cally very expensive, which often annuls the advantage of doing the migration. The major reason for the high cost is not so much moving the page (which with the block transfer engine in the Hub takes about 25–30 μs for a 16-KB page) as changing the virtual-to-physical mappings in the TLBs of all processors that have referenced the page. Migrating a page keeps the virtual address the same but changes the physical address, so the old mappings in the page tables of processes are now invalid. As page table entries are changed, it is important that the cached versions of those entries in the TLBs of processors be invalidated (much like TLB shootdown discussed in Chapter 6). In fact, all processors must be sent a TLB invalidation message since we don't know which ones have a mapping for the page cached in their TLB. The processors are interrupted, and the invalidating processor has to wait for the last among them to respond before it can update the page table entry and continue. This process typically takes over 100 μs, in addition to the cost to move the page itself.

To reduce this cost, Origin uses a distributed, lazy TLB invalidation mechanism supported by its seventh directory state, the poisoned state. The idea is not to invalidate TLB entries when the page is moved but rather to invalidate a processor's TLB entry only when that processor next references the page. Not only is the time to invalidate all TLBs removed from the critical path of the processor that manages the migration, but TLB entries end up being invalidated for only those processors that subsequently reference the page. Let's see how this works. To migrate a page, a block transfer engine reads all cache blocks from the source page location using special "uncached read-exclusive" requests. This request type returns the latest coherent copy of the data and invalidates any existing cached copies (like a regular read-exclusive request), but it also causes the destination main memory to be updated with the latest version of the block and puts the directory in the poisoned state. The migration itself takes only the time to do this block transfer. When a processor next tries to access a block from the old physical page, using its stale TLB entry, it will miss in the cache and will find the block in poisoned state at the directory. At that time, the poisoned state will cause the requesting processor to see a bus error. The special OS handler for this bus error invalidates the processor's TLB entry so that it will obtain the new mapping from the page table when it retries the access. Of course, the old physical page must be reclaimed by the system at some point to avoid wasting storage. Once the block transfer has completed, the OS invalidates one TLB entry per time quantum of the OS scheduler so that after some fixed amount of time the old page can be moved on to the free list.

Support for Synchronization

Origin provides two types of support for synchronization. First, the load-locked store-conditional (LL-SC) instructions of the R10000 processor are available to compose synchronization operations, as we saw in the previous chapters. Second, for situations in which many processors contend to update a location, such as a global counter or a barrier, uncached fetch&op primitives are provided. These fetch&op operations are performed at the main memory; the block is not replicated in the

caches, so successive nodes trying to update the location do not have to retrieve it from the previous writer's cache. The cacheable LL-SC is better when the same node tends to repeatedly access the shared (synchronization) variable, and the uncached fetch&op is better when different nodes tend to update in an interleaved or contended way. Producer-consumer communication of event synchronization can also be aided by uncached write and read operations since at most two network transactions are needed instead of four and since the producer and consumer transactions may even overlap on their way to the home node. However, spinning on a remote uncached location may cause a lot of traffic.

8.5.5 Overview of the Origin2000 Hardware

The preceding protocol discussion has provided us with a fairly complete picture of how a flat, memory-based directory protocol is implemented out of network transactions and state transitions, just as a bus-based protocol was implemented out of bus transactions and state transitions. Let us now turn our attention to the actual hardware of the Origin2000 machine that implements this protocol. This subsection provides an overview of the system hardware organization and is followed by a deeper examination of how the Hub controller is actually implemented (in Section 8.5.6). Finally, the performance of the machine is discussed in Section 8.5.7. (Readers interested in only the protocol can skip the rest of this section without loss of continuity.)

In addition to the two MIPS R10000 processors connected by a system bus, each node of the Origin2000 contains a fraction of the main memory on the machine (1–4 GB per node), the Hub (which is the combined communication/coherence controller and network interface), and an I/O interface called the Xbow. All components but the Xbow are on a single 16" × 11" printed circuit board. Each processor in a node has its own separate L_1 and L_2 caches, with the L_2 cache configurable from 1 to 4 MB with a cache block size of 128 bytes and two-way set associativity. There is one directory entry per main memory block. Memory is interleaved from 4 ways to 32 ways, depending on the number of modules plugged in (4-way interleaving at 4-KB granularity within a module and up to 32-way at 512-MB granularity across modules). The system has up to 512 such nodes, that is, up to 1,024 processors. With a 195-MHz R10000 processor, the peak performance per processor is 390 MFLOPS or 780 MIPS (four instructions per cycle), leading to an aggregate peak performance of almost 500 GFLOPS in a maximally sized machine. The peak bandwidth of the SysAD bus that connects the two processors is 780 MB/s, as is that of the Hub's connection to memory. Memory bandwidth itself for data is about 670 MB/s. The Hub connections to the off-board network router chip and Xbow I/O interface are 1.56 GB/s each, using the same link technology. A detailed picture of the node board is shown in Figure 8.19.

The Hub chip is the heart of the machine. It sits on the system bus of the node and connects the processors, local memory, network, and Xbow, which communicate with one another through it. All cache misses, whether to local or remote memory, go through the Hub (which implements the coherence protocol), as do all

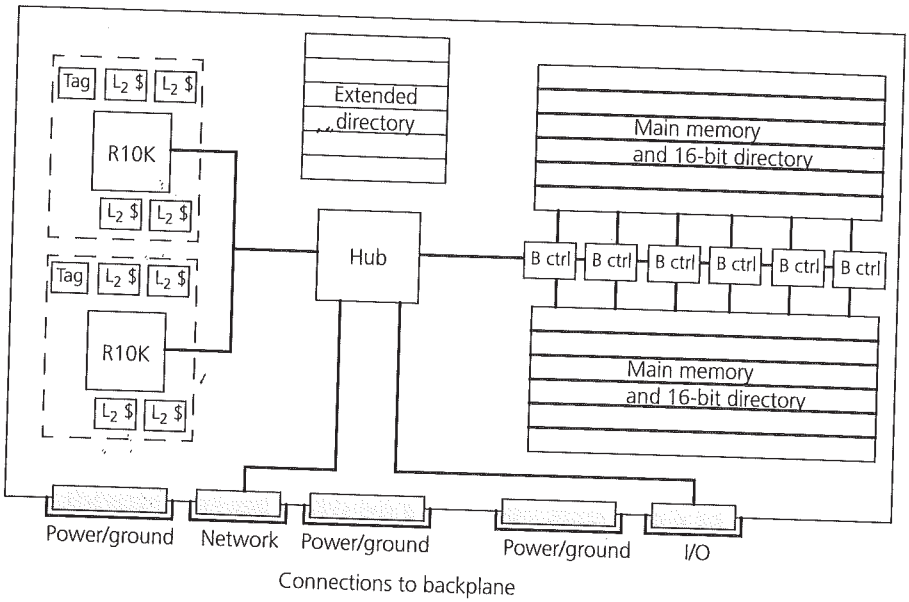


FIGURE 8.19 A node board on the Origin multiprocessor. "L_2 $" stands for secondary cache chips and "B ctrl" for memory bank controller.

uncached operations. It is a highly integrated, 500-K gate standard-cell design in 0.5-μ CMOS technology. It contains outstanding transaction buffers for each of its two processors (each processor itself allows four outstanding requests), a pair of block transfer engines that support block memory copy and fill operations at full system bus bandwidth, and interfaces for the network, the SysAD bus, the memory/directory, and the I/O subsystem. The Hub also implements the at-memory, uncached fetch&op instructions and page migration support discussed earlier.

The interconnection network has a hypercube topology for machines with up to 64 processors but a different topology, called a *fat cube*, beyond that. (This topology is discussed in Chapter 10.) Each router supports six links. The network links have high bandwidth (1.56 GB/s total per link in the two directions) and low latency (41 ns pin-to-pin through a router) and can use flexible cabling up to three feet long for the links. Each link supports four *virtual channels*. Virtual channels are described in Chapter 10; for now, we can think of the machine as having four distinct networks such that each has about one-fourth of the physical link bandwidth. One of these virtual channels is reserved for request network transactions, one for responses. Two can be used for congestion relief and high-priority transactions, thereby violating point-to-point order, or can be reserved for I/O as is usually done.

The Xbow chip connects the Hub to other I/O interfaces. It is itself implemented as a crossbar with eight ports. Typically, two nodes (Hubs) might be connected to one Xbow and, through it, to six external I/O cards as shown in Figure 8.20. The Xbow is quite similar to the router chip (called SPIDER) but with simpler buffering

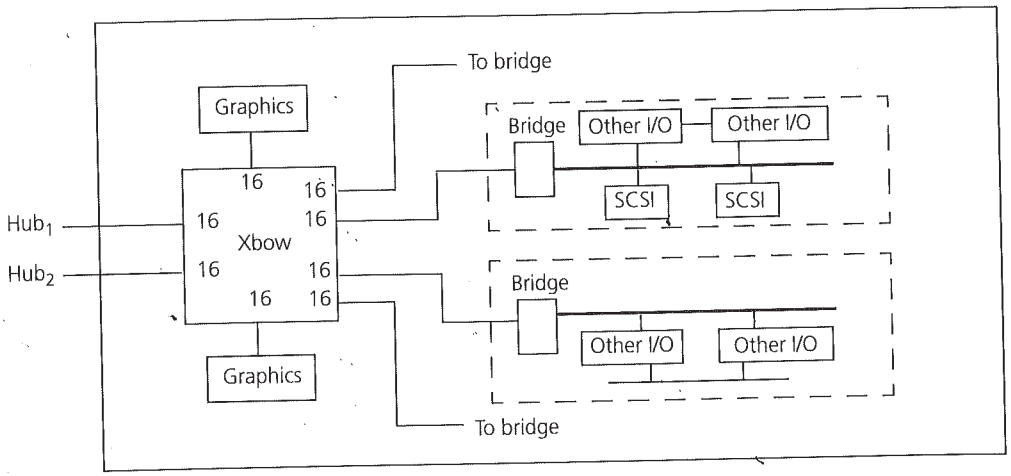


FIGURE 8.20 Typical Origin I/O configuration shared by two nodes. High-performance graphics devices connect directly to the Xbow, while other I/O devices connect to I/O buses that are linked to the Xbow through bridges.

and arbitration that allow eight ports to fit on the chip rather than six. The arbiter also supports the reservation of bandwidth for certain devices to support real-time needs like video I/O. High-performance I/O cards like graphics connect directly to the Xbow ports, but most other ports are connected through a bridge and an I/O bus that allows multiple cards to plug into it. Any processor can reference any physical I/O device in the machine, either through uncached references to a special I/O address space or through coherent DMA operations. An I/O device, too, can transfer data to and from any memory in the system, not just the memory on the node to which it is directly connected through the Xbow, thus taking advantage of the shared address space. Communication between the processor and the appropriate Xbow is handled transparently by the Hubs and network routers. Thus, like memory, I/O is physically distributed but globally accessible, so locality in I/O distribution is also a performance rather than correctness issue.

8.5.6 Hub Implementation

The communication assist—the Hub—must have certain basic abilities to implement the coherence protocol. It must be able to observe all cache misses, synchronization events, and uncached operations; keep track of outgoing requests while moving on to handle other outgoing and incoming transactions; guarantee the sinking of responses coming in from the network; invalidate cache blocks; and intervene in the caches to retrieve data. It must also coordinate the activities and dependences of all the different types of transactions that flow through it from different components and implement the necessary pathways and control. The design of such controllers is, therefore, challenging. This subsection briefly describes the major

components of the Hub controller used in the Origin2000 and points out some of its salient features used to implement the coherence protocol. Further details of the actual data and control pathways through the Hub, as well as the mechanisms used to actually control the interactions among messages, are also useful for understanding how scalable cache coherence is implemented and can be read elsewhere (Singh 1997).

The Hub is divided into four major interfaces, one for each type of external entity that it connects together: the processor interface or PI, the memory/directory interface or MI, the network interface or NI, and the I/O interface or II (see Figure 8.21). These interfaces communicate with one another through an on-chip crossbar switch. Each interface is divided into a few major structures, including FIFO queues to buffer messages to/from other interfaces and to/from external entities. A key property of the design is for each interface to shield its external entity from the details of other interfaces and entities (and vice versa). For example, the PI hides the processors from the rest of the world, so any other interface must only know the behavior of the PI and not of the processors and SysAD bus themselves. Let us discuss the structures of the PI, MI, and NI briefly, as well as some examples of the shielding provided by the interfaces.

The Processor Interface (PI)

The PI has the most complex control mechanisms among the interfaces since it keeps track of outstanding protocol requests and responses and must match them. The PI interfaces with the memory (SysAD) buses of the two R10000 processors on one side and with incoming and outgoing FIFO queues connecting it to each of the other Hub interfaces on the other side (Figure 8.21). Each physical FIFO is logically separated into independent request and response "virtual FIFOs" by providing separate logic and staging buffers. In addition, the PI itself contains three pairs of coherence control buffers that keep track of outstanding transactions, control the flow of messages through the PI, and implement the interactions among messages dictated by the protocol. These buffers do not, however, hold the messages themselves. There are two read request buffers (RRBs) that track outstanding read requests from each processor, two write request buffers (WRBs) that track outstanding write requests, and two intervention request buffers (IRBs) that track incoming invalidation and intervention requests. Access to the three sets of buffers is through a single bus, so all messages contend for access to them.

A message that is recorded in one type of buffer may also need to look up another type to check for conflicting accesses or interventions to the same address from the processor. For example, an outgoing read request performs an associative lookup in the WRB to see if a write back to the same address is pending as well. If there is a conflicting WRB entry, a read request is not placed in the PI's outgoing request FIFO; rather, a bit is set in the RRB entry to indicate that when the WRB entry is freed, the read request should be reissued (i.e., when the write back is acknowledged or is canceled by an incoming invalidation as per the protocol). Buffers are also looked up to close an outstanding PI transaction in them when a completion response comes in

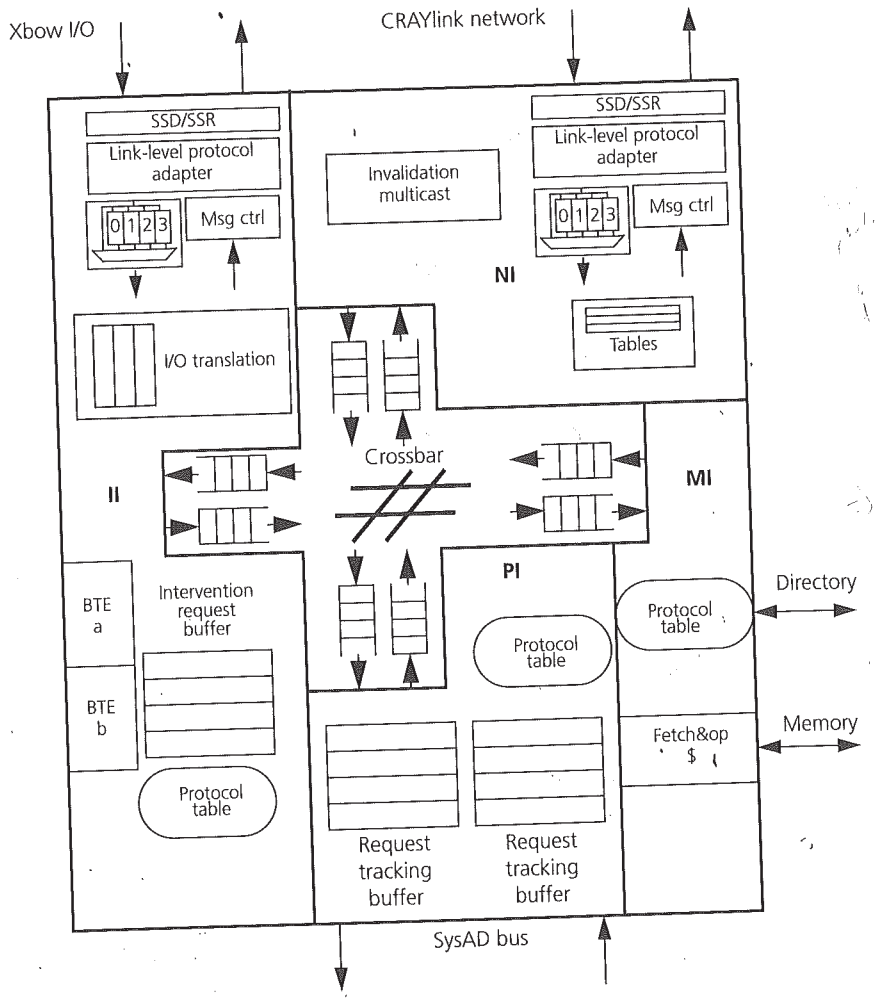


FIGURE 8.21 Layout of the Hub chip. The crossbar at the center connects the buffers of the four different interfaces. Clockwise from the bottom left, the BTEs are the block transfer engines. The top left corner is the I/O interface or II (the SSD and SSR translate signals to and from the I/O ports). Next is the network interface (NI), including the routing tables. The bottom right is the memory/directory interface (MI), and at the bottom is the processor interface (PI) with its request tracking buffers.

from either the processors in the node or from another interface. Since the order of transactions closing is not deterministic, a new transaction must go into any available slot, so these tracking buffers are implemented as fully associative rather than FIFO buffers (the queues that hold the actual messages are FIFO). The buffer lookups determine whether the PI should issue a request to either a processor or the other interfaces.

The PI is a good example of the shielding provided by interfaces. If the processor (or cache) provides data as a reply to an incoming intervention, it is the logic in the PI's outgoing FIFO that expands the reply into the two responses required by the protocol, one to the home as a sharing write-back revision message and one to the requestor. The processor itself does not have to be modified to generate two replies. Another example is in the mechanisms used to keep track of and match incoming and outgoing requests and responses. All requests passing through the PI in either direction are given request numbers, and responses carry these request numbers as well. However, the processor itself does not know about request numbers, and it is the PI's job to ensure that when it passes on incoming requests (interventions or invalidations) to the processor, it can match the processor's responses to the outstanding interventions/invalidations without the processor having to deal with request numbers.

The Memory/Directory Interface (MI)

The MI also has FIFOs between it and the Hub crossbar. The FIFO from the Hub crossbar to the MI separates headers from data so that the header of the next message can be examined by the directory while the current one is being serviced; this allows writes to be pipelined and performed at peak memory bandwidth. The MI also contains a directory interface, a memory interface, and a controller. The directory interface contains the logic and tables that determine what protocol actions to take and hence implement the coherence protocol. It also contains the logic that generates outgoing message headers, while the memory interface contains the logic that generates outgoing message data. Both the memory and directory RAMS have their own address and data buses. Some messages, like revision messages coming to the home, may not access the memory but only the directory.

On a read request, the read is issued to memory at the home speculatively, simultaneously with starting the directory operation. The directory state is available a cycle before the memory data, and the controller uses this (plus the message type and initiator) to look up the directory protocol table. This hardwired table directs the controller to the action to be taken and the message to send. The directory block sends the latter information to the memory interface, where the message headers are assembled and inserted into the outgoing FIFO together with the data returning from memory. The directory lookup itself is a read-modify-write operation. For this, the MI provides support for partial writes of memory blocks and a one-entry merge buffer to hold the bytes from the time they are read from memory to the time they are written back. Finally, to speed up the at-memory fetch&op accesses provided for synchronization, the MI contains a four-entry LRU fetch&op cache to hold the data for recent fetch&op variables and, hence, to avoid a memory or directory access. This reduces the best-case serialization time at memory for a fetch&op to 41 ns, about four 100-MHz Hub cycles.

The Network Interface (NI)

The NI interfaces the Hub crossbar to the network router for that node. The router and the Hub internals use different data transport formats, protocols, and speeds (100 MHz in the Hub versus 400 MHz in the router), so one major function of the NI is to translate between the two. Toward the router side, the NI implements a flow control mechanism to avoid network congestion (Singh 1997). The FIFOs between the NI and the network also implement separate virtual FIFOs for requests and responses, thus implementing separate virtual networks. The outgoing FIFO also has an invalidation destination generator that takes the bit vector of nodes to be invalidated and generates individual messages for them, a routing table that predetermines the routing decisions based on source and destination nodes, and virtual channel selection logic.

8.5.7 Performance Characteristics

The peak hardware bandwidths of the Origin2000 system were stated earlier: 780-MB/s SysAD bus, 670-MB/s local memory, and 780-MB/s node-to-network each way. The occupancy of the Hub at the home for a transaction on a cache block is about 20 Hub cycles (about 40 processor cycles), though it varies between 18 and 30 Hub cycles depending on whether successive directory pages accessed are in the same bank of the directory RAM and on the exact pattern of successive transactions. The latencies of memory operations depend on many factors, such as the type of operation, whether the home is local or not, where and in what state the data is currently cached, and how much contention there is for resources along the way. The latencies can be measured using microbenchmarks. Let us examine microbenchmark results for latency and bandwidth first, followed by the performance and scaling of our six parallel applications.

Characterization with Microbenchmarks

Unlike the MIPS R4400 processor used in the SGI Challenge, the Origin's MIPS R10000 processor is dynamically scheduled and does not stall on a read miss. This makes it more difficult to measure read latency, raising an interesting methodological issue. We cannot, for example, measure the unloaded latency of a read miss by simply executing the microbenchmark from Chapter 4 that reads the elements of an array with stride greater than the cache block size. Since the misses are to different locations, subsequent misses will simply be overlapped with one another and the processor will not see their full latency. Instead, this microbenchmark will give us a measure of the throughput that the system can provide on successive read misses issued from a processor. The throughput is the inverse of the latency remaining after overlap, which we can call the *pipelined latency*.

To measure the full latency, we need to ensure that subsequent operations are dependent on each other. To do this, we can use a microbenchmark that chases pointers down a linked list: the address for the next read is not available to the pro-

Table 8.1 Back-to-Back and True Unloaded Latencies for Different System Sizes

Where Miss is Satisfied	Network Routers Traversed	Back-to-Back Latency (ns)	True Unloaded Latency (ns)
L_1 cache	0	5.5	5.5
L_2 cache	0	56.9	56.9
Local memory	0	472	329
4P remote memory	1	582	449
8P remote memory	2	775	621
16P remote memory	3	826	702

The first column shows where in the extended memory hierarchy the misses are satisfied. For the 8P case, for example, the misses are satisfied in the node furthest away from the requestor in a system of 8 processors. Given the Origin2000 topology, this means traversing through two network routers in this case.

cessor until the previous read (of the pointer) completes, so the reads cannot be overlapped. However, it turns out this is a little pessimistic in determining the unloaded read latency. The reason is that the processor implements critical word restart; that is, it can use the value returned by a read as soon as that word is returned to the processor, without waiting for the rest of the cache block to be loaded in the caches. With the pointer-chasing microbenchmark, the next read will be issued before the previous block has been loaded and will contend for cache access with the loading of the rest of that block. The latency obtained from this microbenchmark, which includes this contention, can be called *back-to-back latency* (one read miss issued just as the previous one completes). Avoiding this contention between successive accesses requires that we put some computation between the read misses; the computation should depend on the data being read, so it cannot execute in parallel with the read miss, but should not access the cache between two misses. The goal is to have this computation overlap the time it takes for the rest of the cache block to load into the caches after a read miss so that the next read miss will not have to stall on cache access. The time for this overlap computation must, of course, be subtracted from the elapsed time of the microbenchmark to measure the true unloaded read-miss latency, assuming critical word restart. We can call this the *true unloaded latency*. Table 8.1 shows the back-to-back and true unloaded latencies measured on the Origin2000. Only one processor executes the microbenchmark, but the data that is accessed is distributed among the memories of different numbers of processors. The back-to-back latency is usually about 13 SysAD bus cycles (133 ns) longer because the L_2 cache block size (128 B) is 12 double words longer than the L_1 cache block size (32 B) and there is one cycle for bus turnaround.

Table 8.2 lists the back-to-back latencies for different initial states of the block being referenced (Hristea, Lenoski, and Keen 1997). Recall that the owner node is the home node when the block is in unowned or shared state at the directory and is the node that has a cached copy when the block is in exclusive state. The true unloaded latency for the case where both the home and the owner are the local node

Table 8.2 Back-to-Back Latencies (in ns) for Different Initial States of the Block

		State of Block		
Home	Owner	Unowned	Clean-Exclusive	Modified
Local	Local	472	707	1,036
Remote	Local	704	930	1,272
Local	Remote	472	930	1,159
Remote	Remote	704	917	1,097

The first column indicates whether the home of the block is local or not, the second indicates whether the current owner is local or not, and the last three columns give the latencies for the block being in different states. Of course, the owner node should be ignored for the unowned state.

(i.e., if the block is owned by main memory, the other processor in the same node) is 338 ns for the unowned state, 656 ns for the clean-exclusive state, and 892 ns for the modified state. Note that no contention is encountered with operations from other processors in this microbenchmark; latencies under real workloads will be larger.

Application Speedups

Figure 8.22 shows the speedups for the six parallel applications on a 32-processor Origin2000, using two problem sizes for each application. We see that most of the applications speed up well, especially once the problem size is large enough. The dependence on problem size is particularly stark in applications like Ocean and Raytrace. The exceptions to good speedup at this scale are Radiosity and, to an extent, Radix. In the case of Radiosity, even the larger problem is relatively small for a machine of this size and power. We can expect to see better speedups for larger scenes. For Radix, the problem is the highly scattered, bursty pattern of writes in the permutation phase. These writes are mostly to locations that are allocated remotely, and the flood of requests to and from the directories, invalidations, acknowledgments, and replies that they generate causes tremendous contention and hot spotting at Hubs and memories. Running larger problems alleviates only false sharing, since there is no other computation than the data permutation during this phase so the communication-to-computation ratio is essentially independent of problem size; in fact, the situation worsens once a processor's partition of the keys does not fit in its cache, at which point frequent write-back transactions are also thrown into the mix. For applications like Radix (and an FFT, not shown) that exhibit all-to-all bursty communication, the fact that two processors share a Hub and two Hubs share a router also causes contention at these resources, despite their high peak bandwidths (Jiang and Singh 1998). For these applications, the machine would perform better if it had only a single processor per Hub and per router. However, the sharing of resources does reduce cost and does not get in the way of the other applications.

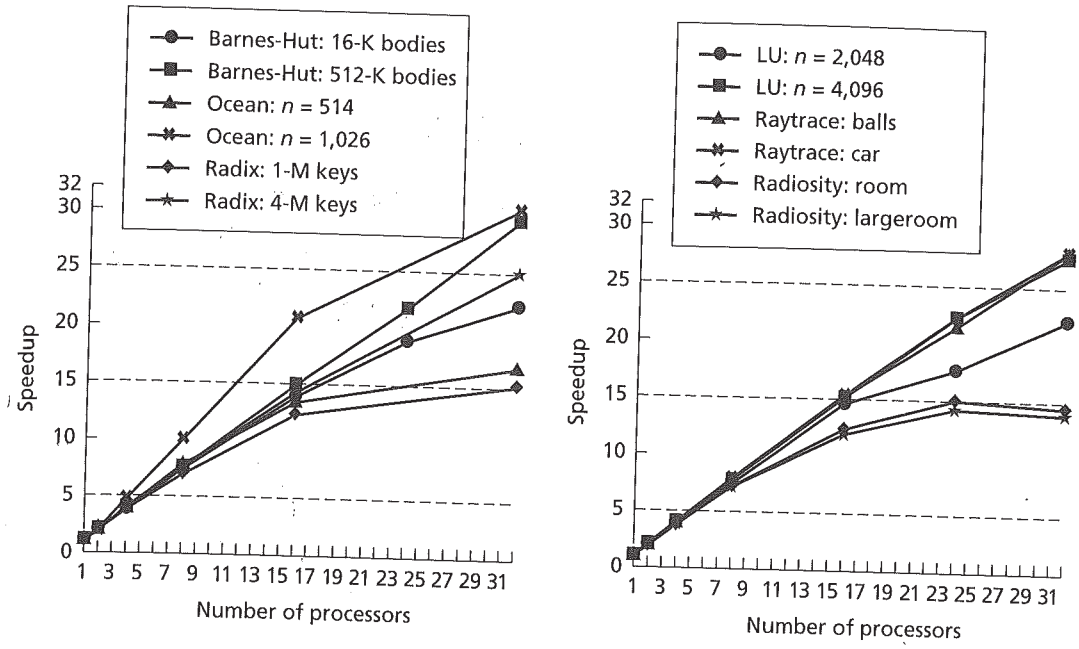


FIGURE 8.22 Speedups for the parallel applications on the Origin2000. Two problem sizes are shown for each application. The Radix sorting program does not scale well, and the Radiosity application is limited by the available input problem sizes. The other applications speed up quite well when reasonably large problem sizes are used.

Breakdowns of execution time into components on a per-processor basis on this machine were shown in Chapters 3 and 4, giving us a good idea of where time is spent.

Scaling

Figure 8.23 shows the speedups under different scaling models for the Barnes-Hut galaxy simulation on the Origin2000. The results are quite similar to those on the SGI Challenge in Chapter 6—although extended to more processors—and the analysis there largely applies. For applications like Ocean (not shown), in which an important working set is proportional to the data set size per processor, machines like the Origin2000 display an interesting effect in comparing scaling models when we start from a problem size where the working set does not fit in the cache on a uniprocessor. Under PC and TC scaling, the data set size per processor diminishes with an increasing number of processors. Thus, although the communication-to-computation ratio increases, we observe superlinear speedups once the working set starts to fit in the cache (since the performance within each node becomes much

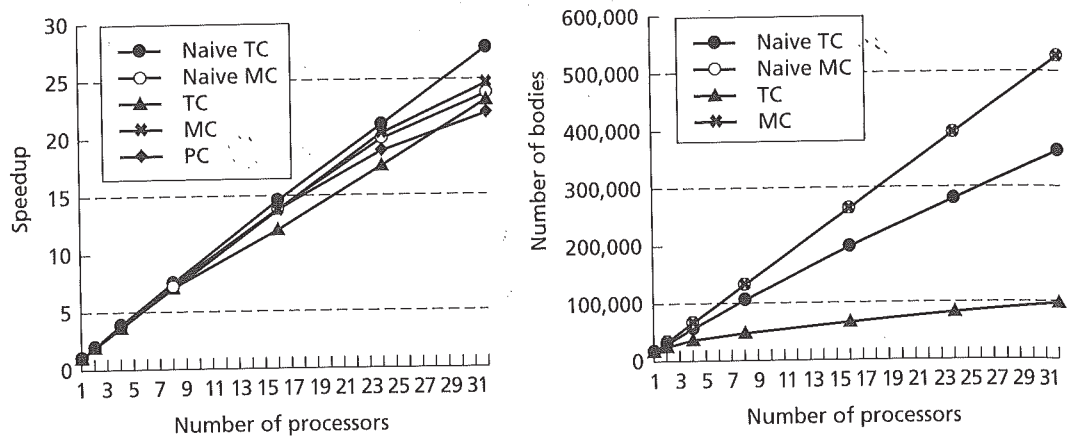


FIGURE 8.23 Scaling of speedups and number of bodies simulated under different scaling models for the Barnes-Hut galaxy simulation on the Origin2000. As with the results for bus-based machines in Chapter 6, the speedups are very good under all scaling models, and the number of bodies that can be simulated grows much more slowly under realistic TC scaling than under MC or naive TC scaling.

better when the working set fits in the cache). Under MC scaling, the communication-to-computation ratio does not change, but neither does the working set size per processor. As a result, although the demands on the communication architecture scale more favorably under MC scaling than under TC or PC scaling (the capacity misses due to the working sets are almost entirely local), speedups are not so good because the beneficial effect on node performance of the working set suddenly fitting in the cache is no longer observed. Also, even local capacity misses occupy the Hub and memory, contributing to contention.

8.6 CACHE-BASED DIRECTORY PROTOCOLS: THE SEQUENT NUMA-Q

The flat, cache-based directory protocol described in our second case study is the IEEE standard Scalable Coherent Interface (SCI) protocol (Gustavson 1992). As a case study of this protocol, we examine the NUMA-Q machine from Sequent Computer Systems, Inc., a machine targeted toward commercial workloads such as databases and transaction processing (Lovett and Clapp 1996). This machine relies heavily on third-party commodity hardware, using stock Intel SMPs as the processing nodes, stock I/O links, and the DataPump network interface from Vitesse Semiconductor Corporation to move data between the node and the network. The only customization is in the IQ-Link board used to implement the SCI directory protocol. A similar directory protocol is also used (with much more customization) in the Convex Exemplar series of machines (Convex Computer Corporation 1993; Thekkath et al. 1997), which, like the SGI Origin, is targeted more toward scientific computing.

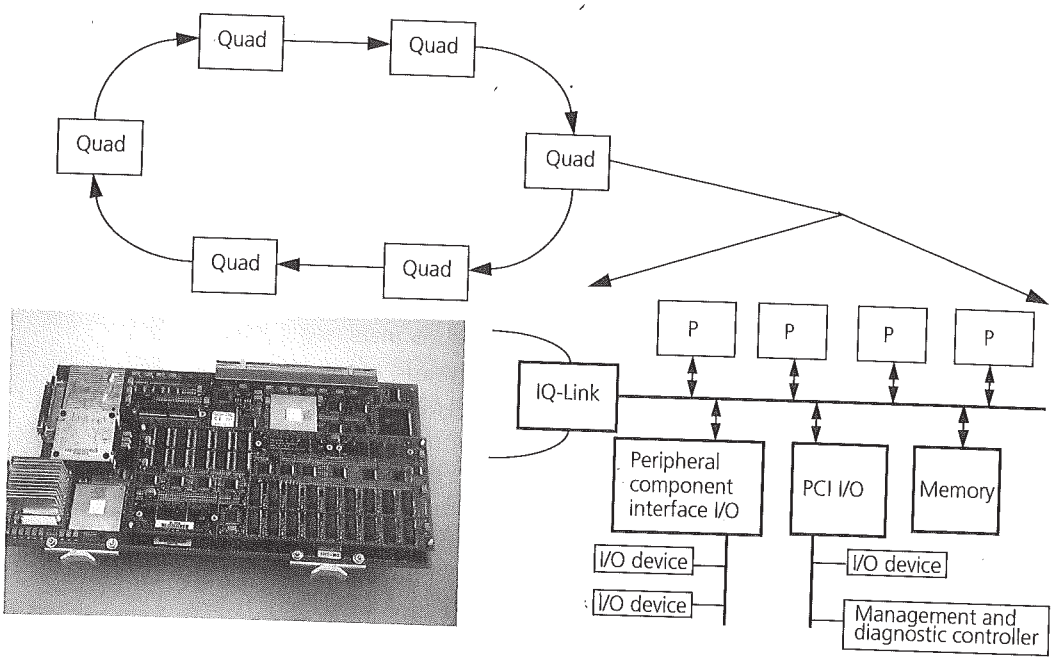


FIGURE 8.24 Block diagram of the Sequent NUMA-Q multiprocessor. The diagram shows the high-level organization of the machine, both across nodes and within a node. The photograph shows an IQ-Link board. *Source:* Photo courtesy of Sequent Computer Systems, Inc.

NUMA-Q is a collection of homogeneous processing nodes interconnected by high-speed links in a ring configuration (Figure 8.24). Each processing node is an inexpensive Intel quad bus-based multiprocessor with four Intel Pentium Pro microprocessors, which illustrates the use of high-volume SMPs as building blocks for larger systems. Systems from Data General (Clark and Alnes 1996) and from HAL Computer Systems (Weber et al. 1997) also use Pentium Pro quads as their processing nodes, the former also using an SCI protocol similar to NUMA-Q across quads and the latter using a memory-based protocol inspired by the Stanford DASH protocol. (In the Convex Exemplar series, the individual nodes connected by the SCI protocol are not bus based but are small directory-based multiprocessors kept internally coherent by a different directory protocol.) We described the quad SMP node in Chapter 1 (see Figure 1.17) and so do not discuss it further.

The IQ-Link board in each quad plugs into the quad memory bus and takes the place of the Hub in the SGI Origin. In addition to the directory logic and storage and the datapath between the quad bus and the network, it also contains a large (expandable) 32-MB, four-way set-associative remote access cache for blocks that are fetched to the node from remote memory. This *remote access cache,* hereafter called the *remote cache,* represents the quad to the cross-node SCI directory protocol. It is

the only cache in the quad that is visible to that protocol; the individual processor caches are kept coherent with the remote cache through the snooping bus protocol within the quad. The directory protocol is for the most part oblivious to how many processors there are within a node and even to the bus protocol itself. Inclusion is preserved between the remote cache and the processor caches within the node, so if a block is replaced from the remote cache it is invalidated in the processor caches, and if a block is placed in modified state in a processor cache then the state in the remote cache reflects this. The cache block size of the remote cache is 64 bytes, which is therefore the granularity of both communication and coherence across quads.

8.6.1 Cache Coherence Protocol

While two interacting coherence protocols are used in the Sequent NUMA-Q machine, this section focuses on the SCI directory protocol across remote caches and ignores the multiprocessor nature of the quad nodes. Interactions with the snooping MESI protocol within the quads are discussed in Section 8.6.5.

Directory Structure

The directory structure of SCI is the flat, cache-based distributed doubly linked-list scheme that was described in Section 8.2.3 and illustrated in Figure 8.8. There is a linked list of sharers per block, and the pointer to the head of this list is stored with the main memory that is the home of the corresponding memory block. An entry in the list corresponds to a remote cache in a quad. The remote cache is stored in synchronous DRAM memory in the IQ-Link board of that quad, together with the forward and backward pointers for the list. Figure 8.25 shows a simplified representation of a list. The first element (node) is called the head of the list and the last node the tail. The head node has both read and write permission on its cached block whereas the other nodes have only read permission (except in a special-case extension, called pairwise sharing, that we discuss briefly in Section 8.6.3). The pointer in a node that points to its neighbor in the direction toward the tail of the list is called the forward or downstream pointer, and the other is called the backward or upstream pointer. Let us see how the cross-node SCI coherence protocol uses this directory representation.

States

Since processor caches are not visible to the directory protocol, and since a block never enters the remote cache at its home node, unlike in the Origin, the directory protocol in the NUMA-Q does not keep track of cached copies at the home. Keeping the copy in the home memory coherent with these cached copies is the job of the bus protocol. A block in main memory can be in one of three directory states whose names are defined by the SCI protocol as follows. The states are similar to but not the same as the directory states in the Origin protocol.

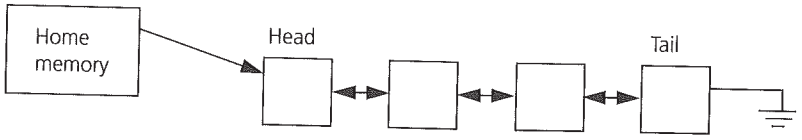


FIGURE 8.25 An SCI sharing list. Each element of the list in NUMA-Q is a multiprocessor node, represented by its remote cache.

- *Home*: No remote cache (quad) in the system contains a copy of the block (of course, a processor cache in the home quad itself may have a copy since this is not visible to the SCI coherence protocol but is managed by the bus protocol within the quad). This is like the unowned directory state in the Origin.
- *Fresh*: One or more remote caches may have a read-only copy, and the copy in memory is valid.This is like the shared state in the Origin.
- *Gone*: Another remote cache contains a writable (exclusive or dirty) copy. No valid copy exists on the local node. This is like the exclusive directory state in the Origin.

Consider the cache states for blocks in a remote cache. While the processor caches within a quad use the standard MESI stable states, the SCI scheme that governs the remote caches has a large number of possible cache states. In fact, 7 bits are used to represent the state of a block in a remote cache, and the standard describes 29 stable states and many pending (busy) or transient states. Each stable state can be thought of as having two parts, which is reflected in the naming structure of the states. The first part describes where that cache entry is located in the sharing list for that block. This may be `ONLY` (for a single-node list), `HEAD`, `TAIL`, or `MID` (which means neither the head nor the tail of a multiple-node list). The second part describes the actual state of the cached block. This includes states like dirty (modified and writable); clean (unmodified, same contents as memory, but writable, like the exclusive state in MESI); fresh (data may be read but may not be written until memory is informed); copy (unmodified and readable); and several others. A full description can be found in the SCI standards document (IEEE Computer Society 1993). We shall encounter some of these states (such as `HEAD-DIRTY`, `TAIL-CLEAN`, etc.) as we go along.

The SCI standard defines three primitive operations that can be performed on a distributed sharing list. Memory operations such as read misses, write misses, write backs, and replacements are implemented using these three primitive operations:

1. *List construction*: adding a new node (sharer) to the head of a sharing list.
2. *Rollout*: removing a node from a list, which requires that a node communicate with its upstream and downstream neighbors, informing them of their new neighbors so they can update their pointers.
3. *Purging (invalidation)*: the node at the head may purge or invalidate all other nodes, thus resulting in a single-element list. Only the head node of a list can issue a purge.

The SCI standard also describes three levels of increasingly sophisticated SCI protocols. The *minimal* protocol does not permit even read sharing; that is, only one node at a time can have a cached copy of a block. The *typical* protocol is what most systems are expected to implement. It has provisions for read sharing (multiple copies), efficient access to data that is in FRESH state in memory, as well as options for efficient DMA transfers and robust recovery from errors. Finally, the *full* protocol implements all of the options defined by the standard, including optimizations for pairwise sharing between only two nodes and queue-on-lock-bit (QOLB) synchronization (to be discussed later). The NUMA-Q system implements the typical protocol, and this is the one we discuss. Let us see how different types of memory operations—read misses, write misses, and replacements (including write backs)—are handled. In each case, the identity of the home node is first determined from the address of the block.

Handling Read Requests

Suppose the read request needs to be propagated off quad. We can think of this node's remote cache as the requesting cache as far as the SCI protocol is concerned. The requesting cache first allocates an entry for the block if necessary and sets the cache state of the block to a pending (busy) state; in this state, it will not process other requests for that block that come to it. (The SCI protocol often puts cached blocks in busy states at requestors in this way, to keep transactions for a block atomic and to facilitate serialization, much like the Origin protocol did with its busy states at the directory. However, it does not use NACKs, as we shall see.) It then begins a list construction operation to add itself to the head of the sharing list by sending a request to the home node. When the home receives the request, its block may be in one of the three directory states identified earlier: HOME, FRESH, or GONE.

If the directory state is HOME, there are no cached copies and the copy in memory is valid. On receiving the read request, the home updates its state for the block to FRESH and sets its head pointer to point to the requesting node. The home then replies to the requestor with the data, which upon receipt updates its state from PENDING to ONLY_FRESH. All actions at a node in response to a given transaction are atomic (the processing for one is completed before the next one is handled), and a strict request-response protocol is followed in all cases (unlike in Origin).

If the directory state is FRESH, there is already a sharing list, but the copy at the home is also valid. The home changes its head pointer to point to the requesting cache instead of the previous head of the list. It then sends back a transaction to the requestor containing the data as well as a pointer to the previous head. On receipt, the requestor moves to a different pending state and sends a transaction to that previous head asking to be attached as the new head of the list (the list construction operation). The previous head reacts to this message by changing its state from HEAD_FRESH to MID_VALID or from ONLY_FRESH to TAIL_VALID as the case may be, updating its backward pointer to point to the requestor and sending an acknowledgment to the requestor. When the requestor receives this acknowledgment, it sets its forward pointer to point to the previous head and changes its state

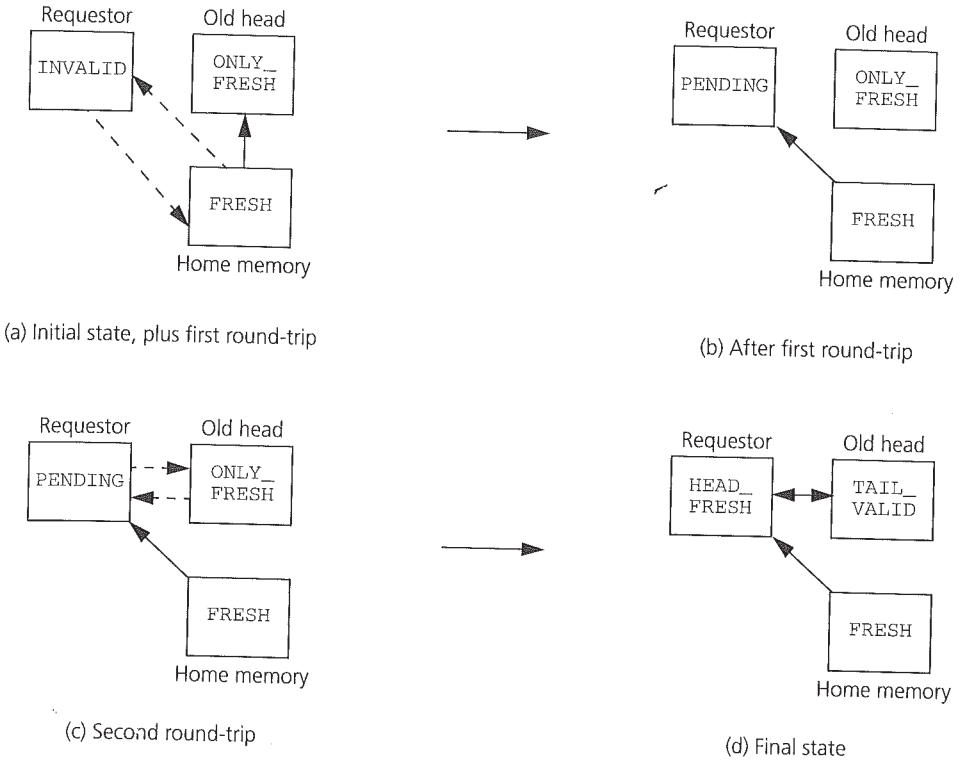


FIGURE 8.26 An example of a read miss in the SCI protocol. The figure shows the messages and state transitions for a read miss to a block that is initially in the FRESH state at home, with one node on the sharing list. Solid lines are the pointers in the sharing list, whereas dotted lines represent network transactions. Null pointers are not shown.

from the pending state to HEAD_FRESH. The sequence of transactions and actions is shown in Figure 8.26 for the case where the previous head is in state HEAD_FRESH when the request comes to it.

If the directory state is GONE, the cache at the head of the sharing list has an exclusive (clean or modified) copy of the block. Now, the memory does not reply with the data but simply stays in the GONE state and sends a pointer to the previous head back to the requestor. The requestor goes to a new pending state and sends a request to the previous head, asking both for the data and to attach to the head of the list (list construction). The previous head changes its state from HEAD_DIRTY to MID_VALID or from ONLY_DIRTY to TAIL_VALID (or whatever is appropriate), sets its backward pointer to point to the requestor, and returns the data to the requestor. (The data may have to be retrieved from a processor cache in the previous head node.) The requestor then updates its copy, sets its state to HEAD_DIRTY, and sets its forward pointer to point to the new head, all in a single atomic action as always. Note that even though the reference was a read, the head of the sharing list is

left in HEAD_DIRTY state. This does not have the standard meaning of dirty that we are familiar with; that is, that the head node can write that data without having to invalidate any other caches. It means that it can indeed write the data into the cache without communicating with the home (and even before sending out the invalidations), but it must invalidate the other nodes in the sharing list since they are in valid state.

It is possible to fetch a block in HEAD_DIRTY state even when the directory state is not GONE, for example, when the requesting node is expected to write that block soon afterward. In this case, if the directory state is FRESH the memory returns the data to the requestor, together with a pointer to the old head of the sharing list, and then puts itself in GONE state. The requestor then prepends itself to the sharing list by sending a request to the old head and puts itself in the HEAD_DIRTY state. The old head changes its state from HEAD_FRESH to MID_VALID or from ONLY_FRESH to TAIL_VALID as appropriate, and other nodes on the sharing list remain unchanged.

In the preceding cases, a requestor is always directed by the home to the old head. It is possible that the old head (let's call it *A*) is in a pending state when the request from the new requestor (*B*) reaches it since it may itself have a memory operation outstanding on that block. This is dealt with not by buffering the request at the old head or NACKing it but by extending the sharing list backward into a (still distributed) *pending list*. That is, node *B* will indeed be physically attached to the head of the list but in a pending state waiting to truly become the head. If another node *C* now makes a request to the home, it will be forwarded to node *B* and will also attach itself to the pending list (the home will now point to *C*, so subsequent requests will be directed there, and so on). At any time, we call the "true head" (here *A*) simply the *head* of the sharing list, we call the part of the list before the true head the *pending list*, and we call the latest element to have joined the pending list (here *C*) the *pending head* (see Figure 8.27). When *A* leaves the pending state and completes its operation, it will pass on the "true head" status to *B*, which will in turn pass it on to *C* when its request is completed. Note also that, unlike in the Origin, no pending or busy state exists at the directory, which always simply takes atomic actions to change its state and head pointer and returns the previous state/pointer information to the requestor, a point we will revisit when discussing how correctness issues are addressed.

Handling Write Requests

The head node of a sharing list is assumed to always have the latest copy of the block (unless the head node is in a pending state). Thus, only the head node is allowed to write a block and issue invalidations. When a node incurs a write miss, three cases are possible. In the first case, the writer is already at the head of the list, but it does not have the sole modified copy (e.g., there may be other sharers). It first ensures that it is in the appropriate state for this case, by communicating with the home if necessary (and in the process ensuring that the home block is already in or transitions to the GONE state). It then modifies the data locally and invalidates the rest of the nodes in the sharing list. (This case is elaborated on in the next two para-

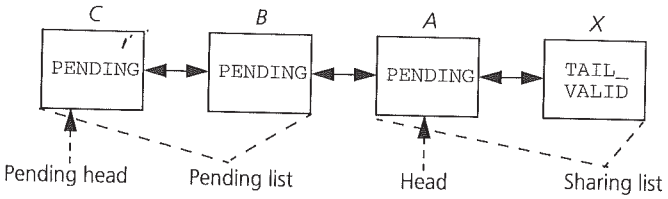


FIGURE 8.27 Pending lists in the SCI protocol. The pending list is a continuation (in the reverse direction) of the regular sharing list. The true head (called the head) and the nodes in the pending list are in pending states.

graphs.) In the second case, the writer is not in the sharing list at all. The writer must first allocate space for and obtain a copy of the block, then add itself to the head of the list using the list construction operation, and then perform the preceding steps to complete the write. The third case is when the writer is in the sharing list but not at the head. In this case, it must remove itself from the list (rollout), then add itself to the head (list construction), and finally perform the preceding steps. We discuss rollout further in the context of replacement, where it is also needed, and we have already seen list construction. Let us focus now on the case where the writing node is already at the head of the list.

If the block is in the HEAD_DIRTY state in the writer's cache, it is modified right away (since the directory must already be in GONE state) and then the writing node purges the rest of the sharing list. The purge operation is done in a serialized request-response manner: an invalidation request is sent to the next node in the sharing list, which rolls itself out from the list and sends back to the head a pointer to the next node in the list. The head then sends this node a similar request, and so on until all entries are purged (i.e., until the response to the head contains a null pointer; see also Figure 8.28). The writer, or head node, stays in a pending state while the purging is in progress. During this time, new attempts to add to the sharing list are delayed in a pending list as usual. The latency of purging a sharing list is a few serialized round-trips (invalidation request, acknowledgment, and the rollout transactions) plus the associated actions per sharing list entry, so it is important that long sharing lists are not encountered often on writes. It is possible to reduce the number of network transactions in the critical path by having each node pass on an invalidation request to the next node and perhaps acknowledge the previous node rather than return the identity to the writer. This is not part of the SCI standard since it distributes the state of the invalidation progress and hence complicates protocol-level recovery from errors; however, practical systems may be tempted to take advantage of this shortcut, especially if sharing lists are long.

If the writer is the head of the sharing list but has the block in HEAD_FRESH state, then it must be changed to HEAD_DIRTY before the block can be modified and the rest of the entries purged. The writer goes into a pending state and sends a request to the home, the home changes from FRESH to GONE state and replies to the message, and then the writer goes into a different pending state and purges the rest

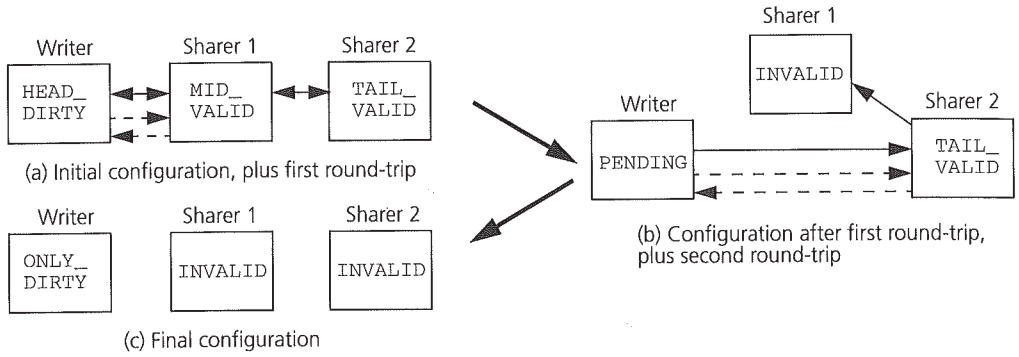


FIGURE 8.28 Purging a sharing list from a HEAD_DIRTY node in SCI. Solid arrows connecting list nodes are list pointers, while dashed arrows indicate network transactions that implement the transition to the next configuration.

of the blocks as was just described. It may be that when the request reaches the home the home is no longer in FRESH state, but it points to a newly queued node that got there in the meantime and has been directed to the writer. When the home looks up its state, it detects this situation and sends the writer a corresponding response that is like a NACK. When the writer receives this response, based on its local pending state it deletes itself from the sharing list (how it does this, given that a request is coming at it, is discussed in the next subsection) and tries to reattach as the head in HEAD_DIRTY or ONLY_DIRTY state by sending the appropriate new request to the home. This is not a retry, in the sense that the writer does not try the same request again, but is a suitably modified request to reflect the new state of itself and the home (similar to modifying an upgrade to a read exclusive in the race condition due to nonatomic state transitions discussed in Chapter 6). The last case for a write by a head node is if the writer has the block in ONLY_DIRTY state, in which case it can modify the block without generating any network transactions.

Handling Write-Back and Replacement Requests

A node that is in a sharing list for a block may need to delete itself, either because it must become the head in order to perform a write operation, or because it must be replaced in its remote cache for capacity or conflict reasons, or because it is being invalidated. In the case of a replacement, even if the block is in shared state and does not have to write data back, the space in the cache (and the pointers) will now be used for another block and its list pointers, so to preserve a correct representation the block being replaced must be removed from its sharing list. These replacements and list removals use the rollout operation.

Consider the general case of a node trying to roll out from the middle of a sharing list. The node first sets itself to a pending state, then sends a request each to its upstream and downstream neighbors asking them to update their forward and back-

ward pointers, respectively, to skip that node. The pending state is needed since there is nothing to prevent two adjacent nodes in a sharing list from trying to roll themselves out at the same time, which can lead to a race condition in the updating of pointers. Even with the pending state, if two adjacent nodes indeed try to roll out at the same time, they may set themselves to pending state simultaneously and send messages to each other. This can cause deadlock since neither will respond while it is in pending state. A simple priority system is used to avoid such deadlock: by convention, the node closer to the tail of the list has priority and is rolled out first. The roll-out operation is completed by setting the state of the rolled-out cache entry to invalid when both the neighbors have replied. The neighbors of the node that is rolling out do not have to change their state except when the node being rolled out is the second in a two-node list; in that case, the head of the list may change its state from HEAD_DIRTY or HEAD_FRESH to ONLY_DIRTY or ONLY_FRESH as appropriate.

If the entry to be rolled out is the head of the list, then the entry may be in dirty state (a write back) or in fresh state (a replacement). The same set of transactions is used in either case. The head puts itself in a pending state and first sends a transaction to its downstream neighbor. This causes the latter to set its backward pointer to the home memory and change its state appropriately (e.g., from TAIL_VALID or MID_VALID to HEAD_DIRTY or from MID_FRESH to HEAD_FRESH). When the replacing (head) node receives a response, it sends a transaction to the home, which updates its pointer to point to the new head but need not change its state. The home sends a response to the replacer, which is now out of the list and sets its state to INVALID. Of course, if the replacer is the only node in the list, then it needs to communicate only with memory, which will set its state to HOME.

This scenario of a head node rolling out provides another example of the state at the recipient of a request not being compatible with that request when it arrives. By the time the message from the replacer gets to the home, the home may have set its head pointer to point to a different node X from which it has received a request for the block in the interim. In general, whenever a transaction comes in, the recipient looks up its local state and the incoming request type; if it detects a mismatch, the general strategy adopted by the protocol is as we saw earlier in the example of a write to a block in HEAD_FRESH state: the recipient does not perform the operation that the request solicits but issues a response that is a lot like a NACK. The requestor will then check its local state again and take an appropriate action. In this specific case, the home detects that the incoming transaction type requires that the requestor be the current head; this is not true, so it NACKs the request. The replacer keeps retrying the request to the home and keeps being NACKed. At some point, the request from node X that was redirected to the replacer will reach the replacer, asking to be prepended to the list. The replacer will look up its (pending) state and send a response to that requestor, telling it to instead go to the downstream neighbor (the real head since the replacer is rolling out of the list). The replacer is now off the list and in a different pending state; it is waiting to go to INVALID state, which it will do when the next NACK from the home reaches it. Thus, the SCI protocol does include NACKs, but not in the traditional sense of asking requests to retry when a node or

resource is busy. NACKs are used just to indicate inappropriate requests and facilitate changes of state at the requestor; the difference is that in this case a request that is NACKed will never succeed in its original form but may cause a new type of request to be generated, which may succeed.

Finally, when a block needs to be written back upon a miss, an important performance question is whether the miss should be satisfied first or the block should be written back first. In discussing bus-based protocols, we saw that most often the miss is serviced first and the block to be written back is put in a write-back buffer. In NUMA-Q, the simplifying decision is made to service the write back (rollout) first and only then satisfy the miss. Although this slows down the miss, the complexity of the buffering solution is greater here than in bus-based systems (where the write-back buffer can simply be snooped). Also, the replacements and hence write backs we are concerned with here are from the remote cache, which is large enough (tens of megabytes) that replacements are likely to be very infrequent.

8.6.2 Dealing with Correctness Issues

A major emphasis in the SCI standard is providing well-defined, uniform mechanisms for preserving serialization, resolving race conditions, and avoiding deadlock, livelock, and starvation. The standard takes a stronger position on starvation and fairness than many other coherence protocols. It was mentioned earlier that most of the correctness considerations are satisfied by the use of distributed lists of sharers as well as pending requests, but let us look at how this works in more detail.

Serialization of Operations to a Given Location

In the SCI protocol, the home node is the entity that determines the order in which cache misses to a block are serialized. However, unlike in the Origin protocol, here the order is that in which the requests first arrive at the home, and the mechanism used for ensuring this order is very different. There is no busy state at the home. Generally (except for some race conditions described earlier), the home accepts every request that comes to it, either satisfying it wholly by itself or directing it to the node that it sees as the current head of the sharing list (the pending head if there is a pending list). Before it directs the request to another node, it first updates its head pointer to point to the current requestor. The next request for the block from any node will see the updated state and pointer (i.e., to the current requestor) even though the operation corresponding to the current request is not globally complete. This ensures that the home does not direct two conflicting requests for a block to the same node at the same time, avoiding race conditions. As we have seen, if a request cannot be satisfied at the head node to which it was directed—that is, if that node is in pending state—the requestor will attach itself to the distributed pending list for that block and await its turn as long as necessary (see Figure 8.27). Nodes in the pending list obtain access to the block in FIFO order, ensuring that the order in which they complete is indeed the same as that in which they first reached the home.

While the home may NACK requests when some race conditions are encountered, those requests will never succeed in their current form, so they do not count in the serialization. They may be modified to new, different requests that will succeed, and in that case those new requests will be serialized in the order in which they first reach the home.

Memory Consistency Model

The SCI standard defines both a coherence protocol and a transport layer, including a network interface design. However, it does not specify many other aspects, like details of the physical implementation or even the memory consistency model. Such matters are left to the system implementor. NUMA-Q does not satisfy sequential consistency but uses a more relaxed memory consistency model called *processor consistency* that we shall discuss in Section 9.1. Interestingly, as in Origin, the consistency model chosen for the system is the one supported by the underlying microprocessor.

Deadlock, Livelock, and Starvation

The fact that a distributed pending list is used to hold waiting requests at the requestors themselves, rather than a hardware queue shared at the home node by all blocks allocated in it, implies that there is no danger of input buffers filling up and, hence, no deadlock problem at the protocol level. A strict request-response protocol is used as well. Since requests are not NACKed from the home to alleviate blockages or contention (only under certain race conditions when they must be altered) but will simply join the pending list and always make progress, livelock does not occur. The list mechanism also ensures that the requests are handled in FIFO order as they first come to the home, thus preventing starvation.

The total number of pending lists that a node can be a part of is the number of requests it can have outstanding, and the storage for the pending lists is already available in the cache entries, so there is little need for extra buffering at the protocol level. (Replacement of a pending entry is not allowed; the memory operation that causes the replacement stalls until the entry is no longer pending.) While the SCI standard does not take a position on queuing and buffering issues at the lower transport level, most implementations, including NUMA-Q, use separate request and response queues on each of the incoming and outgoing paths.

Error Handling

The SCI standard provides some options in the typical protocol to recover from errors at the hardware link level. NUMA-Q does not implement these but, rather, assumes that the hardware links are reliable. Standard ECC and CRC checks are provided to detect and recover from hardware errors in the memory and network links. Robustness to errors at the protocol level often comes at the cost of performance.

For example, SCI's decision to have the writer send all the invalidations one by one, serialized by responses, simplifies error recovery since the writer knows how many invalidations have been completed when an error occurs; however, it but compromises performance. While NUMA-Q retains this feature, other systems may choose not to.

8.6.3 Protocol Extensions

While the SCI protocol is fair and quite robust to errors, many types of operations can generate several serialized network transactions and therefore become quite expensive. A read miss requires two network transactions with the home, at least two with the head node if there is one, and perhaps more with the head node if it is in pending state. A replacement requires a rollout, which requires communication with both neighbors. But, potentially, the most troublesome operation from a scalability viewpoint is invalidation on a write since the cost of the invalidation scales linearly with the number of nodes on the sharing list with a fairly large constant (more than a round-trip time). The use of distributed pending lists can increase latency too, and, in general, the latency of misses tends to be larger than in memory-based protocols. Extensions have been proposed to SCI to deal with widely shared data through a combination of hardware organization and protocol. For example, instead of a single large ring interconnect, the SCI standard envisions building large systems by connecting many smaller rings together in a hierarchy using bridges and switches; the protocol can exploit combining transactions in this hierarchy. Some extensions require changes to the basic protocol and hardware structures whereas others are compatible with the basic SCI protocol and only require new implementations of the bridges. The complexity of the extensions may reduce performance for low degrees of sharing. They are not finalized in the standard and are beyond the scope of this discussion. More information can be found in (IEEE Computer Society 1995; Kaxiras and Goodman 1996; Kaxiras 1996). One extension that is included in the standard specializes the protocol for the case in which only two nodes share a cache block and they ping-pong ownership of it back and forth between themselves by both writing it repeatedly. This is described in the SCI protocol document (IEEE Computer Society 1993). NUMA-Q includes another protocol extension that is a special protocol operation that enables a processor to obtain a copy of a block even while it is invalidating the (nonhome) source of the block.

Unlike Origin, NUMA-Q does not provide hardware or OS support for dynamic page migration. With the very large remote caches, capacity misses in the processor caches to remotely allocated data are almost always satisfied in the remote cache in the local node. However, proper page placement can still be useful when a processor writes and has to obtain ownership for data. If nobody else has a copy (e.g., in the interior portion of a processor's partition in the equation solver kernel or in Ocean), then if the home is local, obtaining ownership does not generate network traffic; however, if home is remote, a round-trip to the home is needed to look up directory state. The NUMA-Q position is that data migration in main memory is the responsibility of user-level software. The exception is when a process migrates, in which case

the OS uses a heuristic to possibly migrate that process's active pages as well, making them local at the new location. The designers considered this to be the important context for page migration. Similarly, little hardware support is provided for synchronization beyond simple atomic exchange primitives like test&set.

8.6.4 Overview of NUMA-Q Hardware

Within a quad multiprocessor node, the second-level caches per processor currently shipped in NUMA-Q systems are 512 KB or 1 MB large and four-way set associative with a 32-byte block size. The quad bus is a 532-MB/s split-transaction in-order bus, with limited facilities for out-of-order responses that are needed by a two-level coherence scheme. (Even if the bus within an SMP node provides in-order responses, when a request must go to a remote node it is infeasible to have its response be in-order with respect to responses generated within the local node.) A quad also contains up to 4 GB of globally addressable main memory; two 32-bit-wide 133-MB/s peripheral component interface (PCI) buses connected to the quad bus by PCI bridges and to which I/O devices and a memory and diagnostic controller can attach; and the IQ-Link board that plugs into the memory bus and includes the communication assist and the network interface.

In addition to the directory information for locally allocated data and the tags for remotely allocated but locally cached data (which it keeps on both the bus side and the directory side), the IQ-Link board consists of four major functional blocks as shown in Figure 8.29: the bus interface controller, the DataPump, the SCI link interface controller, and the RAM arrays. The *Orion bus interface controller* (OBIC) provides the interface to the shared quad bus, managing the remote cache data arrays and the bus snooping and requesting logic. It acts as both a pseudo memory controller that snoops and translates accesses to nonlocal data as well as a pseudo-processor that puts incoming transactions from the network onto the bus. The *DataPump*, a gallium arsenide chip built by Vitesse Semiconductor Corporation, provides the link and packet-level transport protocol of the SCI standard. It provides an interface to a ring interconnect, pulling off packets that are destined for its quad node and letting other packets go by. The *SCI link interface controller* (SCLIC) interfaces to the DataPump and the OBIC as well as to the interrupt controller and the directory tags. Its main function is to manage the SCI coherence protocol, using one or more programmable protocol engines. The *RAM arrays* implement the data and the different tags needed for the remote cache. These components are described further when we discuss the implementation of the IQ-Link in Section 8.6.6.

For the interconnection across quads, the SCI standard defines both a transport layer and a cache coherence protocol. The transport layer defines a functional specification for a node-to-network interface and a network topology that consists of rings made of point-to-point links. In particular, it defines a 1-GB/s ring interconnect and the transactions that can be generated on it. The NUMA-Q system is initially a single-ring topology of up to eight quads as shown in Figure 8.24. Cables from the quads connect to the ports of a ring that is contained in a single box called the IQ-Plus. Larger systems will include multiple eight-quad systems connected

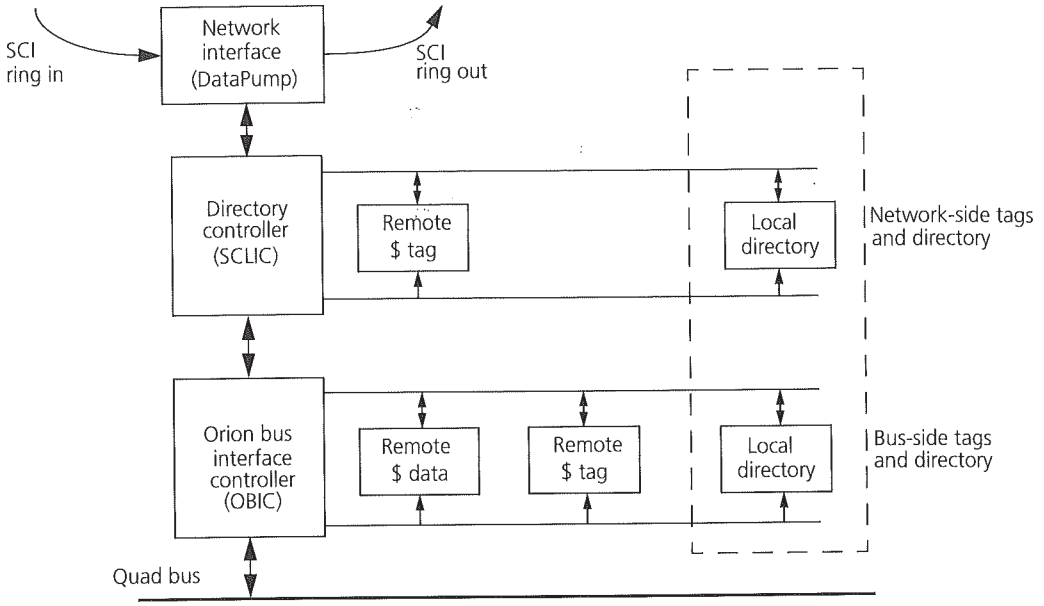


FIGURE 8.29 Functional block diagram of the NUMA-Q IQ-Link board. The remote cache data is implemented in synchronous DRAM (SDRAM). The bus-side tags and directory are implemented in Static RAM (SRAM) whereas the network-side tags and directory can afford to be slower and are therefore implemented in SDRAM.

with local area networks. As mentioned earlier, the SCI standard envisions that, because of the high latency of long rings, larger systems will generally be built out of multiple rings interconnected by switches. With a small number of outstanding requests per node, the latency of a long ring severely limits the node-to-network bandwidth that a node can achieve (see Chapter 11). The transport layer of SCI will be discussed further in Chapter 10.

Since the machine is targeted toward database and transaction processing workloads, I/O is an important focus of the NUMA-Q design. As in Origin, I/O is globally addressable, so any processor can directly write to or read from any I/O device, not just those attached to the local quad. A nonlocal processor does not have to send an explicit message to the quad to which the device is attached and have a processor on that quad issue the access. This is very convenient for commercial applications, which are not often structured so that a processor need only access its local disks. I/O devices are connected to the two PCI buses that attach through PCI bridges to the quad bus. Each PCI bus is clocked at half the speed of the memory bus and is half as wide, yielding roughly one-quarter the bandwidth. Physically, there are two ways for a processor to access I/O devices on other quads. One is through the SCI rings, whether through the cache coherence protocol or through uncached writes, just as Origin does through its Hubs and network. However, bandwidth is a precious resource on a

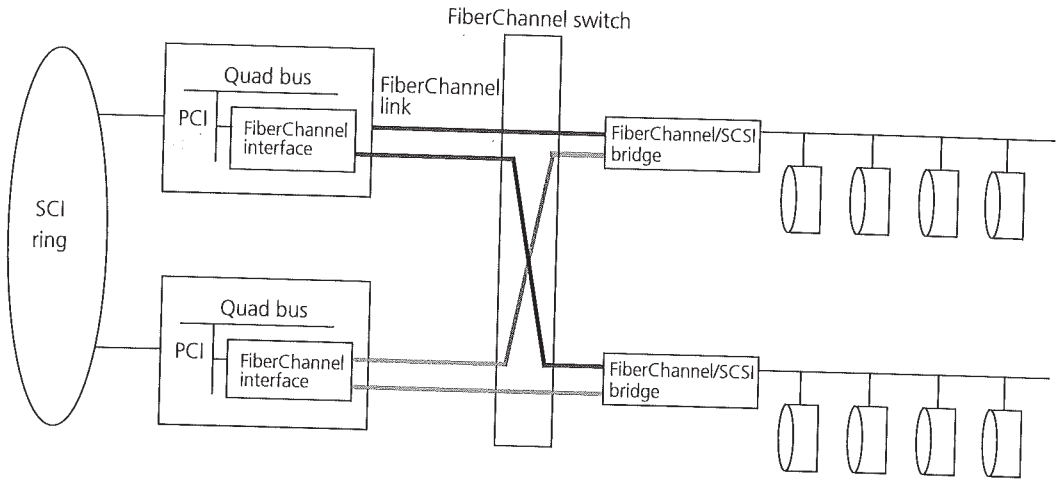


FIGURE 8.30 I/O subsystem of the Sequent NUMA-Q. I/O is globally addressable, and I/O data transfers among nodes can travel through FiberChannel via PCI buses or through the SCI ring used for memory operations.

ring network. I/O transfers can occupy substantial bandwidth, interfering with memory accesses. NUMA-Q therefore provides a separate communication substrate through the PCI buses for interquad I/O transfers, which is the default I/O path. A "FiberChannel" link connects to a PCI bus on each node. These links are connected to all the shared disks in the system through either point-to-point connections, an arbitrated FiberChannel loop, or a FiberChannel switch, depending on the scale of the processing and I/O systems (Figure 8.30).

FiberChannel talks to the disks at over 50 MB/s sustained through a bridge that converts the FiberChannel data format to the SCSI format that the disks accept. I/O to any disk in the system usually takes a path through the local PCI bus and the FiberChannel switch; however, if this path fails for some reason, the operating system causes I/O transfers to go through the SCI ring to another quad and through its PCI bus and FiberChannel link to the disk. FiberChannel may also be used to connect multiple NUMA-Q systems in a loosely coupled fashion and to have multiple systems share disks. Finally, a management and diagnostic controller connects to a PCI bus on each quad; these controllers are linked with one another and to a system console through a private local area network like Ethernet for system maintenance and diagnosis.

8.6.5 Protocol Interactions with SMP Node

The earlier discussion of the SCI protocol ignored the multiprocessor nature of the quad node and the bus-based protocol within it. Now that we understand the hardware structure of the node and the IQ-Link, let us examine the interactions of the two protocols, the requirements that the interacting protocols place upon the quad

and IQ-Link, and some particular problems raised by the use of an off-the-shelf SMP as a node.

A read request illustrates some of the interactions. A read miss in a processor's second-level cache first appears on the quad bus. In addition to being snooped by the other processor caches, it is snooped by the OBIC bus controller on the IQ-Link board. The OBIC looks up the remote cache as well as the directory state bits for locally allocated blocks to see if the read can be satisfied within the quad or if it must be propagated off node. In the former case, main memory or one of the other caches satisfies the read, and the appropriate MESI state changes occur. (Snoop results are reported, in order, after a fixed number of bus cycles [four]; if a controller cannot finish its snoop within this time, it asserts a stall signal for another two bus cycles, after which memory checks for the snoop result again. This continues until all snoop results are available.) The quad bus implements in-order data responses to requests. However, if the OBIC detects that the request must be propagated off node, then it must intervene. It does this by asserting a *deferred response* signal, telling the bus to violate its in-order response property and proceed with other transactions and that the OBIC will take responsibility for responding to this request. This would not have been necessary if the quad bus implemented out-of-order responses. The OBIC then passes on the request to the SCLIC to engage the directory protocol. When the response comes back, it is passed from the SCLIC back to the OBIC, which places it on the bus and completes the deferred transaction. Note that when extending any bus-based system to be the node of a larger cache-coherent machine, it is essential that the bus be split transaction, not only for performance but also to simplify correctness. Otherwise, the bus will be held up for the entire duration of a remote transaction, not allowing even local misses to complete and not allowing incoming network transactions to be serviced by processor caches (potentially causing deadlock).

Writes take a similar path out of and back into a quad. The state of the block in the remote cache, snooped by the OBIC, indicates whether the block is owned by the local quad or a request must be propagated to the home through the SCLIC. Putting the node at the head of the sharing list and invalidating other nodes, if necessary, is taken care of by the SCLIC. When the SCLIC is done, it places a response on the quad bus (via the OBIC), which completes the operation. An interesting situation arises due to a limitation of the quad itself. Consider a read miss or write miss to a locally allocated block that is cached remotely in a modified state. When the response returns and is placed on the bus as a deferred response, it should update the main memory. However, the quad memory was not implemented to deal with deferred requests and responses and does not update itself on seeing a deferred response. Thus, when a deferred response is passed down to the bus through the OBIC, the OBIC must also ensure that it updates the memory through a special action before it gives up the bus. Another limitation arises from how the OBIC uses the quad bus protocol. If two processors in a quad issue read-exclusive requests back to back, and the first one propagates to the SCLIC, we would like the second one to be buffered and accept the response from the first in the appropriate state. However,

the implementation NACKs the second request, which will then have to retry until the first one returns and it succeeds.

Finally, consider serialization. Since serialization at the SCI protocol level is done at the home, incoming transactions at the home have to be serialized not only with respect to one another but also with respect to accesses by the processors in the home quad. For example, suppose a block is in the HOME state at the home. At the SCI protocol level, this means that no remote cache in the system (which must be on some other node) has a valid copy of the block. However, unlike the unowned state in the Origin protocol, this does not mean that no processor cache in the home node has a copy of the block. In fact, the directory will be in HOME state even if one of the processor caches at the home has a dirty copy of the block. Even to obtain the right value, a request coming in for a locally allocated block at a home node must therefore be broadcast on the quad bus as well and cannot be handled entirely by the SCLIC and OBIC. Similarly, an incoming request that makes the directory state change from HOME or FRESH to GONE must be put on the quad bus so that the copies in the processor caches can be invalidated. Since both incoming requests and local misses to data at the home appear on the quad bus, it is natural to let this bus be the actual serializing agent at the home.

Similarly, serialization issues need to be addressed in a requesting quad for accesses to remotely allocated blocks. Activities within a quad relating to remotely allocated blocks are serialized at the local SCLIC rather than the local bus. Thus, requests from local processors for a block in the remote cache and incoming requests from the SCI interconnect for the same block are serialized at the local SCLIC. Similarly, the SCLIC takes care of the local serialization between outstanding invalidations at a requestor and incoming requests. Other interactions with the node protocol are discussed once we have considered the implementation of the IQ-Link board components.

8.6.6 IQ-Link Implementation

Unlike the single-chip Hub in Origin, the SCLIC directory controller, the OBIC bus interface controller, and the DataPump are separate chips on the IQ-Link board, which also contains some SRAM and SDRAM chips for tags, state, and remote cache data (see Figure 8.29).

The data in the remote cache is directly accessible by the OBIC. Two sets of tags are used to reduce communication between the SCLIC and the OBIC: the network-side tags for access by the SCLIC and the bus-side tags for access by the OBIC. The same is true for the directory state for locally allocated blocks. The bus-side tags and directory state contain only the information that is needed for the bus snooping and are implemented in SRAM so they can be looked up at bus speed. The network-side tags and state need more information and can be slower, so they are implemented in synchronous DRAM (SDRAM). The bus-side local directory SRAM contains only the 2 bits of directory state per 64-byte block (to distinguish the HOME, FRESH, and GONE states) whereas the network-side directory contains the 6-bit SCI head pointer

as well. The bus-side remote cache tags also have only 4 bits of state and do not contain the SCI forward and backward list pointers. They keep track of 14 states, some of which are transient states that ensure forward progress within the quad (e.g., that keep track of blocks that are being rolled out or of the particular bus agent that has an outstanding retry on the bus and so must get priority for that block). The network-side remote cache tags, which are part of the directory protocol, contain 7 bits to represent all protocol states plus two 6-bit pointers per block (as well as the 13-bit cache tags themselves).

Unlike the hardwired protocol tables in Origin, the SCLIC coherence controller in NUMA-Q is programmable. This means the protocol can be written in software or firmware rather than hardwired into a finite state machine. Every protocol-invoking operation from a local processor, as well as every incoming transaction from the network, invokes a software "handler" or task that runs on the protocol engine. These software handlers, written in microcode, may manipulate directory state, put interventions on the quad bus, generate network transactions, and so on. The SCLIC engine has multiple register sets to support 12 read/write/invalidate transactions and 1 interrupt transaction concurrently. To allow the standard intraquad interrupt interface to be used across quads, the SCLIC provides a bridge for routing standard intraquad interrupts between quads and provides some extra bits to include the destination quad number when generating such interrupts.

A programmable protocol engine has several potential advantages. It allows the protocol to be debugged in software and corrected by simply downloading new protocol code. It provides the flexibility to experiment with or change protocols even after the machine is built and bottlenecks are discovered, and allows multiple protocols to be supported by the machine. And it enables code to be inserted into the handlers to monitor chosen events for performance debugging, which is especially valuable given the implicit nature of communication and the potential impact of artifactual communication in a shared address space. The disadvantage is that a programmable protocol engine has higher occupancy per transaction than a hardwired one, so a performance cost is associated with this decision. Attempts are made to reduce this performance impact in the NUMA-Q SCLIC. The protocol processor has a three-stage pipeline and issues up to two instructions (a branch and another instruction) every cycle. It uses a cache to hold recently used directory state and tag information rather than accessing the directory RAMs every time. Finally, it is specialized to support the kinds of bit-field manipulation operations that are commonly needed in directory protocols as well as useful instructions that speed up handler dispatch and management, like "queue on buffer full" and "branch on queue space available" instructions. A somewhat different programmable protocol engine is used in the Stanford FLASH multiprocessor (Kuskin et al. 1994), the successor to the hardwired Stanford DASH machine.

Each Pentium Pro processor can have up to four requests outstanding. The quad bus can have eight requests outstanding at a time and ensures that snoop and data responses come in order (except when deferred responses are used, as discussed earlier). The OBIC can have four external requests outstanding to the SCLIC and can buffer two incoming transactions to the quad bus at a time. If a fifth request from the

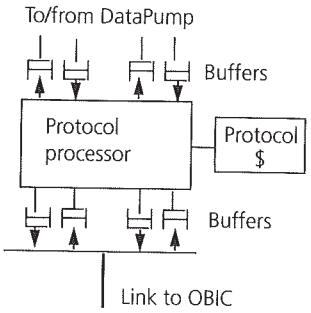


FIGURE 8.31 Simplified block diagram of the SCLIC chip. It contains a programmable protocol processor, a cache for directory information, and buffers to interface with the OBIC (bus) and DataPump (network).

quad bus needs to go off quad, the OBIC will NACK it until a buffer entry is free but will not cause the quad bus to stall for local operations. The SCLIC can have up to eight requests outstanding and can buffer four incoming requests at a time. A simplified illustration of the SCLIC is shown in Figure 8.31. Finally, the DataPump request and response buffers are each two entries deep outgoing to the network and four entries deep incoming. All request and response buffers, whether incoming or outgoing, are physically separate in this implementation.

In addition to the ability to instrument protocol handlers in software, all three components of the IQ-Link board also provide performance counters to enable nonintrusive measurement of various events and statistics. There are three 40-bit memory-mapped counters in the SCLIC and four in the OBIC. Each can be set in software to count any of a large number of events, such as protocol engine utilization, memory and bus utilization, queue occupancies, the occurrence of SCI command types, and the occurrence of transaction types on the quad bus. The counters can be read by software on the main processors at any time or can be programmed to generate interrupts when they cross a predefined threshold value. The Pentium Pro processor module itself provides a number of performance counters to count first- and second-level cache misses as well as the frequencies of request types and the occupancies of internal resources, among other properties. Together with the programmable handlers, these counters can provide a wealth of information about the behavior of the machine when running workloads.

8.6.7 Performance Characteristics

The quad bus has a peak bandwidth of 532 MB/s, and the SCI ring interconnect can transfer 500 MB/s in each direction across the node-to-network interface. The IQ-Link board can transfer data between these two interconnects at about 30 MB/s in each direction (note that only a small fraction of the transactions appearing on the quad bus or on the SCI ring are expected to be relevant to the other interconnect). The latency for a local read miss satisfied in main memory (or the remote cache) is expected to average about 250 ns under ideal conditions. The latency for a read satisfied in remote memory in a two-quad system is expected to be about 2.5 μs, a ratio

Table 8.3 Characteristics of Microbenchmarks and Workloads Running on an Eight-Quad NUMA-Q

Workload	Latency of L_2 Misses		SCLIC Utilization	Percentage of L_2 Misses Satisfied in			
	All	Remotely Satisfied		Local Memory	Other Local Cache	Local "Remote Cache"	Remote Node
Remote Read Misses	8,020 ns	8,300 ns	95%	1.5%	0%	2%	96.5%
Remote Write Misses	9,350 ns	9,625 ns	95%	1%	0%	2%	97%
TPC-B–like	630 ns	4,300 ns	54%	80%	2%	11.5%	6.5%
TPC-D (Q9)	580 ns	3,950 ns	40%	85%	5.5%	4%	5.5%

of about 10 to 1. However, the inclusion of a remote access cache keeps the frequency of artifactual communication very low. The latency through the DataPump network interface for the first 18 bits of a transaction is 16 ns and then 2 ns for every 18 bits thereafter. In the network itself, it takes about 26 ns for the first bit to get from the DataPump output of a quad into the IQ-Plus box that implements the ring and back out to the DataPump of the next quad along the ring.

The designers of the NUMA-Q have performed several experiments on the machine with microbenchmarks and with database and transaction processing workloads. To obtain a flavor for the microbenchmark performance capabilities of the machine, how latencies vary under load, and the characteristics of such workloads, let us take a brief look at the results. For a single-quad system with all four processors simultaneously generating cache misses as quickly as they can, back-to-back read misses are found to take 600 ns each and obtain a combined transfer bandwidth to the processors of 290 MB/s. Under similar conditions, back-to-back write misses, which cause a read followed by a write back, take 585 ns, and sustain 195 MB/s. For a single-quad system with multiple I/O controllers on each PCI I/O bus generating inbound writes from the I/O devices to the local memory as quickly as possible, each cache block transfer takes 360 ns at 111 MB/s sustained bandwidth.

Table 8.3 shows the latencies and characteristics under load as seen in various workloads running on multiple-quad systems. The first two rows are for microbenchmarks designed to have all quads simultaneously issuing read misses that are satisfied in remote memory. The third row is for the Transaction Processing Council's on-line transaction processing benchmark TPC-B (see Appendix). The last row is for Query 9 of the TPC-D benchmark suite, which represents decision support applications. The latencies are measured using the performance counters embedded in the OBIC and SCLIC and are measured not from the processor but from the bus request to the first data response. All workloads are run with four quads (16 processors), except the decision support workload, which is run with eight. Write misses to locally allocated

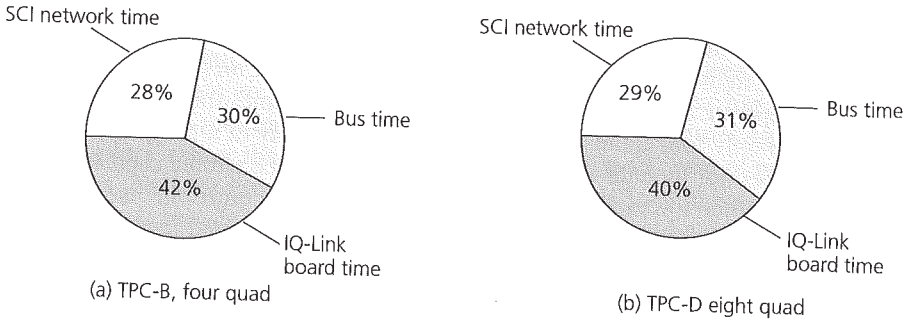


FIGURE 8.32 Components of average remote miss latency in two workloads on an eight-quad NUMA-Q. In both cases, most of the time is spent in the IQ-Link board, which includes data transfers between the SCLIC and the DataPump or the OBIC. Time in the OBIC chip itself is included in bus time in this figure.

data that cause invalidations to be sent remotely are very few and are included in the last column.

Remote data access latencies are clearly significantly higher than the unloaded latencies. In general, the SCI ring and protocol have higher latencies than those of more distributed networks and memory-based protocols, as discussed earlier. However, at least in these transaction processing and decision support workloads, much of the time in a remote access is spent passing through the IQ-Link board itself and not in the bus or ring. Figure 8.32 shows the breakdowns of average remote latency into three components for two workloads on four- and eight-quad systems. The path to improved remote access performance, both under load and not under load, is to make the IQ-Link board more efficient. The designers are considering a number of opportunities, including redesigning the SCLIC, perhaps using two instruction sequencers instead of one in the programmable SCLIC, and optimizing the OBIC, with the hope of reducing the remote access latency to about 2 μs under heavy load in the next generation. The remote cache is found to be very useful in keeping capacity misses local. The TPC-D (Q9) workload has lower SCLIC utilization than the TPC-B workload because it generates fewer invalidations.

8.6.8 Comparison Case Study: The HAL S1 Multiprocessor

The S1 multiprocessor from HAL Computer Systems is an interesting combination of some features of the NUMA-Q and the Origin2000. Like the NUMA-Q, the S1 also uses Pentium Pro quads as the processing nodes; however, it uses a memory-based directory protocol like that of the Origin2000 across quads rather than the cache-based SCI protocol. In addition, to reduce latency and assist occupancy, it integrates the coherence machinery more tightly with the node than the NUMA-Q does, coming closer to the Origin in this regard. Instead of using separate chips for

the directory protocol controller (SCLIC), bus interface controller (OBIC), and network interface (DataPump), the S1 integrates the entire communication assist and the network interface into a single chip called the mesh coherence unit (MCU), with separate chips used for storage. On the other hand, the cache-coherent design scales to only four quads, does not have the flexibility of a programmable controller, and does not include a remote access cache to reduce remote capacity misses.

Since the memory-based protocol does not require the use of forward and backward pointers with each cache entry, there is no need for a quad-level remote data cache to provide this functionality (which processor caches do not provide); in memory-based protocols, remote caches are useful only to reduce capacity misses, and the S1 does not use them. The directory information is maintained in separate SRAM chips, but the directory storage needed is greatly reduced by maintaining directory information not for all memory blocks but only for those blocks that are in fact cached remotely, organizing the directory itself as a cache (as discussed in Section 8.10.1). The MCU also contains a DMA engine to support explicit message passing as well as block data transfers in a cache-coherent shared address space (see Chapter 11). Message passing or explicit data transfers can be implemented either through the DMA engine (preferred for large messages) or through the transfer mechanism used for cache blocks (preferred for small messages). The MCU is hardwired instead of programmable, which reduces its occupancy for protocol processing and hence improves its performance under contention. The MCU also has substantial hardware support for performance monitoring. Other than the MCU, the only custom chip used is the network router, which is a six-ported crossbar with 1.9 million transistors, optimized for speed. The network is clocked at 200 MHz. The latency through a single router is 42 ns, and the usable per-link bandwidth is 1.6 GB/s in each direction—both similar to that of the Origin2000 network. The initial S1 interconnect implementation scales to 32 nodes (128 processors).

A major goal of integrating all the assist functionality into a single chip in S1 was to reduce remote access latency and increase remote bandwidth. From the designers' simulated measurements, the best-case unloaded latency for a read miss that is satisfied in local memory is 240 ns, for a read miss to a block that is clean at a nearby remote home is 1,065 ns, and for a read miss to a block that is dirty in a (nearby) third node is 1,365 ns. The remote-to-local latency ratio ranges from 4 to 5 (including contention), which is a little worse than on the SGI Origin2000 but better than on the NUMA-Q. However, microbenchmark comparisons of latencies are not very meaningful as predictors of overall performance on workloads since they ignore important considerations like remote caches and flexibility that can greatly affect the frequency of communication.

The bandwidths achieved by the HAL S1 in copying a single 4-KB page are instructive. The achieved bandwidth is 105 MB/s from local memory to local memory through processor reads and writes (limited primarily by the quad memory controller that has to handle both the reads and writes of memory), about 70 MB/s between local memory and a remote memory (in either direction) when accomplished through processor reads and writes, and about 270 MB/s in either direction between local and remote memory when performed through the DMA engines in the

MCUs. The case of remote transfers through processor reads and writes is limited primarily by the limit on the number of outstanding memory operations from a processor, which is not an issue for the DMA case. The DMA case has the additional advantage that it requires only one bus transaction at the initiating end for each memory block rather than two split-transaction pairs in the case of processor reads and writes (once for the read and once for the write). At least in the absence of contention across transfers, the local quad bus becomes a bandwidth bottleneck long before the interconnection network does.

Now that we understand the protocol layer that implements the coherent shared address space programming model in some depth for both memory-based and cache-based protocols, let us briefly examine some key interactions of protocols with the basic performance parameters of the communication architecture in determining the performance of applications.

8.7 PERFORMANCE PARAMETERS AND PROTOCOL PERFORMANCE

Recall that there are four major performance parameters in a communication architecture: overhead on the main processor, occupancy of the communication assist, network transit delay, and network bandwidth. Processor overhead is usually quite small on cache-coherent machines (unlike on message-passing systems, where it often dominates) and is determined entirely by the underlying node. In the best case, the portion that we can call processor overhead, and which cannot be hidden from the processor through overlap, is the cost of issuing the memory operation. In the worst case, it is the cost of traversing the processor's cache hierarchy and reaching the assist (which can be quite significant). All other protocol processing actions are off-loaded to the communication assist (e.g., the Hub or the IQ-Link). Network link bandwidth, too, is usually adequate for most applications in high-performance multiprocessor networks (Holt et al. 1995). The more critical issues under the control of the communication architecture are, therefore, network delay and assist occupancy.

As we have seen, the communication assist has many roles in protocol processing, including generating a request, looking up the directory state, accessing the data for a response, and sending out and receiving invalidations and acknowledgments. The occupancy of the assist for processing a transaction not only contributes to the uncontended latency of that transaction but can also cause contention at the assist and hence increase the cost of other transactions. This is especially true in cache-coherent machines because of the large number of small transactions—both data-carrying transactions and others like requests, invalidations, and acknowledgments—which implies that the occupancy is incurred very frequently and not amortized very well. The situation is better than in shared address space machines that are not cache coherent, where a transaction transfers only the referenced word rather than a whole cache block because replication and coherence must be managed by the programmer (see the discussion in Section 3.6), but the amortization is still small. In fact, assist occupancy very often dominates the data transfer bandwidth of the node-to-network interface as the key bottleneck to throughput at the

endpoints (Holt et al. 1995). It is therefore very important to keep assist occupancy small. At the protocol level, it is important both to ensure that the assist is not tied up by an outstanding transaction while other unrelated transactions are available for it to process and to reduce the amount of processing needed from the assist per transaction. For example, if the home forwards a request to a dirty node, the home assist should not be held up until the dirty node returns a response—which would dramatically increase its effective occupancy—but should go on to service the next transaction and deal with the response later when it comes. At the hardware design level, it is important to specialize the assist enough and integrate it tightly with the node's memory system so that its effective occupancy per transaction is low. The tighter the integration and the greater the specialization, the less commodity oriented the design but the lower the occupancy.

Impact of Network Delay and Assist Occupancy

Figure 8.33 shows the impact of assist occupancy and network latency on performance, assuming an efficient memory-based directory protocol similar to that of the SGI Origin2000. In the absence of contention, assist occupancy behaves just like network transit delay or any other component of the latency in a transaction's path: increasing occupancy by d cycles would have the same impact as keeping occupancy constant but increasing network delay by d cycles. Since the x-axis is total uncontended round-trip latency for a remote read miss (including the cost of network delay and assist occupancies incurred along the way), if no contention is induced by increasing occupancy, then all the curves for different values of occupancy will be identical. In fact, they are not, and the separation of the curves indicates the impact of the contention induced by increasing assist occupancy.

The smallest value of occupancy (o) in the graphs is intended to represent that of an aggressive hardwired assist that is tightly integrated with the cache or memory controller, such as the one used in the Origin2000. The least aggressive one represents placing a slow general-purpose processor on the memory bus to play the role of communication assist. The most aggressive network delays used represent modern high-end multiprocessor interconnects whereas the least aggressive ones are closer to using commodity system area networks like asynchronous transfer mode (ATM). We can see that for an aggressive occupancy, the latency curves take the expected $1/l$ shape. The contention induced by assist occupancy has a major impact on performance for applications that stress communication throughput (especially those in which communication is bursty), particularly for the low-delay networks used in multiprocessors. Thus the curves for higher occupancies are far apart from one another toward their left ends. For reasonable occupancies, the curves become closer to one another at larger network delays, since the greater time spent by transactions in the network keeps the assist less busy and hence keeps contention at the assist smaller. For higher occupancies, the curve almost flattens, at least with lower network delays, indicating that the assist is saturated. The problem is especially

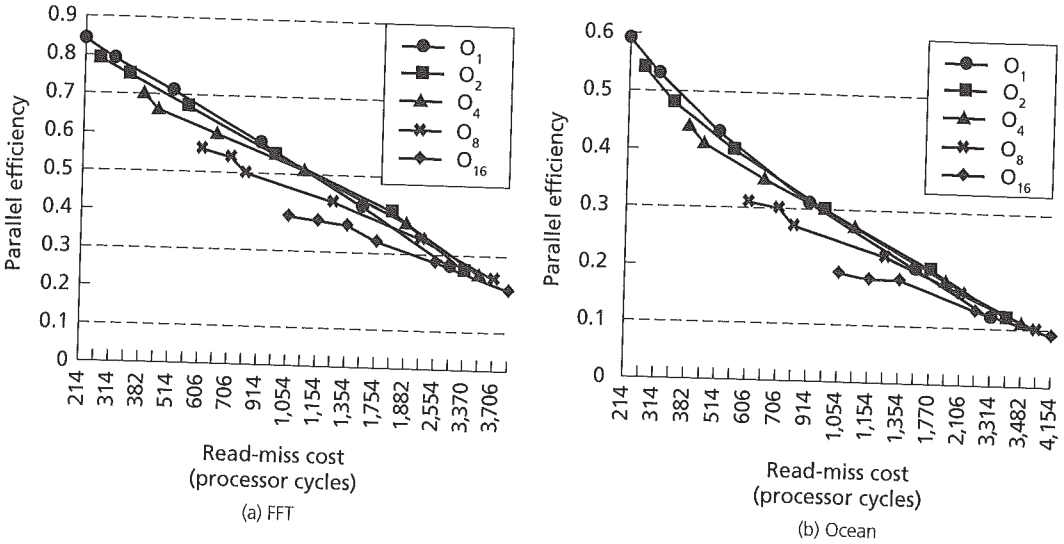


FIGURE 8.33 Impact of assist occupancy and network latency on the performance of memory-based cache coherence protocols. The *y*-axis is the parallel efficiency, which is the speedup over a sequential execution divided by the number of processors used (1 is ideal speedup). The *x*-axis is the uncontended round-trip latency of a read miss that is satisfied in main memory at the home, including all components of cost (occupancy, transit latency, time in buffers, and network bandwidth). Each curve is for a different value of assist occupancy (*o*), while along a curve the only parameter that varies is the network transit delay (*l*). The lowest occupancy assumed is 7 processor cycles, which is labeled O_1. O_2 corresponds to twice that occupancy (14 processor cycles) and so on. All other costs, such as the time to propagate through the cache hierarchy and through buffers and the node-to-network bandwidth, are held constant. The graphs are for simulated 64-processor executions. The main conclusion is that the contention induced by assist occupancy is very important to performance, especially in low-latency networks.

severe for applications with bursty communication, such as sorting and FFTs, since there the rate of communication relative to computation during the communication phase does not change much with problem size, so larger problem sizes do not help alleviate the contention during that phase. Assist occupancy is a less severe problem for applications in which communication events are separated by significant computation and whose communication bandwidth demands are small (e.g., Barnes-Hut). When latency tolerance techniques are used (discussed in Chapter 11), bandwidth is stressed even further, so the impact of assist occupancy is much greater even at higher transit latencies, and the curves at the highest occupancies are almost completely flat for FFT and sorting (Holt et al. 1995). This data shows that it is very important to keep assist occupancy low in machines that communicate and maintain coherence at a fine granularity such as that of cache blocks. The impact of contention due to assist occupancy tends to increase with the number of processors used to solve a given problem since the communication-to-computation ratio tends to increase.

Effects of Assist Occupancy on Protocol Trade-Offs

The occupancy of the assist has an impact not only on the performance of a given protocol but also on the trade-offs among protocols. We have seen that cache-based protocols can have higher latency on write operations than memory-based protocols since the transactions needed to invalidate sharers are serialized. The SCI cache-based protocol also tends to have more protocol processing to do on a given memory operation than a memory-based protocol, so the effective occupancy of the assist tends to be significantly higher, especially when assists are programmable rather than hardwired. Combined with the higher latency on writes, this would tend to cause memory-based protocols to perform better. This difference between the performance of the protocols will become greater as assist occupancy and its performance impact increase. On the other hand, the protocol processing occupancy for a given memory operation (e.g., a write) in SCI is distributed over more nodes and assists, so, depending on the communication patterns of the application, it may experience less contention at a given assist. For example, when hot spotting becomes a problem due to bursty irregular communication in memory-based protocols (as in radix sorting), it may be somewhat alleviated in SCI. How these trade-offs play out in practice will depend on the characteristics of real programs and machines, although overall we might expect memory-based protocols to perform better in optimized implementations.

Improving Performance Parameters in Hardware

There are many ways to use more aggressive, specialized hardware to improve performance characteristics such as delay, occupancy, and bandwidth. Some notable techniques include the following. First, an SRAM directory cache may be placed close to the assist to reduce directory lookup cost, as is done in NUMA-Q and in the Stanford FLASH multiprocessor (Kuskin et al. 1994). Second, a single bit of SRAM can be maintained per memory block at the home to keep track of whether or not the block is in clean state in the local memory. If it is, then on a read miss to a locally allocated block, there is no need to invoke the communication assist any further. Third, if the assist occupancy is high, it can be pipelined into stages of protocol processing, as is also done in the NUMA-Q and Stanford FLASH (e.g., decoding a request, looking up the directory, generating a response), or its occupancy can be overlapped with other actions. Pipelining the assist reduces contention but not the uncontended latency of individual memory operations; the opposite (and complementary) result can be achieved by having the assist generate and send out a response or a forwarded request even before all the cleanup it needs to do is done.

8.8 SYNCHRONIZATION

Software algorithms for synchronization on scalable non-cache-coherent shared address space systems using atomic exchange instructions or LL-SC are discussed in Section 7.9. Recall that the major focus of these algorithms compared to those for

bus-based machines is to exploit the parallelism of independent paths in the interconnect and to ensure that processors will spin on local rather than nonlocal variables. The same algorithms are applicable to scalable cache-coherent machines. However, there are two differences. First, the performance implications of spinning on remotely allocated variables are likely to be much less significant since a processor caches the variable and then spins on it locally until it is invalidated. Having processors spin on different variables rather than the same one is of course useful in preventing all processors from rushing out to the same home memory when the variable is written and invalidated, thereby reducing contention. And good placement of synchronization variables has the benefit of converting the misses that occur after invalidation into two-hop misses from three-hop misses. However, there is only one (very unlikely) situation when it may actually be very important to performance that the variable a processor spins on be allocated locally: if all levels of the cache hierarchy are unified and direct mapped and the instructions for the spin loop conflict with the variable itself, in which case conflict misses will be satisfied locally. Second, while these performance aspects of synchronization algorithms are less critical, implementing atomic primitives and LL-SC is more interesting when it interacts with a coherence protocol. This section examines the performance and implementation aspects, first comparing the performance of the different synchronization algorithms for the locks described in Chapters 5 and 7 on the SGI Origin2000 and then discussing some new implementation issues for atomic primitives beyond the issues already encountered in Chapter 6 for bus-based machines.

8.8.1 Performance of Synchronization Algorithms

The experiments used here to illustrate synchronization performance are the same as those used on the bus-based SGI Challenge in Section 5.5, again using LL-SC as the primitive to construct atomic operations. The delays used are the same in processor cycles and therefore different in actual microseconds. The results for the lock algorithms described in Chapters 5 and 7 are shown in Figure 8.34 for 16-processor executions. Here again, three different sets of values are used for the delays within and after the critical section for which processors repeatedly contend.

Here too, until we use delays between critical sections, the simple locks behave unfairly and yield higher throughput. Exponential backoff often helps the simple LL-SC lock in the event of a null critical section since this is the case where significant contention needs to be alleviated. The ticket lock scales quite poorly in this case, as it did on a bus, but scales very well when proportional backoff is used. The array-based lock also scales very well. With coherent caches, the better placement of lock variables in main memory afforded by the software queuing lock is not particularly useful. If we force the simple locks to behave fairly, they behave much like the ticket lock without proportional backoff.

If we use a non-null critical section and a delay between lock accesses (Figure 8.34[c]), all locks behave fairly. Now the simple LL-SC locks don't have

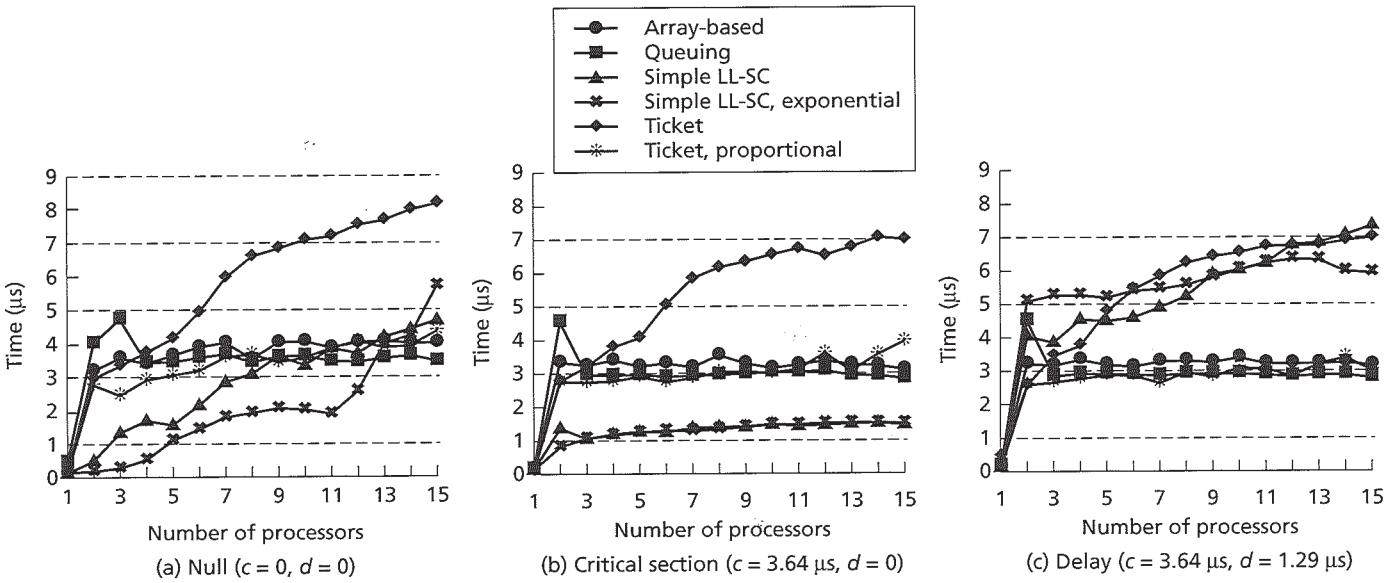


FIGURE 8.34 Performance of locks on the SGI Origin2000 for three different scenarios

their advantage, and their scaling disadvantage shows through. The array-based lock, the queuing lock, and the ticket lock with proportional backoff all scale well (at least to this small number of processors). The better data placement of the queuing lock does not matter, but neither is the contention any worse for it. The bad performance of the queuing lock at two processors is due to a specific interaction in constructing the software queue (Mellor-Crummey and Scott 1991). While experiments with larger-scale machines are warranted, the flattening of the curves indicates that, overall, the array-based lock and the ticket lock perform quite well and robustly for scalable cache-coherent machines, at least when implemented with LL-SC. The simple LL-SC lock with exponential backoff performs best when no delay occurs between an unlock and the next lock due to repeated unfair successful access by a processor in its own cache. The sophisticated queuing lock is unnecessary but also performs well with delays between unlock and lock.

More aggressive hardware support for locks has been proposed. The most prominent example is a hardware version of the queuing lock called QOLB (queue on lock bit). A distributed linked list of nodes waiting on a lock is maintained in hardware, and a releaser grants the lock to the first waiting node without affecting the others (Kägi, Burger, and Goodman 1997). Since the SCI protocol already has hardware support for a distributed list of waiting nodes (namely, the pending list), QOLB locks fit very well with SCI. This aggressive hardware support may reduce the lock transfer time as well as the interference of lock traffic with data access and coherence traffic; however, it is unlikely to change the scaling trends of the lock microbenchmarks, and, as with all system features, its true value to performance is best evaluated with real applications and workloads.

Algorithms and hardware support for barriers are discussed in Section 7.9. Since barriers reached simultaneously by multiple nodes cause contention for read-modify-write access to a shared counter, a number of interesting questions arise: Should this counter variable be a cacheable location or an uncached location accessed at main memory? Or can mechanisms be developed to allow processors to spin in their caches and either be updated at the release or read the release value from main memory rather than from the releaser's cache? Or is the hardware support for at-memory fetch&op operations particularly valuable as provided by machines like the Origin2000?

8.8.2 Implementing Atomic Primitives

Consider implementing atomic exchange (read-modify-write) primitives like test&set performed on a memory location. What matters for atomicity is that a conflicting write to that location by another processor occur either before the read component of the read-modify-write operation or after its write component. As we discussed for bus-based machines in Section 5.5.3, the read component may be allowed to complete as soon as the write component is serialized with respect to other writes and as long as we ensure that no incoming invalidations are applied to the block until the read has completed. If the read-modify-write is implemented at the processor (using cacheable primitives), this means that the read can complete

once the write has obtained ownership and even before invalidation acknowledgments have returned. Atomic operations can also be implemented at the memory, but it is easier to do this if we disallow the block from being cached in dirty state by any processor. Then all writes go to memory, and the read-modify-write can be serialized with respect to other writes as soon as it gets to memory. Memory can send a response to the read component in parallel with sending out invalidations corresponding to the write component.

Implementing LL-SC requires all the same consideration to avoid livelock as it did for bus-based machines, with one further complication. Recall that a store-conditional should not send out invalidations or updates if it fails since, otherwise, two processors may keep invalidating or updating each other and failing, causing livelock. To detect failure of a store-conditional, the requesting processor needs to determine if some other processor's write to the block has been serialized before the store-conditional. In a bus-based system, the cache controller can do this by checking upon a store-conditional whether the cache no longer has a valid copy of the block or whether there are incoming invalidations or updates for the block that have already appeared on the bus. The latter detection of serialization order cannot be done locally by the cache controller with a distributed interconnect, so a different mechanism is necessary. In an invalidation-based protocol, if the block is still in valid state in the cache, then the read-exclusive request corresponding to the store-conditional goes to the directory at the home. There it checks to see if the requestor is still on the sharing list. If it isn't, then the directory knows that another conflicting write has been serialized before the store-conditional, so it does not send out invalidations corresponding to the store-conditional and the store-conditional fails. Otherwise, it succeeds. In an update protocol, this is more difficult since, even if another write has been serialized before the store-conditional, the store-conditional requestor will still be on the sharing list. One solution (Gharachorloo 1995) is to again use a two-phase protocol as was used to provide write atomicity for updates. When the store-conditional reaches the directory, it locks down the entry for that block so that no other requests can access it. Then, the directory sends a message back to the store-conditional requestor, which upon receipt checks to see if the lock flag for the LL-SC has been cleared (by an update that arrived between the current time and the time the store-conditional request was sent out). If so, the store-conditional has failed and a message is sent back to the directory to this effect (and to unlock the directory entry). If not, then as long as point-to-point order is guaranteed in the network, we can conclude that no conflicting write beat the store-conditional to the directory, so the store-conditional should succeed. The requestor sends an acknowledgment back to the directory, which unlocks the directory entry and sends out the updates corresponding to the store-conditional, and the store-conditional succeeds.

8.9 IMPLICATIONS FOR PARALLEL SOFTWARE

Let us now consider the implications for parallel software more generally than for synchronization. What distinguishes the coherent shared address space systems

described in this chapter from those described in Chapters 5 and 6 is that they have physically distributed rather than centralized main memory. Distributed memory is at once an opportunity to improve performance and scalability through data locality and a burden on software to exploit this locality. As we saw in Chapter 3, on cache-coherent architectures with physically distributed memory (or CC-NUMA machines), such as those discussed in this chapter, parallel programs may need to be aware of physically distributed memory, particularly when their important working sets don't fit in the cache. Artifactual communication occurs when data is not allocated in the memory of a node that incurs capacity, conflict, or cold misses on that data. This situation can lead to some artifactual communication even when data does fit in the cache since looking up the directory on write misses (including upgrades) will generate network traffic and contention. Finally, consider a multiprogrammed workload in which application processes are migrated among processing nodes for load balancing. Migrating a process will turn what should be local misses into remote misses unless the system moves all the migrated process's data to the new node's main memory as well. For all these reasons, it may be important that data be allocated appropriately across the distributed memories.

In the CC-NUMA machines discussed in this chapter, the management of main memory is typically done at the fairly large granularity of pages. The large granularity can make it difficult to distribute shared data structures appropriately since data that should be allocated on two different nodes may fall on the same unit of allocation. The operating system may transparently migrate pages to the nodes that incur cache misses on them most often, using information obtained from hardware counters; or the run-time system of a programming language may migrate pages based on user-supplied hints or compiler analysis. (We saw that the Origin2000 provides protocol support for efficient migration.) More commonly today, the programmer may direct the operating system to place pages in the memories closest to particular processes. This may be as simple as providing these directives to the system—such as, "Place the pages in this range of virtual addresses in this process X's local memory"—or it may additionally involve padding and aligning data structures to page boundaries so they can be placed properly, or it may even require that data structures be organized differently to allow such placement at page granularity. We saw examples of the need for all three in using four-dimensional instead of two-dimensional arrays in the equation solver kernel and in Ocean. Simple, regular cases like these may also be handled by sophisticated compilers. In Barnes-Hut, on the other hand, proper placement would require a significant reorganization of data structures as well as code. Instead of having a single linear array for all particles (or cells), each process would have an array or list of its own assigned particles that it could allocate in its local memory; between time-steps, particles that were reassigned would be moved from one array or list to another. However, as we have seen, data placement is not very useful for this application due to the small working sets and low capacity miss rate and may even hurt performance due to its high costs. It is important that we understand the costs and potential benefits of data migration before using it. Similar issues hold for software-controlled replication of data instead

of migration, and the next chapter discusses alternative approaches to coherent replication and migration in main memory.

One of the most difficult problems for a programmer to deal with in a coherent shared address space is contention. Contention can be caused not only by data traffic that is implicit and often unpredictable but also by "invisible" protocol transactions, such as ownership requests, invalidations, and acknowledgments that a programmer is not inclined to think about at all and that are now point-to-point rather than amortized by a broadcast medium. All of these types of transactions occupy the protocol processing portion of the communication assist, reinforcing the importance of keeping the occupancy of the assist per transaction very low to contain endpoint contention. Invisible protocol messages and contention make performance problems like false sharing all the more important for a programmer to avoid, particularly when they cause a lot of protocol transactions to be directed toward the same node. Thus, while the software techniques for inherent communication and for spatial locality and false sharing at cache block granularity are the same as on bus-based machines, the potential impact on performance is different. For example, we are often tempted to structure some kinds of data as an array with one entry per process. If the entries are smaller than a page, several of them will fall on the same page. If these array entries are not padded to avoid false sharing or if they incur conflict misses in the cache, all the misses and traffic will be directed at the home of that page, causing considerable contention. In a distributed-memory machine it is advantageous not only to structure such data as an array of records rather than multiple arrays of scalars (as we do in Chapter 5 to avoid false sharing) but also to pad and align the records to a page and place the pages in the appropriate local memories.

An interesting example of how contention can cause different orchestration strategies to be used in message-passing and shared address space systems is illustrated by a high-performance parallel FFT. Conceptually, the computation is structured in phases. Phases of local computation are separated by phases of communication, which involve the transposition of a matrix. A process reads columns from a source matrix and writes them into its assigned rows of a destination matrix and then performs local computation on its assigned rows of the destination matrix. In a message-passing system, it is important to coalesce data into large messages, so it is necessary for performance to structure the communication this way (as a phase separate from computation). However, in a cache-coherent shared address space there are two differences. First, transfers are always done at cache block granularity. Second, each fine-grained transfer involves invalidations and acknowledgments (each local block that a process writes is likely to be in shared state in the cache of another processor from a previous phase and so must be invalidated), which cause contention at the coherence controllers. It may therefore be preferable to perform the communication on demand at fine grain while the computation is in progress, rather than all at once in a separate transpose phase, thus staggering the communication and easing the contention on the controller: a process that otherwise computes using a row of the destination matrix after the transpose can read the words of the corresponding source matrix column from a remote node on demand while it is

computing, performing the transpose in the process. Which method is better may depend on the architecture.

Finally, synchronization can be expensive in scalable systems, so programs should make a special effort to reduce the frequency of high-contention locks or global barrier synchronization.

8.10 ADVANCED TOPICS

Before concluding the chapter, we cover two additional topics. The first deals with the actual techniques used to reduce directory storage overhead in flat, memory-based schemes. The second addresses techniques for hierarchical coherence, both snooping and directory based.

8.10.1 Reducing Directory Storage Overhead

The discussion of flat, memory-based directories in Section 8.2.3 stated that the size or width of a directory entry can be reduced by using a limited number of pointers rather than a full bit vector and that doing so requires some overflow mechanism when the number of copies of the block exceeds the number of available pointers. Based on the empirical data about sharing patterns, the number of hardware pointers likely to be provided in limited pointer directories is very small, so it is important that the overflow mechanism be efficient. This section first discusses some possible overflow methods. It then examines techniques to reduce the number of directory entries, or directory "height," by organizing the directory as a cache rather than having an entry for every memory block in the system. The limited pointer schemes with i pointers are named Dir_i followed by an abbreviation of their overflow methods, which include broadcast, no broadcast, coarse vector, software overflow, and dynamic pointers.

Overflow Methods for Reduced Directory Width

The overflow strategy in the *broadcast* or *Dir_iB scheme* (Agarwal et al. 1988) is to set a broadcast bit in the directory entry when the number of available pointers i is exceeded. When that block is written again, invalidation messages are sent to all nodes in the system, regardless of whether or not they were caching the block. It is not semantically incorrect to send an invalidation message to a processor not caching the block; however, network bandwidth may be wasted and latency stalls may be increased if the processor performing the write must wait for acknowledgments before proceeding. The advantage of the method is its simplicity.

The *no broadcast* or *Dir_iNB scheme* (Agarwal et al. 1988) avoids broadcast by never allowing the number of valid copies of a block to exceed i. Whenever the number of sharers is i and another node requests a shared copy of the block, the protocol invalidates the copy in one of the existing sharers and frees up that pointer in the directory entry for the new requestor. A major drawback of this scheme is that it

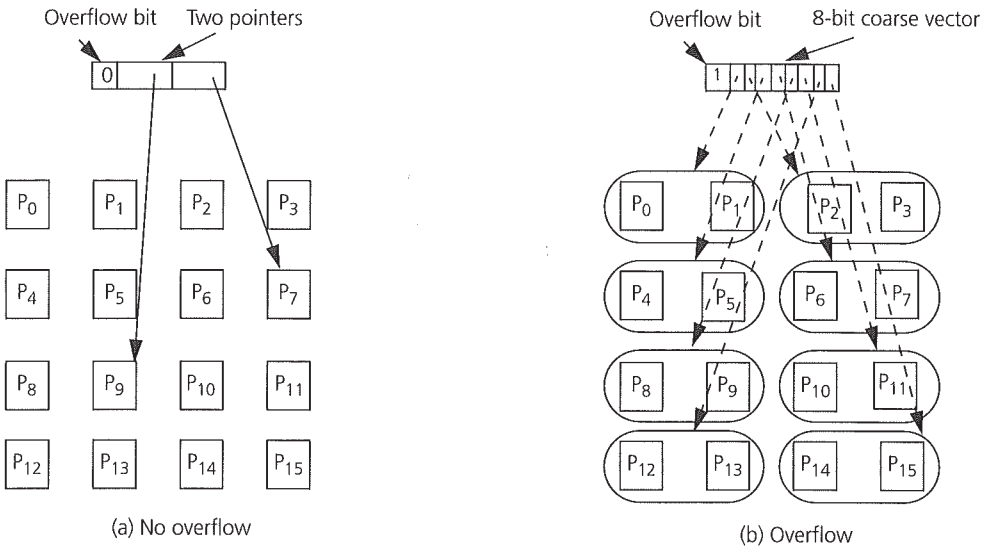


FIGURE 8.35 The change in representation in going from limited pointer representation to coarse vector representation on overflow. Upon overflow, the two 4-bit pointers (for a 16-node system) are viewed as an 8-bit coarse vector, each bit corresponding to a group of two nodes. The overflow bit is also set, so the nature of the representation can be easily determined. The dotted lines in (b) indicate the correspondence between bits and node groups.

does not deal well with data that is actively read by many processors during a period (e.g., tables of precomputed values or even program code), since copies will unnecessarily be invalidated and a continual stream of misses generated. Although special provisions can be made for blocks containing code (e.g., their consistency may be managed by software instead of hardware), it is not clear how to handle widely shared read-mostly data well in this scheme.

The *coarse vector* or *Dir_iCV_r scheme* (Gupta, Weber, and Mowry 1990) also uses i pointers in its initial representation, but on overflow the representation changes to a coarse bit vector like the one used by the Origin2000 for large machines. In this representation, each bit of the directory entry indicates not a node but a unique group of the nodes in the machine (the subscript r in Dir_iCV_r indicates the size of the group), and that bit is turned ON whenever any node in that partition is caching that block (see Figure 8.35). When a processor writes that block, all nodes in the groups whose bits are turned ON are sent an invalidation message, regardless of whether they have actually accessed or are caching the block. As an example, consider a 256-node machine for which we store eight pointers in the directory entry. Since each pointer needs to be 8 bits wide, 64 bits are available for the coarse vector on overflow. Thus, we can implement a Dir_8CV_4 scheme, with each coarse vector bit pointing to a group of 256/64 or four nodes. An additional single bit per entry keeps track of whether the current representation is that of the normal limited pointer or the coarse vector. As shown in Figure 8.36, an advantage of a scheme like Dir_iCV_r (and,

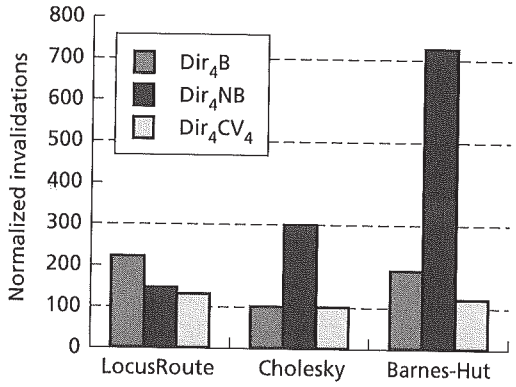


FIGURE 8.36 Robustness of the coarse vector overflow method relative to broadcast and no broadcast. The figure shows a comparison of invalidation traffic generated by Dir_4B, Dir_4NB, and Dir_4CV_4 schemes normalized to that generated by the full bit vector scheme (represented as 100 invalidations). The results are taken from (Weber 1993), so the simulation parameters are different from those used in this book. The number of processors (1 per node) is 64. The data for the LocusRoute wire-routing application, which has data that is written quite frequently and read by many nodes, shows the potential pitfalls of the Dir_iB scheme. Cholesky and Barnes-Hut, which have data that is read shared by large numbers of processors (e.g., nodes close to the root of the tree in Barnes-Hut) show the potential pitfalls of the Dir_iNB scheme. The Dir_iCV_r scheme is found to be reasonably robust.

even more so, of the following schemes) over Dir_iB and Dir_iNB is that its behavior is more robust to different sharing patterns.

The *software overflow* or *Dir_iSW scheme* is different from the previous ones in that it does not throw away the precise caching status of a block when overflow occurs. Rather, the current i pointers and a pointer to the new sharer are saved into a special portion of the node's local main memory by software. This frees up space for new pointers, so i new sharers can be handled by hardware before software must be invoked to store pointers away into memory again. The overflow also causes an overflow bit to be set in hardware. This bit ensures that when a subsequent write is encountered the pointers that were stored away in memory will be read out, and invalidation messages will be sent to those nodes as well. In the absence of a very sophisticated (programmable) communication assist, the overflow situations (both when pointers must be stored into memory and when they must be read out and invalidations sent) are handled by software running on the main processor, so the processor must be interrupted or a trap generated upon these events. The advantages of this scheme are that precise information is kept about sharers even upon overflow, so there is no extra invalidation traffic generated compared to a full bit vector (or unlimited pointer) representation, and that the complexity of overflow handling is managed by software. The major overhead is the cost of the interrupts and software processing. This disadvantage takes three forms: (1) the processor at the home of the block spends time handling the interrupt instead of performing the

user's computation; (2) the overhead of interrupts and of handling these requests is large, thus potentially becoming a bottleneck for contention and slowing down other requests; and (3) the requesting processor may stall longer because of the higher latency of the requests that can cause interrupts as well as increased contention.[5]

Software overflow for limited pointer directories was used in the MIT Alewife research prototype (Agarwal et al. 1995) and was called the LimitLESS scheme (Agarwal et al. 1991). The Alewife machine is designed to scale to 512 processors with one processor per node. Each directory entry is 64 bits wide. It contains five 9-bit pointers to record remote nodes caching the block and 1 dedicated bit to indicate whether the local node is also caching the block (thus saving 8 bits when this is true). Overflow pointers are stored in a hash table in the main memory. The main processor in Alewife has hardware support for multithreading (see Chapter 11), with support for fast handling of traps upon overflow. Nonetheless, although the latency of a request that causes five invalidations and can be handled in hardware is only 84 cycles on a 16-processor system, a request requiring six invalidations and, hence, software intervention takes 707 cycles.

The *dynamic pointers* or *Dir$_i$DP scheme* (Simoni and Horowitz 1991) is a variation of the Dir$_i$SW scheme. In addition to the i hardware pointers, each directory entry in this scheme contains a hardware pointer into a special portion of the local node's main memory. This special memory has a free list associated with it, from which pointer structures can be dynamically allocated to processors as needed. The key difference from Dir$_i$SW is that all linked-list manipulation is done in hardware by a special-purpose protocol processor rather than by the general-purpose processor of the local node. As a result, interrupts are not needed and the overhead of manipulating the linked lists is small. Because it also contains a hardware pointer to memory, the number of hardware pointers i used in this scheme is typically very small. The Dir$_i$DP scheme is the default directory organization for the Stanford FLASH multiprocessor (Kuskin et al. 1994). Because the pool of dynamic pointers is limited and because lists are traversed on invalidations, the use of replacement hints usually accompanies this approach.

Among these many alternative schemes for maintaining directory information in a memory-based protocol, it is quite clear that the Dir$_i$B and Dir$_i$NB schemes are not very robust to different sharing patterns. However, the actual performance (and cost-performance) trade-offs among the schemes are not very well understood for real applications on large-scale machines. The general consensus seems to be that full bit vectors are appropriate for machines that have a moderate number of processing nodes that are visible to the directory protocol. The most likely candidates for hardware overflow schemes are coarse vector and dynamic pointer: the former may suffer from lack of accuracy on overflow, while the latter has greater processing cost due to hardware list manipulation and free list management.

---

5. It is actually possible to respond to a requestor before the trap is handled and thus not affect the latency seen by it. However, that simply means that the next processor's request to that node is delayed and that processor may experience a stall.

Reducing Directory Height

In addition to reducing directory entry width, an orthogonal way to reduce directory memory overhead is to reduce the total number of directory entries used by not using one per memory block (Gupta, Weber, and Mowry 1990; O'Krafka and Newton 1990); that is, to go after the *M* term in the *P*M* expression for directory memory overhead. Since the two methods of reducing overhead are orthogonal, they can be traded off against each other: reducing the number of entries allows us to make entries wider (use more hardware pointers) without increasing cost and vice versa.

The observation that motivates the use of fewer directory entries is that the total amount of cache memory is much less than the total main memory in the machine. This means that only a very small fraction of the memory blocks will be cached at a given time. For example, each processing node may have a 1-MB cache and 64 MB of main memory associated with it. If there were one directory entry per memory block, then across the whole machine 63/64 or 98.5% of the directory entries will correspond to memory blocks that are not cached anywhere in the machine. That is a tremendous number of directory entries lying idle with no bits turned ON (especially when replacement hints are used). This waste of memory can be avoided by organizing the directory as a cache and dynamically allocating the entries in it to directory entries, just as cache lines are allocated to memory blocks containing program data. In fact, if the number of entries in this directory cache is small enough, it may enable us to use fast SRAMs instead of slower DRAMs for directories, thus reducing the access time to directory information. As we know, this access time is in the critical path that determines the latency seen by the processor for many types of memory references. Such a directory organization is called a *sparse directory*, for obvious reasons. (The HAL S1 system, described in Section 8.6.8, uses this approach.)

While a sparse directory operates quite like a regular processor cache, there are some significant differences. First, this cache has no need for a backing store: when an entry is replaced from it, if any node's bits (or pointers) in it are turned on then we can simply send invalidations or flush messages to those nodes. Second, there is only one directory entry per block in this cache, so spatial locality is not an issue. Third, a sparse directory handles references from potentially all processors, whereas a processor cache is only accessed by the processor(s) attached to it. And finally, the references stream that the sparse directory sees is heavily filtered, consisting of only those references that were not satisfied in the processor caches. For a sparse directory not to become a bottleneck, it is essential that it be large enough and have enough associativity that it does not incur too many replacements of actively accessed blocks. Some experiments and analysis studying the sizing of the sparse directory can be found in (Weber 1993).

8.10.2 Hierarchical Coherence

The introduction to this chapter mentions that one way to build scalable coherent machines is to hierarchically extend the snoopy coherence protocols based on the

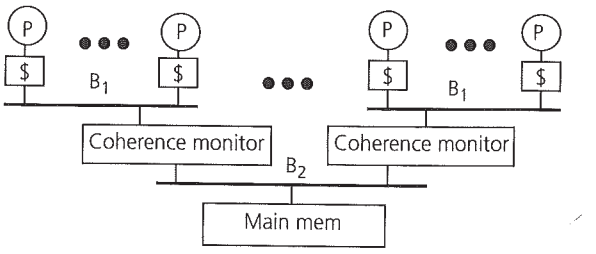
buses and rings that are discussed in Chapters 5 and 6. We have also been introduced to hierarchical directory schemes in this chapter. This section describes these hierarchical approaches to coherence further. Although hierarchical ring-based snooping has been used in commercial systems (e.g., in the Kendall Square Research KSR1 [Frank, Burkhardt, and Rothnie 1993]) as well as research prototypes (e.g., in the University of Toronto's Hector system [Vranesic et al. 1991; Farkas, Vranesic, and Stumm 1992]), and hierarchical directories have been studied in academic research, these approaches have not gained much favor. Nonetheless, building large systems hierarchically out of smaller ones is an attractive abstraction, and it is useful to understand the basic techniques.

Hierarchical Snooping

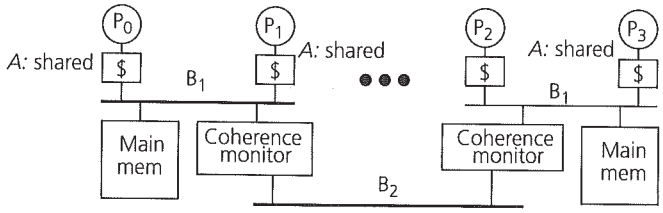
The issues in hierarchical snooping are similar for buses and rings, so we study them mainly through the former. A bus hierarchy is a tree of buses. The leaves are bus-based multiprocessors that contain the processors. The buses that constitute the internal nodes of the tree don't contain processors but are used for interconnection and coherence control: they allow transactions to be snooped and propagated up and down the hierarchy as necessary. Hierarchical machines can be built with main memory either centralized at the root or distributed among the leaf multiprocessors (see Figure 8.37). While a centralized main memory may simplify programming, distributed memory has advantages in bandwidth and performance if locality is exploited. (Note, however, that if data is not distributed such that most cache misses are satisfied locally, remote data may actually be further away than the root of the hierarchy in the worst case, potentially leading to worse performance.) In addition, with distributed memory, a leaf in the hierarchy is a complete bus-based multiprocessor, which is already a commodity product with cost advantages. Let us focus on hierarchies with distributed memory, leaving centralized memory hierarchies to be explored in the exercises.

The processor caches within a leaf node (multiprocessor) are kept coherent by any of the snooping protocols discussed in Chapter 5. In a simple, two-level hierarchy, we connect several of these bus-based systems together using another bus (B_2). (The extension to multilevel hierarchies is straightforward.) What we need is a coherence monitor associated with each B_1 bus that monitors (snoops) the transactions on both buses and decides which transactions on its B_1 bus should be forwarded to the B_2 bus and which ones that appear on the B_2 bus should be forwarded to its B_1 bus. This device acts as a filter, forwarding only the necessary transactions in both directions, and thus reduces the bandwidth demands on the buses.

In a system with distributed memory, the coherence monitor for a node has to worry about two types of data for which transactions may appear on either the B_1 or B_2 bus: data that is allocated remotely but cached by some processor in the local node and data that is allocated locally but cached remotely. To watch for the former data, a *remote access cache* or *remote cache* per node can be used as in the Sequent NUMA-Q. This cache maintains inclusion (see Section 6.3.1) with regard to remote data cached in any of the processor caches on that node, including a dirty-but-stale



(a) Centralized memory



(b) Distributed memory

FIGURE 8.37 Hierarchical bus-based multiprocessors, shown with a two-level hierarchy. Main memory may be centralized at the root or physically distributed, and coherence monitors connect parent and child buses.

bit per block indicating when a processor cache in the node has the block dirty (data allocated in local memory does not enter the remote cache). This gives it enough information to determine which transactions are relevant in each direction and pass them along.

For locally allocated data, bus transactions can be handled entirely by the local memory or caches, except when the data is cached by processors in other (remote) nodes. For the latter data, there is no need to keep the data itself in the coherence monitor since the valid data is either already available locally or is in modified state remotely; in fact, we would not want to keep it there since the amount of data may be as large as the local memory. However, the monitor keeps state information for this data and snoops the local B_1 bus so that relevant transactions for this data can be forwarded to the B_2 bus if necessary. Let's call this part of the coherence monitor the *local state monitor*. Finally, the coherence monitor also watches the B_2 bus for transactions to its local addresses and passes them onto the local B_1 bus unless the local state monitor says they are cached remotely in a modified state. Both the remote cache and the local state monitor are looked up on B_1 and B_2 bus transactions.

Consider the three coherence protocol functions outlined in Section 8.1: (1) enough information about the state in other nodes of the hierarchy is implicitly available in the local node's coherence monitor (remote cache and local state monitor) to determine what action to take; (2) if this information indicates a need to find

other copies beyond the local node, the request or search is broadcast on the next bus (and so on hierarchically in deeper hierarchies), and other relevant monitors will respond; and (3) communication with the other copies is performed simultaneously as part of finding them through the hierarchical broadcasts on buses.

Let us examine the path of a read miss more closely, assuming a shared physical address space. A BusRd request appears on the local B_1 bus. If the remote access cache, the local memory, or another local processor cache has a valid copy of the block, they will supply the data. Otherwise, either the remote cache or the local state monitor will know to pass the request onto the B_2 bus. When the request appears on B_2, the coherence monitors of other nodes will snoop it. If a node's local state monitor determines that a valid copy of the data exists in that node, it will pass the request onto its B_1 bus, wait for the response, and put it back on the B_2 bus. If a node's remote cache contains the data and has it in shared state, it may simply place a reply on the B_2 bus; if in dirty state, it will reply and broadcast a read request on its B_1 bus to have the dirty processor cache downgrade the block to shared; and if dirty-but-stale, it will simply broadcast the read request on its B_1 bus and reply with the result obtained. In the last case, the processor cache that has the data dirty will change its state from dirty to shared and put the data on the B_1 bus. The remote cache will accept the data reply from the B_1 bus, change its state from dirty-but-stale to shared, and pass the reply onto the B_2 bus. When the data reply appears on B_2, the requestor's coherence monitor picks it up, installs it and changes state in its remote cache if appropriate, and places it on its local B_1 bus. (If the block has to be installed in the remote cache, it may replace some other block, which will trigger a flush/invalidation request on that B_1 bus to ensure the inclusion property.) Finally, the requesting cache picks up the response to its BusRd request from the B_1 bus and stores it in shared state.

For writes, consider the specific situation shown in Figure 8.37(b), with P_0 in the left node issuing a write to location *A*, which is allocated in the memory of a third node (not shown). Since P_0's own cache has the data only in shared state, an ownership request (BusUpgr) is issued on the local B_1 bus. As a result, the copy of *A* in P_1's cache is invalidated. Since the block is not available in the remote cache in dirty-but-stale state (which would have been incorrect since P_1 had it in shared state), the monitor passes the BusUpgr request to bus B_2, to invalidate any other copies in the system, and at the same time updates the state for the block in the remote cache to dirty-but-stale. In another node, P_2 and P_3 have the block in their caches in shared state. Because of the inclusion property, their associated remote cache is also guaranteed to have the block in shared state. This remote cache therefore passes the BusUpgr request from B_2 onto its local B_1 bus and invalidates its own copy. When the request appears on the B_1 bus, the copies of *A* in P_2 and P_3's caches are invalidated. If there is a node on the B_2 bus whose processors are not caching the block containing *A*, the upgrade request will not pass onto its B_1 bus. Now suppose another processor P_4 in the left node issues a store to location *B*. This request will be satisfied within the local node, with P_0's cache supplying the data and the remote cache retaining the data in dirty-but-stale state, and no transaction will be passed onto the B_2 bus.

The implementation requirements on the processor caches and cache controllers remain unchanged from those discussed in Chapter 6. However, some constraints do apply to the remote access cache. It should be larger than the sum of the processor caches and quite associative to maintain inclusion without excessive replacements. It should also be *lockup-free;* that is, able to handle multiple requests at a time from processors in the local node while some requests are still outstanding (more on this in Chapter 11). Finally, whenever a block is replaced from the remote cache, an invalidation or flush request must be issued on the B_1 bus, depending on the state of the replaced block (shared or dirty-but-stale, respectively). Minimizing the access time for the remote cache is less critical than increasing its hit rate since it is not in the critical path that affects the clock rate of the processor. Remote caches are therefore more likely to be built out of DRAM than SRAM. The remote cache controller must also deal with the nonatomicity issues in requesting and acquiring the buses that were discussed in Chapter 6.

Finally, consider write serialization and determining store completion. From our earlier discussion of how these work on a single bus in Chapter 6, it should be clear that serialization between two requests will be determined by the order in which those requests appear on the closest bus to the root on which they both appear. For writes that are satisfied entirely within the same leaf node, the order in which they may be seen by other processors—within or without that leaf—is their serialization order provided by the local B_1 bus. Likewise, for writes that are satisfied entirely within the same subtree, the order in which they are seen by other processors—within or without that subtree—is the serialization order determined by the root bus of that subtree. It is easy to see this if we view each bus hanging off a common bus as a processor and recursively use the same reasoning applied to a single bus in Chapters 5 and 6. Similarly, for the store completion detection needed for sequential consistency, a processor cannot assume its store has committed until it appears on the closest bus to the root on which it will appear. An acknowledgment (which now may have to be an explicit bus transaction) cannot be generated until that time, and even then the appropriate orders must be preserved between this acknowledgment and other transactions on the way back to the requesting processor (see Exercise 8.26). Once this acknowledgment is sent back from a bus, the invalidations themselves no longer need to be acknowledged as they make their way down toward the processor caches, as long as the appropriate orders are maintained along this path (just as with multilevel cache hierarchies in Chapter 6).

One of the earliest machines that used the approach of hierarchical snooping buses with distributed memory was the Gigamax (Wilson 1987; Woodbury et al. 1989) from Encore Corporation. The system consisted of up to eight Encore Multimax machines (each a regular snooping bus-based multiprocessor) connected together by fiber-optic links to a ninth global bus, forming a two-level hierarchy. Figure 8.38 shows a block diagram. Each node is augmented with a uniform interconnection card (UIC) and a uniform cluster (node) cache (UCC) card. The UCC is the remote access cache, and the UIC is the local state monitor. The monitoring of the global bus is done differently in the Gigamax due to its particular organization. Nodes are connected to the global bus through a fiber-optic link, so while a node's

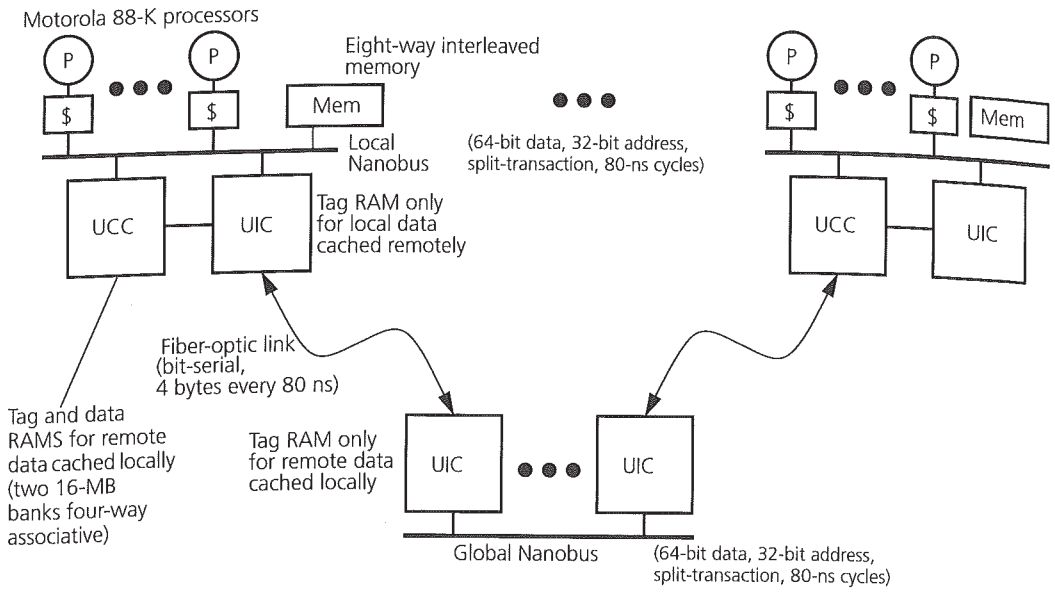


FIGURE 8.38 Block diagram for the Encore Gigamax multiprocessor. A two-level hierarchy of buses is used with memory distributed among the leaf nodes.

remote access cache (the UCC) caches remote data, it does not snoop the global bus directly. Rather, every node also has a second UIC on the global bus, which monitors global bus transactions for remote memory blocks that are cached in this local node. It then passes on the relevant requests to the local bus. If the UCC indeed sat directly on the global bus as well, the UIC on the global bus would not be necessary. The reason the Gigamax uses fiber-optic links and not a single UIC per node that sits on both buses is that high-speed buses are usually short: the Nanobus used in the Encore Multimax and Gigamax is 1 foot long (light travels 1 foot in a nanosecond, hence the name Nanobus). Since each node is at least 1 foot wide and the global bus is also 1 foot wide, flexible cabling is needed to hook these together. With fiber, links can be made quite long without affecting their transmission capabilities.

The extension of snooping cache coherence to hierarchies of rings is much like the extension to hierarchies of buses with distributed memory. Figure 8.39 shows a block diagram. The local rings and the associated processors constitute nodes, and these are connected by one or more global rings. The coherence monitor takes the form of an inter-ring interface, serving the same roles as the coherence monitor in a bus hierarchy.

Hierarchical Directory Schemes

Hierarchical directory schemes use point-to-point network transactions rather than snooping. However, as discussed earlier, unlike in flat directory schemes, the source

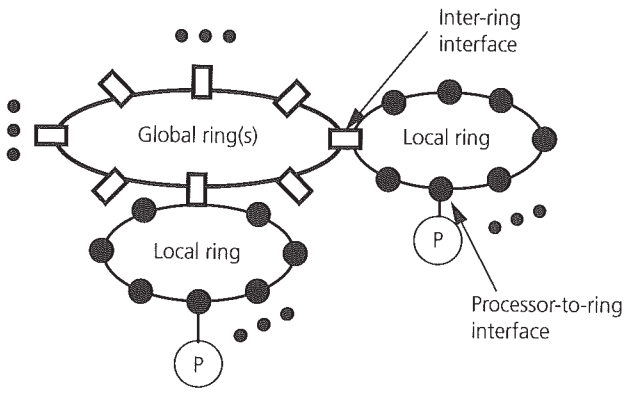


FIGURE 8.39 Block diagram for a hierarchical ring-based multiprocessor. In the two-level hierarchy shown, each local ring is a node as viewed by the global ring, and an inter-ring interface propagates relevant transactions between the two.

of the directory information in hierarchical directories is not found by going to a fixed node. The locations of copies are found neither at a fixed home node nor by traversing a distributed list pointed to by that home. Invalidation messages are not sent directly to the nodes with copies. Rather, all these activities are performed by sending messages up and down a hierarchy (tree) built upon the nodes, with the only direct communication being between parents and children in the tree.

At first blush, the organization of hierarchical directories is much like hierarchical snooping. Consider the example shown in Figure 8.40. The processing nodes are at the leaves of the tree and main memory is distributed along with the processing nodes. Every block has a home memory (leaf) in which it is allocated, but this does not mean that the directory information is maintained or rooted there. The internal nodes of the tree are not processing nodes but only hold directory information. Each such directory node keeps track of all memory blocks that are being cached or recorded by its subtrees. It uses a presence vector per block to tell which of its subtrees have copies of the block and a bit to tell whether one of them has it dirty. It also records information about local memory blocks (i.e., blocks allocated in the local memory of one of its descendants) that are being cached by processing nodes outside its subtree. As with hierarchical snooping, this information is used to decide when requests originating within the subtree should be propagated further up the hierarchy. Since the amount of directory information to be maintained by a directory node that is close to the root can become very large, the directory information is usually organized as a cache to reduce its size and maintains the inclusion property with respect to its children's caches or directories. This requires that on a replacement from a directory cache at a certain level of the tree, the replaced block must be flushed out of all of its descendent directories in the tree as well. Similarly, replacement of the information about a block allocated within that subtree requires that copies of the block in nodes outside the subtree be invalidated or flushed. These operations can be quite expensive.

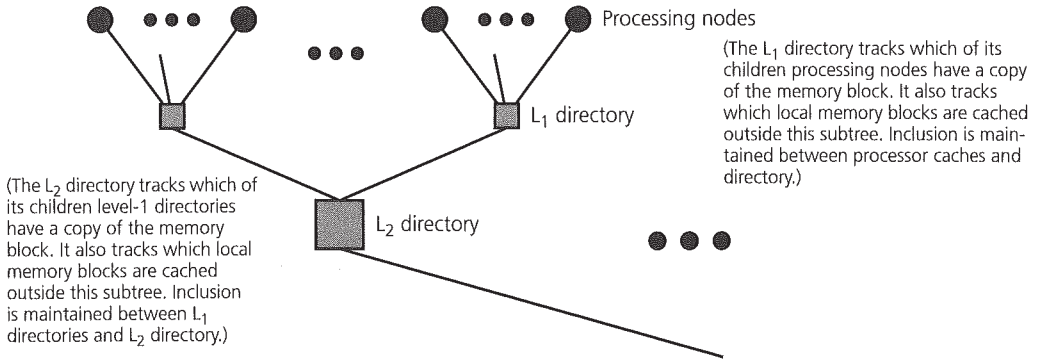


FIGURE 8.40 Organization of hierarchical directories. The processing nodes are at the leaves of the logical tree, and the internal nodes contain only directory information. There is one logical tree for each cached memory block. Logical trees may be embedded in any physical hierarchy.

A read miss from a node flows up the hierarchy either until a directory indicates that its subtree has a copy (clean or dirty) of the memory block being requested or until the request reaches the directory that is the first common ancestor of the requesting node and the home node for that block, and that directory indicates the block is not dirty outside that subtree. The request then flows down the hierarchy to the appropriate processing node to pick up the data. The data reply follows the same path back, updating the directories on its way. If the block was dirty, a copy of the block also finds its way to the home node.

A write miss in the cache flows up the hierarchy until it reaches a directory whose subtree contains the current owner of the requested memory block. The owner is either the home node, if the block is clean, or a dirty cache. The request travels down to the owner to pick up the data, and the requesting node becomes the new owner. If the block was previously in clean state, invalidations are also propagated through the hierarchy to all nodes caching that memory block. Finally, all directories involved in the preceding memory operation are updated to reflect the new owner and the invalidated copies.

In hierarchical snoopy schemes, the interconnection network is physically hierarchical to permit the snooping. With point-to-point communication, hierarchical directories do not need to rely on physically hierarchical interconnects. The hierarchy discussed here is a logical hierarchy, or a hierarchical data structure. It can be implemented either on a network that is physically hierarchical (that is, an actual tree network with directory caches at the internal nodes and processing nodes at the leaves) or on a general, nonhierarchical network such as a mesh with the hierarchical directory embedded in this general network. In fact, there is a separate hierarchical directory structure for every block that is cached. Thus, the same physical node in a general network can be a leaf (processing) node for some blocks and an internal (directory) node for others (see Figure 8.41).

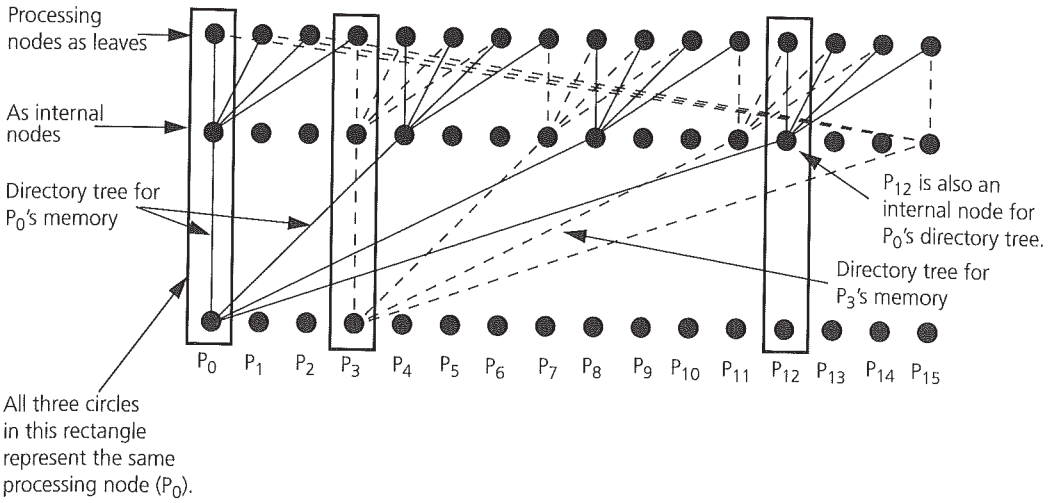


FIGURE 8.41 A multirooted hierarchical directory embedded in an arbitrary network. A 16-node hierarchy is shown. For the blocks in the portion of main memory that is located at a processing node, that node itself is the root of the (logical) directory tree. Thus, for P processing nodes, there are P directory trees. The figure shows only two of these. In addition to being the root for its local memory's directory tree, a processing node is also an internal node in the directory trees for the other processing nodes. The address of a memory block implicitly specifies a particular directory tree and guides the physical traversals to get from parents to children and vice versa in this directory tree.

Finally, the storage overhead of the hierarchical directory has attractive scaling properties. It is the cost of the directory caches at each level. The number of entries in the directory goes up as we go further up the hierarchy toward the root (to maintain inclusion without excessive replacements), but the number of directories becomes smaller. As a result, the total directory memory needed for all directories at any given level of the hierarchy is typically about the same. The directory storage needed is not proportional to the size of main memory but rather to that of the caches in the processing nodes, which is attractive. The overall directory memory overhead relative to main memory is proportional to

$$\frac{C \times \log_b P}{M \times B}$$

where C is the cache size per processing node at the leaf, M is the main memory per node, B is the memory block size in bits, b is the branching factor of the hierarchy, and P is the number of processing nodes at the leaves (so $\log_b P$ is the number of levels in the tree). More information about hierarchical directory schemes can be found in the literature (Scott 1991; Wallach 1992; Hagersten 1992; Joe 1995).

Performance Implications of Hierarchical Coherence

Hierarchical protocols, whether snoopy or directory, have some potential performance advantages that are extensions of the advantages of the two-level protocols discussed earlier. One is the combining of requests for a block as they go up and down the hierarchy. If a processing node is waiting for a memory block to arrive, another processing node that requests the same block can observe at their common ancestor directory that the block has already been requested. It can then wait at the intermediate directory and accept the response when it comes back rather than send a duplicate request. This combining of transactions can reduce traffic and, hence, contention. The sending of invalidations and gathering of invalidation acknowledgments can also be done hierarchically through the tree structure. Another advantage is that upon a miss, if a nearby node in the hierarchy has a cached copy of the block, then the block can be obtained from that nearby node (cache-to-cache sharing) rather than having to go to the home, which may be much further away in the network topology. This can reduce transit latency as well as contention at the home. Of course, this second advantage depends on how well locality in the hierarchy maps to locality in the underlying physical network as well as how well the sharing patterns of the application match the hierarchy.

While locality in the tree network can reduce transit delay on links, particularly for very large machines, the overall latency and bandwidth characteristics are usually not advantageous for hierarchical schemes. Consider hierarchical snooping schemes first. With buses, there is a bus transaction and snooping latency at every bus along the way. With rings, traversing rings at every level of the hierarchy further increases latency to potentially very high levels. For example, the uncontended latency to access a location on a remote ring in a fully populated Kendall Square Research KSR1 machine (Frank, Burkhardt, and Rothnie 1993) was higher than 25 microseconds (Saavedra, Gaines, and Carlton 1993), so other architectural techniques (discussed in Chapter 9) were used to reduce ring remote capacity misses. The commercial systems that have used hierarchical snooping have tended to use quite shallow hierarchies (the largest KSR machine was a two-level ring hierarchy with up to 32 nodes per ring). The fact that there are several processors per node also implies that the bandwidth between a node and its parent or child must be large enough to sustain their combined demands. The processors within a node will compete not only for bus or link bandwidth but also for snoop bandwidth and for the occupancy, buffers, and request tracking mechanisms of the node-to-network interface. To alleviate link bandwidth limitations near the root of the hierarchy, multiple buses or rings can be used closer to the root; however, bandwidth scalability in practical hierarchical systems remains quite limited.

For hierarchical directories, the latency problem is that the number of network transactions sent up and down the hierarchy to satisfy a request tends to be larger than in a flat, memory-based scheme. Even though these transactions may be more localized in the network, each one is a full-fledged network transaction that also requires either looking up or modifying the directory at its (intermediate) destination node. This increased endpoint overhead at the nodes along the critical path

tends to far outweigh any reduction in the total number of network hops traversed and hence network delay, especially given the characteristics of modern networks. Although some pipelining can be used—for example, the data reply can be forwarded toward the requesting node while a directory node is being updated—in practice, the latencies can still become quite large compared to machines with no hierarchy (Hagersten 1992; Joe 1995). Hierarchies with large branching factors can alleviate the latency problem but they increase contention. As with hierarchical snooping, the root of the directory hierarchy can become a bandwidth bottleneck, for both link bandwidth and directory lookup bandwidth. Multiple links may be used closer to the root (particularly appropriate for physically hierarchical networks [Leiserson et al. 1996]), and the directory cache may be interleaved among them. Alternatively, since each block has a separate logical hierarchy, a multirooted directory hierarchy may be embedded in a nonhierarchical, scalable point-to-point interconnect (Scott 1991; Wallach 1992; Scott and Goodman 1993). Figure 8.41 shows a possible organization. Like hierarchical directory schemes themselves, however, these techniques have only been in the realm of research so far.

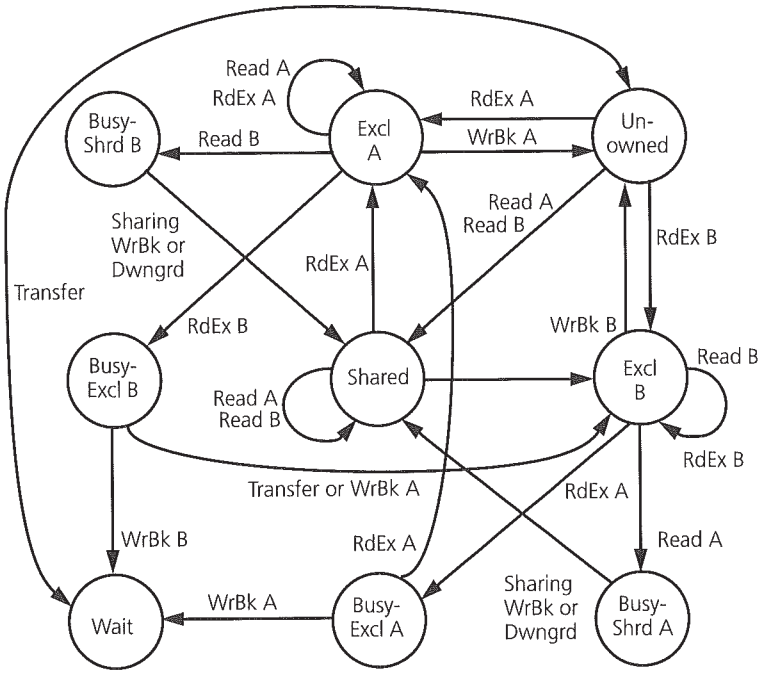
8.11 CONCLUDING REMARKS

Scalable systems that support a coherent shared address space are an increasingly important part of the multiprocessing landscape since they combine the ease of programming of a coherent shared address space programming model with the scaling advantages of a distributed memory and interconnect. Hardware support for cache coherence is becoming increasingly popular in commercial multiprocessors designed for both technical and commercial workloads. Most of these systems use directory-based protocols, whether memory based or cache based. They are found to perform well, at least at the moderate scales at which they have been built so far, and to afford significant ease of programming compared to explicit message passing for many applications.

Directory-based cache coherence protocols are quite complex, with many transient states and "corner cases" to deal with. Figure 8.42 conveys a sense of the complexity by showing the almost complete state transition diagrams of the Origin2000 and NUMA-Q protocols.

While supporting cache coherence in hardware has a significant design cost, it is alleviated by increased experience; the appearance of standards, and the fact that microprocessors themselves provide support for cache coherence. Once the microprocessor coherence protocol is available designers can develop the multiprocessor protocol and communication architecture even before the microprocessor is ready so that not so much of a lag occurs between the two. Commercial multiprocessors today typically use the latest microprocessors available at the time they ship, alleviating the fear that multiprogrammers would have to play catch-up with the processor technology curve.

Some interesting open questions for hardware-coherent shared address space systems include whether their performance on real applications will indeed scale to



(a) Origin2000

FIGURE 8.42 Expanded directory state diagrams for the case study multiprocessors of this chapter. The state diagram for the SGI Origin2000 in (a) is quite simplified: it shows the busy states at the directory but leaves out I/O operations, the poisoned state, and several race conditions. To show the use of busy states, accesses from two nodes A and B are shown. For example, a state labeled "Excl A" means that the directory thinks the block is in exclusive state in node A, and an arc labeled "RdEx B" indicates a read-exclusive operation from node B. The transfer operation and the wait state are used to handle write backs, as described in the text. The state diagram for the Sequent NUMA-Q in (b) is much more complete, though it also excludes a few corner cases. The arcs are not labeled in this diagram and several of the state labels are not explained; the purpose of this diagram is not to convey the complete protocol but simply to show that full-blown state transition diagrams can become quite complex in real systems.

large processor counts (and whether significant changes to current protocols will be needed for this), whether the appropriate node for a scalable system will be a small-scale multiprocessor or a uniprocessor, the extent to which commodity communication architectures will be successful in supporting this abstraction efficiently, and the success with which a communication assist can be designed that supports the most appropriate mechanisms for both cache coherence and explicit message passing. Some critical hardware/software trade-offs for coherent shared address space systems are discussed in the next chapter.

(b) NUMA-Q

FIGURE 8.42 Expanded directory state diagrams for the case study multiprocessors of this chapter

8.12 EXERCISES

8.1 What are the inefficiencies and efficiencies in emulating message passing on a cache-coherent machine compared to the kinds of machines discussed in Chapter 7?

8.2 a. For which of the case study parallel applications used in this book do you expect a substantial advantage in using multiprocessor rather than uniprocessor nodes (assuming the same total number of processors)? For which do you think there might be disadvantages, and under what circumstances?

b. How might your answer to the previous question differ with increasing scale of the machine? That is, how do you expect the performance benefits of using fixed-size multiprocessor nodes to change as the machine size is increased to hundreds of processors?

c. Are there any special benefits that the Illinois MESI coherence scheme offers for organizations with multiprocessor nodes?

8.3 Given a 512-processor system in which each node visible to the directory has 8 processors and 1 GB of main memory and a cache block size of 64 bytes, what is the directory memory overhead for (a) a full bit vector scheme, and (b) Dir_iB with $i = 3$?

8.4 The chapter provided diagrams showing the network transactions for strict request-response, intervention forwarding, and reply forwarding for read operations in a flat, memory-based protocol like that of the SGI Origin (see Figure 8.12). Do the same for write operations.

8.5 The Origin protocol assumed that acknowledgments for invalidations are gathered at the requestor. An alternative is to have the acknowledgments sent back to the home (from where the invalidation requests come) and have the home send a single acknowledgment back to the requestor. This solution is used in the Stanford FLASH multiprocessor. What are the main performance and complexity trade-offs between these two choices?

8.6 Draw the network transaction diagrams (like those in Figure 8.16) for an uncached read-shared request, an uncached read-exclusive request, and a write-invalidate request in the Origin protocol. State one example of a use of each.

8.7 Instead of the doubly linked list used in the SCI protocol, it is possible to use a singly linked list. What is the advantage? Describe what modifications would need to be made to the following operations if a singly linked list were used:

a. Replacement of a cache block that is in a sharing list.

b. Write to a cache block that is in a sharing list.

Qualitatively discuss the effects this might have on large-scale multiprocessor performance.

8.8 How might you reduce the latency of writes that cause invalidations in the SCI protocol? Draw the network transactions. What are the major trade-offs?

8.9 When a variable exhibits migratory sharing, a processor that reads the variable will be the next one to write it. What kinds of protocol optimizations could you use to reduce traffic and latency in this case, and how would you detect the situation dynamically? Describe a scheme or two in some detail.

8.10 Another pattern that might be detected dynamically is a producer-consumer pattern, in which one processor repeatedly writes (produces) a variable and another processor repeatedly reads (consumes) it. Is the standard MESI invalidation-based protocol well suited to this? Why or why not? What enhancements or protocol might be better, and what are the savings in latency or traffic? How would you dynamically detect and employ the changes?

8.11 Why is write atomicity more difficult to provide with update protocols than with invalidation-based protocols in directory-based systems? How would you solve the problem? Does the same difficulty exist in a bus-based system?

8.12 Consider the following program fragment running on a cache-coherent multiprocessor, assuming all values to be 0 initially.

P_1	P_2	P_3	P_4
A = 1	u = A	w = A	A = 2
	v = A	x = A	

There is only one shared variable (A). Suppose that a writer magically knows where the cached copies are and sends updates to them directly without consulting a directory node. Construct a situation in which write atomicity may be violated, assuming an update-based protocol.

a. Show the violation of sequential consistency that occurs in the results.

b. Can you produce a case where coherence is violated as well? How would you solve these problems?

c. Can you construct the same problems for an invalidation-based protocol?

d. Can you construct them for update protocols on a bus?

8.13 In handling write backs in the Origin protocol, we said that when the node doing the write back receives an intervention, it ignores it. Given a network that does not preserve point-to-point order, of what situations do we have to be careful in deciding to ignore the intervention? How do we detect that this intervention should be dropped? Would there be a problem with a network that preserved point-to-point order?

8.14 Can the serialization problems discussed for Origin in Section 8.5.2 arise even with a strict request-response protocol, and do the same guidelines apply? Show example situations, including the examples discussed in that section.

8.15 Consider the serialization of writes in NUMA-Q, given the two-level hierarchical coherence protocol. If a node has the block dirty in its remote cache, how might writes from other nodes that come to it get serialized with respect to writes from

processors in this node? What transactions would have to be generated to ensure the serialization?

8.16 In the Origin implementation, incoming request messages to the memory/directory interface are given priority over incoming responses unless there is a danger of responses being starved. Why do you think this choice of giving priorities to requests was made? Describe some methods for how you might detect when to invert the priority. What would be the danger with responses being starved?

8.17 a. Why is it necessary to flush TLBs when doing migration or replication of pages?

b. For a CC-NUMA multiprocessor with software-reloaded TLBs, suppose a page needs to be migrated. Which one of the following TLB flushing schemes would you pick and why: (i) only TLBs that currently have an entry for a page, (ii) only TLBs that have loaded an entry for a page since the last flush, or (iii) all TLBs in the system. [Hint: the selection should be based on the following two criteria: the cost of doing the actual TLB flush and the difficulty of tracking necessary information to implement the scheme.]

8.18 For a simple two-processor CC-NUMA system, the traces of cache misses for three virtual pages *X, Y, Z* from the two processors P_0 and P_1 are shown. Time goes from left to right. "R" is a read miss and "W" is a write miss. There are two memories M_0 and M_1, local to P_0 and P_1 respectively. A local miss costs 1 time unit and a remote miss costs 4 units. Assume that read misses and write misses cost the same.

Page *X*:
P_0: RRRR R R RRRRR RRR
P_1: R R R R R RRRR RR
Page *Y*:
P_0: no accesses
P_1: RR WW RRRR RWRWRW WWWR
Page *Z*:
P_0: R W RW R R RRWRWRWRW
P_1: WR RW RW W W R

a. In which local memories would you place pages *X, Y,* and *Z*, assuming complete knowledge of the entire trace?

b. Assume that all three pages were initially placed in M_0. You have prior knowledge of the entire trace. You can do one migration, or one replication, or nothing for each page at the beginning of the trace at zero cost. What action would be appropriate for each of the pages?

c. Answer part (b) where a page migration or replication costs 10 units. In addition, give the final memory access cost for each page.

d. Answer part (c) where a migration or replication costs 60 units.

e. Answer part (d) where the cache miss trace for each page is the shown trace repeated 10 times. (You still can only do one migration or replication at the beginning of the entire trace.)

8.19 Full-empty bits, introduced in Section 5.5, provide hardware support for fine-grained synchronization and have been proposed for CC-NUMA machines. What are the advantages and disadvantages of full-empty bits, and why do you think they are not used in modern systems?

8.20 With an invalidation-based protocol, lock transfers take more network transactions than necessary. An alternative to cached locks is to use uncached locks, where the lock variable stays in main memory and is always accessed at the memory itself.

a. Write pseudocode for a simple lock and a ticket lock using uncached operations.

b. What are the advantages and disadvantages relative to using cached locks? Which would you deploy in a production system?

c. Can you describe a scheme that uses both cached and uncached read and write operations to improve the performance of locks? What specific operations would your scheme require?

8.21 Since high-contention and low-contention situations are best served by different lock algorithms, one strategy that has been proposed is to have a library of synchronization algorithms and provide hardware support to switch between them "reactively" at run time based on observed access patterns to the synchronization variable.

a. Which locks would you provide in your library?

b. Assuming a memory-based directory protocol, design simple hardware support and a policy for switching between locks at run time.

c. Describe an example where this support might be particularly useful.

d. What are the potential disadvantages?

8.22 You are performing an architectural study using four applications: Ocean, blocked LU factorization, an FFT that performs local calculations on rows separated by a matrix transposition, and Barnes-Hut. For each application, answer the following questions, assuming a CC-NUMA system:

a. What modifications or enhancements in data structuring or layout would you use to ensure good interactions with the extended memory hierarchy?

b. What are the interactions with cache size and granularities of allocation, coherence, and communication that you would be particularly careful to represent or not represent?

8.23 Consider the example of transposing a matrix of data in parallel, as is used in computations such as high-performance FFTs. Figure 8.43 shows the transpose pictorially. Every process transposes one "patch" of its assigned rows to every other processor, including one to itself. Before the transpose, a process has read and written its assigned rows of the source matrix of the transpose, and after the transpose it reads and writes its assigned rows of the destination matrix. The rows assigned to a process in both the source and destination matrix are allocated in its local memory. There are two ways to perform the transpose: a process can read the local elements from its rows of the source matrix and write them to the appropriate elements of the

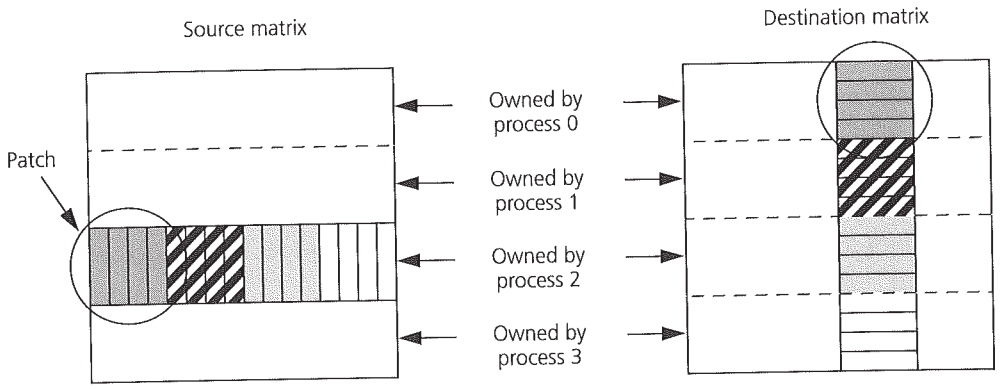


FIGURE 8.43 Sender-initiated matrix transposition. The source and destination matrices are partitioning among processes in groups of contiguous rows. Each process divides its set of n/p rows into p patches of size $(n/p)*(n/p)$. Consider process 2 as a representative example: one patch assigned to it ends up in the assigned set of rows of every other process, and it transposes one patch (third from left, in this case) locally.

destination matrix, whether they are local or remote, as shown in the figure (called a sender-initiated transpose); or a process can write the local rows of the destination matrix and read the appropriate elements of the source matrix, whether they are local or remote (called a receiver-initiated transpose).

a. Given an invalidation-based directory protocol, which method do you think will perform better and why?

b. How do you expect the answer to (a) to change if you assume an update-based directory protocol?

c. Consider the following implementation of a matrix transpose, which you plan to run on eight processors. Each processor has one level of cache, which is fully associative, 8 KB, with 128 byte lines. (*Note:* `AT` and `A` are not the same matrix.)

```
Transpose(double **A, double **AT)
{
   int i,j,mynum;

   GETPID(mynum);

   for (i=mynum*nrows/p; i<((mynum+1)*(nrows/p)); i++) {
      for (j=0; j<1024; j++) {
         AT[i][j] = A[j][i];
      }
   }
}
```

The input data set is a 1,024 × 1,024 matrix of double-precision floating-point numbers (i.e., `nrows` in 1,024), decomposed so that each processor is responsible for generating a contiguous block of rows in the transposed matrix `AT` (i.e., a receiver-initiated transpose). Ignoring the contention problem caused by all processors first going to processor 0, what is the major performance problem with this code? What technique would you use to solve it? Restructure the code to alleviate all performance problems as much as possible. Write the entire restructured loop.

8.24 Consider a hierarchical bus-based system with a centralized memory at the root of the hierarchy rather than distributed memory as discussed in the chapter. What would be the main differences in how reads and writes are satisfied? Briefly describe the path taken by reads and writes.

8.25 Could you construct a hierarchical bus-based system with centralized memory (say) without pursuing the inclusion property between the remote access cache and the L_1 caches in a node? If so, what complications would it cause?

8.26 To ensure sequential consistency in a two-level hierarchical bus design, is it okay to return an acknowledgment when the invalidation request reaches the B_2 bus? If so, what constraints are imposed on the design and implementation of the caches and the orders preserved among transactions? If not, why not? Would it be okay if the hierarchy had more than two levels?

8.27 Suppose two processors in two different nodes of a hierarchical bus-based machine issue an upgrade for a block at the same time. Trace their paths through the system, discussing all state changes and when they must happen as well as what precautions prevent deadlock and prevent both processors from gaining ownership.

8.28 An optimization in distributed-memory bus-based hierarchies is cache-to-cache sharing: if another processor's cache on the local bus can supply the data, we do not have to go to the global bus and remote node. What are the trade-offs of supporting this optimization in ring-based hierarchies?

8.29 What branching factor would you choose in a machine with a hierarchical directory? Highlight the major trade-offs. What techniques might you use to alleviate the performance trade-offs? Be as specific in your description as possible.

8.30 Is it possible to implement hierarchical directories without maintaining inclusion in the directory caches? Design a protocol that does that and discuss the advantages and disadvantages.