5.4 Distributed Shared-Memory and Directory –Based Coherence

A scalable multiprocessor supporting shared memory could choose to exclude or include cache coherence.

Some systems (such as **Cray T3D)** have caches, but to prevent coherence problems, shared data is marked as uncacheable and only private data is kept in the caches.

The advantage of such a mechanism is that little hardware support is required.

There are several disadvantages to this approach:

- ❑compiler mechanisms for transparent software cache coherence are very limited.
- ❑without cache coherence, the multiprocessor loses the advantage of being able to fetch and use multiple words in a single cache block for close to the cost of fetching one word.
- ❑mechanisms for tolerating latency such as prefetch are more useful when they can fetch multiple words, such as a cache block, and where the fetched data remain coherent.

A snooping protocol requires communication with all caches on every cache miss, including writes of potentially shared data.

The absence of any centralized data structure that tracks the state of the caches is the fundamental advantage of a snooping-based scheme.

We could build scalable shared-memory architectures that include cache coherency. The key is to find an alternative coherence protocol to the snooping protocol.

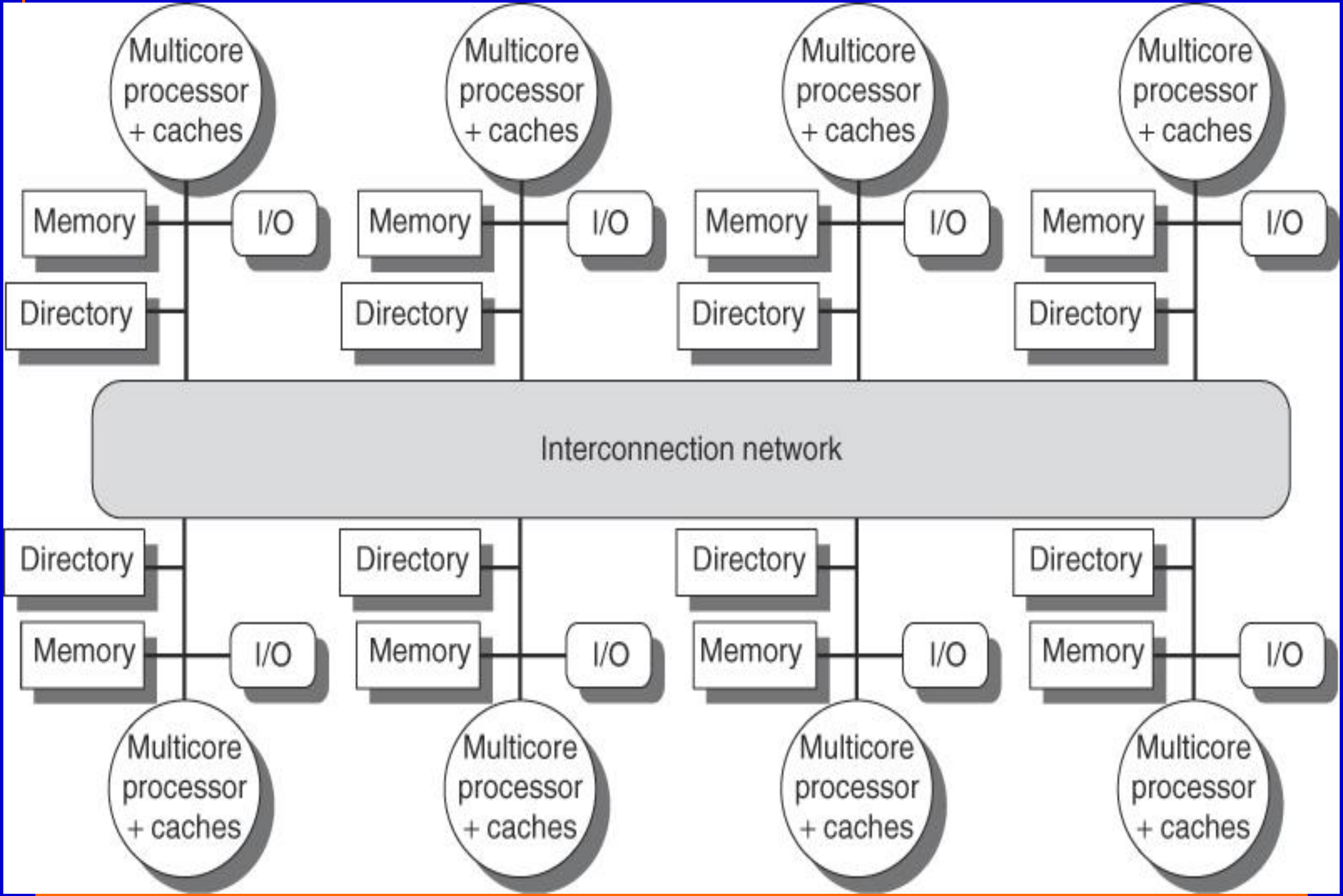
One alternative protocol is a directory protocol. A *directory* keeps the state of every block that may be cached.

Information in the directory includes which caches have copies of the block, whether it is dirty, and so on.

Existing directory implementations associate an entry in the directory with each memory block.

In typical protocols, the amount of information is proportional to the product of the number of memory blocks and the number of processors.

To prevent the directory from becoming the bottleneck, directory entries can be distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories.


Multicore processor + caches
Multicore processor + caches
Multicore processor + caches
Multicore processor + caches
Memory
I/O
Memory
I/O
Memory
I/O
Memory
I/O
Directory
Directory
Directory
Directory
Interconnection network
Directory
Directory
Directory
Directory
Memory
I/O
Memory
I/O
Memory
I/O
Memory
I/O
Multicore processor + caches
Multicore processor + caches
Multicore processor + caches
Multicore processor + caches

Directory-Based Cache-Coherence Protocols: The Basics

Just as with a snooping protocol, there are two primary operations that a directory protocol must implement: handling a read miss and handling a write to a shared, clean cache block. (Handling a write miss to a shared block is a simple combination of these two.)

To implement these operations, a directory must track the state of each cache block. In a simple protocol, these states could be the following:

- *Shared*—One or more processors have the block cached, and the value in memory is up to date (as well as in all the caches).

- *Uncached*—No processor has a copy of the cache block.

- *Exclusive*—Exactly one processor has a copy of the cache block and it has written the block, so the memory copy is out of date. The processor is called the *owner* of the block.

In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared, since they will need to be invalidated on a write.

The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different.

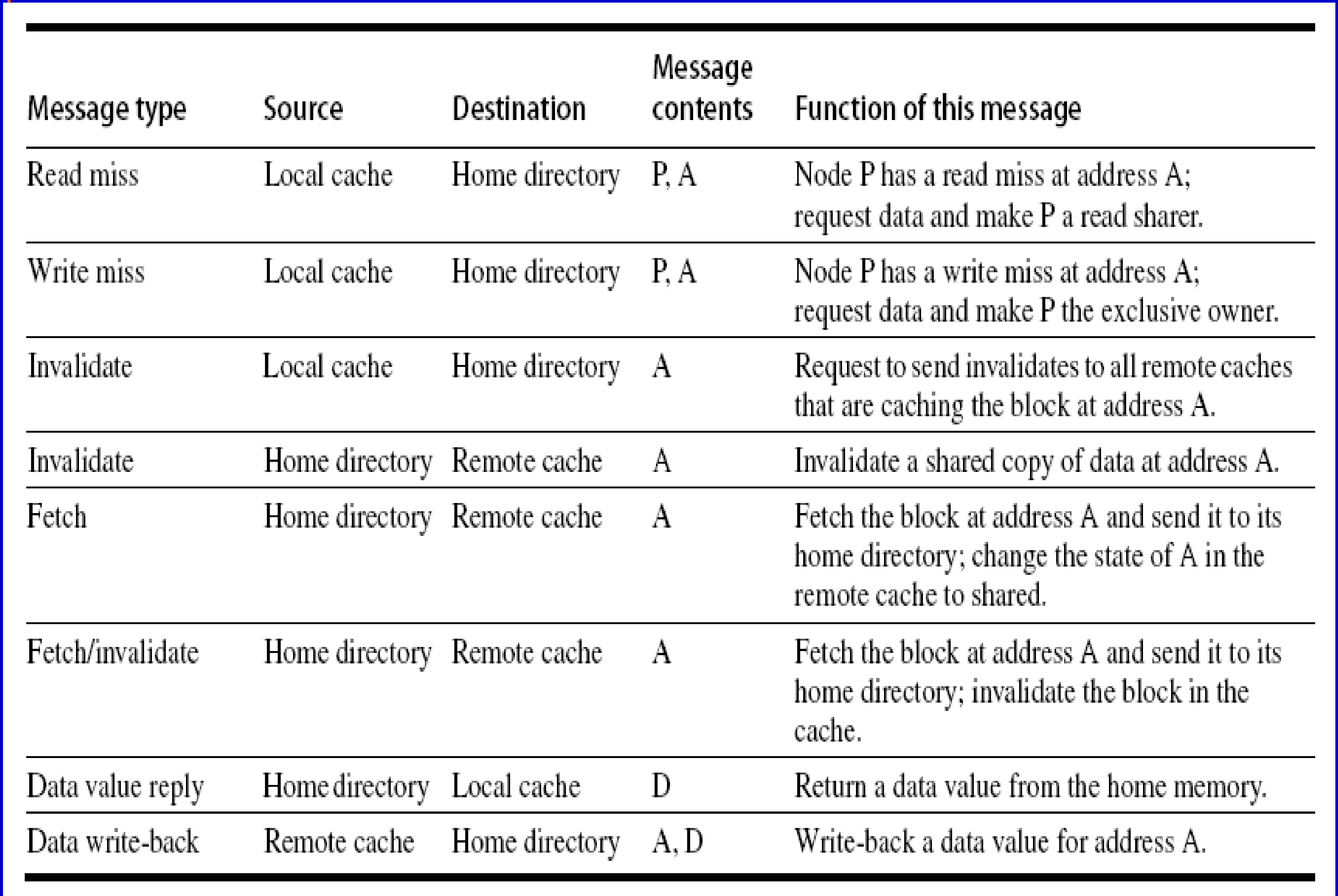
There are two additional complications:

1. We cannot use the interconnect as a single point of arbitration, a function the bus performed in the snooping case.

2. Because the interconnect is message oriented (unlike the bus, which is transaction oriented), many messages must have explicit responses.

The *local* node is the node where a request originates. The *home* node is the node where the memory location and the directory entry of an address reside.

A *remote* node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared.



Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known.

For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node. The local node may also be the home node.

The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a remote node.

An Example Directory Protocol

The basic states of a cache block in a directory-based protocol are exactly like those in a snooping protocol, and the states in the directory are also analogous to those we showed earlier.

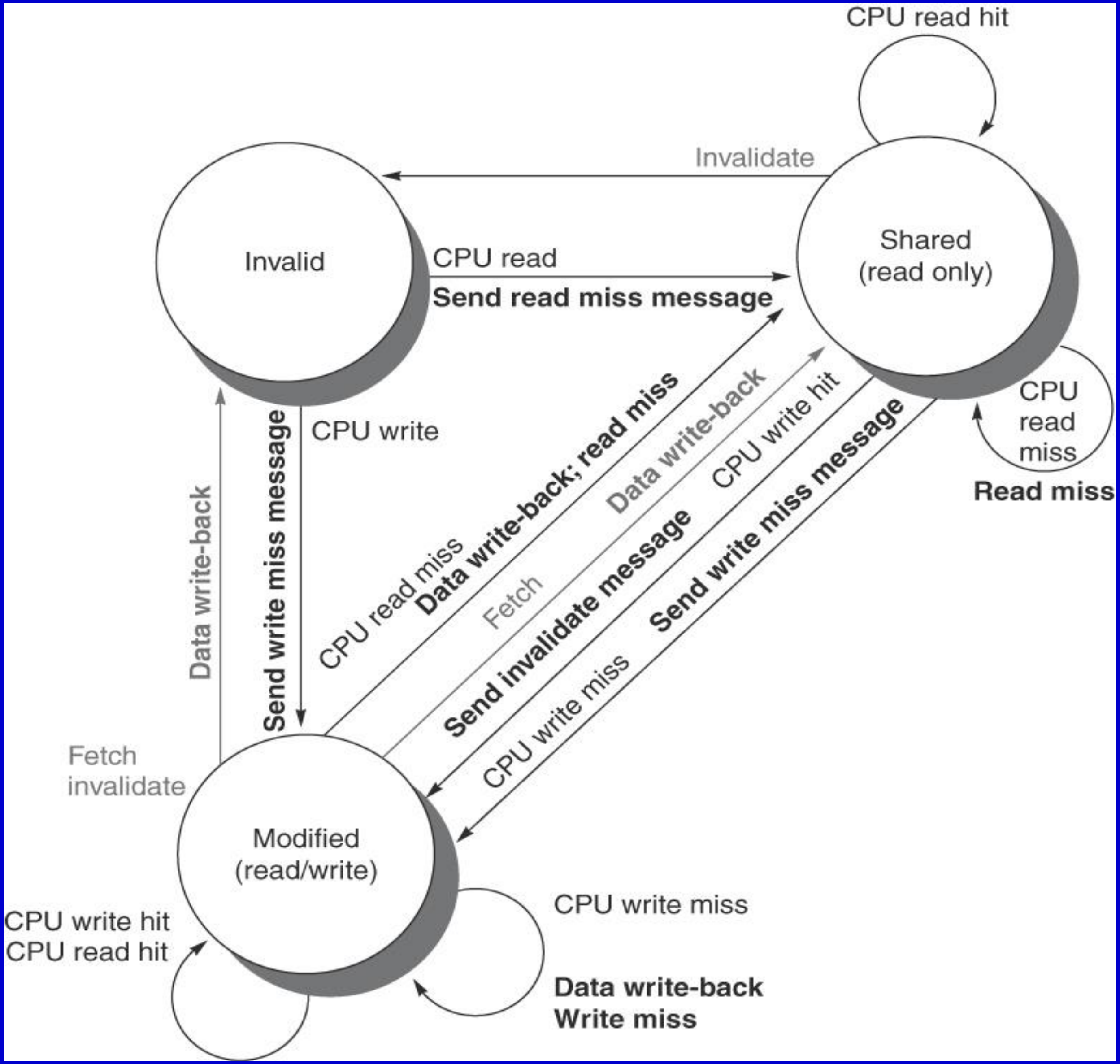


Figure 5.20 State transition diagram for an individual cache block in a directory-based system.

In a directory-based protocol, the directory implements the other half of the coherence protocol. A message sent to a directory causes two different types of actions: updates of the directory state, and sending additional messages to satisfy the request.

The states in the directory represent the three standard states for a block; unlike in a snoopy scheme, however, the directory state indicates the state of all the cached copies of a memory block, rather than for a single cache block.

The memory block may be uncached by any node, cached in multiple nodes and readable (shared), or cached exclusively and writable in exactly one node.

In addition to the state of each block, the directory must track the set of processors that have a copy of a block; we use a set called *Sharers* to perform this function. In multiprocessors with less than 64 nodes (which may represent 2-4 times as many processors), this set is typically kept as a bit vector.

Below figure shows the actions taken at the directory in response to messages received. The directory receives three different requests: read miss, write miss, and data write back.

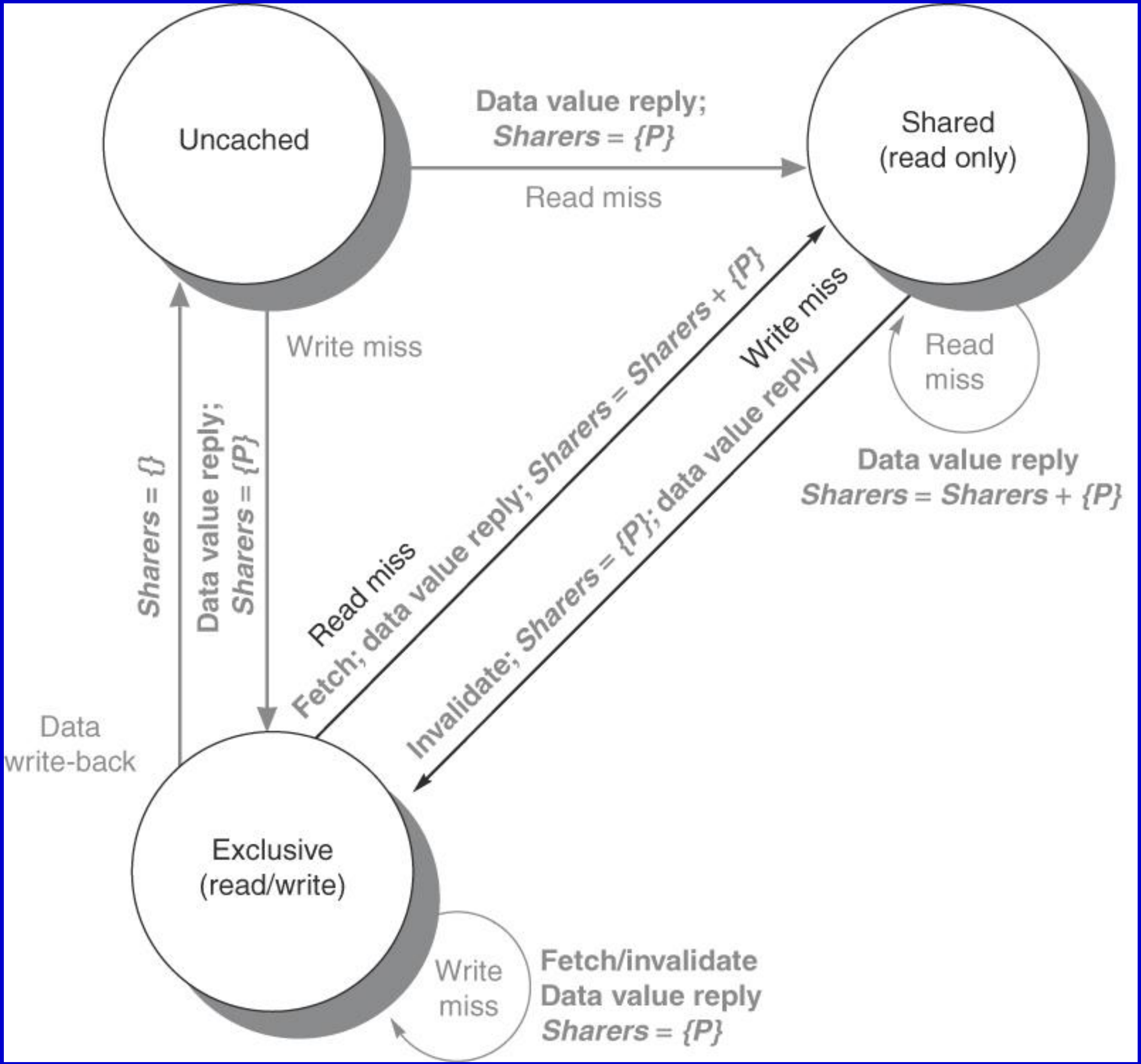


Figure 5.21 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request.

When a block is in the uncached state the copy in memory is the current value, so the only possible requests for that block are:

- ❑ *Read miss*—The requesting processor is sent the requested data from memory and the requestor is made the only sharing node. The state of the block is made shared.
- ❑*Write miss*—The requesting processor is sent the value and becomes the Sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state the memory value is up-to-date, so the same two requests can occur:

- *Read miss*—The requesting processor is sent the requested data from memory and the requesting processor is added to the sharing set.
- *Write miss*—The requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made exclusive.

When the block is in the exclusive state the current value of the block is held in the cache of the processor identified by the set sharers (the owner), so there are three possible directory requests:

- *Read miss*—The owner processor is sent a data fetch message, which causes the state of the block in the owner’s cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).

- *Data write-back*—The owner processor is replacing the block and therefore must write it back. This write-back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the sharer set is empty.
- *Write miss*—The block has a new owner. A message is sent to the old owner causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

When a read or write miss occurs for a block that is exclusive, the block is first sent to the directory at the home node. From there it is stored into the home memory and also sent to the original requesting node.

Many of the protocols in use in commercial multiprocessors forward the data from the owner node to the requesting node directly (as well as performing the write back to the home). Such optimizations often add complexity by increasing the possibility of deadlock and by increasing the types of messages that must be handled.

基于目录的Cache相关性协议采取了以“空间换时间”的策略，减少了访问量但增加了目录存储器，它的大小与系统规模N的平方成正比。改进：

提出了有限映射(Limited-map)和链式结构(Chained)。

1)有限映射假定同一数据在不同Cache中的拷贝数总小于一个常数m(m<<N),m即为目录中位向量的长度，因而大大减小了目录存储器的规模。

有限目录的缺点是：当同一数据的拷贝数大于m时，必须作特殊处理。

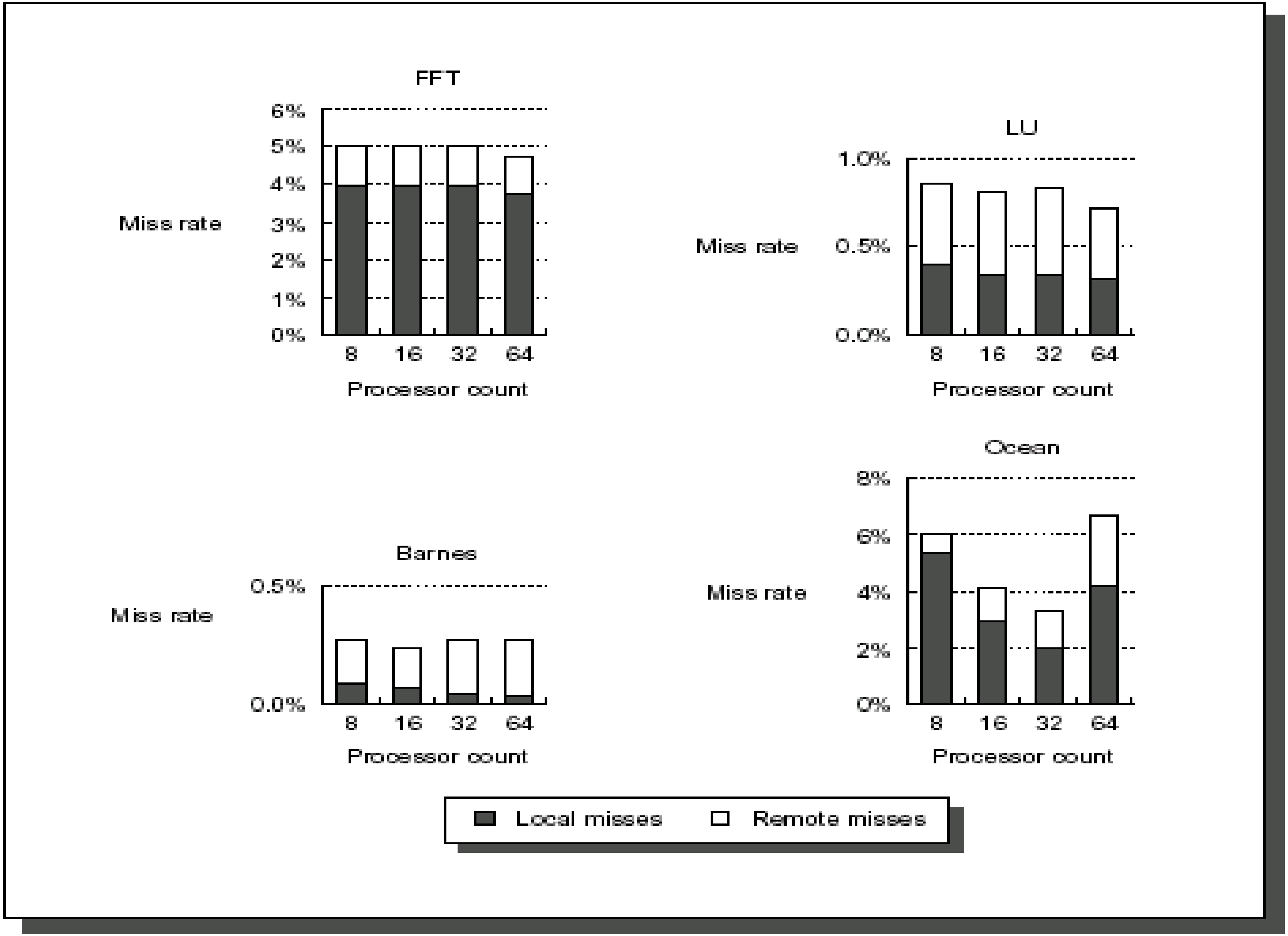
2)链式目录不但目录存储器规模小，而且不存在有限目录关于m的限制，但相关性协议比较复杂。

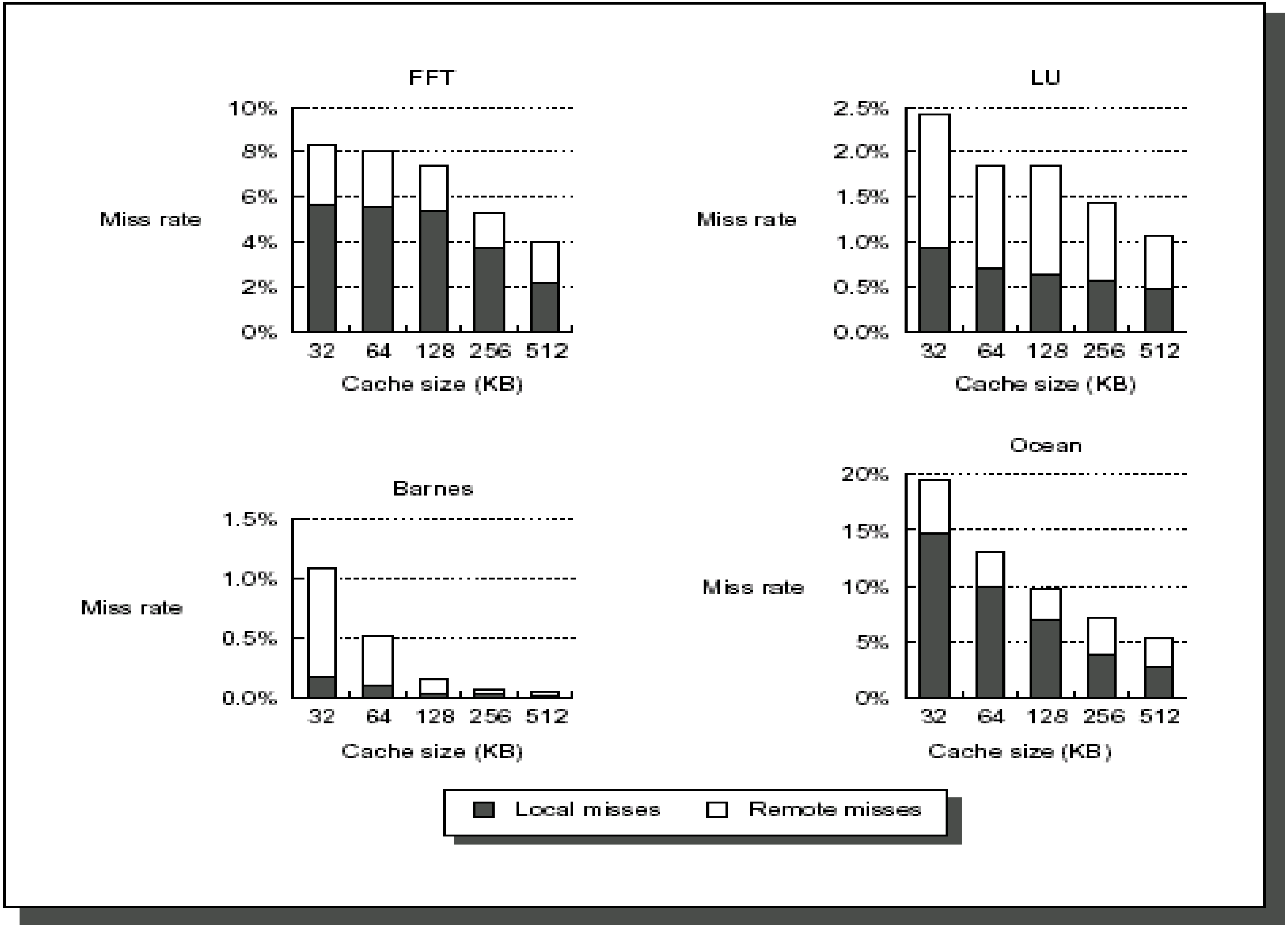
前面介绍的基于目录的相关性协议称为全映射(Full-map)。

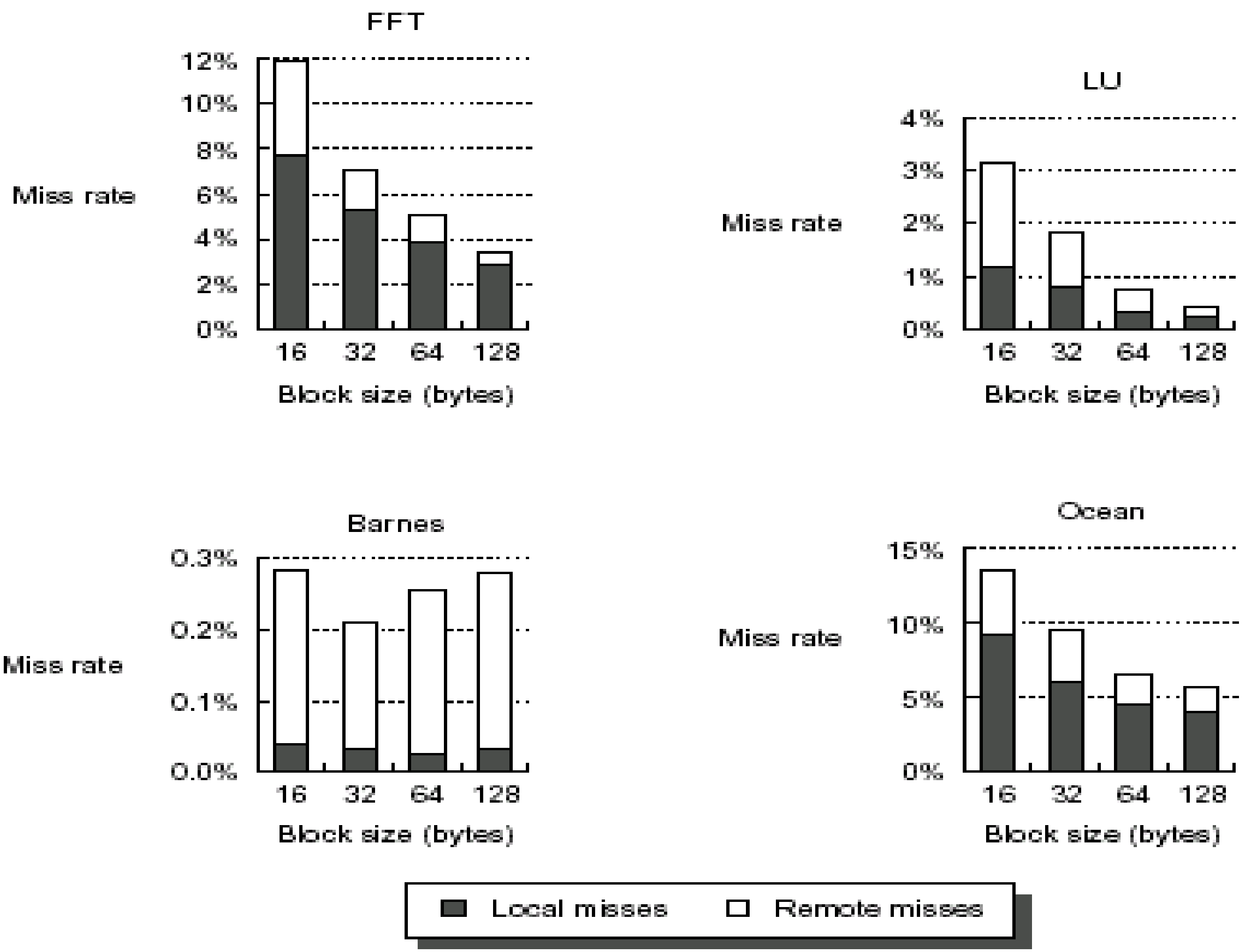
基于目录的Cache相关性协议是完全由硬件实现的。还可以用软硬结合的办法实现，即将一个可编程协议处理器嵌到相关性控制器中，可编程协议处理器可以根据实际应用需要很快开发出来，而相关性协议处理中的异常情况可完全交给软件执行。这种软硬结合的代价是损失了一部分效率。

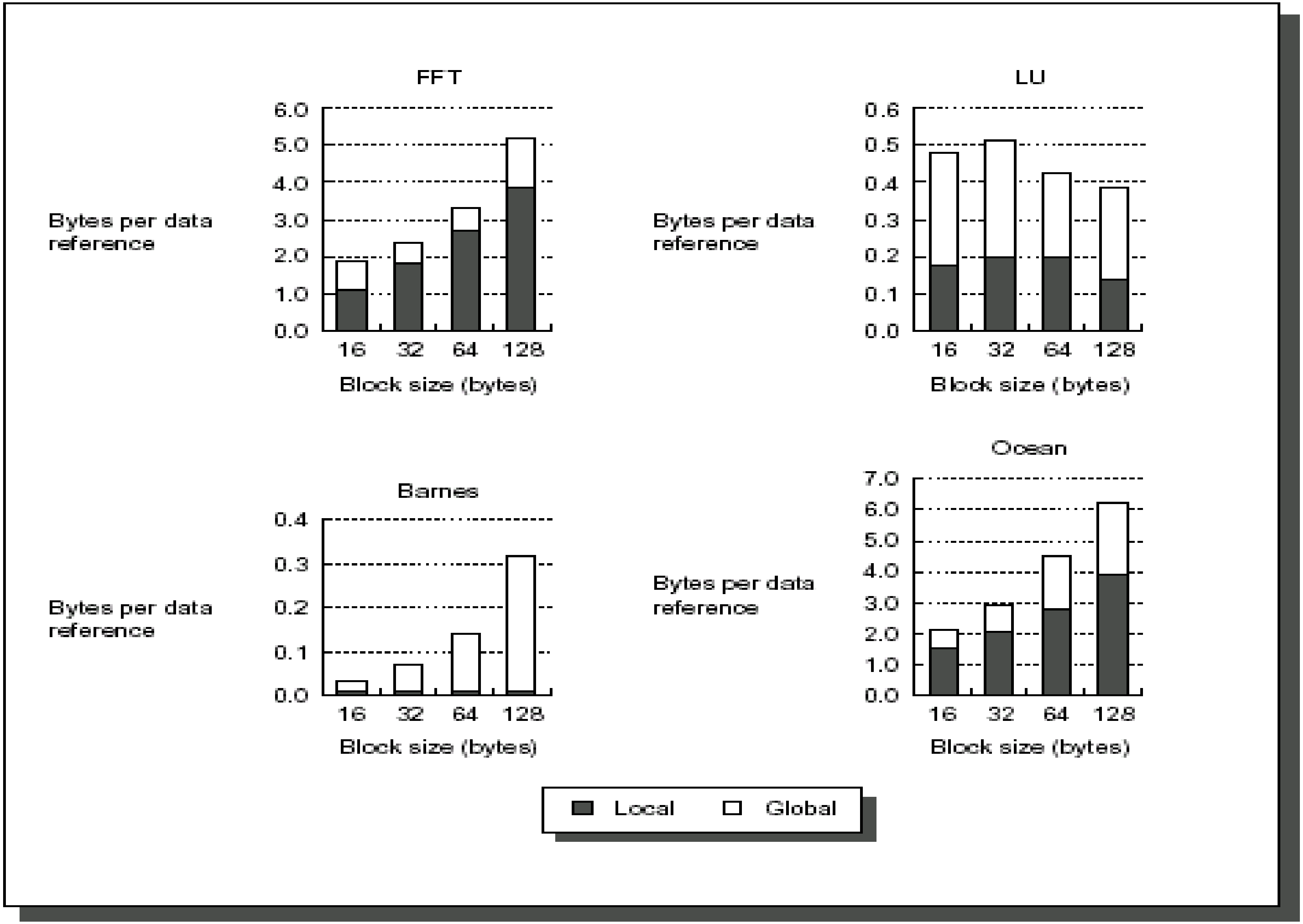
Performance of Distributed Shared-Memory Multiprocessors

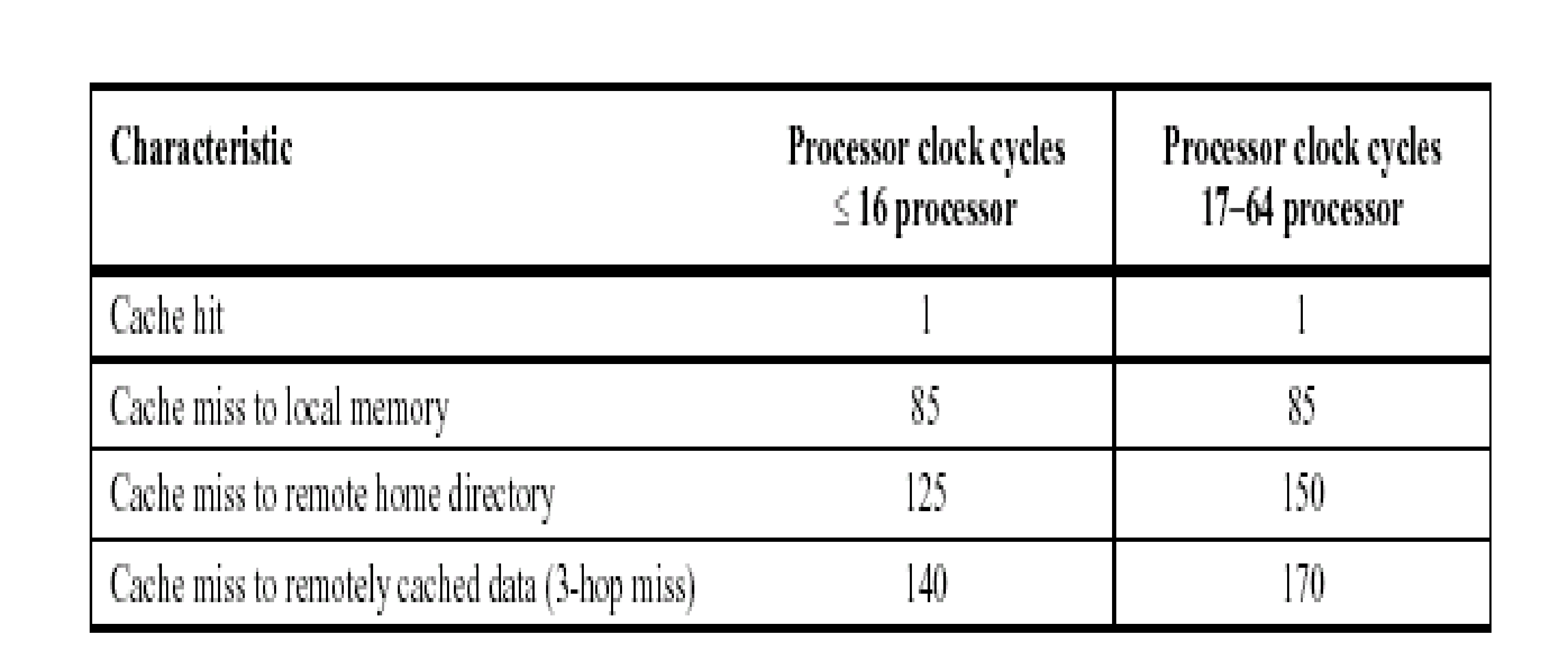
In distributed memory architectures, the distribution of memory requests between local and remote is key to performance, because it affects both the consumption of global bandwidth and the latency seen by requests. Therefore, for the figures in this section we separate the cache misses into local and remote requests.


FFT
6%
5%
4%
3%
2%
1%
0%
Miss rate
8
16
32
64
Processor count
LU
1.0%
0.5%
0.0%
Miss rate
8
16
32
64
Processor count
Barnes
0.5%
0.0%
Miss rate
8
16
32
64
Processor count
Ocean
8%
6%
4%
2%
0%
Miss rate
8
16
32
64
Processor count
■ Local misses
□ Remote misses


FFT
Miss rate
10%
8%
6%
4%
2%
0%
32
64
128
256
512
Cache size (KB)
LU
Miss rate
2.5%
2.0%
1.5%
1.0%
0.5%
0.0%
32
64
128
256
512
Cache size (KB)
Barnes
Miss rate
1.5%
1.0%
0.5%
0.0%
32
64
128
256
512
Cache size (KB)
Ocean
Miss rate
20%
15%
10%
5%
0%
32
64
128
256
512
Cache size (KB)
▨ Local misses
□ Remote misses


FFT
Miss rate
12%
10%
8%
6%
4%
2%
0%
16
32
64
128
Block size (bytes)
LU
Miss rate
4%
3%
2%
1%
0%
16
32
64
128
Block size (bytes)
Barnes
Miss rate
0.3%
0.2%
0.1%
0.0%
16
32
64
128
Block size (bytes)
Ocean
Miss rate
15%
10%
5%
0%
16
32
64
128
Block size (bytes)
■ Local misses
□ Remote misses


FFT
Bytes per data reference
6.0
5.0
4.0
3.0
2.0
1.0
0.0
16
32
64
128
Block size (bytes)
LU
Bytes per data reference
0.6
0.5
0.4
0.3
0.2
0.1
0.0
16
32
64
128
Block size (bytes)
Barnes
Bytes per data reference
0.4
0.3
0.2
0.1
0.0
16
32
64
128
Block size (bytes)
Ocean
Bytes per data reference
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
16
32
64
128
Block size (bytes)
■ Local
□ Global



Characteristic	Processor clock cycles ≤ 16 processor	Processor clock cycles 17–64 processor
Cache hit	1	1
Cache miss to local memory	85	85
Cache miss to remote home directory	125	150
Cache miss to remotely cached data (3-hop miss)	140	170

5.5 Synchronization: the Basics

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions.

In larger-scale multiprocessors or high-contention situations, synchronization can become a performance bottleneck, because contention introduces additional delays and because latency is potentially greater in such a multiprocessor.

Basic Hardware Primitives

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location.

These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers.

In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library.

One typical operation for building synchronization operations is the *atomic exchan*ge, which interchanges a value in a register for a value in memory.

Assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and a 1 is used to indicate that the lock is unavailable.

A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.

The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise.

In the latter case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

Consider two processors that each try to do the exchange simultaneously: This race is broken since exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange.

The key to using the exchange (or swap) primitive to implement synchronization is that the operation is atomic: the exchange is indivisible and two simultaneous exchanges will be ordered by the write serialization mechanisms.

There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically.

One operation, present in many older multiprocessors, is *test-and-set,* which tests a value and sets it if the value passes the test.

Another atomic synchronization primitive is *fetch-and-increment:* it returns the value of a memory location and atomically increments it.

Implementing a single atomic memory operation introduces some challenges, since it requires both a memory read and a write in a single, uninterruptible instruction.

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instructions were atomic.

The pair of instructions includes a special load called a *load linked* or *load locked* and a special store called a *store conditional*.

If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails.

If the processor does a context switch between the two instructions, then the store conditional also fails. The store conditional is defined to return a value indicating whether or not the store was successful.

Since the load linked returns the initial value and the store conditional returns 1 if it succeeds and 0 otherwise, the following sequence implements an atomic exchange on the memory location specified by the contents of R1:

```
try:  mov   x3,x4       ;mov exchange value
      lr    x2,x1       ;load reserved from
      sc    x3,0(x1)    ;store conditional
      bnez  x3,try      ;branch store fails
      mov   x4,x2       ;put load value in x4
```

At the end of this sequence the contents of R4 and the memory location specified by R1 have been atomically exchanged.

Any time a processor intervenes and modifies the value in memory between the LL and SC instructions, the SC returns 0 in R3, causing the code sequence to try again.

An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives. For example, here is an atomic fetch-and-increment:

```
try:   lr      x2,x1        ;load reserved 0(x1)
       addi    x3,x2,1      ;increment
       sc      x3,0(x1)     ;store conditional
       bnez    x3,try       ;branch store fails
```

These instructions are typically implemented by keeping track of the address specified in the LL instruction in a register, often called the *link register.*

If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (for example, by another SC), the link register is cleared.

The SC instruction simply checks that its address matches that in the link register; if so, the SC succeeds; otherwise, it fails.

Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing what instructions are inserted between the two instructions.

In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the SC.

In addition, the number of instructions between the load linked and the store conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store conditional to fail frequently.

Implementing Locks Using Coherence

We can use the coherence mechanisms of a multiprocessor to implement *spin locks:* locks that a processor continuously tries to acquire, spinning around a loop until it suceeds.

Spin locks are used when a programmer expects the lock to be held for a very short amount of time and when she wants the process of locking to be low latency when the lock is available.

The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory.

```
                addi      x2,R0,#1
lockit:  EXCH      x2,0(x1)       ;atomic exchange
         bnez      x2,lockit      ;already locked?
```

If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently.

Two advantages:

1.It allows an implementation where the process of “spinning” (trying to test and acquire the lock in a tight loop) could be done on a local cached copy.

2.There is often locality in lock accesses.

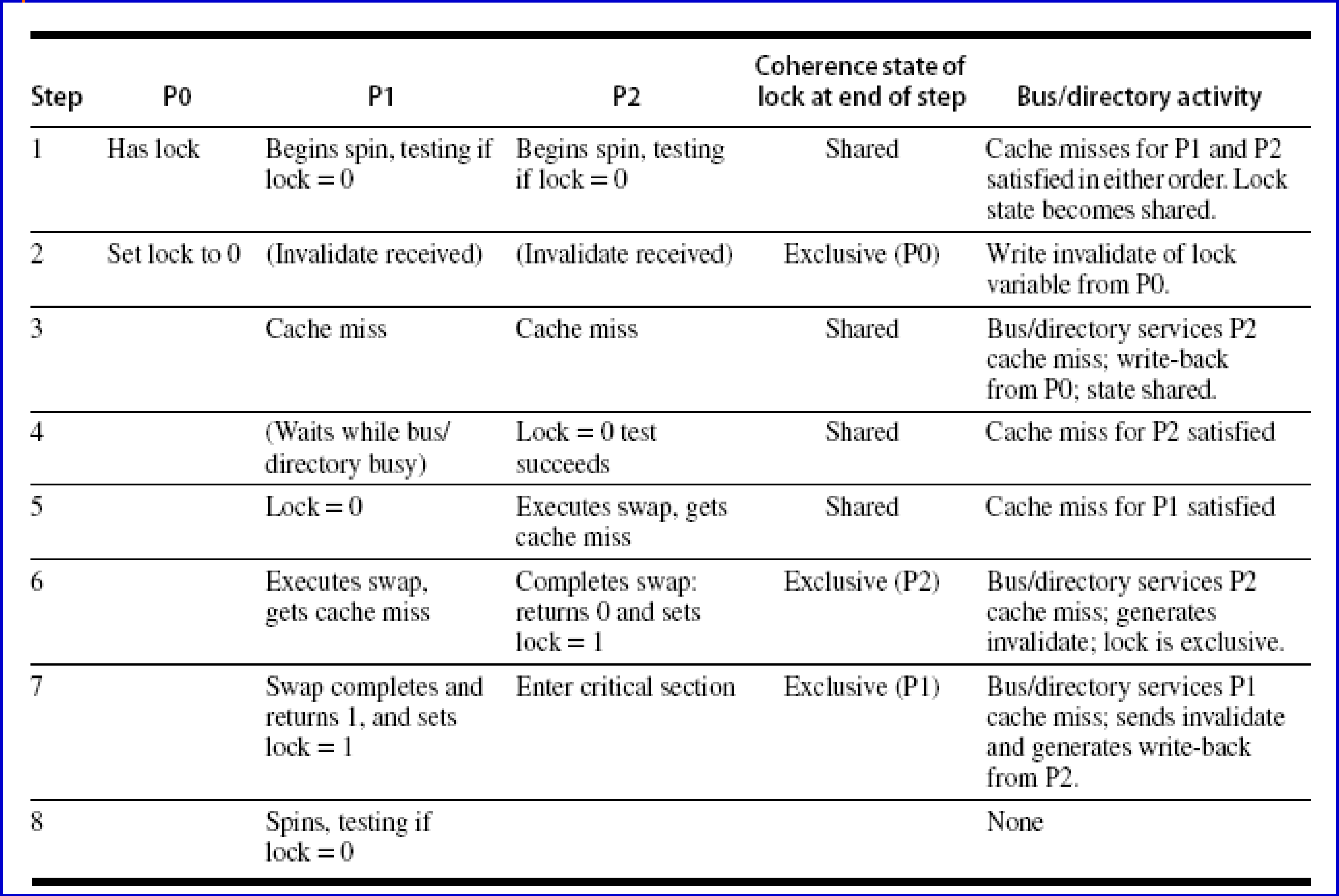
Obtaining the first advantage —requires a change in our simple spin procedure. Each attempt to exchange in the loop directly above requires a write operation.

Thus we should modify our spin-lock procedure so that it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available. Then it attempts to acquire the lock by doing a swap operation.

The winning processor executes the code after the lock and, when finished, stores a 0 into the lock variable to release the lock, which starts the race all over again.

```
lockit:   ld      x2,0(x1)      ;load of lock
          bnez    x2,lockit     ;not available-spin
          addi    x2,R0,#1      ;load locked value
          EXCH    x2,0(x1)      ;swap
          bnez    x2,lockit     ;branch if lock wasn't 0
```

Below figure shows the processor and bus or directory operations for multiple processes trying to lock a variable using an atomic swap.



Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

This example shows another advantage of the load-linked/store-conditional primitives: the read and write operation are explicitly separated. The load linked need not cause any bus traffic.

The following simple code sequence has the same characteristics as the optimized version using exchange (R1 has the address of the lock):

```
lockit:  lr       x2,0(x1)       ;load reserved
         bnez     x2,lockit      ;not available-spin
         addi     x2,R0,#1       ;locked value
         sc       x2,0(x1)       ;store
         bnez     x2,lockit      ;branch if store fails
```

The first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously.

Although our spin lock scheme is simple and compelling, it has difficulty scaling up to handle many processors because of the communication traffic generated when the lock is released.

Synchronization Performance Challenges

To understand why the simple spin-lock scheme of the previous section does not scale well, imagine a large multiprocessor with all processors contending for the same lock.

The directory or bus acts as a point of serialization for all the processors, leading to lots of contention, as well as traffic. The following Example shows how bad things can be.

EXAMPLE Suppose there are 10 processors on a bus that each try to lock a variable simultaneously. Assume that each bus transaction (read miss or write miss) is 100 clock cycles long. You can ignore the time of the actual read or write of a lock held in the cache, as well as the time the lock is held (they won’t matter much!). Determine the number of bus transactions required for all 10 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0. About how long will it take to process the 10 requests? Assume that the bus is totally fair so that every pending request is serviced before a new request and that the processors are equally fast.

ANSWER When *i* process are contending for the lock, they perform the following sequence of actions, each of which generates a bus transaction:

i load linked operations to access the lock

i store conditional operations to try to lock the lock

1 store (to release the lock)

Thus for *i* processes. There are a total of 2*i*+1 bus transactions. Note that this assumes that the critical section time is negligible, so that the lock is released before any other processors whose store conditional failed attempt another load linked.

Thus, for n processes, the total number of bus operation is

$$\sum(2i+1)=n(n+1)+n=n^2+2n$$

For 10 processes there are 120 bus transactions requiring 12000 clock cycles or 120 clock cycles per lock acquisition!

The difficulty in this Example arises from contention for the lock and serialization of lock access, as well as the latency of the bus access.

The key advantages of spin locks, namely that they have low overhead in terms of bus or network cycles and offer good performance when locks are reused by the same processor, are both lost in this example.

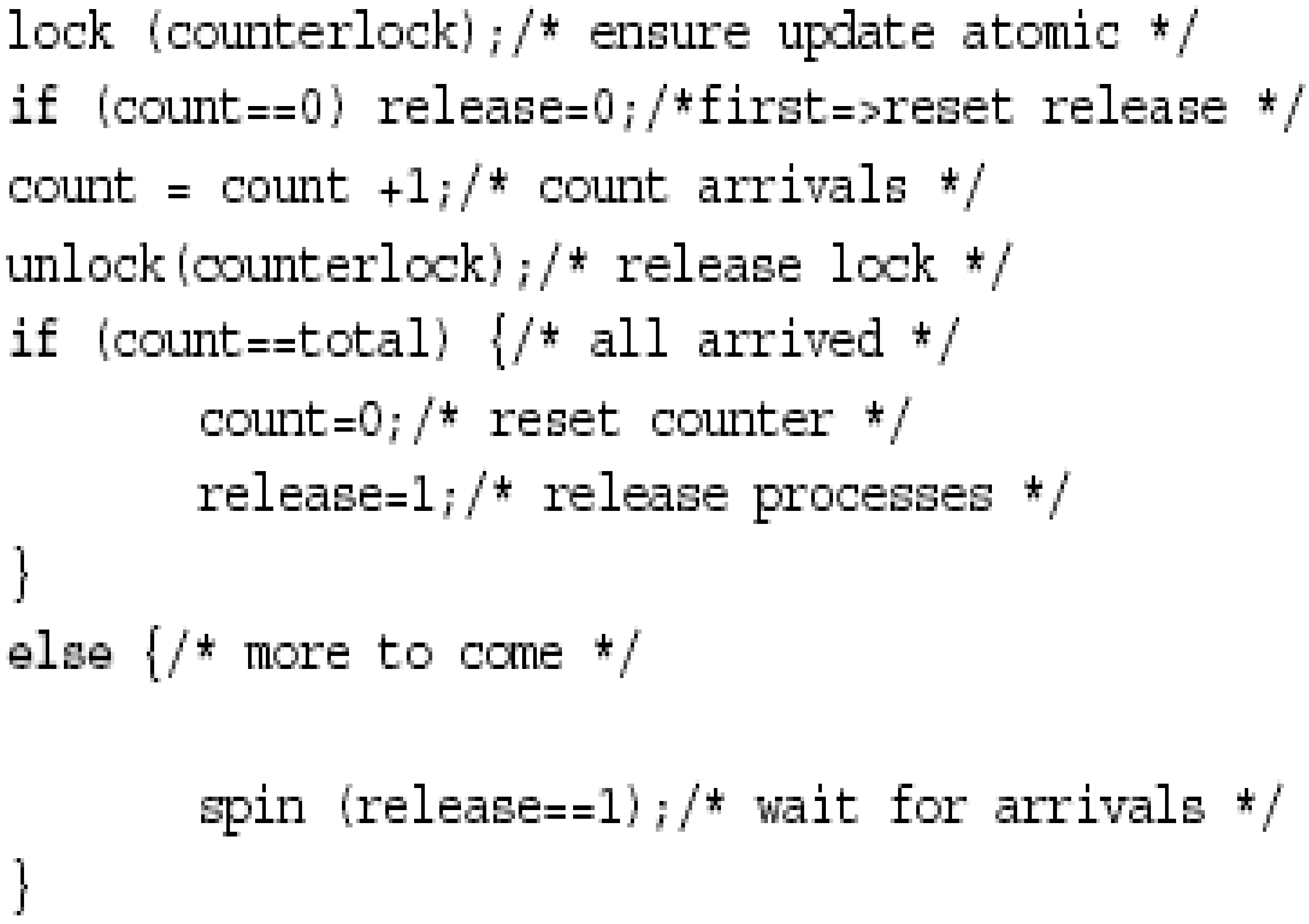
Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier*.

A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes.

A typical implementation of a barrier can be done with two spin locks: one used to protect a counter that tallies the processes arriving at the barrier and one used to hold the processes until the last process arrives at the barrier.

Bellow is a typical implementation, assuming that lock and unlock provide basic spin locks and total is the number of processes that must reach the barrier.



```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/*first=>reset release */
count = count +1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) {/* all arrived */
        count=0;/* reset counter */
        release=1;/* release processes */
}
else {/* more to come */

        spin (release==1);/* wait for arrivals */
}
```

In practice, another complication makes barrier implementation slightly more complex.

Frequently a barrier is used within a loop, so that processes released from the barrier would do some work and then reach the barrier again.

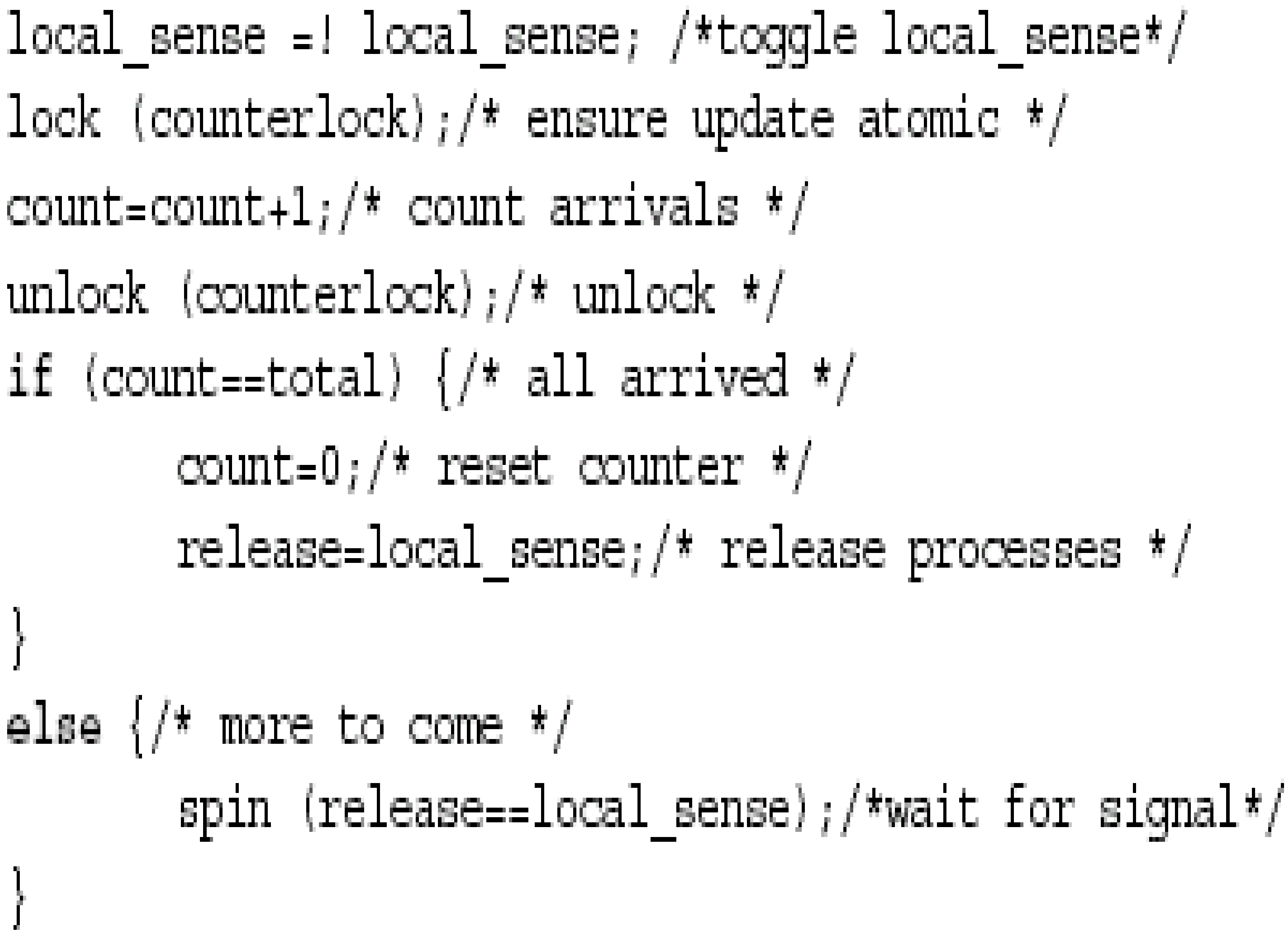
Assume that one of the processes never actually leaves the barrier (it stays at the spin operation), which could happen if the OS scheduled another process, for example.

Now it is possible that one process races ahead and gets to the barrier again before the last process has left. The “fast” process then traps the remaining “slow” process in the barrier by resetting the flag release.

Now all the processes will wait infinitely at the next instance of this barrier, because one process is trapped at the last instance, and the number of processes can never reach the value of total.

One obvious solution to this is to count the processes as they exit the barrier (just as we did on entry) and not to allow any process to reenter and reinitialize the barrier until all processes have left the prior instance of this barrier.

An alternative solution is a *sense-reversing barrier,* which makes use of a private perprocess variable, *local_sense*, which is initialized to 1 for each process.



```
local_sense =! local_sense; /*toggle local_sense*/
lock (counterlock);/* ensure update atomic */
count=count+1;/* count arrivals */
unlock (counterlock);/* unlock */
if (count==total) {/* all arrived */
        count=0;/* reset counter */
        release=local_sense;/* release processes */
}
else {/* more to come */
        spin (release==local_sense);/*wait for signal*/
}
```

EXAMPLE Suppose there are 10 processors on a bus that each try to execute a barrier simultaneously. Assume that each bus transaction is 100 clock cycles, as before. You can ignore the time of the actual read or write of a lock held in the cache as the time to execute other nonsynchronization operations in the barrier implementation. Determine the number of bus transactions required for all 10 processors to reach the barrier, be released from the barrier, and exit the barrier. Assume that the bus is totally fair, so that every pending request is serviced before a new request and that the processors are equally fast. Don’t worry about counting the processors out of the barrier. How long will the entire process take?

ANSWER: We assume that load linked and store conditional are used to implement lock and unlock. Figure 6.40 shows the sequence of bus events for a processor to traverse the barrier, assuming that the first process to grab the bus does not have the lock. There is a slight diffrence for the last process to reach the barrier, as described in the caption.

For the *i*th process, the number of bus transactions is: $3i+4$. The last process to reach the barrier requires one less. Thus for *n* processes, the number of bus transactions is:

$$(\sum(3i+4))-1=(3n^2+11n)/2-1$$

For 10 processes, this is 204 bus cycles or 20400 clock cycles! Our barrier operations takes almost twice as long as the 10-peocessor lock-unlock sequence.

Synchronization performance can be a real bottleneck when there is substantial contention among multiple processes.

When there is little contention and synchronization operations are infrequent, we are primarily concerned about the latency of a synchronization primitive—that is, how long it takes an individual process to complete a synchronization operation.

Basic spin-lock operation can do this in two bus cycles: one to initially read the lock and one to write it.

We could improve this to a single bus cycle by a variety of methods. For example, we could simply spin on the swap operation.

The more serious problem in these examples is the serialization of each process’s attempt to complete the synchronization. This serialization is a problem when there is contention, because it greatly increases the time to complete the synchronization operation.

Synchronization Mechanisms for Larger-Scale Multiprocessors

What we would like are synchronization mechanisms that have low latency in uncontended cases and that minimize serialization in the case where contention is significant.

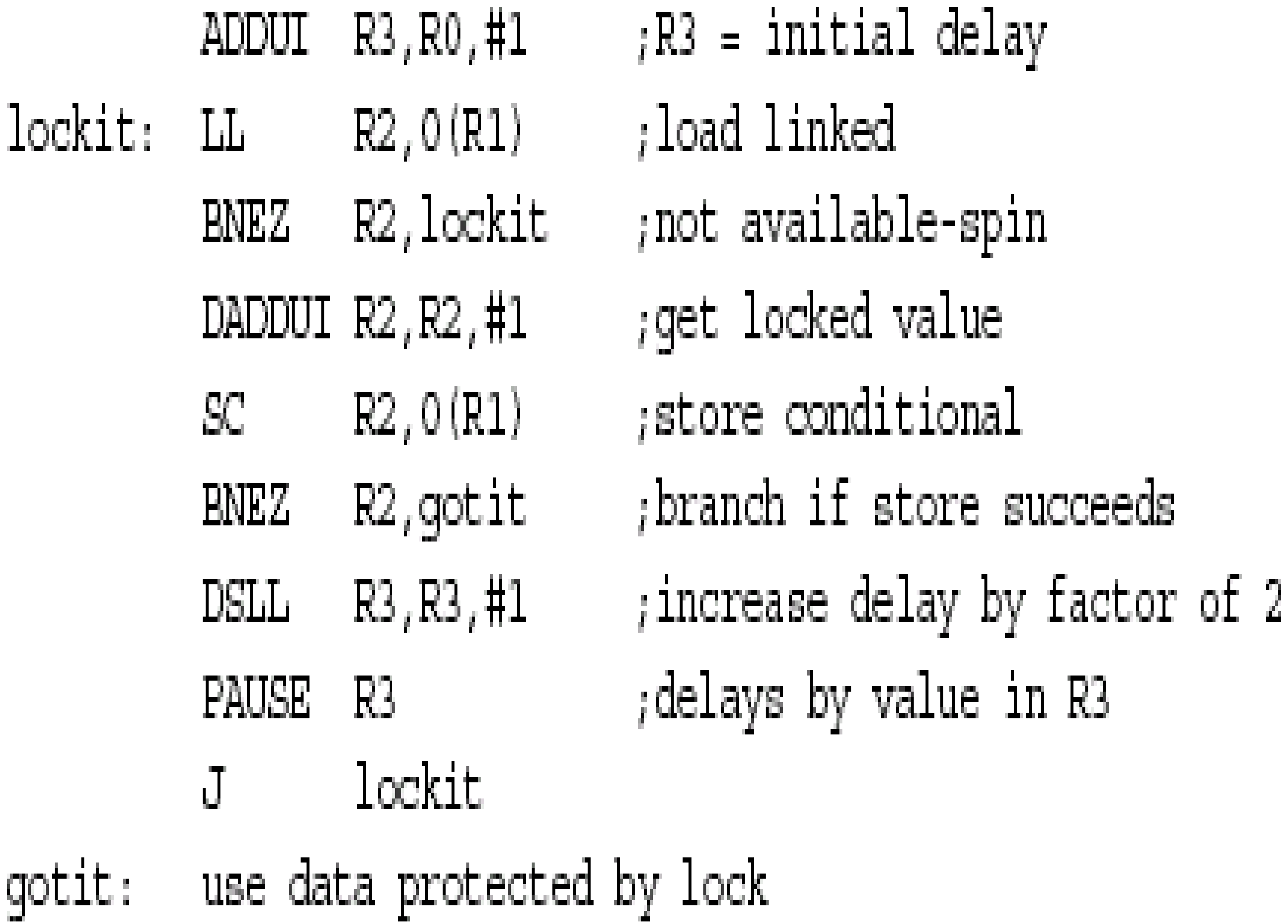
Software Implementations

The major difficulty with our spin-lock implementation is the delay due to contention when many processes are spinning on the lock.

One solution is to artificially delay processes when they fail to acquire the lock. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails.

Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses.

This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop.



```
        ADDUI  R3,R0,#1     ;R3 = initial delay
lockit: LL     R2,0(R1)     ;load linked
        BNEZ   R2,lockit    ;not available-spin
        DADDUI R2,R2,#1     ;get locked value
        SC     R2,0(R1)     ;store conditional
        BNEZ   R2,gotit     ;branch if store succeeds
        DSLL   R3,R3,#1     ;increase delay by factor of 2
        PAUSE  R3           ;delays by value in R3
        J      lockit
gotit:  use data protected by lock
```

Another technique for implementing locks is to use queuing locks.

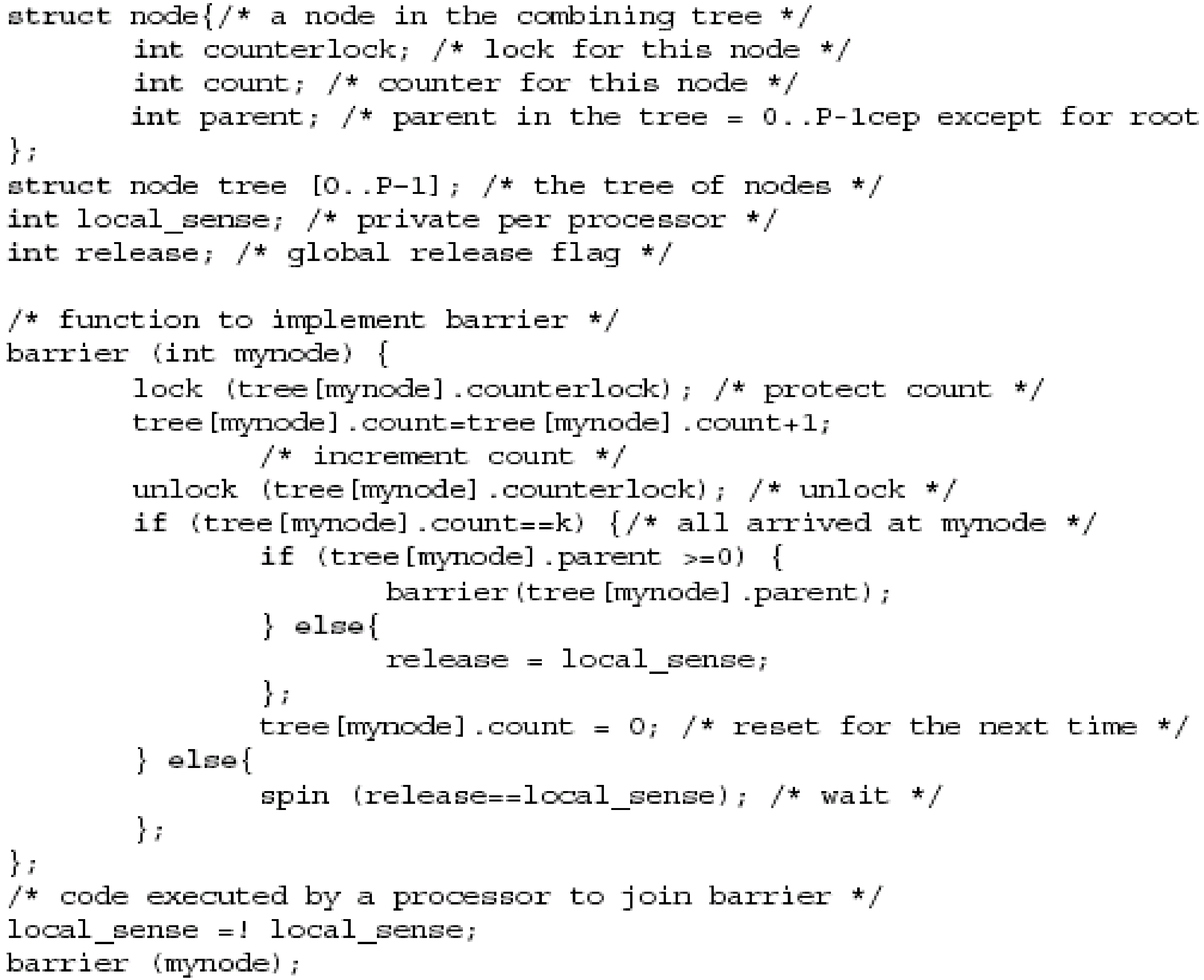
Queuing locks work by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access.

Our barrier implementation suffers from contention both during the *gather* stage, when we must atomically update the count, and at the *release* stage, when all the processes must read the release flag.

We can reduce the contention by using a *combining tree,* a structure where multiple requests are locally combined in tree fashion.

Our combining tree barrier uses a predetermined *n*-ary tree structure. We use the variable *k* to stand for the fan-in; in practice *k* = 4 seems to work well.

When the *k*th process arrives at a node in the tree, we signal the next level in the tree. When a process arrives at the root, we release all waiting processes.



```
struct node{/* a node in the combining tree */
        int counterlock; /* lock for this node */
        int count; /* counter for this node */
        int parent; /* parent in the tree = 0..P-1cep except for root
};
struct node tree [0..P-1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release; /* global release flag */

/* function to implement barrier */
barrier (int mynode) {
        lock (tree[mynode].counterlock); /* protect count */
        tree[mynode].count=tree[mynode].count+1;
                /* increment count */
        unlock (tree[mynode].counterlock); /* unlock */
        if (tree[mynode].count==k) {/* all arrived at mynode */
                if (tree[mynode].parent >=0) {
                        barrier(tree[mynode].parent);
                } else{
                        release = local_sense;
                };
                tree[mynode].count = 0; /* reset for the next time */
        } else{
                spin (release==local_sense); /* wait */
        };
};
/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);
```

Some MPPs (e.g., the T3D and CM-5) have also included hardware support for barriers, but more recent machines have relied on software libraries for this support.

Hardware Primitives

We look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices.

We can hand explicitly the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released.

We keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock*.

Queuing locks can be implemented either in hardware, which we describe here, or in software using an array to keep track of the waiting processes.

The basic concepts are the same in either case. Hardware implementation assumes a directory-based multiprocessor where the individual processor caches are addressable.

A queuing lock work:

On the first miss to the lock variable, the miss is sent to a synchronization controller, which may be integrated with the memory controller (in a bus-based system) or with the directory controller. If the lock is free, it is simply returned to the processor.

If the lock is unavailable, the controller creates a record of the node’s request (such as a bit in a vector) and sends the processor back a locked value for the variable, which the processor then spins on.

When the lock is freed, the controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processor’s cache or invalidate the copy, causing the processor to miss and fetch an available copy of the lock.

EXAMPLE How many bus transaction and how long does it take to have 10 processors lock and unlock the variable using a queuing lock that updates the lock on a miss? Make the other assumptions about the system the same as those in the earlier example on page 596.

ANSWER: For n processors, each will initially attenmpt a lock access, generating a bus transaction, one will succeed and free up the lock, for a total of n+1 transactions for the first processor. Each subsequent processor requires two bus transactions, one to receive the lock and one to free it up.Thus, the total number of bus transactions is (n+1)+2(n-1)=3n-1. Note that the number of bus transactions is now linear in the number of processors contending for the lock, rather than quadratic,as it was with the spin lock we examined earlier. For 10 processors, this requires 29 bus cycles or 2900 clock cycles.

Two key insights in implementing such a queuing lock capability:

1. We need to be able to distinguish the initial access to the lock, so we can perform the queuing operation, and also the lock release.

2. The queue of waiting processes can be implemented by a variety of mechanisms. In a directory-based multiprocessor, this queue is akin to the sharing set, and similar hardware can be used to implement the directory and queuing lock operations.

One complication is that the hardware must be prepared to reclaim such locks, since the process that requested the lock may have been context-switched and may not even be scheduled again on the same processor.

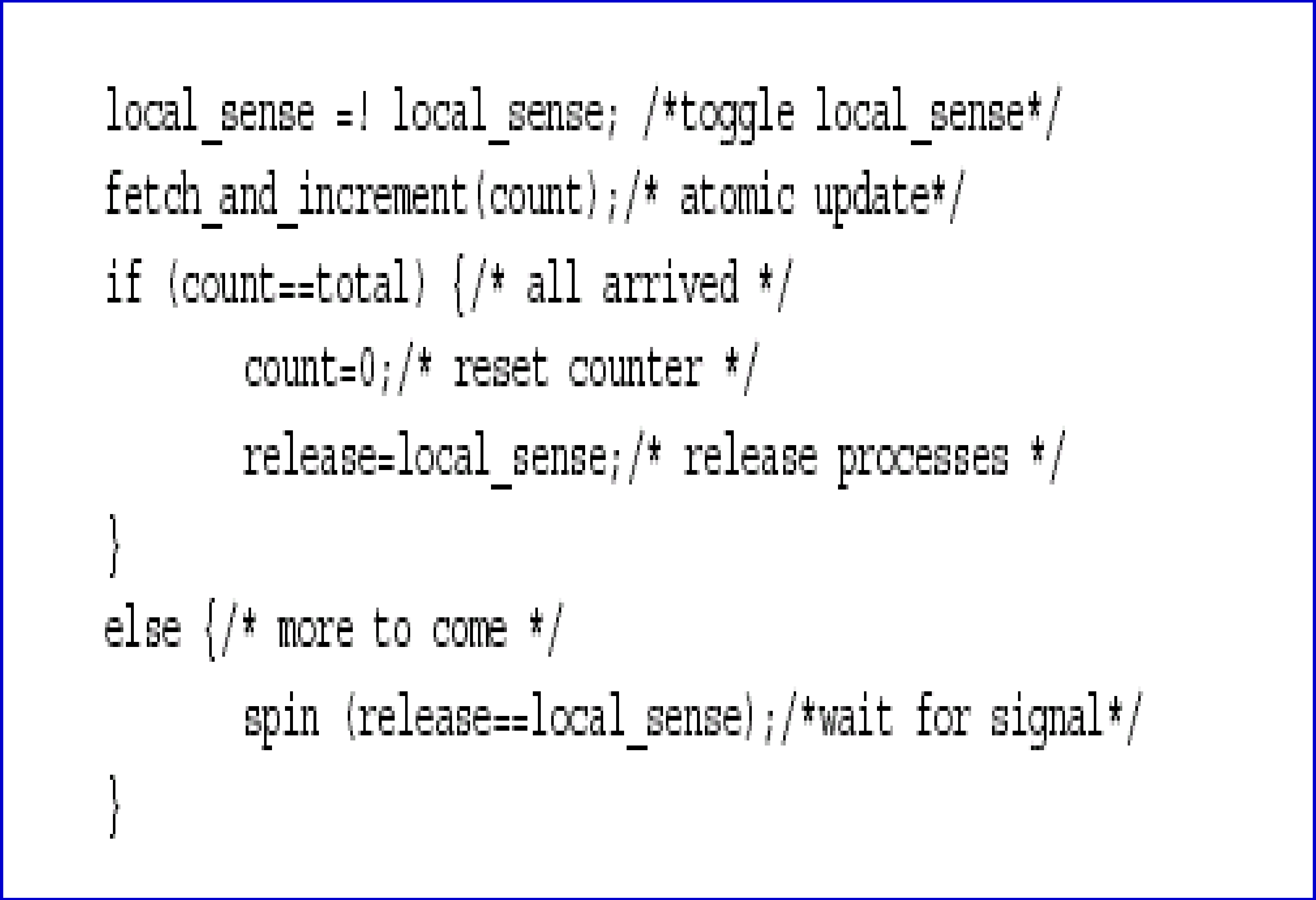
We can introduce a primitive that reduces the amount of time needed to increment the barrier count, thus reducing the serialization at this bottleneck, which should yield comparable performance to using queuing locks.

One primitive that has been introduced for this and for building other synchronization operations is *fetch-and-increment,* which atomically fetches a variable and increments its value.

The returned value can be either the incremented value or the fetched value. Using fetch-and-increment we can dramatically improve our barrier implementation, compared to the simple code sensing barrier.

EXAMPLE Write the code for the barrier using fetch-and-increment. Making the same assumptions as in our earlier example and also assuming that a fetch-and-increment operation takes 100 clock cycles, determine the time for 10 processors to traverse the barrier. How many bus cycles are required?

ANSWER Accoding to the code for the barrier, for n processors, this implementation requires n fetch-and-increment operations, n cache misses to access the count, and n cache misses for the release operation for a total of 3n bus transactions. For 10 processors, this is 30 bus transactions or 3000 clock cycles. Like the queueing lock, the time is linear in the number of processors. Of course, fetch-and-increment can also be used in implementing the combining tree barrier, reducing the serialization at each node in the tree.



```
local_sense =! local_sense; /*toggle local_sense*/
fetch_and_increment(count);/* atomic update*/
if (count==total) {/* all arrived */
        count=0;/* reset counter */
        release=local_sense;/* release processes */
}
else {/* more to come */
        spin (release==local_sense);/*wait for signal*/
}
```

As we have seen, synchronization problems can become quite acute in largerscale multiprocessors.

When the challenges posed by synchronization are combined with the challenges posed by long memory latency and potential load imbalance in computations, we can see why getting efficient usage of large-scale parallel processors is very challenging.