A Survey of Cache Coherence Schemes for Multiprocessors

Per Stenström

Lund University

Shared-memory multiprocessors have emerged as an especially cost-effective way to provide increased computing power and speed, mainly because they use low-cost microprocessors economically interconnected with shared memory modules.

Figure 1 shows a shared-memory multiprocessor consisting of processors connected with the shared memory modules by an interconnection network. This system organization has three problems[1]:

(1) Memory contention. Since a memory module can handle only one memory request at a time, several requests from different processors will be serialized.

(2) Communication contention. Contention for individual links in the interconnection network can result even if requests are directed to different memory modules.

(3) Latency time. Multiprocessors with a large number of processors tend to have complex interconnection networks. The latency time for such networks (that is, the time a memory request takes to traverse the network) is long.

These problems all contribute to increased memory access times and hence slow down the processors' execution speeds.

Cache memories have served as an important way to reduce the average memory access time in uniprocessors. The locality of memory references over time (*temporal locality*) and space (*spatial locality*) allows the cache to perform a vast majority of all memory requests (typically more than 95 percent); memory handles only a small fraction. It is therefore not surprising that multiprocessor architects also have employed cache techniques to address the problems pointed out above. Figure 2 shows a multiprocessor organization with caches attached to all processors. This cache organization is often called *private*, as opposed to *shared*, because each cache is private to one or a few of the total number of processors.

Cache coherence schemes tackle the problem of maintaining data consistency in shared-memory multiprocessors. They rely on software, hardware, or a combination of both.

The private cache organization appears in a number of multiprocessors, including Encore Computer's Multimax, Sequent Computer Systems' Symmetry, and Digital Equipment's Firefly multiprocessor workstation. These systems use a common bus as the interconnection network. Communication contention therefore becomes a primary concern, and the cache serves mainly to reduce bus contention.

Other systems worth mentioning are RP3 from IBM, Cedar from the University of Illinois at Urbana-Champaign, and Butterfly from BBN Laboratories. These systems contain about 100 processors connected to the memory modules by a multistage interconnection network with a considerable latency. RP3 and Cedar also use caches to reduce the average memory access time.

Shared-memory multiprocessors have an advantage: the simplicity of sharing code and data structures among the processes comprising the parallel application. Process communication, for instance, can be implemented by exchanging information through shared variables. This sharing can result in several copies of a shared block in one or more caches at the same time. To maintain a coherent view of the memory, these copies must be consistent. This is the *cache coherence problem* or the *cache consistency problem.* A large num-

0018-9162/90/0600-0012$01.00 © 1990 IEEE

ber of solutions to this problem have been proposed.

This article surveys schemes for cache coherence. These schemes exhibit various degrees of hardware complexity, ranging from protocols that maintain coherence in hardware to software policies that prevent the existence of copies of shared, writable data. First we'll look at some examples of how shared data is used. These examples help point out a number of performance issues. Then we'll look at hardware protocols. We'll see that consistency can be maintained efficiently, but in some cases with considerable hardware complexity, especially for multiprocessors with many processors. We'll investigate software schemes as an alternative capable of reducing the hardware cost.

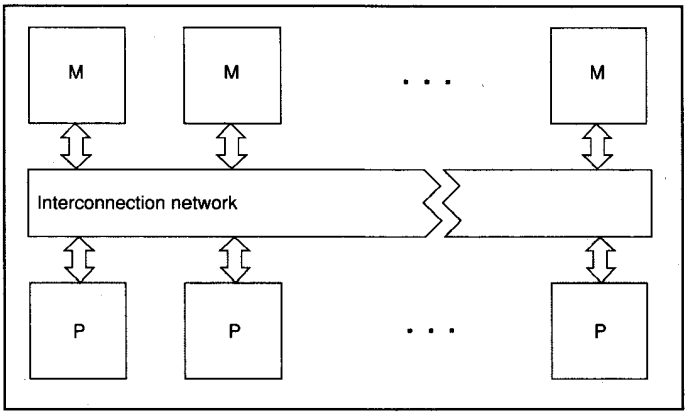

Figure 1. An example of a shared memory multiprocessor.[1]

Example of algorithms with data sharing

Cache coherence poses a problem mainly for shared, read-write data structures. Read-only data structures (such as shared code) can be safely replicated without cache coherence enforcement mechanisms. Private, read-write data structures might impose a cache coherence problem if we allow processes to migrate from one processor to another. Many commercial multiprocessors help increase throughput for multiuser operating systems where user processes execute independently with no (or little) data sharing. In this case, we need to efficiently maintain consistency for private, read-write data in the context of process migration.

We will concentrate on the behavior of cache coherence schemes for parallel applications using shared, read-write data structures. To understand how the schemes work and how they perform for different uses of shared data structures, we will investigate two parallel applications that use shared data structures differently. These examples highlight a number of performance issues.

We can find the first example — the well-known bounded-buffer producer and consumer problem — in any ordinary text on operating systems. Figure 3 shows it in a Pascal-like notation. The producer inserts a data item in the shared buffer if the buffer is not full. The buffer can store *N* items. The consumer removes an item if the buffer is not empty. We can choose the number of producers and consumers arbitrarily. The buffer is managed by a shared array, which implements the buffer, and

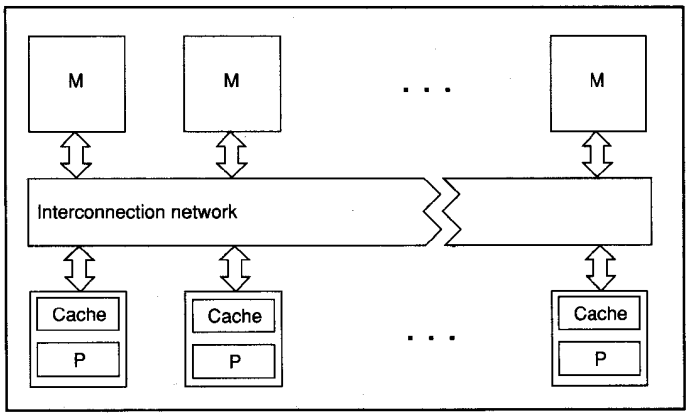

Figure 2. An example of a multiprocessor with private caches.[1]

```
Producer:                          Consumer:
  if count <= N then                 if count <> 0 then
  mutexbegin                         mutexbegin
    buffer[ in ] := item;              item := buffer[ out ];
    in := in + 1 mod N;                out := out + 1 mod N;
    count := count + 1;                count := count - 1;
  mutexend                           mutexend
```

Figure 3. Pascal-like code for the bounded-buffer problem.

three shared variables: *in*, *out*, and *count*, which keep track of the next item and the number of items stored in the buffer. Semaphores (implemented by mutexbegin and mutexend) protect buffer operations, which means that one process at most can enter the critical section at a time.

The second example to consider is a

```
repeat
   par_for J := 1 to N do
   begin
      xtemp[ J ] := b[ J ];
      for K := 1 to N do
         xtemp[ J ] := xtemp[ J ] + A[ J,K ] * x[ K ];
   end;
   barrier_sync;
   par_for J := 1 to N do
      x[ J ] := xtemp[ J ];
   barrier_sync;
until false;
```

Figure 4. Pascal-like code for one iteration of the parallel algorithm for solving a linear system of equations by iteration.

parallel algorithm for solving a linear system of equations by iteration. It takes the form

$$\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i + \mathbf{b}$$

where $\mathbf{x}_{i+1}$, $\mathbf{x}_i$, and **b** are vectors of size N and **A** is a matrix of size $N \times N$. Suppose that each iteration (the calculation of vector $\mathbf{x}_{i+1}$) is performed by N processes, where each process calculates one vector element. The code for this algorithm appears in Figure 4. The termination condition does not concern us here. Therefore, we assume that it never terminates.

The par_for statement initiates N processes. Each process calculates a new value, which is stored in **xtemp**. The last parallel loop in the iteration copies back the elements of **xtemp** to vector **x**. This requires a barrier synchronization. The most important observations are

(1) Vector **b** and matrix **A** are read-shared and can be safely cached.

(2) All elements of vector **x** are read to calculate a new vector element.

(3) All elements of vector **x** are updated in each iteration.

With these examples in mind, we will consider how the proposed schemes for cache coherence manage copies of the data structures.

Proposed solutions range from hardware-implemented cache consistency protocols, which give software a coherent view of the memory system, to schemes providing varied hardware support but with cache coherence enforcement policies implemented in software. We will focus on the implementation cost and performance issues of the surveyed schemes.

Hardware-based protocols

Hardware-based protocols include snoopy cache protocols, directory schemes, and cache-coherent network architectures. They all rely on a certain cache coherence policy. Let's start to look at different policies.

Table 1. Comparison of the number of consistency actions generated by the cache coherence policies for the example algorithms.

Communication Cost		Bounded-Buffer Problem	Iterative Algorithm
Write-invalidate	Invalidations	1	N
	Misses	1	N
Write-update	Updates	K	N

Cache coherence policies. Hardware-based protocols for maintaining cache coherence guarantee memory system coherence without software-implemented mechanisms. Typically, hardware mechanisms detect inconsistency conditions and perform actions according to a hardware-implemented protocol.

Data is decomposed into a number of equally sized blocks. A block is the unit of transfer between memory and caches. Hardware protocols allow an arbitrary number of copies of a block to exist at the same time. There are two policies for maintaining cache consistency: *write-invalidate* and *write-update*.

The write-invalidate policy maintains consistency of multiple copies in the following way: Read requests are carried out locally if a copy of the block exists. When a processor updates a block, however, all other copies are invalidated. How this is done depends on the interconnection network used. (Ignore it for the moment.) A subsequent update by the same processor can then be performed locally in the cache, since copies no longer exist. Figure 5 shows how this policy works. In Figure 5a, four copies of block X are present in the system (the memory copy and three cached copies). In Figure 5b, processor 1 has updated an item in block X (the updated block is denoted X') and all other copies are invalidated (denoted I). If processor 2 issues a read request to an item in block X', then the cache attached to processor 1 supplies it.

The write-update policy maintains consistency differently. Instead of invalidating all copies, it updates them as shown in Figure 5c. Whether the memory copy is updated or not depends on how this protocol is implemented. We will look at that later.

Consider the write-invalidate policy for the bounded-buffer problem, recalling the code in Figure 3. Suppose a producer process P and a consumer process C, executing on different physical processors, alternately enter the critical section in the following way: P enters the critical section K times in a row, then C enters the critical section K times in a row, and so forth. If K=1, then *count* will be read and written by P and C, then P again, etc. This means there will be a miss on the read, then an invalidation on the write. Referred to as the *ping-pong effect*, this means data migrates back and forth between the caches, resulting in heavy network traffic. However, if the producer process inserts consecutive items in the buffer — that is, if K>1 — then the

reads and writes to *count* will be local. The same holds for the consumer process.

Now consider the write-update policy applied to the bounded-buffer problem. Here, note that the write to *count* generates a global update independent of the order of execution of P and C. Table 1 shows the communication cost associated with accesses to the variable *count* for K consecutive executions of the critical section. Under the assumption that the communication cost is the same for an invalidation as for an update and that the communication cost for a miss is twice that for an invalidation, then the break-even point of the communication cost between the two policies is K=3.

Now consider the iterative algorithm of Figure 4 and the write-invalidate protocol. Suppose the block size is exactly one vector element and the cache is infinitely large. Observe first that accesses to matrix **A** and vector **b** will be local, since they are read-shared and will not be invalidated. However, each process will realize a read miss on every access to vector **x**, since all elements of **x** are updated (that is, all copies are invalidated) in each iteration. Each process generates exactly one invalidation. Thus, each process will have N read misses and N invalidations in each iteration.

If we instead consider the write-update policy, then all reads will be local but N global updates will be generated for each process. These observations are summarized in Table 1. Write-update performs better in terms of communication cost than does write-invalidate for this algorithm, with the same assumptions as for the bounded-buffer problem.

The write-invalidate and write-update policies require that cache invalidation and update commands (collectively referred to as *consistency commands*) be sent to at least those caches having copies of the block. Until now we have not considered the implications of this for different networks. In some networks (such as buses), it is feasible to broadcast consistency commands to all caches. This means that every cache must process every consistency command to find out whether it refers to data in the cache. These protocols are called *snoopy cache protocols* because each cache "snoops" on the network for every incoming consistency command.

In other networks (such as multistage networks), the network traffic generated by broadcasts is prohibitive. Such systems prefer to multicast consistency commands exactly to those caches having a copy of the block. This requires bookkeeping by

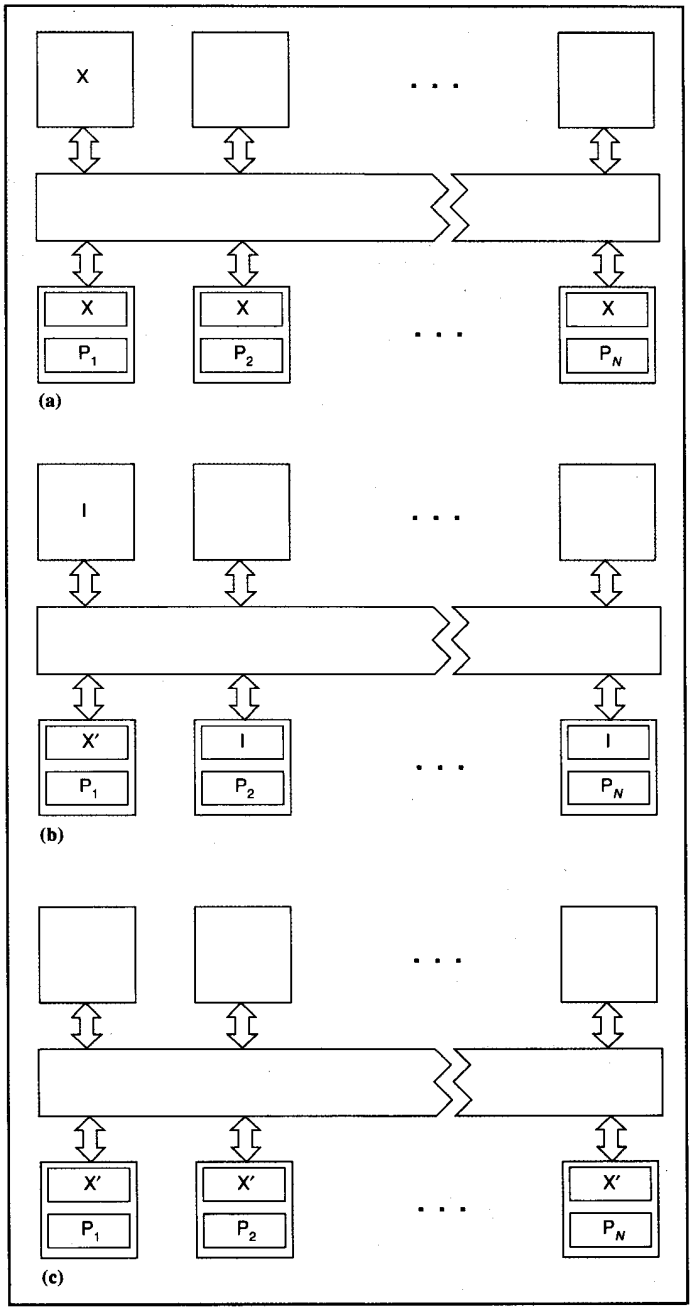

Figure 5. (a) Memory and three processor caches store consistent copies of block X. (b) All copies except the one stored in processor 1's cache are invalidated (I) when processor 1 updates X (denoted X') if the write-invalidate policy is used. (c) All copies (except the memory copy, which is ignored) are updated if the write-update policy is used.

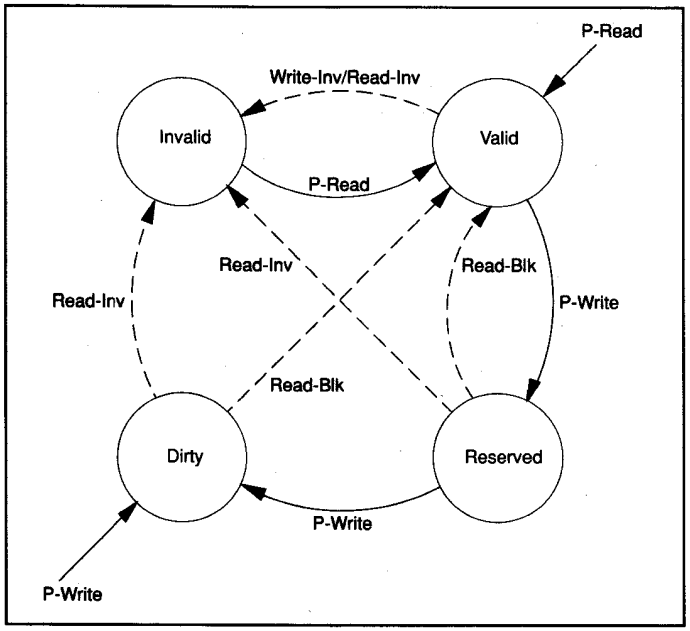

Figure 6. State-transition graph for states of cached copies for the write-once protocol. Solid lines mark processor-initiated actions, and dashed lines mark consistency actions initiated by other caches.

means of a directory that tracks all copies of blocks. Hence, these protocols are called *directory schemes*.

First we'll look at different implementations of snoopy cache protocols. Then we'll look at directory schemes. While snoopy cache protocols rely on the use of buses, directory schemes can be used for general interconnection networks. Past work has also yielded proposals for cache-coherent network architectures supporting a large number of processors.

Write-invalidate snoopy cache protocols. Historically, Goodman proposed the first write-invalidate snoopy cache protocol, called *write-once* and reviewed by Archibald and Baer.[2] To understand the hardware complexity of the reviewed protocols, and certain possible optimizations, we will take a rather detailed look at this protocol.

The write-once protocol associates a state with each cached copy of a block. Possible states for a copy are

- Invalid. The copy is inconsistent.
- Valid. There exists a valid copy consistent with the memory copy.
- Reserved. Data has been written exactly once and the copy is consistent with the memory copy, which is the only other copy.
- Dirty. Data has been modified more than once and the copy is the only one in the system.

Write-once uses a copy-back memory update policy, which means that the entire copy of the block must be written back to memory when it is replaced, provided that it has been modified during its cache residence time (that is, the state is dirty). To maintain consistency, the protocol requires the following consistency commands besides the normal memory read block (Read-Blk) and write block (Write-Blk) commands:

- Write-Inv. Invalidates all other copies of a block.
- Read-Inv. Reads a block and invalidates all other copies.

State transitions result either from the local processor read and write commands (P-Read and P-Write) or the consistency commands (Read-Blk, Write-Blk, Write-Inv, and Read-Inv) incoming from the global bus. Figure 6 shows a state-transition graph summarizing the actions taken by the write-once protocol. Solid lines mark processor-initiated actions, and dashed lines mark consistency actions initiated by other caches and sent over the bus.

The operation of the protocol can also be specified by making clear the actions taken on processor reads and writes. Read hits can always be performed locally in the cache and do not result in state transitions. For read misses, write hits, and write misses the actions occur as follows:

- Read miss. If no dirty copy exists, then memory has a consistent copy and supplies a copy to the cache. This copy will be in the valid state. If a dirty copy exists, then the corresponding cache inhibits memory and sends a copy to the requesting cache. Both copies will change to valid and the memory is updated.
- Write hit. If the copy is in the dirty or reserved states, then the write can be carried out locally and the new state is dirty. If the state is valid, then a Write-Inv consistency command is broadcast to all caches, invalidating their copies. The memory copy is updated and the resulting state is Reserved.
- Write miss. The copy either comes from a cache with a dirty copy, which then updates memory, or from memory. This is accomplished by sending a Read-Inv consistency command, which invalidates all cached copies. The copy is updated locally and the resulting state is dirty.
- Replacement. If the copy is dirty, then it has to be written back to main memory. Otherwise, no actions are taken.

Other examples of proposed write-invalidate protocols include the Illinois protocol proposed by Papamarcos and Patel and the Berkeley protocol specifically designed for the SPUR (Symbolic Processing Using RISC) multiprocessor workstation at the University of California at Berkeley (reviewed by Archibald and Baer[2]). They improve on the management of private data (Illinois) and take into account the discrepancy between the memory and cache access times to optimize cache-to-cache transfers (Berkeley).

Write-update snoopy cache protocols. An example of a write-update protocol, the Firefly protocol, has been implemented in

the Firefly multiprocessor workstation from Digital Equipment (reviewed by Archibald and Baer[2]). It associates three possible states with a cached copy of a block:

- Valid-exclusive. The only cached copy, it is consistent with the memory copy.
- Shared. The copy is consistent, and there are other consistent copies.
- Dirty. This is the only copy. The memory copy is inconsistent.

The Firefly protocol uses copy-back update policy for private blocks and write-through for shared blocks. The notion of shared and private is determined at run-time.

To maintain consistency, a write-update consistency command updates all copies. A dedicated bus line, denoted "shared line," is used by the snooping mechanisms to tell the writer that copies exist. Figure 7 summarizes the state transitions.

The actions taken on a processor read or write follow:

- Read miss. If there are shared copies, then these caches supply the block by synchronizing the transmission on the bus. If a dirty copy exists, then this cache supplies the copy and updates main memory. The new state in these cases is shared. If there is no cached copy, then memory supplies the copy and the new state is valid-exclusive.
- Write hit. If the block is dirty or valid-exclusive, then the write can be carried out locally and the resulting state is dirty. If the copy is shared, all other copies (including the memory copy) are updated. If sharing has ceased (indicated by the shared line), then the next state is valid-exclusive.
- Write miss. The copy is supplied either from other caches or from memory. If it comes from memory, then its loaded-in state is dirty. Otherwise, all other copies (including the memory copy) are updated and the resulting state is shared.
- Replacement. If the state is dirty, then the copy is written back to main memory. Otherwise, no actions are taken.

Another write-update protocol, the Dragon protocol (reviewed by Archibald and Baer[2]), has been proposed for the Dragon multiprocessor workstation from Xerox PARC. To improve the efficiency of cache-to-cache transfers, it avoids updating memory until a block is replaced.

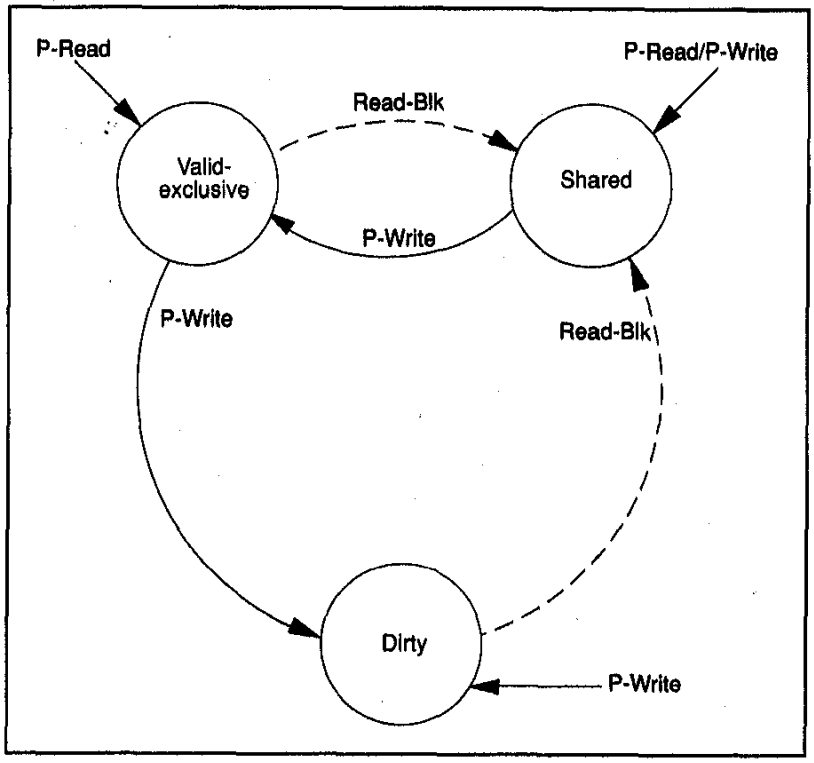

Figure 7. State-transition graph for states of cached copies for the Firefly protocol.

Implementation and performance issues for snoopy cache protocols. Snoopy cache protocols are extremely popular because of the ease of implementation. Many commercial, bus-based multiprocessors have used the protocols we have investigated here.[1] For example, Sequent Computer Systems' Symmetry multiprocessor and Alliant Computer Systems' Alliant FX use write-invalidate policies to maintain cache consistency. The DEC Firefly uses the write-update policy, as does the experimental Dragon workstation mentioned above.

The main differences between a snoopy cache and a uniprocessor cache are the cache controller, the information stored in the cache directory, and the bus controller. The cache controller is a finite-state machine that implements the cache coherence protocol according to the state transition graphs of Figures 6 and 7.

The cache directory needs to store the state for each block. Only two bits are needed for the protocols we have reviewed. The bus controller implements the bus-snooping mechanisms, which must monitor every bus operation to discover whether an action is needed. Since the snooping mechanism must have access to the directory, contention for the directory can arise between local requests and requests coming in from the bus. For that reason the directory is often duplicated.

Another implementation issue concerns the bus design. To efficiently support the protocols reviewed here, certain bus lines are needed. One example we have seen is the shared line to support write-update policies. Therefore, dedicated bus standards have been proposed such as the IEEE Futurebus (IEEE standard P896.1).

Now let's discuss the impact of certain cache parameters on the performance of snoopy cache protocols. We would use snoopy cache protocols mainly to reduce bus traffic, with a secondary goal of reducing the average memory access time. An important question is how these metrics are affected by the block (line) size when using a write-invalidate protocol.

For uniprocessor caches, bus traffic and average access time mainly result from cache misses, that is, references to data that are not cache resident. Uniprocessor cache studies have demonstrated that the

Table 2. Bit overhead for proposed directory schemes.

Scheme	Overhead (No. of Bits)
Tang	CB
Censier	$M(B+N)$
Stenström	$C(B+N) + M \log_2 N$

miss ratio decreases when the block size increases. This results from the spatial locality of code, in particular, and for data. The miss ratio decreases until the block size reaches a certain point — the data pollution point — then it starts to increase. For larger caches the data pollution point appears at a larger block size.

Bus traffic per reference (in number of bus cycles) is proportional both to the miss ratio, M, and the number of words that must be transferred to serve a cache miss. If this number matches the block size, L, then average bus traffic per reference is $B = M L$. Hence, if the miss ratio decreases when the block size increases, bus traffic will not necessarily decrease. In fact, simulations have shown that bus traffic increases with block size for data references,[3] which suggests using a small block size in bus-based multiprocessors.

For write-invalidate protocols, a cache miss can result from an invalidation initiated by another processor prior to the cache access — an *invalidation miss*. Such misses increase bus traffic. Note that increasing the cache size cannot reduce invalidation misses. Eggers and Katz[4] have done extensive simulations based on parallel program traces (trace-driven simulation) to investigate the impact of block size on the miss ratio and bus traffic (see also their references to earlier work). One of their conclusions is that the total miss ratio generally exceeds that in uniprocessors. Moreover, it does not necessarily decrease when the block size increases, unlike uniprocessor cache behavior. This means that bus traffic in multiprocessors may increase dramatically when the block size increases.

We can explain the main results by using the example algorithms from Figures 3 and 4. Consider the bounded-buffer problem and the use of the shared array, buffer. If the line size matches the size of each item, then the consumer will experience an invalidation miss on every access, assuming that producers and consumers access the critical section alternately. Note that if the block size increases, the invalidation miss ratio remains the same but bus traffic increases. However, with a larger block size, consumers could benefit from a decreased miss ratio if the same consumer process accessed the critical section several times in a row.

For the iterative algorithm from Figure 4, increasing the block size reduces the miss ratio for accesses to vector **x**, since all elements of the block are accessed once. Accesses to vector **xtemp**, however, experience a higher miss ratio because each write to **xtemp** invalidates all copies. This means that, in the worst case, a read miss for **xtemp** results for each iteration in the inner loop.

Even if the spatial locality with respect to a process is high, this does not necessarily suggest a large block size. It depends on the effect of accesses by all processes sharing the block. For shared data usage where data are exclusively accessed by one process for a considerable amount of time, increasing the block size may reduce the invalidation miss ratio.

For write-update protocols, the block size is not an issue because misses are not caused by consistency-related actions. Moreover, the frequency of global updates does not depend on the block size. A potential problem, however, is that write-update protocols tend to update copies even if they are not actively used. Note that a copy remains in the cache until replaced, since write-update protocols never invalidate copies. This effect is more emphasized for large caches, which help multiprocessors reduce the miss ratio and the resulting bus traffic.

An important performance issue for write-invalidate policies concerns reducing the number of invalidation misses. For write-update policies, an important issue concerns reducing the sharing of data to lessen bus traffic. Now let's survey some extensions to the two types of protocols that address these issues.

Snoopy cache protocol extensions. The write-invalidate protocol may lead to heavy bus traffic caused by read misses resulting from iterations where one processor updates a variable and a number of processors read the same variable. This happens with the iterative algorithm shown in Figure 4. The number of read misses could be reduced considerably if, upon a read miss, the copy were distributed to all caches with invalid copies. In that case, all N read misses per iteration and per process could be eliminated for all processes less one. Such an extension to the read-invalidate protocol, called *read-broadcast*, was proposed by Rudolph and Segall.[5]

As noted for the write-update protocol, data items might be updated even if never accessed by other processors. This could happen with the bounded-buffer problem of Figure 3 if a consumer process migrates from one processor to another. In this case, parts of the buffer might remain in the old cache until replaced. This can take a very long time if the cache is large. While in the cache, the buffer generates heavy network traffic because of the broadcast updates.

One approach measures the break-even point when the communication cost (in terms of bus cycles for updating a block) exceeds the cost of handling an invalidation miss. Assuming that a miss costs twice as much as a global update, then the break-even point appears when two consecutive updates have taken place with no intervening local accesses. We could implement this scheme by adding two cache states that determine when the break-even point is reached. Karlin et al.[6] proposed and evaluated a number of such extensions, called *competitive snooping*, and Eggers and Katz evaluated the performance benefits of these extensions.[4]

Directory schemes. We have seen that even using large caches cannot entirely eliminate bus traffic because of the consistency actions introduced as a result of data sharing. This puts an upper limit on the number of processors that a bus can accommodate. For multiprocessors with a large number of processors — say, 100 — we must use other interconnection networks, such as multistage networks.

Snoopy cache protocols do not suit general interconnection networks, mainly because broadcasting reduces their performance to that of a bus. Instead, consistency commands should be sent to only those caches that have a copy of the block. To do that requires storing exact information about which caches have copies of all cached blocks.

We will survey different approaches proposed in the literature. Note that this issue can be considered orthogonal to the choice of cache coherence policy. Therefore, keep in mind that either write-invalidate or write-update would serve. Cache coherence protocols that somehow store information on where copies of blocks reside are called *directory schemes*. Agarwal et al. surveyed and evaluated directory

schemes in their work.[7]

The proposed schemes differ mainly in how the directory maintains the information and which information the directory stores. Tang proposed the first directory scheme (reviewed by Agarwal et al.[7]). He suggested a central directory containing duplicates of all cache directories. The information stored in the cache directory depends on the coherence policy employed. The main point is that the directory controller can find out which caches have a copy of a particular block by searching through all duplicates.

Censier and Feautrier proposed another organization of the directory (also reviewed by Agarwal et al.[7]). Associated with each memory block is a bit vector, called the *presence flag vector*. One bit for each cache indicates which caches have a copy of the block. Some status bits are needed, depending on the cache coherence policy used.

In an earlier work, I proposed a different way of storing the directory information.[8] Instead of associating the state information and the presence flag vector with the memory copy, this information is associated with the cached copy. Let's call this the Stenström scheme.

First we compare the implementation cost in terms of the number of bits needed to store the information. Given M memory blocks, C cache lines, N caches, and B bits for state information for each cache block, Table 2 shows the overhead for each scheme.

From Table 2 we see that the Tang scheme has the least overhead, provided that $C \leq M$. However, this scheme has two major disadvantages. First, the directory is centralized, which can introduce severe contention. Second, the directory controller must search through all duplicates to find which caches have copies of a block.

In the other schemes, state information is distributed over memory or cache modules, which reduces contention. Furthermore, for both schemes the presence flag vector stores the residency of copies, eliminating the need for the search associated with the Tang scheme. This simplification does not come for free. In the Censier scheme, overhead is proportional to memory size; in the Stenström scheme, it is proportional to cache size. The last scheme needs the identity of the current owner in memory. This requires an additional $\log_2 N$ bits. The bit overhead for both schemes is prohibitive for multiprocessors with a large number of processors because of the size of the presence flag vector.

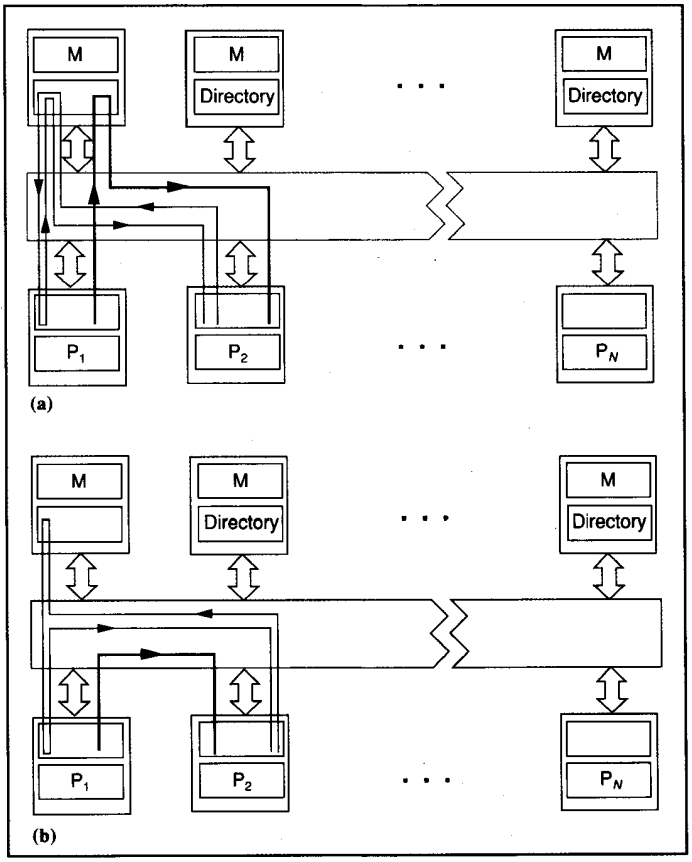

Figure 8. Actions taken on a read miss (thin lines) and a write hit (bold lines) for the write-invalidate implementation of (a) the Censier scheme and (b) the Stenström scheme.

To get an insight into the reduction of network traffic over snoopy cache protocols, assume that the directory organizations presented above support the write-invalidate cache coherence scheme. Since the Tang and Censier schemes differ only in the directory implementation, we will consider only the Censier and Stenström schemes.

Let's concentrate on how read misses and write hits are handled. Previous protocol descriptions have already shown how other actions are handled. In the following, assume the system contains exactly one dirty copy. Figure 8 shows the control flow of consistency actions.

In the Censier scheme, a read miss at cache 2 results in a request sent to the memory module. The memory controller retransmits the request to the dirty cache. This cache writes back its copy. The memory module can then supply a copy to the requesting cache. These actions appear in Figure 8a as thin lines. If a write hit is generated at cache 1, then a command is sent to the memory controller, which sends invalidations to all caches marked in the presence flag vector (cache 2) in Figure 8a. Bold lines mark these actions in Figure 8a.

Considering the Stenström scheme, a read miss at cache 2 results in a request sent to the memory module. The memory controller retransmits the request to the dirty cache. Instead of writing back its copy, the cache supplies the copy directly to cache 2. These actions appear in Figure 8b as thin lines. If a write hit is generated at cache 1, then invalidation commands are sent di-

Table 3. Cache pointer bit overhead for a full-map, limited, and chained directory scheme.

Scheme	Overhead (No. of Bits)
Stenström	$M\log_2 N + CN$
Limited	$iM\log_2 N$
Chained	$M\log_2 N + C\log_2 N$

rectly to all caches marked in the presence flag vector (cache 2) in Figure 8b. Bold lines mark these actions in Figure 8b.

Assuming the bounded-buffer problem of Figure 3 and the *count* variable, some important points come up. First, invalidations will be sent to only one cache because there will be at most one copy of *count*. This is very important because if broadcasts were generated, this would result in immense network traffic. Second, in both cases read misses to dirty blocks must traverse the network twice. Third, the Censier scheme requires sending a request to the memory controller for each invalidation. In the Stenström scheme, invalidations can be sent directly because the presence flag vector is stored in the cache. The price for this, however, is that the presence flag vector must be fetched from the current owner if the block is not owned (the processor does not have write permission), which results in considerable network traffic for large presence flag vectors. For applications with one writer to a block, as is the case for the iterative algorithm in Figure 4, this overhead stays small.

The directory schemes presented have the main advantage of being able to restrict the consistency commands to those caches having copies of a block. They are called *full-map directory schemes* because they can track copies of an arbitrary number of caches. However, they are expensive to implement, especially for multiprocessors containing many processors.

There are different alternatives to reduce the directory size. One method, called the *limited directory scheme*, restricts the number of cache pointers to less than the actual number of caches. Given N caches and i pointers in each directory entry, where $i < N$, then $i\log_2 N$ bits are needed to track copies of blocks for each memory block. A key question for limited directory schemes is how to handle cases where more than i copies are requested. Two alternatives are possible: Either disallow more than i copies or start to broadcast when more than i copies exist. Clearly the success of a limited directory scheme depends on the degree of sharing, that is, the number of processors that simultaneously share data.

Agarwal et al.[7] introduced a classification of directory schemes. They referred to a directory scheme as $Dir_i X$, where i is the number of cache pointers for each block and X denotes whether the scheme broadcasts consistency commands ($X = B$) when the number of copies exceeds the number of cache pointers, or whether it disallows more than i copies ($X = NB$). Their terminology denotes the full-map schemes as $Dir_N NB$ and the limited directory schemes with broadcast capability as $Dir_i B$, where $i < N$.

One possible way of reducing the size of the directory for $Dir_N NB$ schemes is to link in a list all caches that store a copy of a block. We could do this by associating an entry including $\log_2 N$ bits with each cache line and memory block. This entry contains a pointer to the next cache that stores a copy. This scheme, called a *chained directory scheme*, routes consistency commands to only those caches having copies of a block. However, when we compare the chained directory scheme with the full-map directory schemes, we find that multicast operations may take longer to perform, thus slowing the processors. The Scalable Coherent Interface (IEEE P1596)[9] proposes a chained directory scheme.

An example of an extremely cost-effective directory scheme that relies on broadcasting consistency commands (denoted $Dir_1 B$) is the one proposed by Archibald and Baer (reviewed by Agarwal et al.[7]). Each directory entry consists of two bits encoding four global states of a memory block: not present in any cache, clean in exactly one cache, clean in an unknown number of caches, and dirty in exactly one cache. When a processor updates a block, an invalidation is broadcast to all caches. This generates immense network traffic. Nevertheless, this scheme is scalable in the sense that the number of caches can increase without changing the directory structure.

Table 3 compares the bit overhead required for cache pointers for a full-map (Stenström) scheme, a limited scheme with i pointers, and a chained directory scheme. Assume M memory blocks, C cache lines, and N caches. The chained directory is cheaper than the Stenström scheme. However, the Stenström scheme sends consistency commands directly to other caches without having to traverse the chain of cache pointers.

The full-map directory schemes have the advantage of reducing network traffic caused by invalidations or updates by multicasting them only to those caches with copies of a block. However, the amount of memory needed tends to be prohibitive for multiprocessors with many processors. Reducing the number of cache pointers, that is, employing limited directory schemes, alleviates this problem. The price for this is limiting the number of copies that may simultaneously coexist in different caches or introducing peaks of network traffic due to broadcasting of consistency commands. Consequently, a trade-off exists between network traffic and directory size. No commercial implementation yet uses directory schemes.

Another article in this issue, written by Chaiken et al., compares the performance of various directory schemes through a number of benchmark applications.

Cache-coherent network architectures. The real success of shared-memory multiprocessors lies in designs that provide a large number of processors interconnected in an economical way. We have seen that a common bus does not suit hundreds of processors. Multistage networks have problems, too, because of the hardware complexity for many processors. Therefore, researchers have proposed multiprocessors with a hierarchy of buses, in which network traffic is reduced by hierarchical cache-coherence protocols that don't suffer from the implementation complexity of directory schemes. Let's review three novel architectures based on this approach.

The first one, the hierarchical cache/bus architecture proposed by Wilson,[10] appears in Figure 9. We can view this architecture as a hierarchy of caches/buses where a cache contains a copy of all blocks cached underneath it. This requires large higher level cache modules. Memory modules connect to the topmost bus.

To maintain consistency among copies, Wilson proposed an extension to the write-invalidate protocol. Consistency among copies stored at the same level is maintained in the same way as for traditional snoopy cache protocols. However, an invalidation must propagate vertically to invalidate copies in all caches. Suppose that processor P_1 issues a write (see Figure 9). The write request propagates up to the

highest level and invalidates every copy. Consequently, copies in M_c20, M_c22, M_c16, and M_c18 will be invalidated. However, higher order caches such as M_c20 keep track of dirty blocks beneath them. A subsequent read request issued by P_7 will propagate up the hierarchy because no copies exist. When it reaches the topmost level, M_c20 issues a flush request down to M_c11 and the dirty copy is supplied to the cache of processor P_7.

Note that higher level caches act as filters for consistency actions; an invalidation command or a read request will not propagate down to subsystems that don't contain a copy of the corresponding block. This means that M_c21 in the example above acts as a filter for the invalidations on the topmost cache, since this subsystem has no copies.

The next architecture is the Wisconsin Multicube, proposed by Goodman and Woest.[11] As shown in Figure 10, it consists of a grid of buses with a processing element in each switch and a memory module connected to each column bus. A processing element consists of a processor, a cache, and a snoopy cache controller connected to the row and column bus. The snoopy cache is large (comparable to the size of main memory in a uniprocessor) to reduce network traffic. The large caches mean bus traffic results mainly from consistency actions.

Like in the hierarchical cache/bus system, a write-invalidate protocol maintains consistency. Invalidations are broadcast on every row bus, while global read requests are routed to the closest cache with a copy of the requested block. This is supported in the following way: Each block has a "home column" corresponding to the memory module that stores the block. A block can be globally modified or unmodified. If the block is globally modified, then there exists only one copy. Each cache controller stores in its column the identification of all modified blocks, which serves as routing information for read requests. A read request is broadcast on the row bus and routed to the column bus where the modified block is stored.

The Data Diffusion Machine[12] is another hierarchical cache-coherent architecture quite similar to Wilson's architecture. It consists of a hierarchy of buses with large processor caches (on the order of a megabyte) at the lowest level, which is the only type of memory in the system. A hierarchical write-invalidate protocol maintains consistency. Unlike Wilson's architecture, higher order caches (such as

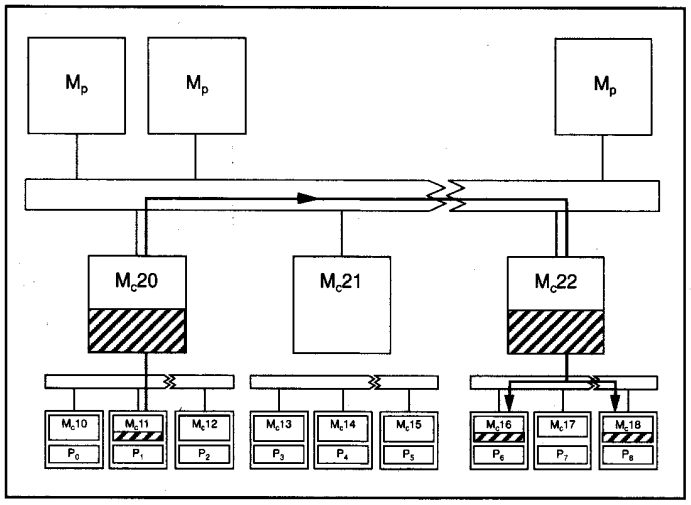

Figure 9. Wilson's hierarchical cache/bus architecture.

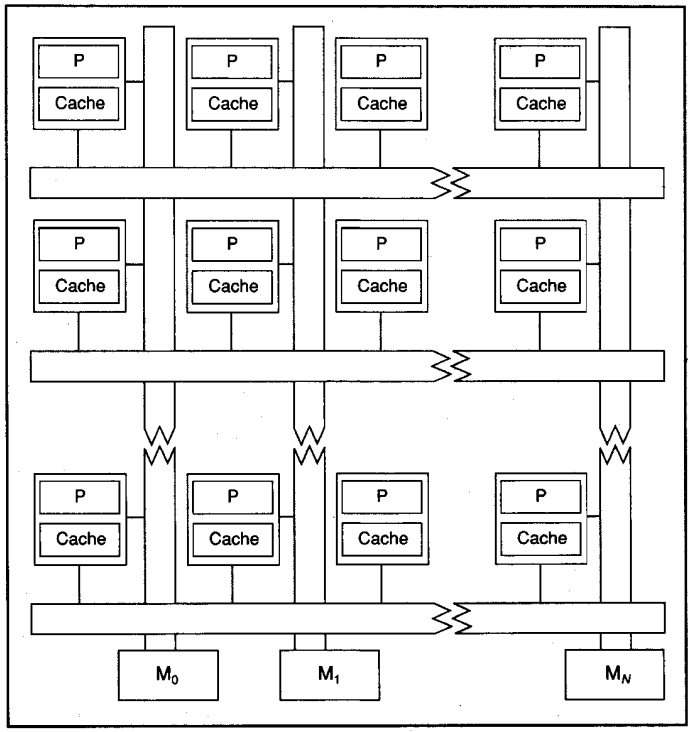

Figure 10. Goodman and Woest's Wisconsin Multicube.

```
repeat
   par_for J := 1 to N do
   begin
      xtemp[ J ] := b[ J ];          — cache-read( b[ J ])
      for K := 1 to N do
         xtemp[ J ] :=
            xtemp[ J ] +             — cache-read(xtemp[ J ])
            A[ J,K ] *               — cache-read(A[ J,K ])
            x[ K ];                  — memory-read(x[ K ])
   end;
   barrier_sync;                     — cache-invalidate
   par_for J := 1 to N do
      x[ J ] := xtemp[ J ];          — memory-read(xtemp[ J ])
   barrier_sync;                     — cache-invalidate
until false;
```

Figure 11. Example of reference marking of the iterative algorithm.

the level-2 caches in Figure 9) contain only state information, which considerably reduces memory overhead. We pay a price for this: Certain read requests must be sent to the root and then down to a leaf and back again because an intermediate-level cache cannot satisfy them.

Interestingly, the global memory has been distributed to the processors. In conjunction with the cache coherence protocol, this allows an arbitrary number of copies. Since data items have no home locations, as opposed to the Wilson and Wisconsin Multicube architectures, they will "diffuse" to those memory modules where they are needed. The Data Diffusion Machine is currently being built at the Swedish Institute of Computer Science.

Compared to the full-map directory schemes, these architectures are more cost-effective in terms of memory overhead and constitute an interesting extension to bus-based architectures for large shared-memory multiprocessors. However, it is too early to tell whether implementations will prove efficient.

Software-based schemes

Software cache-coherence schemes attempt to avoid the need for complex hardware mechanisms. Let's take a look at some of the proposals.

How to prevent inconsistent cache copies. Hardware-based protocols effectively reduce network traffic. However, we pay for this with complex hardware mechanisms, especially for multiprocessors with a large number of processors.

An alternative would prevent the existence of inconsistent cached data by limiting the caching of a data structure to safe times. This makes it necessary to analyze the program to mark variables as cacheable or noncacheable, which a sophisticated compiler or preprocessor can do. The most trivial solution would be to mark all shared read-write variables as noncacheable. This is too conservative, since shared data structures can be exclusively accessed by one process or are read-only during a considerable amount of time. During such intervals it is safe to cache a data structure.

A better approach would let the compiler analyze when it is safe to cache a shared read-write variable. During such intervals it marks the variable as cacheable. At the end of the interval, main memory must be consistent with the cached data, and cached data must be made inaccessible from the cache by invalidation. This raises the following key questions: How does the compiler mark a variable as cacheable, and how is data invalidated?

The following survey of software-based cache coherence schemes will address these issues. Consult Cheong and Veidenbaum[13] for references to further reading. See also the article written by Cheong and Veidenbaum in this issue.

Cacheability marking. We can base the reference marking of a shared variable on static partitioning of the program into computational units. Accesses to a shared variable in one computational unit might differ from those of another computational unit. For example, the accesses might be one of the following types:

(1) Read-only for an arbitrary number of processes.
(2) Read-only for an arbitrary number of processes and read-write for exactly one process.
(3) Read-write for exactly one process.
(4) Read-write for an arbitrary number of processes.

Here, we assume that processes execute on different processors. Type 1 implies that the variable is cacheable, such as all elements of the shared matrix **A** and vector **b** in the iterative algorithm of Figure 4. Type 2 implies that at most the read-write process may cache the variable and that main memory is always made consistent. Using write-through update policy achieves this. Type 3 allows the variable to be cached and updated using copy-back, as for the shared variables in the critical sections of the producer and consumer code in Figure 3. Finally, for type 4 we must mark the variable as noncacheable. Consider, for example, synchronization variables such as those implementing the mutexbegin, mutexend, and barrier synchronization of Figures 3 and 4.

Because synchronizations often delimit a computational unit, we can apply different rules for maintaining a variable's consistency for different computational units. Between computational units, cached shared variables must be invalidated before the next computational unit enters. Moreover, main memory must be updated, either by using write-through update policy or by flushing the content of the cache if a copy-back policy is used.

Computational units are easily identified if they are explicit in the program code, such as the parallel for-loops in the iterative algorithm. In the first parallel loop, all elements of **xtemp** are type 3, while all elements of **A**, **b**, and **x** are type 1. In the second parallel loop, all elements of vectors **x** and **xtemp** are type 3, which makes it possible to cache all shared variables provided that all elements of vector **x** are invalidated at the end of the second parallel for-loop and main memory is consistent at the beginning of the iteration. The critical sections associated with the bounded-buffer algorithm provide another example of a computational unit.

Typically, the compiler's main task is to analyze data dependencies and generate appropriate cache instructions to control

the cacheability and invalidation of shared variables. The data dependence analysis itself, an important and sometimes complex task, lies outside the scope of this article. Interested readers should consult the references in Cheong and Veidenbaum.[13] Instead, we will look at different approaches to enforcing cache coherence and investigate the hardware support implied by these schemes. The first three approaches rely on parallel for-loops to classify the cacheability of shared variables. The last approach relies on critical sections as a model for accessing shared read-write data.

Cache coherence enforcement schemes. In the first approach, proposed by Cheong and Veidenbaum, all shared variables accessed within a computational unit receive equal treatment; either all or none can be cached. This scheme assumes a write-through cache, which guarantees up-to-date memory content. Moreover, it assumes three cache instructions: Cache-On, Cache-Off, and Cache-Invalidate. Cache-On turns caching on for all shared variables when all shared accesses are read-only (type 1) or exclusively accessed by one process (type 3). Cache-Off results in all shared accesses bypassing cache and going to the shared memory.

After execution of a computational unit, the Cache-Invalidate instruction invalidates all cache content. Invalidating the whole cache content, called *indiscriminate invalidation*, has the advantage of being easy to implement efficiently. However, indiscriminate invalidation is too conservative and leads to an unnecessarily high cache-miss ratio. For instance, in the iterative algorithm, caching is turned on, allowing all variables to be cached. However, invalidations needed after each barrier synchronization result in unnecessary misses for accesses to the read-only matrix **A** and vector **b**.

Selective invalidation of only those variables that can introduce inconsistency would improve this scheme. It is important to implement selective invalidation efficiently. Cheong and Veidenbaum[13] proposed a scheme with these objectives. In this scheme, shared-variable accesses within a computational unit are classified as always up to date or possibly stale. The scheme assumes three types of cache instructions to support this, namely, Memory-Read, Cache-Read, and Cache-Invalidate. Memory-Read means possibly stale cached copy, whereas Cache-Read guarantees up-to-date cached copy. Furthermore, the scheme assumes the cache uses write-through.

Associated with each cache line is a change bit. The Cache-Invalidate instruction sets all change bits true. If a Memory-Read is issued to a cache block with its change bit set true, then the read request will be passed to memory. When the requested block is loaded into the cache, the change bit is set false and subsequent accesses will hit in the cache.

To demonstrate this method, consider once again the iterative algorithm of Figure 4. Assume that the block size is one vector element and that $n = N/L$ processors cooperate in the execution of the parallel loops. Each processor executes L iterations. Figure 11 includes comments for all read operations to shared data with the cache instructions supported by the scheme. The only sources of inconsistency are the accesses to vectors **x** and **xtemp**. This means the only references that need marking as Memory-Reads are when **x** is read in the first parallel loop and when **xtemp** is read in the second parallel loop. Clearly, this scheme eliminates cache misses on the accesses to vector **b** and matrix **A**. Just turning on all change bits efficiently accomplishes the fast selective invalidation scheme in one cycle.

Even if this scheme reduces the number of invalidation misses, it is still conservative because the same processor might execute the same iterations in the two parallel loops. If so, the corresponding elements of **xtemp** will be unnecessarily invalidated and reread from memory in the second parallel loop.

A third scheme takes advantage of this temporal locality: the timestamp-based scheme proposed by Min and Baer.[14] This scheme associates a "clock" (a counter) with each data structure, such as vectors **x** and **xtemp** in the iterative algorithm. This clock is updated at the end of each computational unit (at the barrier synchronizations in the algorithm) in which the corresponding variable is updated. For example, the clock for vector **xtemp** is updated after the first parallel loop, and the clock for vector **x** is updated after the second loop. A timestamp associated with each block in the cache (for example, with each vector element) is set to the value of the corresponding clock+1 when the block is updated in the cache. A reference to a cache word is valid if its timestamp exceeds its associated clock value. Otherwise, the block must be fetched from memory.

This scheme eliminates invalidations associated with variables local to a processor between two computational units because the timestamp value for these variables exceeds their clock value. The hardware support for this scheme consists of a number of clock registers and a timestamp entry for each cache line in the cache directory.

Let's compare the number of invalidation misses generated by the schemes presented so far and compare these numbers with the corresponding number for a write-invalidate hardware scheme. Assume that $n = N/L$ processors execute the iterations and that each processor always executes the same iteration. This means that processor i executes iterations $(i\text{-}1)L + 1$ to iL in the parallel loops in the iterative algorithm. Assume a block size corresponding to one vector element. Table 4 shows the result.

Table 4. Comparison of the number of invalidation misses for the software-based schemes and a write-invalidate hardware-based scheme for the iterative algorithm.

	Indiscriminate Invalidation	Fast Selective Invalidation	Timestamp Scheme	Write-Invalidate Hardware Scheme
Loop 1	$N(L+1) + L$	N	N	N
Loop 2	L	L		
Sum	$N(L+1) + 2L$	$N + L$	N	N

The table makes it clear that the last scheme results in the same number of invalidation misses as does any write-invalidate hardware-based scheme. The hardware support for the different schemes differs in complexity. The indiscriminate invalidation scheme requires only a mechanism for turning on or off and invalidating the cache. The fast selective invalidation scheme requires one bit for each cache line (the change bit), whereas the timestamp-based scheme requires a time-