are marked according to specific rules. For cache hit, not only must the address and validity tags match, but also the reference must be either a cache-read or a memory-read with the change bit set to 0.

In this scheme, the invalidation instructions are inserted into the program statically. The same applies for marking of references. However, the Invalidate instruction here only points to the occurrence of a situation in which stale contents may appear in cache memory. Decisions about actual invalidations are made at execution time. The hit is prevented only for those memory-read references with the change bit in the accessed word set. This approach has two benefits. First, invalidation has no impact on cache-read references; that is, the invalidation is selective. Second, when a variable is read more than once within a single iteration of a DoAll loop, although all reads are marked as memory-read, the invalidation will refer to the first read only; the other reads will be satisfied in the cache memory (intraepoch restrictive invalidation). However, the algorithm for inserting the Invalidate instruction is conservative—preventive invalidation is done at the boundaries of parallel loops (interepoch conservative invalidation).

Classification: (D:s,S:s,R:c,A:f,L:l,G:l,B:e,P:b,U:t,C:c/+s+d+r/).

4.1.7. Programmable cache from the IBM T.J.Watson Research Center

Cytron, Karlovsky, and McAuliffe [CYTRO88] proposed a solution for cache consistency maintenance in shared-memory multiprocessors that is based on a detailed static analysis of data dependencies for all instructions in the program. The authors were with the IBM T.J. Watson Research Center and the University of Illinois at the time when the work was published.

As in other static schemes, special instructions for cache memory management are inserted into the program at compile time, as follows: (1) Post is the instruction that passes the value of the local data copy into the location of the original data in main memory; (2) Invalidate is the instruction that destroys the local data copy; and (3) Flush is the instruction that performs a combination of the previous two instructions. A write-back cache memory is assumed, one which is large enough so that replacement of lines can be neglected. The line size is equal to one memory word. Each address in the global address space is marked in one of the following three ways: (a) Cachable—data copies on these addresses can exist in private cache memories at all times (read-only shared data and private data); (b) Temporarily cachable—data at these addresses can be placed into the cache memory, and the compiler is responsible for making decisions about when these data are to be flushed or invalidated (read-write shared data); (c) Noncachable—data from these locations are not allowed to be placed into the cache memory (shared data with activity dynamics such that the consistency maintenance does not pay off, due to a relatively large overhead). The cachability status of an address is determined in three steps: first, all variables are marked as

Temporarily cachable; second, the locations where the cache management instructions are to be inserted are determined; and third, the variables that require no consistency maintenance are marked as Cachable, while those accessed only after invalidation are marked as Noncachable; other variables remain Temporarily cachable (these would require a special analysis to determine how profitable it would be to perform the consistency maintenance activity for each particular variable). The proposed algorithm relies on some information obtained from the parallelizing compiler, such as information about the ability to parallelize a loop (if the loop is parallel or serial), the data dependency graph, and the like. Instruction Post(X) is inserted after the instruction that writes into a shared variable X if it (the instruction that writes) is a source of the crossing flow dependency. The term crossing refers to the case in which the sink of the dependency can be executed on another processor. Also, if the variable X is inter-procedurally live after the procedure under analysis, the instruction Post(X) is inserted after the last write to X within the same procedure. Hence, the instruction Write is followed by the instruction Post(X) in the case of the following sequence:

$$Write\ (Pi,X) \rightarrow Read(Pj,X) \qquad (j \neq i)$$

The authors propose two different algorithms for insertion of the instruction Invalidate. The first one is less complex. It is linearly dependent on the number of flow dependencies in the analyzed program. However, it can generate unnecessary invalidations because it is based on a conservative invalidation strategy: Invalidate (X) is inserted before each reading of the variable X if the variable X is a sink of a crossing flow dependence. Also, if a shared variable X has been defined before the entry into the procedure, the Invalidate(X) instruction is inserted before the first reading of the variable X in the procedure. Hence, instruction Invalidate(X) is inserted before the read operation when the above-mentioned write-read sequence exists. However, it would be enough to invalidate the variable X before the second reading, only when the following sequence exists:

$$Ref(Pj,X) \rightarrow Write\ (Pi,X) \rightarrow Read(Pj,X) \quad (j \neq i)$$

Consequently, the authors propose a better, but more complex algorithm for insertion of the invalidation instructions (an algorithm quadratically dependent on the number of flow dependencies). The Invalidate(X) instruction is inserted before the sink of the crossing flow dependency if the source of the analyzed flow dependency acts as a sink in at least one output dependency, or in at least one antidependency, in relation to the variable X. Instruction Flush(X) generates both operations—the passing of X into the shared memory and the invalidation of X in cache memory (of the processor that executed the instruction). Instruction Flush is inserted after the source of the direct dependency, substituting for the instructions Post and Invalidate when such substitution can be done without causing unnecessary misses.

This scheme is very restrictive to the invalidations within a procedure. This is due to the fact that both the program structure analysis and the data dependency analysis are performed at compile time. However, the problem of unnecessary invalidations on the interprocedure level is still there.

Classification: (D:s,S:s,R:r,A:f,L:g,G:l,B:s,P:d,U:b,C:d/+s+r/).

4.1.8. Version control from the University of Illinois

Cheong and Veidenbaum [CHEON89] proposed a scheme that represents an improvement of their "fast selective invalidation" scheme. The new scheme was also developed at the University of Illinois. It is based on the static analysis of the parallel program tasking structure and a dynamic control of the variable version using appropriate hardware support.

The essence of the approach is in the notion that, whenever a write is made to a shared variable, a new "version" of its contents is formed. In the case of DoAll loops, there is no data dependency between iterations (iterations of the DoAll loop form tasks on the same level of the task execution graph), and there is no hazard of another processor needing the new version of the shared variable. Therefore, no consistency maintenance related activities are needed within a DoAll loop—a temporary inconsistency of the memory system is allowed. However, before the processor passes the task level boundary, the most recent version of the variable in cache must be stored into the main memory. Each processor keeps a private information about the current version of each shared variable (scalar or vector) in its local memory. For that purpose, a current version number (CVN) is maintained. When passing to the next task level (for example, after the exit from a DoAll loop), processors execute the instructions (inserted by the compiler, based on the static program analysis) for incrementing the CVN for all variables being written at the previous task level. At the same time, each cache line contains a field with the information about the birth version number (BVN). The BVN is updated at the loading of each cache line from main memory. After a read miss and the loading of the cache line, BVN is equal to CVN. At each cache write, BVN is made equal to CVN + 1 (the number of the next version). At each access to a shared data item in cache, CVN and BVN are compared. If BVN is smaller than CVN, the data item in cache is stale, and the access is declared a miss. If BVN is equal to or greater than CVN, the data item in cache is valid, and the access is declared a hit (after a write to a variable, BVN = CVN + 1, and BVN is greater than CVN on the same task level).

The version control scheme [CHEON89] demonstrates a better performance than the previous two schemes [VEIDE86 and CHEON88] because it respects the temporal locality of references over the task execution level boundary [CHEON90]. A direct consequence of this is an increased hit ratio. This advantage is paid for by a more complex hardware support for consistency maintenance in comparison to other schemes.

Classification: (D:s,S:s,R:r,A:f,L:g,G:l,B:e,P:b,U:a,C:c/+s/).

4.1.9. Timestamps from the University of Washington

Min and Baer [MIN89, MIN92], from the University of Washington, proposed, independently of Cheong and Veidenbaum, a scheme which is essentially similar to the version control scheme.

Each shared data structure is assigned a counter, which is incremented at the end of each epoch in which the data structure can be modified. Each word in cache is assigned a "timestamp." This timestamp is set to "counter + 1" when

that word is modified. At the access to cache memory, a word is valid if the value of the timestamp is equal to or greater than the value of the corresponding counter. The Min-Baer counter corresponds to the Cheong–Veidenbaum CVN, and the Min-Baer timestamp corresponds to the Cheong–Veidenbaum BVN. The Min–Baer epoch corresponds to the Cheong–Veidenbaum task level.

The primary difference between the timestamps approach and the version control approach lies in the fact that Min and Baer also propose a very sophisticated algorithm of marking references, which better supports the localities between dependent tasks in a DoAcross loop.

Classification: (D:s,S:s,R:r,A:f,L:g,G:l,B:e,P:b,U:t,C:c/+s+r/).

4.1.10. Generational approach from the State University of New York at Stony Brook

Chiueh [CHIUE93] proposed an improvement of the idea about control of shared-data versions. His consistency scheme takes care of the current generation of tasks using a levelized task graph, instead of the separate management of each variable version. The work was performed at the State University of New York at Stony Brook.

Each cache line is extended with a valid generation number (VGN) field. On the other hand, a new register, to indicate the current generation number (CGN) of tasks, is added to each processor register set. At the end of each task, where a shared variable was written, the appropriate instruction to modify the appropriate VGN field in the cache line is inserted by compiler. The VGN field is set to the generation number, when this variable is written again by an arbitrary processor; thus, the VGN field indicates the last generation number, until the appropriate shared variable can be considered as valid. When a task is scheduled to a processor, CGN is written into the CGN register of that processor, to indicate the current generation of the task. An access to a shared variable X is treated as a hit, if $VGN(X) \geq CGN$.

Compared to the Cheong-Veidenbaum version control scheme or the Min-Baer timestamping scheme, this generational approach simplifies the hardware support. Instead of a local table for each processor to keep information on the global current version of each shared variable, only one additional register (CGN) is used here. Additionally, the Chiueh's approach naturally overcomes some inherent problems of the version control scheme, such as DoAccross loop handling or inefficiency caused by conditional writes.

Classification: (D:s,S:s,R:r,A:f, L:g,G:l,B:e,P:b,U:t,C:c/+s+r/)

4.1.11. One-bit time stamps from Rice University

Darnell and Kennedy [DARNE93], from Rice University, proposed a new scheme based on timestamping. Their scheme requires considerably less complex hardware support then appropriate schemes [CHEON89] and [MIN89], although it achieves at least the same hit ratio.

One bit, called "epoch bit," is associated with each cache line and notifies any arbitrary access to the cache line during current epoch of the program execution. All epoch bits are reset at each epoch boundary. Invalidate instructions are inserted statically by compiler at the end of each epoch (before reset of the epoch bits). Invalidation can be related to the whole array (conservative analysis of the program) or only to some parts of the array that have been written during the current epoch. In run time, these instructions actually invalidate only the cache lines that have not been accessed during the epoch. Invalidate instruction practically copies the epoch bit to the valid bit.

As it has been shown through simulation study, this scheme has nearly always a better hit ratio than the scheme proposed by Min and Baer [MIN89]. Hardware support, on the other hand, is very simple—each cache line is enlarged with one bit (epoch) and no additional bits are required in instructions or private memory.

Classification: (D:s,S:s,R:r,A:f,L:g,G:l,B:e,P:b,U:*,C:c/+s/).

4.2. Dynamic solutions

Like hardware solutions, dynamic software solutions maintain the consistency of private caches entirely at run time. They are implemented in the kernel of the operating system. Consequently, they do not contribute to compiler complexity (in static schemes, one multiprocessor system requires a number of different compilers—one for each language). Since the consistency maintenance decisions are performed at execution time, dynamic approaches do not require preventive actions; thus it is possible to carry out only absolutely necessary actions. Dynamic approaches are difficult to apply when the parallelism is modeled via parallel loops. Consequently, dynamic solutions are not suitable for numeric applications, which are primarily based on a programming model characterized by parallel loops. On the other hand, dynamic solutions offer a natural solution for nonnumeric concurrent applications, which are frequently based on a programming model with an explicit management of sharing (using critical regions, monitors, or similar mechanisms).

4.2.1. One time identifiers from the University of California at Berkeley

Smith [SMITH85] proposed a solution for dynamic maintenance of cache consistency that is entirely based on the application of operating-system-level primitive operations for control of exclusive access to critical regions. This research was done at the University of California at Berkeley.

In the case of conventional page-organized memory, the hashing function is used at each access to a page to calculate the real page address from the virtual page address. To avoid unnecessary repetitions of this operation, a translation lookaside buffer (TLB) can be introduced. This buffer is a cache in which the address tag contains the virtual page address and the data field contains the real page address. Smith proposed that each TLB entry and each line in the processor cache be expanded with another field which provides a unique marking of a shared data page. This field is referred to as OTI (one-time identifier). When a new TLB

entry is loaded, a new and unique value is placed into its OTI field. This value is read from a special incrementing register. When an entry for a shared page is accessed for the first time, the entry is loaded into the cache from the main memory. At the same time, the OTI field value from the TLB is passed to the OTI field in the cache. All subsequent accesses to this variable check the value of the OTI field in cache for a match with the value of the OTI field in the TLB. If a match occurs, the access is treated as a hit; otherwise it is a miss. After the exit from a critical region, the processor executes the instructions that invalidate all TLB entries of the pages that belong to shared data being protected by the just-exited critical region. At the first next access to shared data, the corresponding TLB entry is loaded again, and a new value for the OTI field is obtained. This allows all later accesses to stale data to be treated as misses because the value of the OTI field in the TLB is different from the value of the OTI field in the cache. Next, Smith proposed a selective write-through, as follows. Updating of shared data is done with the write-through approach (which guarantees that the main memory is always updated for shared data), while the updating of the private data is done using the write-back approach.

The advantage of the OTI scheme is the complete decentralization of cache consistency maintenance, as is the case with most static schemes and not the case with most hardware schemes. Consistency is maintained at each processor autonomously, without any communications with other processors in the system. However, the scheme requires a relatively complex hardware support (the OTI extension in the TLB and the cache, comparators for the OTI fields, and so forth). Also, at the exit from a critical region no possibility remains for restriction in executing the invalidation instructions (here, the invalidation is done as a preventive action).

Classification: (D:d,S:s,R:c,A:f,L:l,G:l,B:r,P:e,U:h,C:c).

4.2.2. Consistency on interrupt request from Stanford University

Cheriton, Slavenburg, and Boyle [CHERI86] proposed a combined hardware–software solution for the maintenance of cache consistency in virtually addressed private cache memories. This research was done at Stanford University and was a part of the VMP multiprocessor project. The essence of the solution is in the hardware-based determination of the inconsistency conditions and generation of the interrupt request, followed by the software-based activities aimed at consistency maintenance.

This scheme is based on the concept of virtual memory. Page faults are determined in hardware, whereas fetching of new pages from the secondary memory is done in software. The proposed cache memory page size is relatively large (up to 512 bytes), and the proposed hardware for the transfer of data blocks is relatively fast (40 Mbytes/sec). Consequently, cache memory misses are relatively rare and can be processed in software on the operating-system level. The same concept is extended into the domain of cache consistency maintenance. The hardware-based bus monitor detects the inconsistency conditions and informs the local processor about that, by issuing an interrupt request. In response, the processor executes the interrupt routine and performs the consistency maintenance

activities (within the interrupt routine). The same consistency maintenance mechanism is used both for the shared pages and the pages that contains entries of the page table. The main memory is viewed as a sequence of cache page frames. A page can be in one of two states: shared or private. In the shared state, the page frame contains the most recently written page value, while the local cache memories can contain copies of that particular page. In the private state, only one cache memory contains a copy of the page. Each bus monitor maintains a private table of actions. For each page frame, this table defines the activity to be done by the monitor when the address from that page appears on the bus. If the corresponding page is not in the processor's private cache memory, no action is needed. If the page in the cache memory is shared, the monitor ignores all read-shared requests; all read-private and assert-ownership requests have to be aborted, followed by an interrupt to the local processor (during the processing of that interrupt request, the processor invalidates the page in cache memory). If the page in cache memory is private, the monitor has to abort the transaction and issue an interrupt request for each transaction related to that page (during the processing of that interrupt, if the page is dirty, the processor performs a write-back into the main memory; if the transaction is read-private or assert-ownership, the processor invalidates the page in cache; if the transaction is read-shared, the processor changes the page state into shared). The processor that was performing the aborted transaction detects the abortion and starts the transaction again. As the authors briefly summarize, using this protocol, a request for a shared copy of a shared page is satisfied immediately. A request for a shared copy of a private page results in an abortion of the transaction and the relinquishing of ownership. Consequently, the processor that requested the page is allowed to succeed in the next try. A request for a private copy of a shared page is satisfied immediately, but it induces the invalidation of all cache copies of that page. A request for a private copy of a private page is aborted but causes the relinquishing of ownership, which allows success on the next try.

The exclusively hardware-based detection of inconsistency conditions results in a very restrictively applied invalidation; however, the size of the cache page (which must be relatively large so that interrupts related to misses are relatively rare) makes the invalidation poorly selective in the case of shared data of fine granularity.

Classification: (D:d,S:s,R:r,A:f,L:g,G:p,B:p,P:i,U:b,C:m).

4.2.3. Conditional invalidation from the University of Belgrade

The authors of this survey, in their paper [TARTA92], proposed a class of dynamic software schemes for maintenance of cache consistency based on testing for inconsistency conditions at the entry into a critical region, thereby avoiding unnecessary invalidations. The work was performed at the University of Belgrade in cooperation with the NCR Corporation.

Consistency maintenance activities are performed at the entry into /exit from the critical region (the program code region with exclusive access to the shared data segment). Of the three schemes they proposed, the "version verification" scheme is the most invalidation-restrictive, and the one that utilizes the existing inter-

regional locality of references the best. At the entry into the critical region, the real version of the shared segment (information from the shared-segment table) is explicitly compared with local and private information about the most recently used segment version. If the versions do not match, selective invalidation of the shared segment is done. If the shared memory is updated using the write-back approach, the updating of the shared-segment original (in main memory) is done at the exit from the critical region.

A simulation analysis has shown the advantages of the restrictively applied invalidation, even if the number of processors is relatively small (up to 16). This is especially the case if the mostly-read shared segments predominate.

Classification: (D:d,S:s,R:r,A:f,L:g,G:s,B:r,P:e,U:a,C:c).

4.2.4. Adaptive software cache management from Rice University

Bennett, Carter, and Zwaenepoel [BENN90a, BENN90b] proposed a dynamic adaptive software maintenance of consistency for the Munin system at Rice University. Munin supports several consistency maintenance mechanisms, and in each case uses the one that best fits the prevailing class of shared objects for that particular case (the access type is relevant when making the decision about the specific mechanism to use).

Generally, in distributed shared memory (DSM) systems [PROTI95] the memory inconsistency problem exists because the virtual address space is physically distributed on several local memories. Consequently, one memory address can be mapped into a number of memory modules. Munin is a program system that makes this problem completely transparent for the application. For the purpose of memory consistency maintenance, the programmer informs the system about the expected way of access to shared objects. The classes of objects are (a) write-once, (b) private, (c) write-many, (d) result, (e) synchronization, (f) migratory, (g) producer-consumer, (h) read-mostly, and (i) general read-write. A study [BENN90b] has shown that the number of general read-write objects is relatively small. It was also shown that parallel programs behave differently in different phases of their execution. Except in the initialization phase, most of the accesses are of the read type. Finally, it was noticed that the average period between two accesses to synchronization objects is considerably longer than the average access period for other shared objects.

In addition to traditional mechanisms for consistency maintenance in DSM systems (replication, invalidation, migration, and remote load/store), Munin supports one new mechanism—delayed update. When a process modifies a shared-data item, the new value is not immediately sent to remote servers to update their copies. Instead, the sending is postponed until the first next synchronization, and all changes are sent in one package. Between the synchronizations, each process forms its own queue of delayed updates. This queue is flushed at the time of the next synchronization.

Some objects are written only once (at the initialization), and later only read. These objects are maintained by replication and accessed only locally. Private

objects are those accessible to all processes, but they are accessed by one processor only. Accesses to private objects are local, and they are not managed by Munin. The write-many objects are those being written into many times between two synchronization points. The delayed update mechanism proves to be very efficient for the consistency maintenance of these objects (weak consistency). The result objects collect the writes from several processes and make them available for reading to only one process. The delayed update mechanism is very useful here, as well. For the synchronization objects, the consistency is maintained using the distributed locks mechanism. Migratory objects are accessed in one time interval by one process only (within the critical region). Consistency is achieved efficiently if, at the exit from the critical region of one process, the object and the associated distributed lock for that critical region are migrated onto the processor to execute the first next process from the waiting queue for the corresponding critical region. The producer-consumer objects are those being written into by one process and being read by a number of other processes. The consistency is maintained by the so-called *eager object movement* to the processors that need them before the actual need arises (and the related request is issued). The mostly-read objects are replicated and then maintained by updating via broadcast after the writes which are relatively rare. The general read-write objects are those that cannot be put into any of the previously described classes. Consistency of these objects is maintained through a mechanism based on the Berkeley Ownership scheme for consistency maintenance of cache memories.

The adaptivity of this consistency scheme (based on the dynamic nature of objects) has a very positive impact on the overall performance of the system and the application. Simulation studies [BENN90b] have shown that, for a given application (such as Quicksort), this adaptive method can decrease the bus traffic by more than 50 percent compared to the conventional hardware write-update, and more than 85 percent compared to the write-invalidate mechanism.

Classification: (D:d,S:s,R:r,A:a,L:g,G:f,B:p,P:*,U:*,C:*).

5. Generalization of the Proposed Classification

The set of criteria introduced earlier in the paper enables each software solution to be described with a set of attributes. These attributes are related to the 10 criteria and correspond to different possible classes of the applied criteria. A 10-tuple of attributes was associated with each of the presented solutions. The proposed classification method can be easily generalized through the introduction of an abstract space of criteria. Each one of the presented solutions can be treated as a point in such a space.

The coordinates of the multidimensional discrete space of criteria correspond to the chosen criteria, and the discrete values on these coordinates correspond to the classes of the applied criteria, that is, to the attributes. The number of criteria determines the dimension of the space. The number of classes for each criterion determines the number of discrete values that exist on the corresponding coordinate.

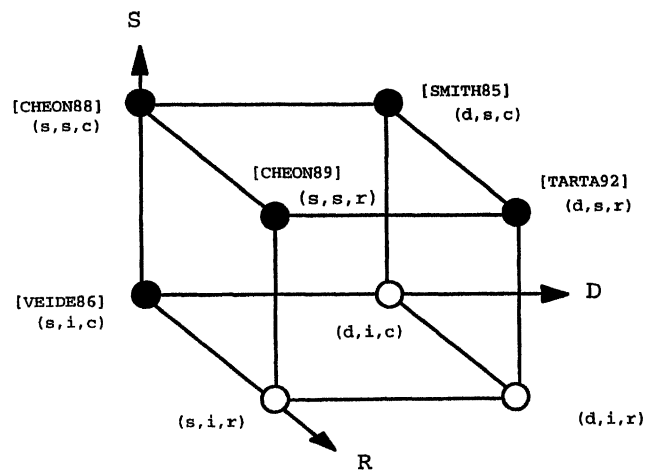
If the abstract space is described with a sufficiently large number of criteria

(complete space), each point corresponds to one of the existing solutions. If the abstract space is described with a smaller number of criteria (reduced space), one point may correspond to a larger number of existing solutions. This situation indicates either that some important criterion was not taken into consideration or that the solutions are practically the same (such as [CHEON89] and [MIN89]) or semantically similar (such as [MIN89] and [CHIUE93]), but different in implementation, resulting in different performance and/or complexity. On the other hand, if some criterion is not relevant for some solution, the axes related to this criterion must be extended so that the value (attribute) is "irrelevant." This action prevents the propagation of the solution over all points along the coordinate of the irrelevant criterion. Thus all points in the complete space come into one-to-one correspondence with the existing solutions, and the classification boils down to positioning a given solution to a given point in the space.

For example, Figure 2 shows a reduced three-dimensional (3D) space formed from the first three criteria (dynamism, selectivity, and restrictiveness). Some points in this reduced space correspond to several solutions; however, only one representative solution is shown. For example, the point (s,s,r) contains not only the scheme from [CHEON89], but also the schemes from [CYTRO88] and [MIN89]. In addition, the point (d,s,r) contains not only the scheme from [TARTA92], but also the scheme from [BENN90b].

If the coordinates of the space are treated as discrete and bounded, as is the case with the abstract criterion space discussed here, the total possible number of points in this space is given by

$$Q = \prod_{i=1}^{N} Ci \qquad (1)$$



1. Dynamism (D): static (s) or dynamic (d)
2. Selectivity (S): indiscriminative (i) or selective (s)
3. Restrictiveness (R): conservative (c) or restrictive (r)

Figure 2. An example of a reduced 3D abstract criterion space.

where Ci refers to the number of classes for the ith criterion ($I = 1,\ldots, N$). According to equation (1), the number of theoretically different software schemes for the concrete case of the 10-criterion abstract space presented here, is Q=40,960. Of course, this number should be considered as an illustration for one possible classification, not as a general conclusion.

The combinations of attributes (N-tuples of attributes) that correspond to nonexisting solutions are referred to as "free points." Some of these free points can serve as guides to new solutions that are potentially useful. (Some of the free points correspond to solutions that make no sense; and of those that do make sense, some may not be very useful in given circumstances.) It should be noted that, given the characteristics of classes that describe a new solution, one could roughly estimate the performance of the solution without analyzing its details.

A relatively large number of new and potentially useful solutions could be found in the "plane" of adaptive static schemes. Using the semantic information about the access methods to a shared object, appropriate instructions could be inserted into the code at compile time to support the mechanism that is "optimal" for the given access method.

6. Evaluation Studies of Software Schemes

This section is devoted to an overview of existing evaluation studies of software solutions. Most of these are aimed at the following two issues: (a) obtaining a better understanding about the impact of various parameters (especially the workload related parameters) on the performance of software solutions and (b) gaining an impression about the real bottlenecks of software solutions. In addition, one of our goals is to present different methods for evaluation of software solutions.

The following evaluation studies have been selected for presentation here: (a) one simulation analysis based on real address traces [MIN90], (b) one simulation study based on the probabilistic synthetic workload [TARTA92], (c) one analytical study based on the mean value analysis (MVA) model [OWICK89], and (d) one analytical study that compares hardware directory-based schemes and software static schemes [ADVE91]. Each study is presented in two paragraphs, first describing the method and conditions and the second describing the main results and conclusions.

In [MIN90], Min and Baer compared a hardware scheme based on a centralized full-map directory, developed by Censier and Feautrier with a static software scheme based on timestamps developed by Min and Baer. In a follow-up paper, it was shown that the static software scheme based on timestamps demonstrates a lower miss ratio than do other static software schemes [MIN92]. The comparison in [MIN90] is based on simulation with three address traces of three different applications in which the model of parallelism is expressed through parallel DoAll loops (the program paradigm is referred to as SPMD—single program multiple data). At run time, the program forms a sequence of epochs. One epoch can have multiple instances, as is the case with epochs in which DoAll loops are executed—each instance represents one iteration and is executed on a different

processor. The simulation was done both for the prescheduled processes (the ith iteration of DoAll is executed on the processor (i mod P)+1, and the serial code between two loops is always on the same processor) and for the randomly scheduled processes (the processor that executes an instance of an epoch is randomly selected). The following performance measures were shown: (a) miss ratio, (b) write traffic, and (c) network traffic. All of these performance measures were viewed as functions of the number of processors in the system.

For the analyzed applications, the miss ratio was about the same for both schemes for any number of processors. In general, the miss ratio is an increasing function of the number of processors because the larger the number of processors, the greater the probability that one processor writes to a location between two successive reads from the same location by another processor. However, the larger the number of processors, the lower the probability of benefiting from interiteration locality because the iterations are spread over a larger number of processors. For two out of three analyzed applications, the miss ratio is considerably higher if the iterations are randomly distributed because the ability to reuse the cache contents outside the epoch boundary is lost. The opposite behavior was noticed with just one out of three applications. The write traffic is considerably higher with the software scheme (compared to the hardware scheme) and is fairly independent of the number of processors and the process scheduling policy. The relatively large write traffic of the software scheme is a consequence of the write-back of dirty lines at the end of each instance of one epoch. The network traffic is larger with the hardware scheme because it is directory-based. Also, the network traffic increase as a function of the number of processors is faster with the hardware scheme (compared to the software scheme). This is because the traffic increase is related to the inductive invalidation typical of hardware schemes in general, whereas the software schemes are characterized by self-invalidation, which has no impact on the network traffic. With software schemes, the increase of the network traffic is minimal when the number of processors goes up. This is because the write traffic and write misses have a relatively small impact on the network traffic.

In [TARTA92], the authors presented the results of their study of the impact of the restrictiveness of invalidation conditions on processing power. They used a modified Archibald–Baer model of the probabilistically synthesized workload. The modifications consist of the introduction of (a) new parameters that model the spatial and the processor locality as well as (b) an operating system model that is rather simplified, but adequate for this purpose. The authors first propose a class of schemes referred to as the selective and conditional self-invalidation class (SCSIC). The schemes of this class differ only in the restrictiveness of the invalidation conditions. The schemes are compared on a realistic multiprocessor simulator having a 32-bit bus and a set-associative cache of a relatively small size with several words per cache line. The selected performance measures are (a) the average number of invalidations during the passage through a critical region, (b) cache hit ratio, and (c) the processing power. All performance measures were treated as functions of the number of processors in the system. An important parameter in the entire analysis is the probability that, in a given passage through the critical region, the shared data are only read (Pr-o). This parameter has an important impact on the overall performance of all SCSIC schemes.

The preliminary results of the simulation analysis demonstrate the advantages of the restrictively applied invalidation, even in systems with only 16 processors. The advantage is higher when the value of the parameter (*Pr*-o) is higher, which is obvious. However, what is not obvious is the quantitative value of the real advantage. When the number of processors is as low as 16 and the value of *Pr*-o is equal to 0.95, the advantage is more than 5 percent for the scheme with the most restrictive invalidation (verification of the segment version) compared with a similar, OTI-like scheme. For a much larger number of processors, which is more realistic for future systems, the advantage is expected to be considerably higher.

In [OWICK89], Owicki and Agarwal used the analytical MVA model for performance comparison of four representative schemes for the maintenance of cache consistency. The chosen schemes include the following: (a) Base—a scheme with no consistency maintenance, which defines the upper limit for performance; (b) No-cache—a very conservative scheme that does not allow the caching of pages with shared data (this scheme corresponds to the scheme proposed for the C.mmp [WULF72]); (c) Software-flush—a more sophisticated scheme based on static program analysis and insertion of the flush instruction (this scheme corresponds to the scheme from [CYTRO88] and [CHEON88]); and (d) Dragon [ATKIN84]—one of the best hardware schemes based on snooping and write-broadcast. The developed analytical model includes the system model, the workload model, and the contention model. The system model defines the number of cycles required for each given operation in hardware. The workload model defines the frequencies (probabilities) of specific actions. For example, *ls* refers to the probability that the instruction is a load or store, *shd* refers to the probability that load or store use shared data, and *apl* refers to the number of references to a shared data item before it is flushed from the cache using the flush instruction. The contention model determines the additional time spent due to the contention on the common bus or the multistage interconnection network. The basic performance measure used in this research is processing power. It is viewed as a function of the number of processors in the system. For each relevant parameter, three values are used: low, medium, and high. If the impact of one parameter is studied, the value of that parameter is varied from low to high, while the values of all other parameters are kept at the medium value.

For the common bus case, the analysis has shown that the No-cache scheme has the worst performance and the Dragon scheme the best. Also, it was shown that the software schemes are relatively sensitive to the changing values of the *ls*, *shd*, and *apl* parameters. For low values of parameters *ls* and *shd*, the performance of the Software-flush scheme is comparable with that of the Dragon scheme. For medium and high values of these parameters, the Dragon scheme has an advantage. The software schemes show an increased sensitivity to the *apl* parameter. For relatively high values of this parameter (*apl* = 25) the performance of the Software-flush scheme is even better than the performance of the Dragon scheme. For the multistage interconnection network case, the analysis has shown that the processing power of the software schemes increases linearly with the number of processors, which is an indication of the high level of scalability of these schemes.

In [ADVE91], the authors compared the directory-based hardware schemes with

static software schemes. Their applied analytical method starts from a general program behavior model, which specifies the access dynamics for different classes of shared data. The following classes of shared data have been observed: (a) passively shared objects (the shared read-only data, and the shared read-write data actually accessed by one processor only), (b) mostly-read objects, (c) frequently read-written objects, (d) migratory objects (the data being accessed within critical regions), and (e) synchronization objects (for example, the lock of the critical regions). The model includes high-level parameters and low-level parameters. The high-level parameters include (a) the fraction of memory accesses related to data references (f_{data}), (b) the fraction of data references related to private data (f_{pvt}), (c) the fraction of shared references related to a particular class of data (f_{PS}, f_{MR}, f_{RW}, f_{MIG}), (d) the fraction of accesses to mostly-read ($f_{w/MR}$) or read-written ($f_{w/RW}$) data that are writes, (e) the runtime average number of read accesses by a processor to a mostly-read data element between consecutive compiler-inserted invalidations executed on that element by the same processor (l_{MR}), (f) the average number of read accesses by a processor to a frequently read-written data element between potential writes by other processors (l_{RW}), (g) the mean number of processors that access a data element of appropriate data class between consecutive actual writes to that element (n_{MR} and n_{RW}), and so on. These high-level parameters are independent of the cache consistency scheme, describe the workload, and enable different classes of data to be analyzed independently of each other with respect to their impact on the efficiency of a given scheme. The low-level parameters for the hardware scheme in [AGARW88] include (a) the fraction of references that are read misses or write misses to lines in shared ($p_{r/sh}$ or $p_{w/sh}$) or modified ($p_{r/mod}$ or $p_{w/mod}$) state, (b) the probability that invalidations are sent individually ($p_{ind.inv}$), and (c) the average number of processors to which individual invalidations are sent (n_{inv}). The low-level parameters for the software scheme in [CYTRO88] include (a) the fraction of references that are read (p_r) or write (p_w) misses and (b) the fraction of references that are posts (p_{post}) or invalidates (p_{inv}). All low-level parameters used in direct comparison of different schemes are derived from the general high-level model, and therefore the comparisons are expected to be fair. These low-level parameters serve as input data into the approximate MVA model of the scheme being analyzed. In this study, the basic performance measure is the processor efficiency. It is defined as the "average fraction of time each processor spends executing locally out of its cache."

For different classes of data, the paper gives the contours of the constant ratio of software to hardware efficiency method. From these contours, one can easily identify the parameter domains where the software scheme exhibits an advantage in comparison with the hardware scheme, and vice versa. For the class of migratory data, the software scheme is always more efficient. For the class of mostly-read objects, the software scheme is slightly better only for very small values (close to 1) of the average number of compiler-inserted invalidates done by a processor in the interval between two successive actual writes into the given data item, averaged over the intervals when the processor does execute such invalidates ($ratio_{MR}$). Larger values of the parameter $ratio_{MR}$ correspond to a larger number of unrealized potential writes that cause unnecessary invalidations. For larger values of the $ratio_{MR}$ parameter ($ratio_{MR} \geq 3$), a relatively small value of the parameter l_{MR} ($l_{MR}=1$), and a fraction of the mostly-read objects $f_{MR}>0.2$, the hardware scheme is more 20 percent better than the software scheme. However, for a larger value of the parameter l_{MR} ($l_{MR} \geq 8$), the hardware scheme is less

than 20 percent better than the software scheme for almost all values of $ratio_{MR}$ and f_{MR}<0.85; therefore, the software scheme is comparable in performance with the hardware scheme for a much lower production cost. In general, the software scheme is relatively sensitive to changes of the parameter l_{MR}. On the other hand, the hardware scheme is sensitive to the changes of the parameters $f_{w/MR}$ and n_{MR}. For the frequently read-written data class with the expected fraction in real applications f_{RW}<0.2, it was concluded that the software scheme is also comparable with the hardware scheme because it is within the 20 percent range of the hardware scheme performance. Finally, the paper deals with an impact of reduction in the hit ratio (due to the conservative analysis of memory conflicts typical of static software schemes) on the software/hardware scheme efficiency ratio. If the values of all relevant parameters are set in such a way that the hardware and the software scheme are equal in performance, the following interesting impact of the hit ratio reduction is observed. If the hit ratio reduction is more than 15 percent, the hardware scheme is more than 20 percent better in performance. If the reduction of the hit ratio is 10 percent, the hardware scheme is about 10 percent better in performance. However, if the hit ratio reduction is less than 5 percent, and if most of the potential writes are actually executed, the performance advantage of the software scheme goes up to about 20 percent.

It should be underlined here that the study by Adve et al. [ADVE91] uses the term "software schemes" to mean only the static software schemes. However, dynamic software schemes merit attention as well, for a number of reasons. For example, the imprecise prediction of memory conflicts and potential writes at compile time, which can significantly degrade the performance of static schemes, is not an issue with the dynamic schemes. Also, in the case of nonnumeric applications characterized by tasks communicating through exclusive usage of shared variables inside critical regions—where dynamic software schemes (as [SMITH85] or [TARTA92]) offer a natural solution by virtue of the dominance of migratory data objects—a relatively good performances can be expected.

7. Conclusions

This paper represents an effort to encompass and classify the existing work in the field of software-based algorithms for the maintenance of cache consistency in shared-memory multiprocessor systems. A set of 10 classification criteria is proposed. Each criterion results in several classes. These classes serve as attribute values for description and classification of existing approaches.

The basic criterion, the dynamism of the approach, is used to divide the surveyed schemes into two essential groups: (a) static, which do the program analysis for detection of inconsistency conditions at compile time, and (b) dynamic, which detect the inconsistency conditions at run time.

After the field is broadly surveyed and the correspondence between schemes and attributes is established, a generalization of the proposed classification is introduced. It is based on an N-dimensional criterion space. Each scheme that is essentially different from other existing schemes corresponds to a point in that space. This generalization allows the following to be accomplished:

- With the state of the art in the field, it can be detected that some solutions are essentially the same (because they belong to the same point in the criterion space), although they are declared as formally different (because they have been introduced independently of each other in time and space).
- With the development of the field, it can be determined if a set of criteria (and related classes) is defined widely enough so that essentially different solutions can be associated with different points in the space.
- With respect to future contributions in the field, the "empty" points in the criterion space can serve as guides toward new solutions (as was the case with the Mendeleev periodic classification system in chemistry).

Through a careful inspection of the criterion space, one can recognize that a part of the criterion space that corresponds to static and adaptive schemes is "empty" at the time. This means that some potentially good new solutions can be found in this part of the criterion space, thus encouraging research in that direction.

Finally, the paper gives a survey of performance evaluation studies in the field of software-based consistency maintenance. Representative evaluation techniques and their basic results have been described in two different domains: (a) simulation analysis based on the real address traces or the synthetic workload models, and (b) analytical evaluation based on MVA models.

The presented studies also yield good insight into the real performance differences between software and hardware schemes for different values of technology- and application-related parameters. One especially important conclusion is that in several applications (mostly, but not only, those characterized by migratory data) software schemes demonstrate better performance. It is expected that in real implementations, this performance advantage may be even slightly higher because the complexity of hardware support for software schemes is relatively low. Consequently, VLSI systems that utilize software schemes potentially have a slightly better internal timing, and operate with a slightly faster system clock.

8. References

[ADVE91] Adve, S.V., et al., "Comparison of Hardware and Software Cache Coherence Schemes," *Proc. 18th Ann. Int'l Symp. Computer Architecture,* ACM Press, New York, N.Y., 1991, pp. 298–308.

[AGARW88] Agarwal, A., et al., "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 280-289.

[ARCHI86] Archibald, J., and Baer, J.-L., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems,* Vol. 4, No. 4, Nov. 1986, pp. 273–298.

[BENN90a] Bennett, J.K., Carter, J.B., and Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. 1990 Conf. Principles and Practice of Parallel Programming,* 1990, pp. 168–176.

[BENN90b] Bennett, J.K., Carter, J.B., and Zwaenepoel, W., "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1990, pp. 125–134.

[BRANT85] Brantley, W.C., McAuliffe, K.P., and Weiss, J., "RP3 Processor-Memory Element," *Proc. 1985 Int'l Conf. Parallel Processing,* IEEE CS Press, Los Alamitos, Calif., 1985, pp. 782–789.

[CENSI78] Censier, L.M., and Feautrier, P., "A New Solution to Coherence Problem in Multicache Systems," *IEEE Trans. Computers,* Vol. C-27, No. 12, Dec. 1978, pp. 1112–1118.

[CHERI86] Cheriton, D.R., Slavenburg, G.A., and Boyle, P.D., "Software-Controlled Caches in the VMP Multiprocessor," *Proc. 13th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1986, pp. 366–374.

[CHEON88] Cheong, H., and Veidenbaum, A.V., "A Cache Coherence Scheme with Fast Selective Invalidation," *Proc. 15th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1988, pp. 299–307.

[CHEON89] Cheong, H., and Veidenbaum, A.V., "A Version Control Approach to Cache Coherence," *Proc. Int'l Conf. Supercomputing 89,* ACM Press, New York, N.Y., 1989, pp. 322–330.

[CHEON90] Cheong, H., and Veidenbaum, A.V., "Compiler-Directed Cache Management in Multiprocessors," *Computer,* Vol. 23, No. 6, June 1990, pp. 39–47.

[CHEON92] Cheong, H., "Life Span Strategy—A Compiler-Based Approach to Cache Coherence," *Proc. 1992 Int'l Conf. Supercomputing,* ACM Press, New York, N.Y., 1992, pp. 139–148.

[CHIUE93] Chiueh, T.-C., "A Generational Algorithm to Multiprocessor Cache Coherence," *Proc. 1993 Int'l Conf. Parallel Processing,* Vol. 1, CRC Press, Boca Raton, Fla., 1993, pp. 20–24.

[CYTRO88] Cytron, R., Karlovsky, S., and McAuliffe, K.P., "Automatic Management of Programmable Caches," *Proc. 1988 Int'l Conf. Parallel Processing,* Penn State Press, University Park, Pa., 1988, pp. 229–238.

[DAHLG95] Dahlgren, F., Skeppstedt, J., and Stenström, P., "Effectiveness of Hardware-Based and Compiler-Controlled Snooping Cache Protocol Extensions," *Proc. Int'l Conf. High-Performance Computing,* 1995, (to appear).

[DARNE93] Darnell, E., and Kennedy, K., "Cache Coherence Using Local Knowledge," *Proc. Supercomputing '93,* ACM Press, New York, N.Y., 1993, pp. 720–729.

[EDLER85] Edler, J., et al., "Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach," *Proc. 12th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1985, pp. 126–135.

[EGGER89] Eggers, S.J., and Katz, R.H., "Evaluating the Performance of Four Snooping Cache Coherency Protocols," *Proc. 16th Ann. Int'l Symp. Computer Architecture,* ACM Press, New York, N.Y., 1989, pp. 2–15.

[GOTTL82] Gottlieb, A., et al., "The NYU Ultracomputer—Designing a MIMD, Shared-Memory Parallel Machine (Extended abstract)," *Proc. 9th Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1982, pp. 27–42.

[GOTTL83] Gottlieb, A., et al., "The NYU Ultracomputer—Designing an MIMD, Shared Memory Parallel Computer," *IEEE Trans. Computers,* Vol. C-32, No. 2, Sept. 1983, pp. 175–189.

[LEE87a] Lee, R.L., Yew, P.-C., and Lawrie, D.H., "Multiprocessor Cache Design Considerations," *Proc. 14th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, N.Y., 1987, pp. 253–262.

[LEE87b] Lee, R.L., "The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors," PhD thesis, TR 670, Center of Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, Aug. 1987.

[LILJA93] Lilja, D., "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons," *ACM Computing Surveys*, Vol. 25, No. 3, Sept. 1993, pp. 303–338.

[LOURI92] Louri, A., and Sung, H., "A Compiler Directed Cache Coherence Scheme with Fast and Parallel Explicit Invalidation," *Proc. 1992 Int'l Conf. Parallel Processing,* Vol. 1, CRC Press, Boca Raton, Fla., 1992, pp. 2–9.

[MCAUL86] McAuliffe, K., "Analysis of Cache Memories in Highly Parallel Systems," PhD Thesis, TR 269, Courant Institute of Mathematical Sciences, New York University, May 1986.

[MIN89] Min, S.L., and Baer, J.-L., "A Timestamp-Based Cache Coherence Scheme," *Proc. Int'l Conf. Parallel Processing*, Vol. 1, Penn State Press, University Park, Pa., 1989, pp. I23–I32.

[MIN90] Min, S.L., and Baer, J.-L., "A Performance Comparison of Directory-Based and Timestamp-Based Cache Coherence Schemes," *Proc. 1990 Int'l Conf. Parallel Processing,* Vol. I, Penn State Press, University Park, Pa., 1990, pp. 305–311.

[MIN92] Min, S.L., and Baer, J.-L., "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps," *IEEE Trans. Parallel and Distributed Systems,* Vol. 3, No. 1, Jan. 1992, pp. 25–44.

[OWICK89] Owicki, S., and Agarwal, A., "Evaluating the Performance of Software Cache Coherence," *Proc. 3rd Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, N.Y., 1989, pp. 230–242.

[PFIST85] Pfister, G.F., et.al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 Parallel Processing Conf.,* IEEE CS Press, Los Alamitos, Calif., 1985, pp. 764–771.

[PROTI95] Protić, J., Tomašević, M., and Milutinović, V.," A Survey of Distributed Shared Memory Systems," *Proc. 28th Ann. Hawaii Int'l Conf. System Sciences,* Vol. 1, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 74–84.

[SKEPP95] Skeppstedt, J., and Stenström, P., "A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* 1995, pp. 69–78.

[SMITH85] Smith, A.J., "CPU Cache Consistency with Software Support and Using One Time Identifiers," *Proc. Pacific Computer Communication Symp.,* 1985, pp. 142–150.

[STENS90] Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer,* Vol. 23, No. 6, June 1990, pp. 12–24.

[TARTA92] Tartalja, I., and Milutinović, V., "An Approach to Dynamic Software Cache Consistency Maintenance Based on Conditional Invalidation," *Proc. 25th Ann. Hawaii Int'l Conf. System Sciences,* Vol. 1, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 457–466.

[TOMAŠ93] Tomašević, M., and Milutinović, V., *The Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions,* IEEE CS Press, Los Alamitos, Calif., 1993.

[TOMA94a] Tomašević, M., and Milutinović, V., "Hardware Approaches to Cache Coherence Problem in Shared-Memory Multiprocessors, Part 1," *IEEE Micro*, Vol. 14, No. 5, Oct. 1994, pp. 52–59.

[TOMA94b] Tomašević, M., and Milutinović, V., "Hardware Approaches to Cache Coherence Problem in Shared-Memory Multiprocessors, Part 2," *IEEE Micro*, Vol. 14, No. 6, Dec. 1994, pp. 61–66.

[VEIDE86] Veidenbaum, A.V., "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *Proc. 1986 Int'l Conf. Parallel Processing,* IEEE CS Press, Los Alamitos, Calif., 1986, pp. 1029–1036.

[WULF72] Wulf, W.A., and Bell, C.G., "C.mmp—A Multi-Mini Processor," *Proc. Fall Joint Computer Conf.,* IEEE Press, New York, NY, 1972, pp. 765–777.

[YEH83] Yeh, P.C.C., Patel, J.H., and Davidson, E.S., "Shared Cache for Multiple-Stream Computer Systems," *IEEE Trans. Computers,* Vol. C-32, No. 1, Jan. 1983, pp. 38–47.