CHAPTER 6 Snoop-based Multiprocessor Design

6.1 Introduction

The large differences we see in the performance, cost, and scale of symmetric multiprocessors on the market rest not so much on the choice of the cache coherence protocol, but rather on the design and implementation of the oganizational structure that supports the logical operation of the protocol. The performance impact of protocol trade-offs are well understood and most machines use a variant of the protocols described in the previous chapter. However, the latency and bandwidth that is achieved on a protocol depend on the bus design, the cache design, and the integration with memory, as does the engineering cost of the system. This chapter studies the more detailed physical design issues in snoop-based cache coherent symmetric multiprocessors.

The abstract state transition diagrams for coherence protocols are conceptually simple, however, subtle correctness and performance issues arise at the hardware level. The correctness issues arise mainly because actions that are considered atomic at the abstract level are not necessarily atomic

at the organizational level. The performance issues arise mainly because we want to want to pipeline memory operations and allow many operations to be outstanding, using different components of the memory hierarchy, rather than waiting for each operation to complete before starting the next one. These performance enhancements present further correctness challenges. Modern communication assist design for aggressive cache-coherent multiprocessors presents a set of challenges that are similar in complexity and in form to those of modern processor design, which allows a large number of outstanding instructions and out of order execution.

We need to peel off another layer of the design of snoop-based multiprocessors to understand the practical requirements embodied by state transition diagrams. We must contend with at least three issues. First, the implementation must be correct; second, it should offer high performance, and third, it should require minimal extra hardware. The issues are related. For example, providing high performance often requires that multiple low-level events be outstanding (in progress) at a time, so that their latencies can be overlapped. Unfortunately, it is exactly in these situations—due to the numerous complex interactions between these events—that correctness is likely to be compromised. The product shipping dates for several commercial systems, even for microprocessors that have on-chip coherence controllers, have been delayed significantly because of subtle bugs in the coherence hardware.

Our study begins by enumerating the basic correctness requirements on a cache coherent memory system. A base design, using single level caches and a one transaction at a time atomic bus, is developed in *** and the critical events in processing individual transactions are outlined. We assumes an invalidation protocol for concreteness, but the main issues apply directly to update protocols.*** expands this design to address multilevel cache hierarchies, showing how protocol events propagate up and down the hierarchy. *** expands the base design to utilize a split transaction bus, which allows multiple transactions to be performed in a pipelined fashion, and then brings together multilevel caches and split-transactions. From this design point, it is a small step to support multiple outstanding misses from each processor, since all transactions are heavily pipelined and many take place concurrently. The fundamental underlying challenge throughout is maintaining the illusion of sequential order, as required by the sequential consistency model. Having understood the key design issues in general terms, we are ready to study concrete designs in some detail. *** presents to case studies, the SGI Challenge and the Sun Enterprise, and ties the machine performance back to software trade-offs through a workload driven study of our sample applications. Finally, Section *** examines a number of advanced topics which extend the design techniques in functionality and scale.

6.2 Correctness Requirements

A cache-coherent memory system must, of course, satisfy the requirements of coherence and preserve the semantics dictated by the memory consistency model. In particular, for coherence it should ensure that stale copies are found and invalidated/updated on writes and it should provide write serialization. If sequential consistency is to be preserved by satisfying the sufficient conditions, the design should provide write atomicity and the ability to detect the completion of writes. In addition, it should have the desirable properties of any protocol implementation, including cache coherence, which means it should be free of deadlock and livelock, and should either eliminate starvation or make it very unlikely. Finally, it should cope with error conditions beyond its control (e.g., parity errors), and try to recover from them where possible.

Deadlock occurs when operations are still outstanding but all system activity has ceased. In general, deadlock can potentially arise where multiple concurrent entities incrementally obtain shared resources and hold them in a non-preemptible fashion. The potential for deadlock arises when there is a cycle of resource dependences. A simple analogy is in traffic at an intersection, as shown in Figure 6-1. In the traffic example, the entities are cars and the resources are lanes. Each car needs to acquire two lane resources to proceed through the intersection, but each is holding one.

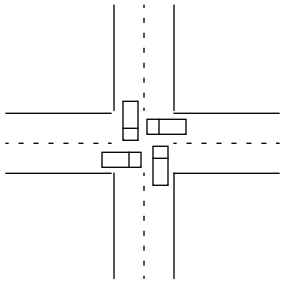

Figure 6-1 Deadlock at a traffic intersection.

Four cars arrive at an intersection and all proceed one lane each into the intersection. They block one another, since each is occupying a resource that another needs to make progress. Even if each yields to the car on the right, the intersection is deadlocked. To break the deadlock, some cars must retreat to allow others to make progress, so that then they themselves can make progress too.

In computer systems, the entities are typically controllers and the resources are buffers. For example, suppose two controllers A and B communicate with each other via buffers, as shown in Figure 6-2(a). A's input buffer is full, and it refuses all incoming requests until B accepts a request from it (thus freeing up buffer space in A to accept requests from other controllers), but B's input buffer is full too, and it refuses all incoming requests until A accepts a request from it. Neither controller can accept a request, so deadlock sets in. To illustrate the problem with more than two controllers, a three-controller example is shown in Figure 6-2(b). It is essential to either avoid such dependence cycles or break them when they occur.

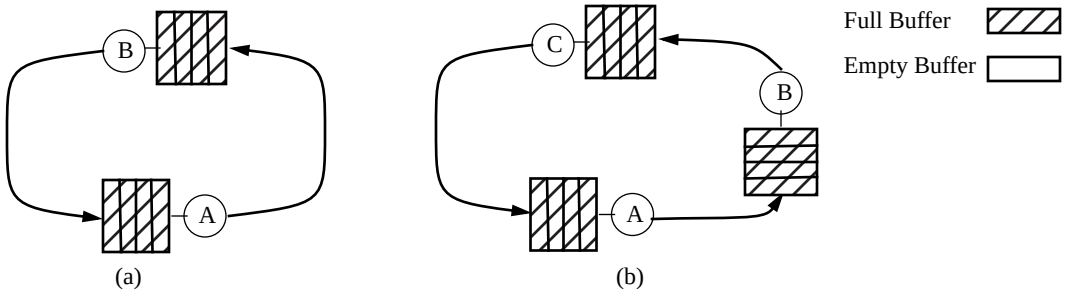

Figure 6-2 Deadlock can easily occur in a system if independent controllers with finite buffering need to communicate with each other.

If cycles are possible in the communication graph, then each controller can be stalled waiting for the one in front to free up resources. We illustrate cases for two and three controllers.

The system is in *livelock* when no processor is making forward progress in its computation even though transactions are being executed in the system. Continuing the traffic analogy, each of the

vehicles might elect to back up, clearing the intersection, and then try again to move forward. However, if they all repeatedly move back and forward at the same time, there will be a lot of activity but they will end up in the same situation repeatedly with no real progress. In computer systems, livelock typically arises when independent controllers compete for a common resource, with one snatching it away from the other before the other has finished with its use for the current operation.

Starvation does not stop overall progress, but is an extreme form of unfairness in which one or more processors make no progress while others continue to do so. In the traffic example, the livelock problem can be solved by a simple priority scheme. If a northbound car is given higher priority than an eastbound car, the latter must pull back and let the former through before trying to move forward again; similarly, a southbound car may have higher priority than a westbound car. Unfortunately, this does not solve starvation: In heavy traffic, an eastbound car may never pass the intersection since there may always be a new northbound car ready to go through. Northbound cars make progress, while eastbound cars are starved. The remedy here is to place an arbiter (e.g. a police officer or traffic light) to orchestrate the resource usage in a fair manner. The analogy extends easily to computer systems.

In general, the possibility of starvation is considered a less catastrophic problem than livelock or deadlock. Starvation does not cause the entire system to stop making progress, and is usually not a permanent state. That is, just because a processor has been starved for some time in the past it does not mean that it will be starved for all future time (at some point northbound traffic will ease up, and eastbound cars will get through). In fact, starvation is much less likely in computer systems than in this traffic example, since it is usually timing dependent and the necessary pathological timing conditions usually do not persist. Starvation often turns out to be quite easy to eliminate in bus-based systems, by having the bus arbitration be fair and using FIFO queues to other hardware resources, rather than rejecting requests and having them be retried. However, in scalable systems that we will see in later chapters, eliminating starvation entirely can add substantial complexity to the protocols and can slow down common-case transactions. Many systems therefore do not completely eliminate starvation, though almost all try to reduce the potential for it to occur.

In the discussions of how cache coherence protocols ensure write serialization and can satisfy the sufficient conditions for sequential consistency, the assumption was made that memory operations are atomic. Here this assumption is made somewhat more realistic: there is a single level of cache per processor and transactions on the bus are atomic. The cache can hold the processor while it performs the series of steps involved in a memory operation, so these operations appear atomic to the processor. The basic issues and tradeoffs that arise in implementing snooping and state transitions are discussed, along with new issues that arise in providing write serialization, detecting write completion, and preserving write atomicity. Then, more aggressive systems are considered, starting with multi-level cache hierarchies, going on to split-transaction (pipelined) buses, and then examining the combination of the two. In split-transaction buses, a bus transaction is split into request and response phases that arbitrate for the bus separately, so multiple transactions can be outstanding at a time on the bus. In all cases, writeback caches are assumed, at least for the caches closest to the bus so they can reduce bus traffic.

6.3 Base Design: Single-level Caches with an Atomic Bus

Several design decisions must be made even for the simple case of single-level caches and an atomic bus. First, how should we design the cache tags and controller, given that both the processor and the snooping agent from the bus side need access to the tags? Second, the results of a snoop from the cache controllers need to be presented as part of the bus transaction; how and when should this be done? Third, even though the memory bus is atomic, the overall set of actions needed to satisfy a processor's request uses other resources as well (such as cache controllers) and is not atomic, introducing possible race conditions. How should we design protocol state machines for the cache controllers given this lack of atomicity? Do these factors introduce new issues with regard to write serialization, write completion detection or write atomicity? And what deadlock, livelock and starvation issues arise? Finally, writebacks from the caches can introduce interesting race conditions as well. The next few subsections address these issues one by one.

6.3.1 Cache controller and tags

Consider first a conventional uniprocessor cache. It consists of a storage array containing data blocks, tags, and state bits, as well as a comparator, a controller, and a bus interface. When the processor performs an operation against the cache, a portion of the address is used to access a cache set that potentially contains the block. The tag is compared against the remaining address bits to determine if the addressed block is indeed present. Then the appropriate operation is performed on the data and the state bits are updated. For example, a write hit to a clean cache block causes a word to be updated and the state to be set to modified. The cache controller sequences the reads and writes of the cache storage array. If the operation requires that a block be transferred from the cache to memory or vice versa, the cache controller initiates a bus operation. The bus operation involves a sequence of steps from the bus interface; these are typically (1) assert request for bus, (2) wait for bus grant, (3) drive address and command, (4) wait for command to be accepted by the relevant device, and (5) transfer data. The sequence of actions taken by the cache controller is itself implemented as a finite state machine, as is the sequencing of steps in a bus transaction. It is important not to confuse these state machines with the transition diagram of the protocol followed by each cache block.

To support a snoopy coherence protocol, the basic uniprocessor cache controller design must be enhanced. First, since the cache controller must monitor bus operations as well as respond to processor operations, it is simplest to view the cache as having two controllers, a bus-side controller and a processor-side controller, each monitoring one kind of external event. In either case, when an operation occurs the controller must access the cache tags. On every bus transaction the bus-side controller must capture the address from the bus and use it to perform a tag check. If the check fails (a snoop miss), no action need be taken; the bus operation is irrelevant to this cache. If the snoop "hits," the controller may have to intervene in the bus transaction according to the cache coherence protocol. This may involve a read-modify-write operation on the state bits and/or trying to obtain the bus to place a block on it.

With only a single array of tags, it is difficult to allow two controllers to access the array at the same time. During a bus transaction, the processor will be locked out from accessing the cache, which will degrade processor performance. If the processor is given priority, effective bus bandwidth will decrease because the snoop controller will have to delay the bus transaction until it

gains access to the tags. To alleviate this problem, many coherent cache designs utilize a *dual-ported* tag and state store or duplicate the tag and state for every block. The data portion of the cache is not duplicated. If tags are duplicated, the contents of the two sets of tags are exactly the same, except one set is used by the processor-side controller for its lookups and the other is used by the bus-side controller for its snoops (see Figure 6-3). The two controllers can read the tags and perform checks simultaneously. Of course, when the tag for a block is updated (e.g. when the state changes on a write or a new block is brought into the cache) both copies must ultimately be modified, so one of the controllers may have to be locked out for a time. For example, if the bus-side tags are updated by a bus transaction, at some point the processor-side tags will have to be updated as well. Machine designs can play several tricks to reduce the time for which a controller is locked out, for example in the above case by having the processor-side tags be updated only when the cache data are later modified, rather than immediately when the bus-side tags are updated. The frequency of tag updates is also much smaller than read accesses, so the coherence snoops are expected to have little impact on processor cache access.

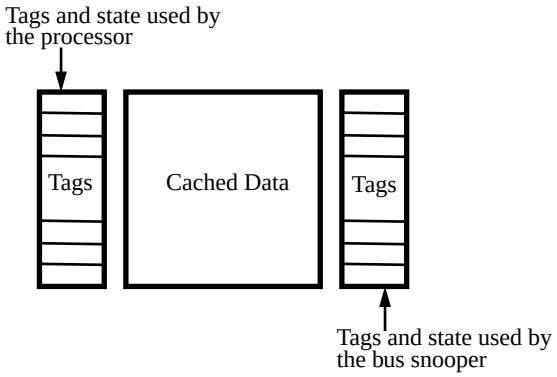

Figure 6-3 Organization of single-level snoopy caches.

For single-level caches there is a duplicate set of tags and state provided to reduce contention. One set is used exclusively by the processor, while another is used by the bus-snooper. Any changes to cache content or state, however, involves updating both sets of tags simultaneously.

Another major enhancement from a uniprocessor cache controller is that the controller now acts not only as an initiator of bus transactions, but also as a responder to them. A conventional responding device, such as the controller for a memory bank, monitors the bus for transactions on the fixed subset of addresses that it contains, and responds to relevant read or write operations possibly after some number of "wait" cycles. It may even have to place data on the bus. The cache controller is not responsible for a fixed subset of addresses, but must monitor the bus and perform a tag check on every transaction to determine if the transaction is relevant. For an update-based protocol, many caches may need to snoop the new data off the bus as well. Most modern microprocessors implement such enhanced cache controllers so that they are "multiprocessor-ready".

6.3.2 Reporting snoop results

Snooping introduces a new element to the bus transaction as well. In a conventional bus transaction on a uniprocessor system, one device (the initiator) places an address on the bus, all other devices monitor the address, and one device (the responder) recognizes it as being relevant. Then data is transferred between the two devices. The responder acknowledges its role by raising a

wired-OR signal; if no device decides to respond within a time-out window, a bus error occurs. For snooping caches, each cache must check the address against its tags and the collective result of the snoop from *all* caches must be reported on the bus before the transaction can proceed. In particular, the snoop result informs main memory whether it should respond to the request or some cache is holding a modified copy of the block so an alternative action is necessary. The questions are when the snoop result is reported on the bus, and how.

Let us focus first on the "when" question. Obviously, it is desirable to keep this delay as small as possible so main memory can decide quickly what to do.[1] There are three major options:

1. The design could guarantee that the snoop results are available within a *fixed* number of clock cycles from the issue of the address on the bus. This, in general, requires the use of a dual set of tags because otherwise the processor, which usually has priority, could be accessing the tags heavily when the bus transaction appears. Even with a dual set of tags, one may need to be conservative about the fixed snoop latency, because both sets of tags are made inaccessible when the processor updates the tags; for example in the E --> M state transition in the Illinois MESI protocol.[2] The advantages of this option are a simple memory system design, and disadvantages are extra hardware and potentially longer snoop latency. The Pentium Pro Quads use this approach—with the ability to extend the snoop phase when necessary (see Chapter 8) as do the HP Corporate Business Servers [CAH+93] and the Sun Enterprise.
2. The design could alternatively support a *variable* delay snoop. The main memory assumes that one of the caches will supply the data, until all the cache controllers have snooped and indicated otherwise. This option may be easier to implement since cache controllers do not have to worry about tag-access conflicts inhibiting a timely lookup, and it can offer higher performance since the designer does not have to conservatively assume the worst-case delay for snoop results. The SGI Challenge multiprocessors use a slight variant of this approach, where the memory subsystem goes ahead and fetches the data to service the request, but then stalls if the snoops have not completed by then [GaW93].
3. A third alternative is for the main memory subsystem to maintain a bit per block that indicates whether this block is modified in one of the caches or not. This way the memory subsystem does not have to rely on snooping to decide what action to take, and when controllers return their snoop results is a performance issue. The disadvantage is the extra complexity added to the main-memory subsystem.

How should snoop results be reported on the bus? For the MESI scheme, the requesting cache controller needs to know whether the requested memory block is in other processors' caches, so it can decide whether to load the block in exclusive (E) or shared (S) state. Also, the memory system needs to know whether any cache has the block in modified state, so that memory need not

1. Note, that on an atomic bus there are ways to make the system less sensitive to the snoop delay. Since only one memory transaction can be outstanding at any given time, the main memory can start fetching the memory block regardless of whether it or the cache would eventually supply the data; the main-memory subsystem would have sit idle otherwise. Reducing this delay, however, is very important for a split transaction-bus, discussed later. There, multiple bus transactions can be outstanding, so the memory subsystem can be used in the meantime to service another request, for which *it* (and not the cache) may have to supply the data.

2. It is interesting that in the base 3-state invalidation protocol we described, a cache-block state is never updated unless a corresponding bus-transaction is also involved. This usually gives plenty of time to update the tags.

respond. One reasonable option is to use three wired-or signals, two for the snoop results and one indicating that the snoop result is valid. The first signal is asserted when any of the processors' caches (excluding the requesting processor) has a copy of the block. The second is asserted if any processor has the block modified in its cache. We don't need to know the identity of that processor, since it itself knows what action to take. The third signal is an inhibit signal, asserted until all processors have completed their snoop; when it is de-asserted, the requestor and memory can safely examine the other two signals. The full Illinois MESI protocol is more complex because a block can be preferentially retrieved from another cache rather than from memory even if it is in shared state. If multiple caches have a copy, a priority mechanism is needed to decide which cache will supply the data. This is one reason why most commercial machines that use the MESI protocol limit cache-to-cache transfers. The Silicon Graphics Challenge and the Sun Enterprise Server use cache-to-cache transfers only for data that are potentially in modified state in the cache (i.e. are either in exclusive or modified state), in which case there is a single supplier. The Challenge updates memory in the process of a cache to cache transfer, so it does not need a shared-modified state, while the Enterprise does not update memory and uses a shared modified state.

6.3.3 Dealing with Writebacks

Writebacks also complicate implementation, since they involve an incoming block as well as an outgoing one. In general, to allow the processor to continue as soon as possible on a cache miss that causes a writeback (replacement of modified data in the cache), we would like to delay the writeback and instead first service the miss that caused it. This optimization imposes two requirements. First, it requires the machine to provide additional storage, a *writeback buffer*, where the block being replaced can be temporarily stored while the new block is brought into the cache and before the bus can be re-acquired for a second transaction to complete the writeback. Second, before the writeback is completed, it is possible that we see a bus transaction containing the address of that block. In that case, the controller must supply the data from the writeback buffer and cancel its earlier pending request to the bus for a writeback. This requires that an address-comparator be added to snoop on the writeback buffer as well. We will see in Chapter 6 that writebacks introduce further correctness subtleties in machines with physically distributed memory.

6.3.4 Base Organization

Figure 6-4 shows a block-diagram for our resulting base snooping architecture. Each processor has a single-level write-back cache. The cache is dual tagged, so the bus-side controller and the processor-side controller can do tag checks in parallel. The processor-side controller initiates a transaction by placing an address and command on the bus. On a writeback transaction, data is conveyed from the write-back buffer. On a read transaction, it is captured in the data buffer. The bus-side controller snoops the write-back tag as well as the cache tags. Bus arbitration places the requests that go on the bus in a total order. For each transaction, the command and address in the request phase drive the snoop operation—in this total order. The wired-OR snoop results serve as acknowledgment to the initiator that all caches have seen the request and taken relevant action. This defines the operation as being completed, as all subsequent reads and writes are treated as occurring "after" this transaction. It does not matter that the data transfer occurs a few cycles latter, because all the caches know that it is coming and can defer servicing additional processor requests until it arrives.

Using this simple design, let us examine more subtle correctness concerns that either require the state machines and protocols to be extended or require care in implementation: non-atomic state transitions, serialization, deadlock, and starvation.

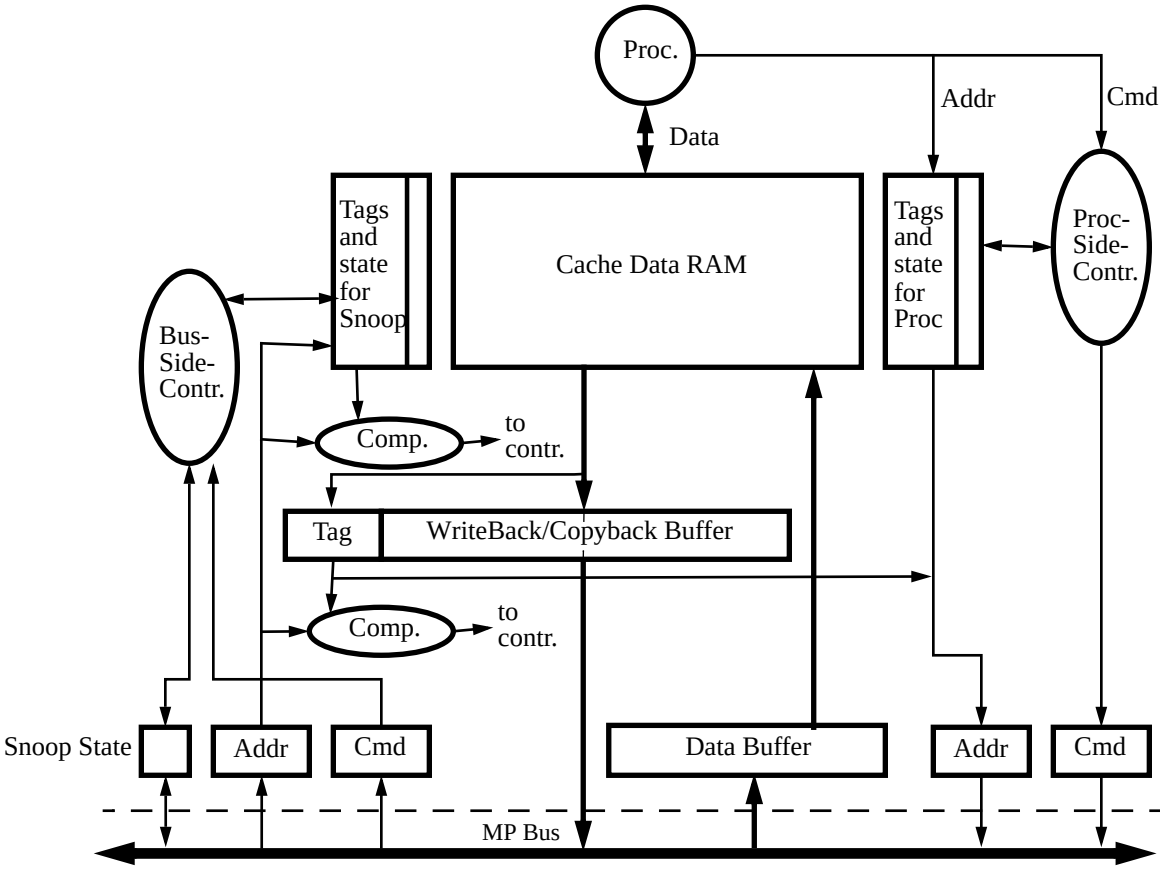

Figure 6-4 Design of a snooping cache for the base machine.

We assume each processor has a single-level writeback cache, an invalidation protocol is used, the processor can have only one memory request outstanding, and that the system bus is atomic. To keep the figure simple, we do not show the bus-arbitration logic and some of the low-level signals and buffers that are needed. We also do not show the coordination signals needed between the bus-side controller and the processor-side controller.

6.3.5 Non-atomic State Transitions

The state transitions and their associated actions in our diagrams have so far been assumed to happen instantaneously or at least atomically. In fact, a request issued by a processor takes some time to complete, often including a bus transaction. While the bus transaction itself is atomic in our simple system, it is only one among the set of actions needed to satisfy a processor's request. These actions include looking up the cache tags, arbitrating for the bus, the actions taken by other controllers at their caches, and the action taken by the issuing processor's controller at the end of the bus transaction (which may include actually writing data into the block). Taken as a whole, the set is not atomic. Even with an atomic bus, multiple requests from different processors may be outstanding in different parts of the system at a time, and it is possible that while a processor

(or controller) P has a request outstanding—for example waiting to obtain bus access—a request from another processor may appear on the bus and need some service from P, perhaps even for the same memory block as P's outstanding request. The types of complications that arise are best illustrated through an example.

Example 6-1 Suppose two processors P1 and P2 cache the same memory block A in shared state, and both simultaneously issue a write to block A. Show how P1 may have a request outstanding waiting for the bus while a transaction from P2 appears on the bus, and how you might solve the complication that results.

Answer Here is a possible scenario. P1's write will check its cache, determine that it needs to elevate the block's state from shared to modified before it can actually write new data into the block, and issue an upgrade bus request. In the meantime, P2 has also issued a similar upgrade or read-exclusive transaction for A, and it may have won arbitration for the bus first. P1's snoop will see the bus transaction, and must downgrade the state of block A from shared to invalid in its cache. Otherwise when P2's transaction is over A will be in modified state in P2's cache and in shared state in P1's cache, which violates the protocol. To solve this problem, a controller must be able to check addresses snooped from the bus against its own outstanding request, and modify the latter if necessary. Thus, when P1 observes the transaction from P2 and downgrades its block state to invalid, the upgrade bus request it has outstanding is no longer appropriate and must be replaced with a read-exclusive request. (If there were no upgrade transactions in the protocol and read-exclusives were used even on writes to blocks in shared state, the request would not have to be changed in this case even though the block state would have to be changed. These implementation requirements should be considered when assessing the complexity of protocol optimizations.)

A convenient way to deal with the "non-atomic" nature of state transitions, and the consequent need to sometimes revise requests based on observed events, is to expand the protocol state diagram with intermediate or *transient* states. For example, a separate state can be used to indicate that an upgrade request is outstanding. Figure 6-5 shows an expanded state diagram for a MESI protocol. In response to a processor write operation the cache controller begins arbitration for the bus by asserting a bus request, and transitions to the S->M intermediate state. The transition out of this state occurs when the bus arbiter asserts BusGrant for this device. At this point the BusUpgr transaction is placed on the bus and the cache block state is updated. However, if a BusRdX or BusUpgr is observed for this block while in the S->M state, the controller treats its block as having been invalidated before the transaction. (We could instead retract the bus request and transition to the I state, whereupon the still pending PrWr would be detected again.) On a processor read from invalid state, the controller advances to an intermediate state (I-->S,E); the next stable state to transition to is determined by the value of the shared line when the read is granted the bus. These intermediate states are typically not encoded in the cache block state bits, which are still MESI, since it would be wasteful to expend bits in every cache slot to indicate the one block in the cache that may be in a transient state. They are reflected in the combination of state bits and controller state. However, when we consider caches that allow multiple outstanding transactions, it will be necessary to have an explicit representation for the (multiple) blocks from a cache that may be in a transient state.

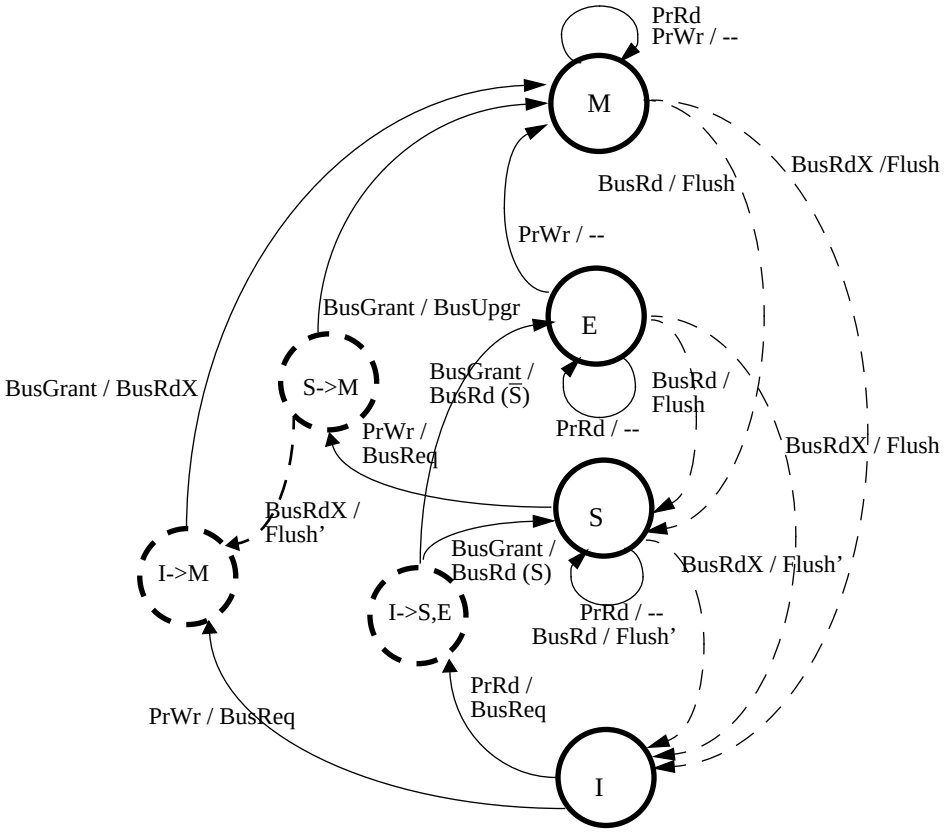

Figure 6-5 Expanded MESI protocol state diagram indicating transient states for bus acquisition

Expanding the number of states in the protocol greatly increases the difficulty of proving that an implementation is correct, or of testing the design. Thus, designers seek mechanisms that avoid transient states. The Sun Enterprise, for example, does not use a BusUpgr transaction in the MESI protocol, but uses the result of the snoop to eliminate unnecessary data transfers in the BusRdX. Recall, that on a BusRdX the caches holding the block invalidate their copy. If a cache has the block in the dirty state, it raises the dirty line, thereby preventing the memory from supplying the data, and flushes the data onto the bus. No use is made of the shared line. The trick is to have the processor that issues the BusRdX to snoop its own tags when the transaction actually goes on the bus. If the block is still in its cache, it raises the shared line, which waves off the memory. Since it already has the block, the data phase of the transaction is skipped. The cache controller does not need a transient state, because regardless of what happens it has one action to take—place a BusRdX transaction on the bus.

6.3.6 Serialization

With the non-atomicity of an entire memory operation, care must be taken in the processor-cache handshake to preserve the order determined by the serialization of bus transactions. For a read, the processor needs the result of the operation. To gain greater performance on writes, it is tempt-

ing to update the cache block and allow the processor continue with useful instructions while the cache controller acquires exclusive ownership of the block—and possibly loads the rest of the block—via a bus transaction. The problem is that there is a window between the time the processor gives the write to the cache and when the cache controller acquires the bus for the read-exclusive (or upgrade) transaction. Other bus transactions may occur in this window, which may change the state of this or other blocks in the cache. To provide write serialization and SC, these transactions must appear to the processor as occurring before the write, since that is how they are serialized by the bus and appear to other processors. Conservatively, the cache controller should not allow the processor issuing the write to proceed past the write until the read-exclusive transaction occurs on the bus and makes the write visible to other processors. In particular, even for write serialization the issuing processor should not consider the write complete until it has appeared on the bus and thus been serialized with respect to other writes.

In fact, the cache does not have to wait until the read-exclusive transaction is finished—i.e. other copies have actually been invalidated in their caches—before allowing the processor to continue; it can even service read and write hits once the transaction is on the bus, assuming access to the block in transit is handled properly. The crux of the argument for coherence and for sequential consistency presented in Section 5.4 was that all cache controllers observe the exclusive ownership transactions (generated by write operations) in the same order and that the write occurs immediately after the exclusive ownership transaction. Once the bus transaction starts, the writer knows that all other caches will invalidate their copies before another bus transaction occurs. The position of the write in the serial bus order is *committed*. The writer never knows where exactly the invalidation is inserted in the local program order of the other processors; it knows only that it is before whatever operation generates the next bus transaction and that all processors insert the invalidations in the same order. Similarly, the writer's local sequence of memory operations only become visible at the next bus transaction. This is what is important to maintain the necessary orderings for coherence and SC, and it allows the writer to *substitute commitment for actual completion* in following the sufficient conditions for SC. In fact, it is the basic observation that makes it possible to implement cache coherency and sequential consistency with pipelined busses, multilevel memory hierarchies, and write buffers. Write atomicity follows the same argument as presented before in Section 5.4.

This discussion of serialization raises an important, but somewhat subtle point. Write serialization and write atomicity have very little to do with when the write-backs occur or with when the actual location in memory is updated. Either a write or a read can cause a write-back, if it causes a dirty block to be replaced. The write-backs are bus transactions, but they do not need to be ordered. On the other hand, a write does not necessarily cause a write-back, even if it misses; it causes a read-exclusive. What is important to the program is when the new value is bound to the address. The write completes, in the sense that any subsequent read will return the new or later value once the BusRdX or BusUpgr transaction takes place. By invalidating the old cache blocks, it ensures that all reads of the old value precede the transaction. The controller issuing the transaction ensures that the write occurs after the bus transaction, and that no other memory operations intervene.

6.3.7 Deadlock

A two-phase protocol, such as the request-response protocol of a memory operation, presents a form of protocol-level deadlock, sometimes called *fetch-deadlock*[Lei*92], that is not simply a question of buffer usage. While an entity is attempting to issue its request, it needs to service

incoming transactions. The place where this arises in an atomic bus based SMP, is that while the cache controller is awaiting the bus grant, it needs to continue performing snoops and handling requests which may cause it to flush blocks onto the bus. Otherwise, the system may deadlock if two controllers have outstanding transactions that need to be responded to by each other, and both are refusing to handle requests. For example, suppose a BusRd for a block B appears on the bus while a processor P1 has a read-exclusive request outstanding to another block A and is waiting for the bus. If P1 has a modified copy of B, its controller should be able to supply the data and change the state from modified to shared while it is waiting to acquire the bus. Otherwise the current bus transaction is waiting for P1's controller, while P1's controller is waiting for the bus transaction to release the bus.

6.3.8 Livelock and Starvation

The classic potential livelock problem in an invalidation-based cache-coherent memory system is caused by all processors attempting to write to the same memory location. Suppose that initially no processor has a copy of the location in its cache. A processor's write requires the following non-atomic set of events: Its cache obtains exclusive ownership for the corresponding memory block—i.e. invalidates other copies and obtains the block in modified state—a state machine in the processor realizes that the block is now present in the cache in the appropriate state, and the state-machine re-attempts the write. Unless the processor-cache handshake is designed carefully, it is possible that the block is brought into the cache in modified state, but before the processor is able to complete its write the block is invalidated by a BusRdX request from another processor. The processor misses again and this cycle can repeat indefinitely. To avoid livelock, a write that has obtained exclusive ownership must be allowed to complete before the exclusive ownership is taken away.

With multiple processors competing for a bus, it is possible that some processors may be granted the bus repeatedly while others may not and may become starved. Starvation can be avoided by using first-come-first-serve service policies at the bus and elsewhere. These usually require additional buffering, so sometimes heuristic techniques are used to reduce the likelihood of starvation instead. For example, a count can be maintained of the number of times that a request has been denied and, after a certain threshold, action is taken so that no other new request is serviced until this request is serviced.

6.3.9 Implementing Atomic Primitives

The atomic operations that the above locks need, test&set and fetch&increment, can be implemented either as individual atomic read-modify-write instructions or using an LL-SC pair. Let us discuss some implementation issues for these primitives, before we discuss all-hardware implementations of locks. Let us first consider the implementation of atomic exchange or read-modify-write instructions first, and then LL-SC.

Consider a simple test&set instruction. It has a read component (the test) and a write component (the set). The first question is whether the test&set (lock) variable should be cacheable so the test&set can be performed in the processor cache, or it should be uncacheable so the atomic operation is performed at main memory. The discussion above has assumed cacheable lock variables. This has the advantage of allowing locality to be exploited and hence reducing latency and traffic when the lock is repeatedly acquired by the same processor: The lock variable remains dirty in the cache and no invalidations or misses are generated. It also allows processors to spin in their

caches, thus reducing useless bus traffic when the lock is not ready. However, performing the operations at memory can cause faster transfer of a lock from one processor to another. With cacheable locks, the processor that is busy-waiting will first be invalidated, at which point it will try to access the lock from the other processor's cache or memory. With uncached locks the release goes only to memory, and by the time it gets there the next attempt by the waiting processor is likely to be on its way to memory already, so it will obtain the lock from memory with low latency. Overall, traffic and locality considerations tend to dominate, and lock variables are usually cacheable so processors can busy-wait without loading the bus.

A conceptually natural way to implement a cacheable test&set that is not satisfied in the cache itself is with two bus transactions: a read transaction for the test component and a write transaction for the set component. One strategy to keep this sequence atomic is to lock down the bus at the read transaction until the write completes, keeping other processors from putting accesses (especially to that variable) on the bus between the read and write components. While this can be done quite easily with an atomic bus, it is much more difficult with a split-transaction bus: Not only does locking down the bus impact performance substantially, but it can cause deadlock if one of the transactions cannot be immediately satisfied without giving up the bus.

Fortunately, there are better approaches. Consider an invalidation-based protocol with write-back caches. What a processor really needs to do is obtain exclusive ownership of the cache block (e.g. by issuing a single read-exclusive bus transaction), and then it can perform the read component and the write component in the cache as long as it does not give up exclusive ownership of the block in between, i.e. it simply buffers and delays other incoming accesses from the bus to that block until the write component is performed. More complex atomic operations such as fetch-and-op must retain exclusive ownership until the operation is completed.

An atomic instruction that is more complex to implement is compare-and-swap. This requires specifying three operands in a memory instruction: the memory location, the register to compare with, and the value/register to be swapped in. RISC instruction sets are usually not equipped for this.

Implementing LL-SC requires a little special support. A typical implementation uses a so called lock-flag and a lock-address register at each processor. An LL operation reads the block, but also sets the lock-flag and puts the address of the block in the lock-address register. Incoming invalidation (or update) requests from the bus are matched against the lock-address register, and a successful match (a conflicting write) resets the lock flag. An SC checks the lock flag as the indicator for whether an intervening conflicting write has occurred; if so it fails, and if not it succeeds. The lock flag is also reset (and the SC will fail) if the lock variable is replaced from the cache, since then the processor may no longer see invalidations or updates to that variable. Finally, the lock-flag is reset at context switches, since a context switch between an LL and its SC may incorrectly cause the LL of the old process to lead to the success of an SC in the new process that is switched in.

Some subtle issues arise in avoiding livelock when implementing LL-SC. First, we should in fact not allow replacement of the cache block that holds the lock variable between the LL and the SC. Replacement would clear the lock flag, as discussed above, and could establish a situation where a processor keeps trying the SC but never succeeds due to continual replacement of the block between LL and SC. To disallow replacements due to conflicts with instruction fetches, we can use split instruction and data caches or set-associative unified caches. For conflicts with other data references, a common solution is to simply disallow memory instructions between an LL

and an SC. Techniques to hide latency (e.g. out-of-order issue) can complicate matters, since memory operations that are not between the LL and SC in the program code may be so in the execution. A simple solution to this problem is to not allow reorderings past LL or SC operations.

The second potential livelock situation would occur if two processes continually fail on their SCs, and each process's failing SC invalidated or updated the other process's block thus clearing the lock-flag. Neither of the two processes would ever succeed if this pathological situation persisted. This is why it is important that an SC not be treated as an ordinary write, but that it not issuing invalidations or updates when it fails.

Note that compared to implementing an atomic read-modify-write instruction, LL-SC can have a performance disadvantage since both the LL and the SC can miss in the cache, leading to two misses instead of one. For better performance, it may be desirable to obtain (or prefetch) the block in exclusive or modified state at the LL, so the SC does not miss unless it fails. However, this reintroduces the second livelock situation above: Other copies are invalidated to obtain exclusive ownership, so their SCs may fail without guarantee of this processor's SC succeeding. If this optimization is used, some form of backoff should be used between failed operations to minimize (though not completely eliminate) the probability of livelock.

6.4 Multi-level Cache Hierarchies

The simple design above was illustrative, but it made two simplifying assumptions that are not valid on most modern systems: It assumed single-level caches, and an atomic bus. This section relaxes the first assumption and examines the resulting design issues.

The trend in microprocessor design since the early 90's is to have an on-chip first-level cache and a much larger second-level cache, either on-chip or off-chip.[1] Some systems, like the DEC Alpha, use on-chip secondary caches as well and an off-chip tertiary cache. Multilevel cache hierarchies would seem to complicate coherence since changes made by the processor to the first-level cache may not be visible to the second-level cache controller responsible for bus operations, and bus transactions are not directly visible to the first level cache. However, the basic mechanisms for cache coherence extend naturally to multi-level cache hierarchies. Let us consider a two-level hierarchy for concreteness; the extension to the multi-level case is straightforward.

One obvious way to handle multi-level caches is to have independent bus snooping hardware for each level of the cache hierarchy. This is unattractive for several reasons. First, the first-level cache is usually on the processor chip, and an on-chip snooper will consume precious pins to monitor the addresses on the shared bus. Second, duplicating the tags to allow concurrent access by the snooper and the processor may consume too much precious on-chip real estate. Third, and more importantly, there is duplication of effort between the second-level and first-level snoops, since most of the time blocks present in the first-level cache are also present in the second-level cache so the snoop of the first-level cache is unnecessary.

1. The HP PA-RISC microprocessors are a notable exception, maintaining a large off-chip first level cache for many years after other vendors went to small on-chip first level caches.

The solution used in practice is based on this last observation. When using multi-level caches, designers ensure that they preserve the inclusion property. The *inclusion property* requires that:

1. If a memory block is in the first-level cache, then it must also be present in the second-level cache. In other words, the contents of the first-level cache must be a subset of the contents of the second-level cache.
2. If the block is in a modified state (e.g., modified or shared-modified) in the first-level cache, then it must also be marked modified in the second-level cache.

The first requirement ensures that all bus transactions that are relevant to the L1 cache are also relevant to L2 cache, so having the L2 cache controller snoop the bus is sufficient. The second ensures that if a BusRd transaction requests a block that is in modified state in the L1 or L2 cache, then the L2 snoop can immediately respond to the bus.

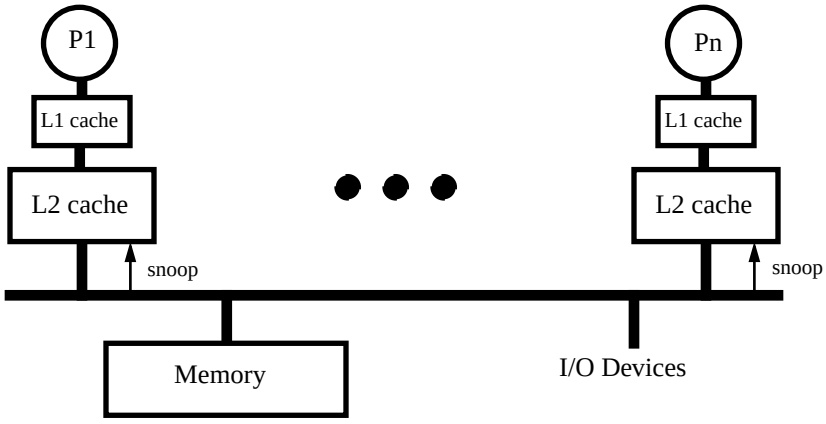

Figure 6-6 A bus-based machine containing processors with multi-level caches.

Coherence is maintained by ensuring the inclusion property, whereby all blocks in the L1 cache are also present in the L2 cache. With the inclusion property, a L2-snoop also suffices for the L1 cache.

6.4.1 Maintaining inclusion

The requirements for inclusion are not trivial to maintain. Three aspects need to be considered. Processor references to the first level cache cause it to change state and perform replacements; these need to be handled in a manner that maintains inclusion. Bus transactions cause the second level cache to change stage and flush blocks; these need to be forwarded to the first level. Finally, the modified state must be propagated out to the second level cache.

At first glance it might appear that inclusion would be satisfied automatically, since all first level cache misses go to the second level cache. The problem, however, is that two caches may choose different blocks to replace. Inclusion falls out automatically only for certain combinations of cache configuration. It is an interesting exercise to see what conditions can cause inclusion to be violated if no special care is taken, so let us do that a little here before we look at how inclusion is typically maintained. Consider some examples of typical cache hierarchies [BaW88]. For notational purposes, assume that the L1 cache has associativity a_1, number of sets n_1, block size b_1, and thus a total capacity of $S_1 = a_1.b_1.n_1$, where a_1, b_1 and n_1 are all powers of two. The corre-

sponding parameters for the L2 cache are a_2, n_2, b_2, and S_2. We also assume that all parameter values are powers of 2.

Set-associative L1 caches with history-based replacement. The problem with replacement polices based on the history of accesses to a block, such as least recently used (LRU) replacement, is that the L1 cache sees a different history of accesses than L2 and other caches, since all processor references look up the L1 cache but not all get to lower-level caches. Suppose the L1 cache is two-way set-associative with LRU replacement, both L1 and L2 caches have the same block size ($b_1 = b_2$), and L2 is k times larger than L1 ($n_2 = k.n_1$). It is easy to show that inclusion does *not* hold in this simple case. Consider three distinct memory blocks m_1, m_2, and m_3 that map to the same set in the L1 cache. Assume that m_1 and m_2 are currently in the two available slots within that set in the L1 cache, and are present in the L2 cache as well. Now consider what happens when the processor references m_3, which happens to collide with and replace one of m_1 and m_2 in the L2 cache as well. Since the L2 cache is oblivious of the L1 cache's access history which determines whether the latter replaces m_1 or m_2, it is easy to see that the L2 cache may replace one of m_1 and m_2 while the L1 cache may replace the other. This is true even if the L2 cache is two-way set associative and *m1* and *m2* fall into the same set in it as well. In fact, we can generalize the above example to see that inclusion can be violated if L1 is not direct-mapped and uses an LRU replacement policy, regardless of the associativity, block size, or cache size of the L2 cache.

Multiple caches at a level. A similar problem with replacements is observed when the first level caches are split between instructions and data, even if they are direct mapped, and are backed up by a unified second-level cache. Suppose first that the L2 cache is direct mapped as well. An instruction block m1 and a data block m2 that conflict in the L2 cache do not conflict in the L1 caches since they go into different caches. If m2 resides in the L2 cache and m1 is referenced, m2 will be replaced from the L2 cache but not from the L1 data cache, violating inclusion. Set associativity in the unified L2 cache doesn't solve the problem. For example, suppose the L1 caches are direct-mapped and the L2 cache is 2-way set associative. Consider three distinct memory blocks m_1, m_2, and m_3, where m_1 is an instruction block, m_2 and m_3 are data blocks that map to the same set in the L1 data cache, and all three map to a single set in the L2 cache. Assume that at a given time m_1 is in the L1 instruction cache, m_2 is in the L1 data cache, and they occupy the two slots within a set in the L2 cache. Now when the processor references m_3, it will replace m_2 in the L1 data cache, but the L2 cache may decide to replace m_1 or m_2 depending on the past history of accesses to it, thus violating inclusion. This can be generalized to show that if there are multiple independent caches that connect to even a highly associative cache below, inclusion is not guaranteed.

Different cache block sizes. Finally, consider caches with different block sizes. Consider a miniature system with a direct-mapped, unified L1 and L2 caches ($a_1 = a_2 = 1$), with block sizes 1-word and 2-words respectively ($b_1 = 1$, $b_2 = 2$), and number of sets 4 and 8 respectively ($n_1 = 4$, $n_2 = 8$). Thus, the size of L1 is 4 words, and locations 0, 4, 8,... map to set 0, locations 1, 5, 9,... map to set 1, and so on. The size of L2 is 16 words, and locations 0&1, 16&17, 32&33,... map to set-0, locations 2&3, 18&19, 34&35,... map to set-1, and so on. It is now easy to see that while the L1 cache can contain both locations 0 and 17 at the same time (they map to sets 0 and 1 respectively), the L2 cache can not do so because they map to the same set (set-0) and they are not consecutive locations (so block size of 2-words does not help). Hence inclusion can be violated. Inclusion can be shown to be violated even if the L2 cache is much larger or has greater associativity, and we have already seen the problems when the L1 cache has greater associativity.

In one of the most commonly encountered cases, inclusion *is* maintained automatically. This is when the L1 cache is direct-mapped ($a_1 = 1$), L2 can be direct-mapped or set associative ($a_2 >= 1$) with any replacement policy (e.g., LRU, FIFO, random) as long as the new block brought in is put in both L1 and L2 caches, the block-size is the same ($b_1 = b_2$), and the number of sets in the L1 cache is equal to or smaller than in L2 cache ($n_1 =< n_2$). Arranging this configuration is one popular way to get around the inclusion problem.

However, many of the cache configurations used in practice do not automatically maintain inclusion on replacements. Instead, the mechanism used for propagating coherency events is extended to explicitly maintain inclusion. Whenever a block in the L2 cache is replaced, the address of that block is sent to the L1 cache, asking it to invalidate or flush (if dirty) the corresponding blocks (there can be multiple blocks if $b_2 > b_1$).

Now consider bus transactions seen by the L2 cache. Some, but not all, of the bus transactions relevant to the L2 cache are also relevant to the L1 cache and must be propagated to it. For example, if a block is invalidated in the L2 cache due to an observed bus transaction (e.g., BusRdX), the invalidation must also be propagated to the L1 cache if the data are present in it.There are several ways to do this. One is to inform the L1 cache of all transactions that were relevant to the L2 cache, and to let it ignore the ones whose addresses do not match any of its tags. This sends a large number of unnecessary interventions to the L1 cache and can hurt performance by making cache-tags unavailable for processor accesses. A more attractive solution is for the L2 cache to keep extra state (inclusion bits) with cache blocks, which record whether the block is also present in the L1 cache. It can then suitably filter interventions to the L1 cache, at the cost of slightly extra hardware and complexity.

Finally, on a L1 write hit, the modification needs to be communicated to the L2 cache. One solution is to make the L1 cache write-through. This has the advantage that single-cycle writes are simple to implement [HP95]. However, writes can consume a substantial fraction of the L2 cache bandwidth, and a write-buffer is needed between the L1 and L2 caches to avoid processor stalls. The requirement can also be satisfied with writeback L1 caches, since it is not necessary that the data in the L2 cache be up-to-date but only that the L2 cache knows when the L1 cache has more recent data. Thus, the state of the L2 cache blocks is augmented so that blocks can be marked "*modified-but-stale*." The block behaves as modified for the coherency protocol, but data is fetched from the L1 cache on a flush. (One simple approach is to set both the modified and invalid bits.) Both the write-through and writeback solutions have been used in many bus-based multiprocessors. More information on maintaining cache inclusion can be found in [BaW88].

6.4.2 Propagating transactions for coherence in the hierarchy

Given that we have inclusion and we propagate invalidations and flushes up to the L1 cache as necessary, let us see how transactions percolate up and down a processor's cache hierarchy. The intra-hierarchy protocol handles processor requests by percolating them downwards (away from the processor) until either they encounter a cache which has the requested block in the proper state (shared or modified for a read request, and modified for a write/read-exclusive request) or they reach the bus. Responses to these processor requests are sent up the cache hierarchy, updating each cache as they progress towards the processor. Shared responses are loaded into each cache in the hierarchy in the shared state, while read-exclusive responses are loaded into all levels, except the innermost (L1) in the modified-but-stale state. In the innermost cache, read-exclusive data are loaded in the modified state, as this will be the most up-to-date copy.

External flush, copyback, and invalidate requests from the bus percolate upward from the external interface (the bus), modifying the state of the cache blocks as they progress (changing the state to invalid for flush and copyback requests, and to shared for copyback requests). Flush and copyback requests percolate upwards until they encounter the modified copy, at which point a response is generated for the external interface. For invalidations, it is not necessary for the bus transaction to be held up until all the copies are actually invalidated. The lowest-level cache controller (closest to the bus) sees the transaction when it appears on the bus, and this serves as a point of commitment to the requestor that the invalidation will be performed in the appropriate order. The response to the invalidation may be sent to the processor from its own bus interface as soon as the invalidation request is placed on the bus, so no responses are generated within the destination cache hierarchies. All that is required is that certain orders be maintained between the incoming invalidations and other transactions flowing through the cache hierarchy, which we shall discuss further in the context of split transaction busses that allow many transactions to be outstanding at a time.

Interestingly, dual tags are less critical when we have multi-level caches. The L2 cache acts as a filter for the L1 cache, screening out irrelevant transactions from the bus, so the tags of the L1 cache are available almost wholly to the processor. Similarly, since the L1 cache acts as a filter for the L2 cache from the processor side (hopefully satisfying most of processor's requests), the L2 tags are almost wholly available for bus snooper's queries (see Figure 6-7). Nonetheless, many machines retain dual tags even in multilevel cache designs.

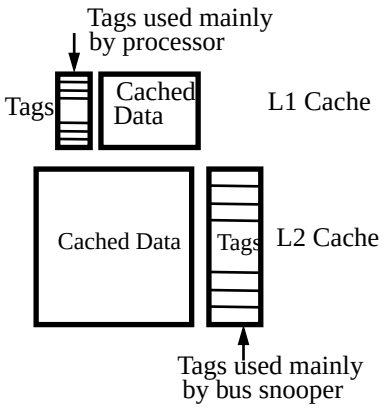

Figure 6-7 Organization of two-level snoopy caches.

With only one outstanding transaction on the bus at a time, the major correctness issues do not change much by using a multi-level hierarchy as long as inclusion is maintained. The necessary transactions are propagated up and down the hierarchy, and bus transactions may be held up until the necessary propagation occurs. Of course, the performance penalty for holding the processor write until the BusRdX has been granted is more onerous, so we are motivated to try to decouple these operations (the penalty for holding the bus until the L1 cache is queried is avoided by the early commitment optimization above). Before going further down this path, let us remove the second simplifying assumption, of an atomic bus, and examine a more aggressive, split-transaction bus. We will first return to assuming a single-level processor cache for simplicity, and then incorporate multi-level cache hierarchies.

6.5 Split-transaction Bus

An atomic bus limits the achievable bus bandwidth substantially, since the bus wires are idle for the duration between when the address is sent on the bus and when the memory system or another cache supply the data or response. In a split-transaction bus, transactions that require a response are split into two independent sub-transactions, a *request* transaction and a *response* transaction. Other transactions (or sub-transactions) are allowed to intervene between them, so that the bus can be used while the response to the original request is being generated. Buffering is used between the bus and the cache controllers to allow multiple transactions to be outstanding on the bus waiting for snoop and/or data responses from the controllers. The advantage, of course, is that by pipelining bus operations the bus is utilized more effectively and more processors can share the same bus. The disadvantage is increased complexity.

As examples of request-response pairs, a BusRd transaction is now a request that needs a data response. A BusUpgr does not need a data response, but it does require an acknowledgment indicating that it has committed and hence been serialized. This acknowledgment is usually sent down toward the processor by the requesting processor's bus controller when it is granted the bus for the BusUpgr request, so it does not appear on the bus as a separate transaction. A BusRdX needs a data response and an acknowledgment of commitment; typically, these are combined as part of the data response. Finally, a writeback usually does not have a response.

The major complications caused by split transaction buses are:

1. A new request can appear on the bus before the snoop and/or servicing of an earlier request are complete. In particular, *conflicting* requests (two requests to the same memory block, at least one of which is due to a write operation) may be outstanding on the bus at the same time, a case which must be handled very carefully. This is different from the earlier case of non-atomicity of overall actions using an atomic bus; there, a conflicting request could be observed by a processor but before its request even obtained the bus, so the request could be suitably modified before being placed on the bus.
2. The number of buffers for incoming requests and potentially data responses from bus to cache controller is usually fixed and small, so we must either avoid or handle buffers filling up. This is called *flow control*, since it affects the flow of transactions through the system.
3. Since bus/snoop requests are buffered, we need to revisit the issue of when and how snoop responses and data responses are produced on the bus. For example, are they generated in order with respect to the requests appearing on the bus or not, and are the snoop and the data part of the same response transaction?

Example 6-2 Consider the previous example of two processors P1 and P2 having the block cached in shared state and deciding to write it at the same time (Example 6-1). Show how a split-transaction bus may introduce complications which would not arise with an atomic bus.

Answer With a split-transaction bus, P1 and P2 may generate BusUpgr requests that are granted the bus on successive cycles. For example, P2 may get the bus before it has been able to look up the cache for P1's request and detect it to be conflicting. If they both assume that they have acquired exclusive ownership, the protocol breaks down

because both P1 and P2 now think they have the block in modified state. On an atomic bus this would never happen because the first BusUpgr transaction would complete—snoops, responses, and all—before the second one got on the bus, and the latter would have been forced to change its request from BusUpgr to BusRdX. (Note that even the breakdown on the atomic bus discussed in Example 6-1 resulted in only one processor having the block in modified state, and the other having it in shared state.)

The design space for split-transaction, cache-coherent busses is large, and a great deal of innovation is on-going in the industry. Since bus operations are pipelined, one high level design decision is whether responses need to be kept in the same order as the requests. The Intel Pentium Pro and DEC Turbo Laser busses are examples of the "in order" approach, while the SGI Challenge and Sun Enterprise busses allow responses to be out of order. The latter approach is more tolerant of variations in memory access times (memory may be able to satisfy a later request quicker than an earlier one, due to memory bank conflicts or off-page DRAM access), but is more complex. A second key decision is how many outstanding requests are permitted: few or many. In general, a larger number of outstanding requests allows better bus utilization, but requires more buffering and design complexity. Perhaps the most critical issue from the viewpoint of the cache coherence protocol is how ordering is established and when snoop results are reported. Are they part of the request phase or the response phase? The position adopted on this issue determines how conflicting operations can be handled. Let us first examine one concrete design fully, and then discuss alternatives.

6.5.1 An Example Split-transaction Design

The example is based loosely on the Silicon Graphics Challenge bus architecture, the PowerPath 2. It takes the following positions on the three issues. Conflicting requests are dealt with very simply, if conservatively: the design disallows multiple requests for a block from being outstanding on the bus at once. In fact, it allows only eight outstanding requests at a time on the bus, thus making the conflict detection tractable. Limited buffering is provided between the bus and the cache controllers, and flow-control for these buffers is implemented through *negative acknowledgment* or *NACK* lines on the bus. That is, if a buffer is full when a request is observed, which can be detected as soon as the request appears on the bus, the request is rejected and NACKed; this renders the request invalid, and asks the requestor to retry. Finally, responses are allowed to be provided in a different order than that in which the original requests appeared on the bus. The request phase establishes the total order on coherence transactions, however, snoop results from the cache controllers are presented on the bus as part of the response phase, together with the data if any.

Let us examine the high-level bus design and how responses are matched up with requests. Then, we shall look at the flow control and snoop result issues in more depth. Finally, we shall examine the path of a request through the system, including how conflicting requests are kept from being simultaneously outstanding on the bus.

6.5.2 Bus Design and Request-response Matching

The split-transaction bus design essentially consists of two separate busses, a request bus for command and address and a response bus for data. The request bus provides the type of request (e.g., BusRd, BusWB) and the target address. Since responses may arrive out of order with regard

to requests, there needs to be a way to identify returning responses with their outstanding requests. When a request (command-address pair) is granted the bus by the arbiter, it is also assigned a unique tag (3-bits, since there are eight outstanding requests in the base design). A response consists of data on the data bus as well as the original request tags. The use of tags means that responses do not need to use the address lines, keeping them available for other requests. The address and the data buses can be arbitrated for separately. There are separate bus lines for arbitration, as well as for flow control and snoop results.

Cache blocks are 128 bytes (1024 bits) and the data bus is 256 bits wide in this particular design, so four cycles plus a one cycle 'turn-around' time are required for the response phase. A uniform pipeline strategy is followed, so the request phase is also five cycles: arbitration, resolution, address, decode, and acknowledgment. Overall, a complete request-response transaction takes three or more of these five-cycle phases, as we shall see. This basic pipelining strategy underlies several of the higher level design decisions. To understand this strategy, let's follow a single read operation through to completion, shown in Figure 6-8. In the request arbitration cycle a cache controller presents its request for the bus. In the request resolution cycle all requests are considered, a single one is granted, and a tag is assigned. The winner drives the address in the following address cycle and then all controllers have a cycle to decode it and look up the cache tags to determine whether there is a snoop hit (the snoop result will be presented on the bus later). At this point, cache controllers can take the action which makes the operation 'visible' to the processor. On a BusRd, an exclusive block is downgraded to shared; on a BusRdX or BusUpgr blocks are invalidated. In either case, a controller owning the block as dirty knows that it will need to flush the block in the response phase. If a cache controller was not able to take the required action during the Address phase, say if it was unable to gain access to the cache tags, it can inhibit the completion of the bus transaction in the A-ack cycle. (During the ack cycle, the first data transfer cycle for the previous data arbitration cycle can take place, occupying the data lines for four cycles, see Figure 6-8.)

After the address phase it is determined which module should respond with the data: the memory or a cache. The responder may request the data bus during a following arbitration cycle. (Note that in this cycle a requestor also initiates a new request on the address bus). The data bus arbitration is resolved in the next cycle and in the address cycle the tag can be checked. If the target is ready, the data transfer starts on the ack cycle and continues for three additional cycles. After a single turn-around cycle, the next data transfer (whose arbitration was proceeding in parallel) can start. The cache block sharing state (snoop result) is conveyed with the response phase and state bits are set when the data is updated in the cache.

As discussed earlier, writebacks (BusWB) consist only of a request phase. They require use of both the address and data lines together, and thus must arbitrate for simultaneous use of both resources. Finally, upgrades (BusUpgr) performed to acquire exclusive ownership for a block also have only a request part since no data response is needed on the bus. The processor performing a write that generates the BusUpgr is sent a response by its own bus controller when the BusUpgr is actually placed on the bus, indicating that the write is committed and has been serialized in the bus order.

To keep track of the outstanding requests on the bus, each cache controller maintains an eight-entry buffer, called a *request table* (see Figure 6-9), Whenever a new request is issued on the bus, it is added to all request tables at the same index, which is the three-bit tag assigned to that request, as part of the arbitration process. A request table entry contains the block address associated with the request, the request type, the state of the block in the local cache (if it has already

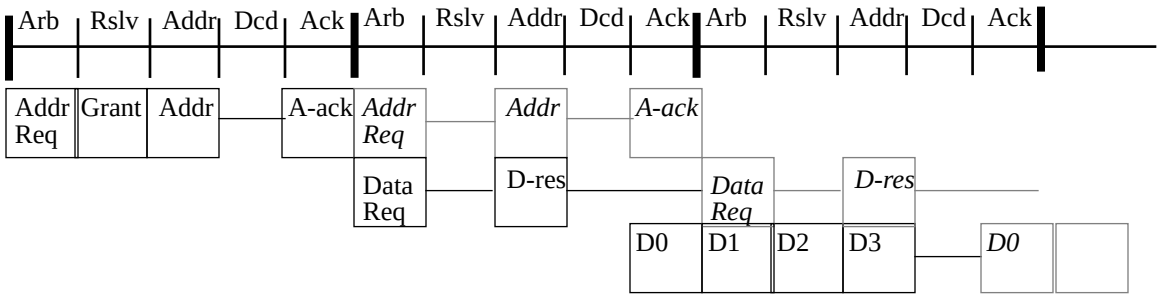

Figure 6-8 Complete read transaction for a split-transaction bus

A pair of consecutive read operations are performed on consecutive phases. Each phase consist of five specific cycles: arbitration, resolution, address, decode, and acknowledgment. Transactions are split into to phases: address and data.

been determined), and a few other bits. The request table is fully associative, so a new request can be physically placed anywhere in the table; all request table entries are examined for a match by both requests issued by this processor and by other requests and responses observed from the bus. A request table entry is freed when a response to the request is observed on the bus. The tag value associated with that request is reassigned by the bus arbiter only at this point, so there are no conflicts in the request tables.

6.5.3 Snoop Results and Conflicting Requests

Like the SGI Challenge, this base design uses variable delay snooping. The snoop portion of the bus consists of the three wired-or lines discussed earlier: *sharing*, *dirty*, and *inhibit*, which extends the duration of the current response sub-transaction. At the end of the request phase, it is determined which module is to respond with the data. However, it may be many cycles before that data is ready and the responder gains access to the data bus. During this time, other requests and responses may take place.The snoop results in this design are presented on the bus by all controllers at the time they see the actual response to a request being put on the bus, i.e. during the response phase. Writeback and upgrade requests do not have a data response, but then they do not require a snoop response either.

Avoiding conflicting requests is easy: Since every controller has a record of the pending reads in its request table, no request is issued for a block that has a response outstanding. Writes are performed during the request phase, so even though the bus is pipelined the operations for an individual location are serialized as in the atomic case. However, this alone does not ensure sequential consistency.

6.5.4 Flow Control

In addition to flow control for incoming requests from the bus, flow control may also be required in other parts of the system. The cache subsystem has a buffer in which responses to its requests can be stored, in addition to the writeback buffer discussed earlier. If the processor or cache allows only one outstanding request at a time, this response buffer is only one entry deep. The number of buffer entries is usually kept small anyway, since a response buffer entry contains not

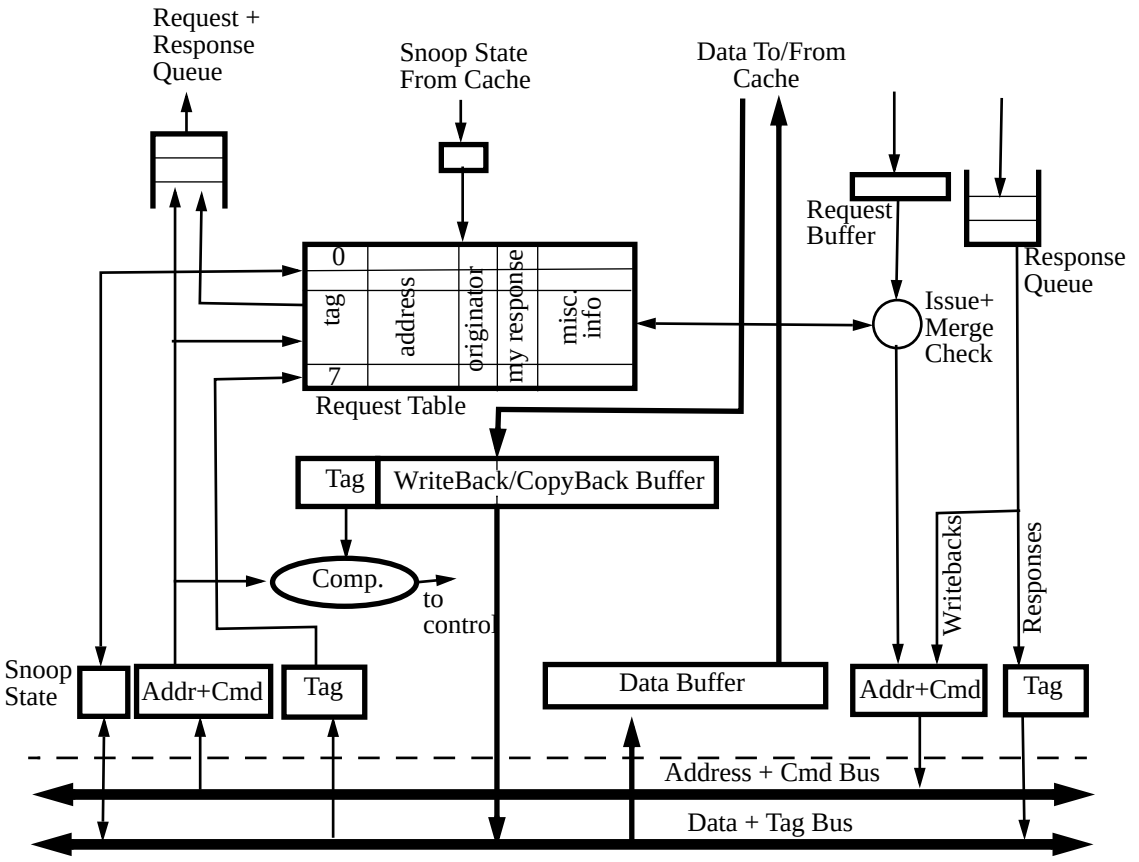

Figure 6-9 Extension of the bus-interface logic shown in Figure 6-4 to accommodate a split-transaction bus.

The key addition is an eight-entry *request table* that keeps track of all outstanding requests on the bus. Whenever a new request is issued on the bus, it is added at the same index in all processors' request table using a common assignment algorithm. The request table serves many purposes, including request merging, and ensuring that only a single request can be outstanding for any given memory block.

only an address but also a cache block of data and is therefore large. The controller limits the number of requests it has outstanding so that there is buffer space available for every response.

Flow control is also needed at main memory. Each of the (eight) pending requests can generate a writeback that main memory must accept. Since writeback transactions do not require a response, they can happen in quick succession on the bus, possibly overflowing buffers in the main memory subsystem. For flow control, the Challenge design provides separate NACK (negative acknowledgment) lines for the address and data portions of bus, since the bus allows independent arbitration for each portion. Before a request or response sub-transaction has reached its ack phase and completed, any other processor or the main memory can assert a NACK signal, for example if it finds its buffers full. The sub-transaction is then canceled everywhere, and must be retried. One common option, used in the Challenge, is to have the requestor retry periodically until it succeeds. Backoff and priorities can be used to reduce bandwidth consumption for failed retries and to avoid starvation. The Sun Enterprise uses an interesting alternative for writes that encounter a full buffer. In this case, the destination buffer—which could not accommodate the

data from the write on the first attempt—itself initiates the retry when it has enough buffer space. The original writer simply keeps watch for the retry transaction on the bus, and places the data on the data bus. The operation of the Enterprise bus ensures that the space in the destination buffer is still available when the data arrive. This guarantees that writes will succeed with only one retry bus transaction, and does not require priority-based arbitration which can slow the bus down.

6.5.5 Path of a Cache Miss

Given this design, we are ready to examine how various requests may be handled, and the race conditions that might occur. Let us first look at the case where a processor has a read miss in the cache, so the request part of a BusRd transaction should be generated. The request first checks the currently pending entries in the request table. If it finds one with a matching address, there are two possible courses of action depending on the nature of the pending request:

1. The earlier request was a BusRd request for the same block. This is great news for this processor, since the request needn't be put on the bus but can just grab the data when the response to the earlier request appears on the bus. To accomplish this, we add two new bits to each entry in the request-table which say: (i) do I wish to grab the data response for this request, and (ii) am I the original generator of this request. In our situation these bits will be set to 1 and 0 respectively. The purpose of the first bit is obvious; the purpose of the second bit is to help determine in which state (valid-exclusive versus shared) to signal the data response. If a processor is not the original requestor, then it must assert the sharing line on the snoop bus when it grabs the response data from the bus, so that all caches will load this block in shared state and not valid-exclusive. If a processor *is* the original requestor, it does not assert the sharing line when it grabs the response from the bus, and if the sharing line is not asserted at all then it will grab the block in valid-exclusive state.
2. The earlier request was incompatible with a BusRd, for example, a BusRdX. In this case, the controller must hold on to the request until it sees a response to the previous request on the bus, and only then attempt the request. The "processor-side" controller is typically responsible for this.

If the controller finds that there are no matching entries in the request-table, it can go ahead and issue the request on the bus. However, it must watch out for race conditions. For example, when the controller first examines the request table it may find that there are no conflicting requests, and so it may request arbitration for the bus. However, before it is granted the bus, a conflicting request may appear on the bus, and then it may be granted the very next use of the bus. Since this design does not allow conflicting requests on the bus, when the controller sees a conflicting request in the slot just before its own it should (i) issue a null request (a no-action request) on the bus to occupy the slot it had been granted and (ii) withdraw from further arbitration until a response to the conflicting request has been generated.

Suppose the processor does manage to issue the BusRd request on the bus. What should other cache controllers and the main memory controller do? The request is entered into the request tables of all cache controllers, including the one that issued this request, as soon as it appears on the bus. The controllers start checking their caches for the requested memory block. The main memory subsystem has no idea whether this block is dirty in one of the processor's caches, so it independently starts fetching this block. There are now three different scenarios to consider:

1. One of the caches may determine that it has the block dirty, and may acquire the bus to generate a response before main memory can respond. On seeing the response on the bus, main

memory simply aborts the fetch that it had initiated, and the cache controllers that are waiting for this block grab the data in a state based on the value of the snooping lines. If a cache controller has not finished snooping by the time the response appears on the bus, it will keep the inhibit line asserted and the response transaction will be extended (i.e. will stay on the bus). Main memory also receives the response since the block was dirty in a cache. If main memory does not have the buffer space needed, it asserts the NACK signal provided for flow control, and it is the responsibility of the controller holding the block dirty to retry the response transaction later.

2. Main memory may fetch the data and acquire the bus before the cache controller holding the block dirty has finished its snoop and/or acquired the bus. The controller holding the block dirty will first assert the inhibit line until it has finished its snoop, and then assert the dirty line and release the inhibit line, indicating to the memory that it has the latest copy and that memory should not actually put its data on the bus. On observing the dirty line, memory cancels its response transaction and does not actually put the data on the bus. The cache with the dirty block will sometime later acquire the bus and put the data response on it.
3. The simplest scenario is that no other cache has the block dirty. Main memory will acquire the bus and generate the response. Cache controllers that have not finished their snoop will assert the inhibit line when they see the response from memory, but once they de-assert it memory can supply the data (Cache-to-cache sharing is not used for data in shared state).

Processor writes are handled similarly to reads. If the writing processor does not find the data in its cache in a valid state, a BusRdX is generated. As before, it checks the request table and then goes on the bus. Everything is the same as for a bus read, except that main memory will not sink the data response if it comes from another cache (since it is going to be dirty again) and no other processor can grab the data. If the block being written is valid but in shared state, a BusUpgr is issued. This requires no response transaction (the data is known to be in main memory as well as in the writer's cache); however, if any other processor was just about to issue a BusUpgr for the same block, it will need to now convert its request to a BusRdX as in the atomic bus.

6.5.6 Serialization and Sequential Consistency

Consider serialization to a single location. If an operation appearing on the bus is a read (miss), no subsequent write appearing on the bus after the read should be able to change the value returned to the read. Despite multiple outstanding transactions on the bus, here this is easy since conflicting requests to the same location are not allowed simultaneously on the bus. If the operation is a BusRdX or BusUpgr generated by a write operation, the requesting cache will perform the write into the cache array after the response phase; subsequent (conflicting) reads to the block are allowed on the bus only after the response phase, so they will obtain the new value. (Recall that the response phase may be a separate action on the bus as in a BusRdX or may be implicitly generated once the request wins arbitration as in a BusUpgr).

Now consider the serialization of operations to different locations needed for sequential consistency. The logical total order on bus transactions is established by the order in which requests are granted for the address bus. Once a BusRdX or BusUpgr has obtained the bus, the associated write is committed. However, with multiple outstanding requests on the bus, the invalidations are buffered and it may be a while before they are applied to the cache. Commitment of a write does not guarantee that the value produced by the write is already visible to other processors; only actual completion guarantees that. The separation between commitment and completion and the need for buffering multiple outstanding transactions imply the need for further mechanisms to

ensure that the necessary orders are preserved between the bus and the processor. The following examples will help make this concrete.

Example 6-3 Consider the two code fragments shown below. What results for (A,B) are disallowed under SC? Assuming a single level of cache per processor and multiple outstanding transactions on the bus, and no special mechanisms to preserve orders between bus and cache or processor, show how the disallowed results may be obtained. Assume an invalidation-based protocol, and that the initial values of A and B are 0.

P1	P2
A = 1	rd B
B = 1	rd A

P1	P2
A = 1	B = 1
rd B	rd A

Answer In the first example, on the left, the result not permitted under SC is (A,B) = (0,1). However, consider the following scenario. P1's write of A commits, so it continues with the write of B (under the sufficient conditions for SC). The invalidation corresponding to B is applied to the cache of P2 before that corresponding to A, since they get reordered in the buffers. P2 incurs a read miss on B and obtains the new value of 1. However, the invalidation for A is still in the buffer and not applied to P2's cache even by the time P2 issues the read of A. The read of A is a hit, and completes returning the old value 0 for A from the cache.

In the example on the right, the disallowed result is (0,0). However, consider the following scenario. P1 issues and commits its write of A, and then goes forward and completes the read of B, reading in the old value of 0. P2 then writes B, which commits, so P2 proceeds to read A. The write of B appears on the bus after the write of A, so they should be serialized in that order and P2 should read the new value of A. However, the invalidation corresponding to the write of A by P1 is sitting in P2's incoming buffer and has not yet been applied to P2's cache. P2 sees a read hit on A and completes returning the old value of A which is 0.

With commitment substituting for commitment and multiple outstanding operations being buffered between bus and processor, the key property that must be preserved for sequential consistency is the following: A processor should not be allowed to actually see the new value due to a write before previous writes (in bus order, as usual) are visible to it. There are two ways to preserve this property: by not letting certain types of incoming transactions from bus to cache be reordered in the incoming queues, and by allowing these reorderings but ensuring that the necessary orders are preserved when necessary. Let us examine each approach briefly.

A simple way to follow the first approach is to ensure that all incoming transactions from the bus (invalidations, read miss replies, write commitment acknowledgments etc.) propagate to the processor in FIFO order. However, such strict ordering is not necessary. Consider an invalidation-based protocol. Here, there are two ways for a new value to be brought into the cache and made available to the processor to read without another bus operation. One is through a read miss, and the other is through a write by the same processor. On the other hand, writes from other proces-

sors become *visible* to a processor (even though the values are not yet local) when the corresponding invalidations are applied to the cache. The invalidations due to writes that are previous to the read miss or local write that provides the new value are already in the queue when the read miss or local write appears on the bus, and therefore are either in the queue or applied to the cache when the reply comes back. All we need to ensure, therefore, is that a reply (read miss or write commitment) does not overtake an invalidation between the bus and the cache; i.e. all previous invalidations are applied before the reply is received.

Note that incoming invalidations may be reordered with regard to one another. This is because the new value corresponding to an invalidation is seen only through the corresponding read miss, and the read miss reply is not allowed to be reordered with respect to the previous invalidation. In an update-based protocol, the new value due to a write does not require a read miss and reply, but rather can be seen as soon as the incoming update has been applied. Writes are made visible through updates as well. This means that not only should replies not overtake updates, but updates should not overtake updates either.

An alternative is to allow incoming transactions from the bus to be reordered on their way to the cache, but to simply ensure that all previously committed writes are applied to the cache (by flushing them from the incoming queue) before an operation from the local processor that enables it to see a new value can be completed. After all, what really matters is not the order in which invalidations/updates are applied, but the order in which the corresponding new values are seen by the processor. There are two natural ways to accomplish this. One is to flush the incoming invalidations/updates every time the processor tries to complete an operation that may enable it to see a new value. In an invalidation-based protocol, this means flushing before the processor is allowed to complete a read miss or a write that generates a bus transaction; in an update-based protocol, it means flushing on every read operation as well. The other way is to flush under the following circumstance: Flush whenever a processor is about to access a value (complete a read hit or miss) if it has indeed seen a new value since the last flush, i.e. if a reply or an update has been applied to the cache since the last flush. The fact that operations are reordered from bus to cache and a new value has been applied to the cache means that there may be invalidations or updates in the queue that correspond to writes that are previous to that new value; those writes should now be applied before the read can complete. Showing that these techniques disallow the undesirable results in the example above is left as an exercise that may help make the techniques concrete. As we will see soon, the extension of the techniques to multi-level cache hierarchies is quite natural.

Regardless of which approach is used, write atomicity is provided naturally by the broadcast nature of the bus. Writes are committed in the same order with respect to all processors, and a read cannot see the value produced by a write until that write has committed with respect to all processors. With the above techniques, we can substitute complete for commit in this statement, ensuring atomicity. We will discuss deadlock, livelock and starvation issues introduced by a split transaction bus after we have also introduced multi-level cache hierarchies in this context. First, let us look at some alternative approaches to organizing a protocol with a split transaction bus.

6.5.7 Alternative design choices

There are reasonable alternative positions for all of request-response ordering, dealing with conflicting requests, and flow control. For example, ensuring that responses are generated in order with respect to requests—as cache controllers are inclined to do—would simplify the design. The

fully-associative request table could be replaced with a simple FIFO buffer that stores pending requests that were observed on the bus. As before, a request is put into the FIFO only when the request actually appears on the bus, ensuring that all entities (processors and memory system) have exactly the same view of pending requests. The processors and the memory system process requests in FIFO order. At the time the response is presented, as in the earlier design, if others have not completed their snoops they assert the inhibit line and extend the transaction duration. That is, snoops are still reported together with responses. The difference is in the case where the memory generates a response first even though a processor has that block dirty in its cache. In the previous design, the cache controller that had the block dirty released the inhibit line and asserted the dirty line, and arbitrated for the bus again later when it had retrieved the data. But now to preserve the FIFO order this response has to be placed on the bus before any other request is serviced. So the dirty controller does not release the inhibit line, but extends the current bus transaction until it has fetched the block from its cache and supplied it on the bus. This does not depend on anyone else accessing the bus, so there is no deadlock problem.

While FIFO request-response ordering is simpler, it can have performance problems. Consider a multiprocessor with an interleaved memory system. Suppose three requests A, B, and C are issued on the bus in that order, and that A and B go to the same memory bank while C goes to a different memory bank. Forcing the system to generate responses in order means that C will have to wait for both A and B to be processed, though data for C will be available much before data for B is available because of B's bank conflict with A. This is the major motivation for allowing out of order responses, since caches are likely to respond to requests in order anyway.

Keeping responses in order also makes it more tractable to allow conflicting requests to the same block to be outstanding on the bus. Suppose two BusRdX requests are issued on a block in rapid succession. The controller issuing the latter request will invalidate its block, as before. The tricky part with a split-transaction bus is that the controller issuing the earlier request sees the latter request appear on the bus before the data response that it awaits. It cannot simply invalidate its block in reaction to the latter request, since the block is in flight and its own write needs to be performed before a flush or invalidate. With out-of-order responses, allowing this conflicting request may be difficult. With in-order responses, the earlier requestor knows its response will appear on the bus first, so this is actually an opportunity for a performance enhancing optimization. The earlier-requesting controller responds to the latter request as usual, but notes that the latter request is pending. When its response block arrives, it updates the word to be written and "short-cuts" the block back out to the bus, leaving its own block invalid. This optimization reduces the latency of ping-ponging a block under write-write false sharing.

If there is a fixed delay from request to snoop result, conflicting requests can be allowed even without requiring data responses to be in order. However, since conflicting requests to a block go into the same queue at the desintation they themselves are usually responded to in order anyway, so they can be handled using the shortcut method described above (this is done in the Sun Enterprise systems). It is left as an exercise to think about how one might allow conflicting requests with fixed-delay snoop results when responses to them may appear on the bus out of order.

In fact, as long as there is a well-defined order among the request transactions, they do not even need to be issued sequentially on the same bus. For example, the Sun SPARCcenter 2000 used two distinct split-phase busses and the Cray 6400 used four to improve bandwidth for large configurations. Multiple requests may be issued on a single cycle. However, a priority is established among the busses so that a logical order is defined even among the concurrent requests.

6.5.8 Putting it together with multi-level caches

We are now ready to combine the two major enhancements to the basic protocol from which we started: multi-level caches and a split-transaction bus. The base design is a (Challenge-like) split-transaction bus and a two-level cache hierarchy. The issues and solutions generalize to deeper hierarchies. We have already seen the basic issues of request, response, and invalidation propagation up and down the hierarchy. The key new issue we need to grapple with is that it takes a considerable number of cycles for a request to propagate through the cache controllers. During this time, we must allow other transactions to propagate up and down the hierarchy as well. To maintain high bandwidth while allowing the individual units to operate at their own rates, queues are placed between the units. However, this raises a family of questions related to deadlock and serialization.

A simple multi-level cache organization is shown in Figure 6-10. Assume that a processor can have only one request outstanding at a time, so there are no queues between the processor and first-level cache. A read request that misses in the L1 cache is passed on to the L2 cache (1). If it misses there, a request is placed on the bus (2). The read request is captured by all other cache controllers in the incoming queue (3). Assuming the block is currently in modified state in the L1 cache of another processor, the request is queued for L1 service (4). The L1 demotes the block to shared and flushes it to the L2 cache (5), which places it on the bus (6). The response is captured by the requestor (7) and passed to the L1 (8), whereupon the word is provided to the processor.

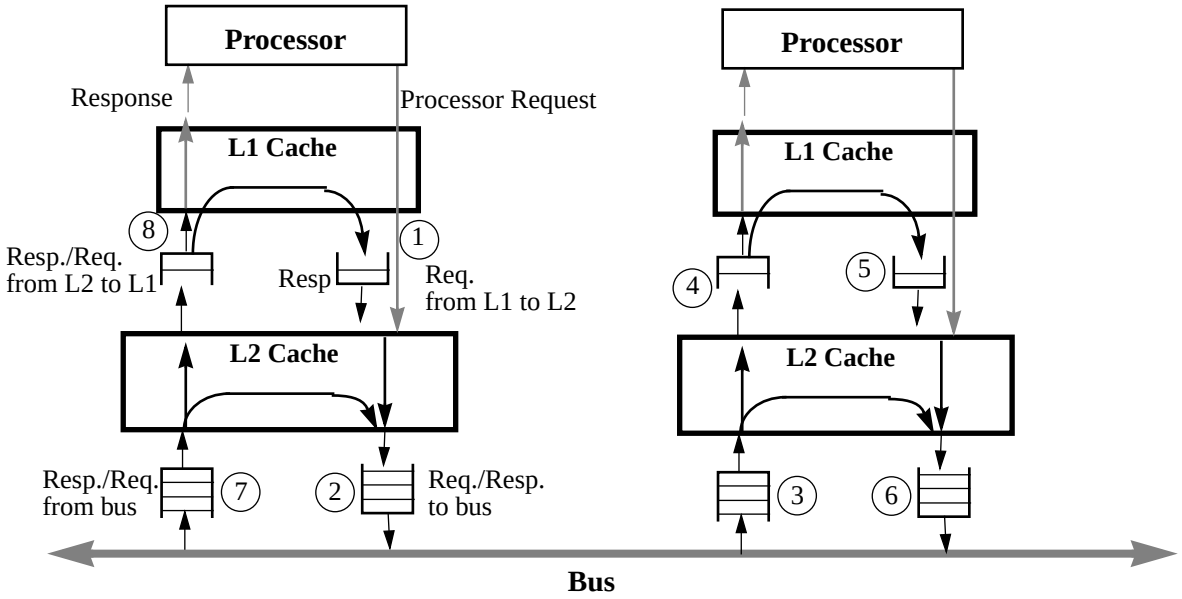

Figure 6-10 Internal queues that may exist inside a multi-level cache hierarchy.

One of the concerns with such queue structures is deadlock. To avoid the fetch deadlock problem discussed earlier, as before an L2 cache needs to be able to buffer incoming requests or responses while it has a request outstanding, so that the bus may be freed up. With one outstanding request

per processor, the incoming queues between the bus and the L2 cache need to be larger enough to hold a request from each other processor (plus a response to its request). That takes care of the case where all processors make a request of this cache while the processor has a request outstanding. If they are made smaller than this to conserve real-estate (or if there are multiple outstanding requests per processor), it is necessary to NACK bus requests when there is not room to enqueue them. Also, one slot in the bus to L2 and in the L2 to L1 queues is reserved for the response to the processor's outstanding request, so that each processor can always drain its outstanding responses. If NACKs are used, the request bus arbitration needs to include a mechanism to ensure forward progress under heavy contention, such as a simple priority scheme.

In addition to fetch deadlock, buffer deadlock can occur within the multilevel cache hierarchy as well. For example, suppose there is a queue in each direction between the L1 and L2 cache, both of which are writeback caches, and each queue can hold two entries. It is possible that the L1->L2 queue holds two outgoing read requests, which can be satisfied in the L2 cache but will generate replies to L1, and the L2->L1 queue holds two incoming read requests, which can be satisfied in the L1 cache. We now have a classical circular buffer dependence, and hence deadlock. Note that this problem occurs only in hierarchies in which there is more than one level of writeback cache, i.e. in which a cache higher than the one closest to the bus is writeback. Otherwise, incoming requests do not generate replies from higher level caches, so there is no circularity and no deadlock problem (recall that invalidations are acknowledged implicitly from the bus itself, and do not need acknowledgments from the caches).

One way to deal with this buffer deadlock problem in a multilevel writeback cache hierarchy is to limit the number of outstanding requests from processors and then provide enough buffering for incoming requests and responses at each level. However, this requires a lot of real estate and is not scalable (each request may need two outgoing buffer entries, one for the request and one for the writeback it might generate, and with a large number of outstanding bus transactions the incoming buffers may need to have many entries as well) . An alternative way uses a general deadlock avoidance technique for situations with limited buffering, which we will discuss in the context of systems with physically distributed memory in the next chapter, where the problem is more acute. The basic idea is to separate transactions (that flow through the hierarchy) into requests and responses. A transaction can be classified as a response if it does not generate any further transactions; a request may generate a response, but not transaction may generate another request (although a request may be transferred to the next level of the hierarchy if it does not generate a reply at the original level). With this classification, we can avoid deadlock if we provide separate queues for requests and responses in each direction, and ensure that responses are always extracted from the buffers. After we have discussed this technique in the next chapter, we shall apply to this particular situation with multilevel writeback caches in the exercises.

There are other potential deadlock considerations that we may have to consider. For example, with a limited number of outstanding transactions on the bus, it may be important for a response from a processor's cache to get to the bus before new outgoing requests from the processor are allowed. Otherwise the existing requests may never be satisfied, and there will be no progress. The outgoing queue or queues must be able to support responses bypassing requests when necessary.

The second major concern is maintaining sequential consistency. With multi-level caches, it is all the more important that the bus not wait for an invalidation to reach all the way up to the first-level cache and return a reply, but consider the write committed when it has been placed on the bus and hence in the input queue to the lowest-level cache. The separation of commitment and

completion is even greater in this case. However, the techniques discussed for single level caches extend very naturally to this case: We simply apply them at each level of the cache hierarchy. Thus, in an invalidation based protocol the first solution extends to ensuring at each level of the hierarchy that replies are not reordered with respect to invalidations in the incoming queues to that level (replies from a lower level cache to a higher-level cache are treated as replies too for this purpose). The second solution extends to either not letting an outgoing memory operation proceed past a level of the hierarchy before the incoming invalidations to that level are applied to that cache, or flushing the incoming invalidations at a level if a reply has been applied to that level since the last flush.

6.5.9 Supporting Multiple Outstanding Misses from a Processor

Although we have examined split transaction buses, which have multiple transactions outstanding on them at a time, so far we have assumed that a given processor can have only one memory request outstanding at a time. This assumption is simplistic for modern processors, which permit multiple outstanding requests to tolerate the latency of cache misses even on uniprocessor systems. While allowing multiple outstanding references from a processor improves performance, it can complicate semantics since memory accesses from the same processor may complete in a different order in the memory system than that in which they were issued.

One example of multiple outstanding references is the use of a write buffer. Since we would like to let the processor proceed to other computation and even memory operations after it issues a write but before it obtains exclusive ownership of the block, we put the write in the write buffer. Until the write is serialized, it should be visible only to the issuing processor and not to other processors, since otherwise it may violate write serialization and coherence. One possibility is to write it into the local cache but not make it available to other processors until exclusive ownership is obtained (i.e. not let the cache respond to requests for it until then). The more common approach is to keep it in the write buffer and put it in the cache (making it available to other processors through the bus) only when exclusive ownership is obtained.

Most processors use write buffers more aggressively, issuing a sequence of writes in rapid succession into the write buffer without stalling the processor. In a uniprocessor this approach is very effective, as long as reads check the write buffer. The problem in the multiprocessor case is that, in general, the processor cannot be allowed to proceed with (or at least complete) memory operations past the write until the exclusive ownership transaction for the block has been placed on the bus. However, there are special cases where the processor can issue a sequence of writes without stalling. One example is if it can be determined that the writes are to blocks that are in the local cache in modified state. Then, they can be buffered between the processor and the cache, as long as the cache processes the writes before servicing a read or exclusive request from the bus side. There is also an important special case in which a sequence of writes can be buffered regardless of the cache state. That is where the writes are all to the same block and no other memory operations are interspersed between those writes. The writes may be coalesced while the controller is obtaining the bus for the read exclusive transaction. When that transaction occurs, it makes the entire sequence of writes visible at once. The behavior is the same as if the writes were performed after the bus transaction, but before the next one. Note that there is no problem with sequences of writebacks, since the protocol does not require them to be ordered.

More generally, to satisfy the sufficient conditions for sequential consistency, a processor having the ability to proceed past outstanding write and even read operations raises the question of who

should wait to "issue" an operation until the previous one in program order completes. Forcing the processor itself to wait can eliminate any benefits of the sophisticated processor mechanisms (such as write buffers and out-of-order execution). Instead, the buffers that hold the outstanding operations— such as the write buffer and the reorder buffer in dynamically scheduled out-of-order execution processors—can serve this purpose. The processor can issue the next operation right after the previous one, and the buffers take charge of not making write operations visible to the memory and interconnect systems (i.e. not issuing them to the externally visible memory system) or not allowing read operations to complete out of program order with respect to outstanding writes even though the processor may issue and execute them out of order. The mechanisms needed in the buffers are often already provided to provide precise interrupts, as we will see in later chapters. Of course, simpler processors that do not proceed past reads or writes make it easier to maintain sequential consistency. Further semantic implications of multiple outstanding references for memory consistency models will be discussed in Chapter 9, when we examine consistency models in detail.

From a design perspective, exploiting multiple outstanding references most effectively requires that the processor caches allow multiple cache misses to be outstanding at a time, so that the latencies of these misses can be overlapped. This in turn requires that either the cache or some auxiliary data structure keep track of the outstanding misses, which can be quite complex since they may return out of order. Caches that allow multiple outstanding misses are called *lockup-free* caches [Kro81,Lau94], as opposed to *blocking* caches that allow only one outstanding miss. We shall discuss the design of lockup-free caches when we discuss latency tolerance in Chapter 11.

Finally, consider the interactions with split transaction busses and multi-level cache hierarchies. Given a design that supports a split-transaction bus with multiple outstanding transactions and a multi-level cache hierarchy, the extensions needed to support multiple outstanding operations per processor are few and are mostly for performance. We simply need to provide deeper request queues from the processor to the bus (the request queues pointing downwards in Figure 6-10), so that the multiple outstanding requests can be buffered and not stall the processor or cache. It may also be useful to have deeper response queues, and more write-back and other types of buffers, since there is now more concurrency in the system. As long as deadlock is handled by separating requests from replies, the exact length of any of these queues is not critical for correctness. The reason for such few changes is that the lockup-free caches themselves perform the complex task of merging requests and managing replies, so to the caches and the bus subsystem below it simply appears that there are multiple requests to distinct blocks coming from the processor. While some potential deadlock scenarios might become exposed that would not have arisen with only one outstanding request per processor—for example, we may now see the situation where the number of requests outstanding from all processors is more than the bus can take, so we have to ensure responses can bypass requests on the way out—the support discussed earlier for split-transaction buses makes the rest of the system capable of handling multiple requests from a processor without deadlock.

6.6 Case Studies: SGI Challenge and Sun Enterprise SMPs

This section places the general design and implementation issues discussed above into a concrete setting by describing two multiprocessor systems, the SGI Challenge and the Sun Enterprise 6000.

The SGI Challenge is designed to support up to 36 MIPS R4400 processors (peak 2.7 GFLOPS) or up to 18 MIPS R8000 processors (peak 5.4 GFLOPS) in the Power Challenge model. Both systems use the same system bus, the Powerpath-2 bus, which provides a peak bandwidth of 1.2 Gbytes/sec and the system supports up to 16 Gbytes of 8-way interleaved main memory. Finally, the system supports up to 4 PowerChannel-2 I/O buses, each providing a peak bandwidth of 320 Mbytes/sec. Each I/O bus in turn can support multiple Ethernet connections, VME/SCSI buses, graphics cards, and other peripherals. The total disk storage on the system can be several Terabytes. The operating system that runs on the hardware is a variant of SVR4 UNIX called IRIX; it is a symmetric multiprocessor kernel in that any of the operating systems' tasks can be done on any of the processors in the system. Figure 6-11 shows a high-level diagram of the SGI Challenge system.

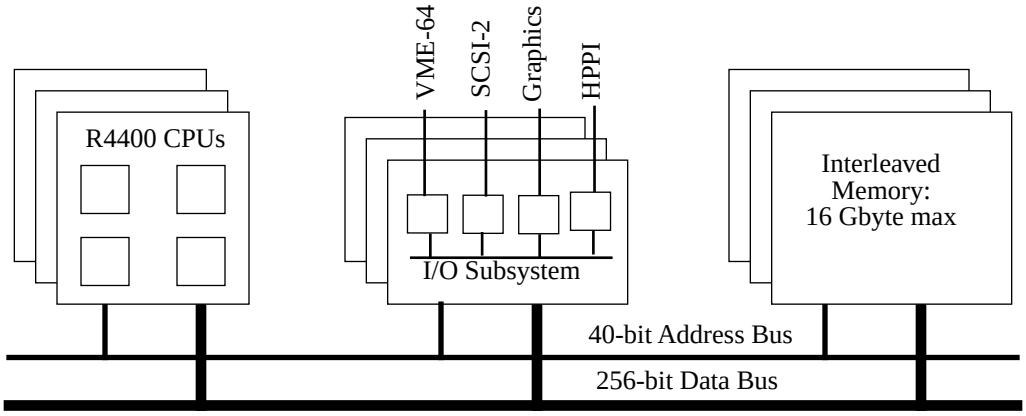

Figure 6-11 The SGI Challenge multiprocessor

With four processors per board, the thirty-six processors consume nine bus slots. It can support up to 16 Gbytes of 8-way interleaved main memory. The I/O boards provide a separate 320 Mbytes/sec I/O bus, to which other standard buses and devices interface. The system bus has a separate 40-bit address path and a 256-bit data path, and supports a peak bandwidth of 1.2 Gbytes/sec. The bus is split-transaction and up to eight requests can be outstanding on the bus at any given time.

The Sun Enterprise 6000 is designed to support up to 30 UltraSPARC processors (peak 9 GFLOPs). The Gigaplane^{TM} system bus provides a peak bandwidth of 2.67 GB/s (83.5MHz times 32 bytes), and the system can support up to 30 GB of up to 16-way interleaved memory. The 16 slots in the machine can be populated with a mix of processing boards and I/O boards, as long as there is a least one of each. Each processing board has two CPU modules and two (512 bit wide) memory banks of up to 1 GB each, so the memory capacity and bandwidth scales with the number of processors. Each I/O card provides two independent 64-bit x 25 MHz SBUS I/O busses, so the I/O bandwidth scales with the number of I/O cards up to more than 2 GB/s. The

total disk storage can be tens of terabytes. The operating system is Solaris UNIX. Figure 6-12 shows a block diagram of the Sun Enterprise system.

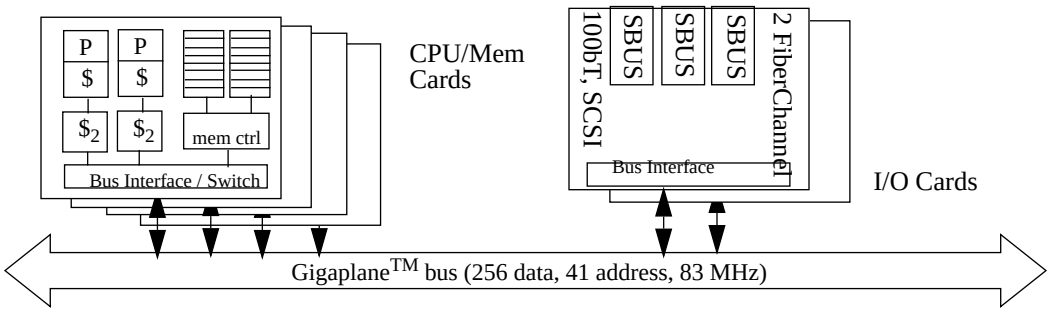

Figure 6-12 The Sun Enterprise 6000 multiprocessor

6.6.1 SGI Powerpath-2 System Bus

The system bus forms the core interconnect for all components in the system. As a result, its design is affected by requirements of all other components, and design choices made for it, in turn, affect back the design of other components. The design choices for buses include: multiplexed versus non-multiplexed address and data buses, wide (e.g., 256 or 128-bit) versus narrower (64-bit) data bus, the clock rate of the bus (affected by signalling technology used, length of bus, the number of slots on bus), split-transaction versus non-split-transaction design, the flow-control strategy, and so on. The powerpath-2 bus is non-multiplexed with 256-bit wide data portion and 40-bit wide address portion, it is clocked at 47.6 MHz, and it is split-transaction supporting 8 outstanding read requests. While the very wide data path implies that the cost of connecting to the bus is higher (it requires multiple bit-sliced chips to interface to it), the benefit is that the high bandwidth of 1.2 Gbytes/sec can be achieved at a reasonable clock rate. At the 50MHz clock rate the bus supports sixteen slots, nine of which can be populated with 4-processor boards to obtain a 36-processor configuration. The width of the bus also affects (and is affected by) the cache block size chosen for the machine. The cache block size in the Challenge machine is 128 bytes, implying that the whole cache block can be transferred in only four bus clocks; a much smaller block size would have resulted in less effective use of the bus pipeline or a more complex design. The bus width affects many other design decisions. For example, the individual board is fairly large in order to support such a large connector. The bus interface occupies roughly 20% of the board, in a strip along the edge, making it natural to place four processors on each board.

Let us look at the Powerpath-2 bus design in a little more detail. The bus consists of a total of 329 signals: 256 data, 8 data parity, 40 address, 8 command, 2 address+command parity, 8 data-resource ID, and 7 miscellaneous. The types and variations of transactions on the bus is small, and all transactions take exactly 5 cycles. System wide, bus controller ASICs execute the following 5-state machine synchronously: arbitration, resolution, address, decode, and acknowledge. When no transactions are occurring, each bus controller drops into a 2-state idle machine. The 2-state idle machine allows new requests to arbitrate immediately, rather than waiting for the arbitrate state to occur in the 5-state machine. Note that 2-states are required to prevent different

requestors from driving arbitration lines on successive cycles. Figure 6-13 shows the state machine underlying the basic bus protocol.

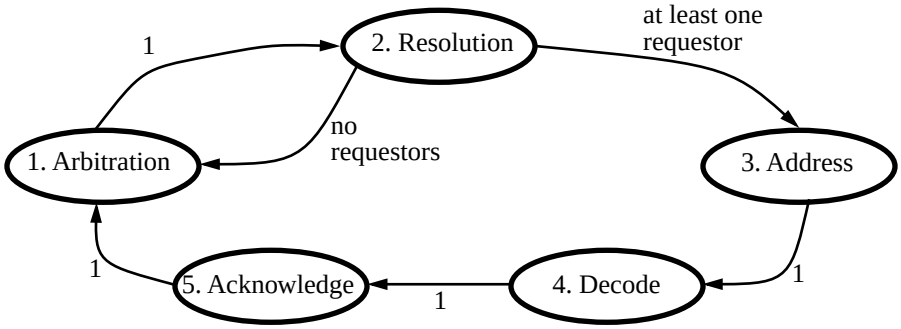

Figure 6-13 Powerpath-2 bus state-transition diagram.

The bus interfaces of all boards attached to the system-bus synchronously cycle through the five states shown in the figure; this is also the duration of all address and data transactions on the bus. When the bus is idle, however, it only loops between states 1 and 2.

Since the bus is split transaction, the address and data buses must be arbitrated for separately. In the arbitration cycle, the 48 address+command lines are used for arbitration. The lower 16 lines are used for data bus arbitration and the middle 16 lines are used for address bus arbitration. For transactions that require both address and data buses together, e.g., writebacks, corresponding bits for both buses can be set high. The top 16 lines are used to make *urgent* or high-priority requests. Urgent requests are used to avoid starvation, for example, if a processor times-out waiting to get access to the bus. The availability of urgent-type requests allowed the designers considerable flexibility in favoring service of some requests over others for performance reasons (e.g., reads are given preference over writes), while still being confident that no requestor will get starved.

At the end of the arbitration cycle, all bus-interface ASICs capture the 48-bit state of the requestors. A distributed arbitration scheme is used, so every bus master sees all of the bus requests, and in the resolution cycle, each one independently computes the same winner. While distributed arbitration consumes more of ASIC's gate resources, it saves the latency incurred by a centralized arbitrator of communicating winners to everybody via bus grant lines.

During the address cycle, the address-bus master drives the address and command buses with corresponding information. Simultaneously, the data-bus master drives the *data resource ID* line corresponding to the response. (The data resource ID corresponds to the global tag assigned to this read request when it was originally issued on the bus. Also see Section 6.5 for details.)

During the decode cycle, no signals are driven on the address bus. Internally, each bus-interface slot decides how to respond to this transaction. For example, if the transaction is a writeback, and the memory system currently has insufficient buffer resources to accept the data, in this cycle it will decide that it must NACK (negative acknowledge or reject) this transaction on the next cycle, so that the transaction can be retried at a later time. In addition all slots prepare to supply the proper cache-coherence information.

During the data acknowledge cycle, each bus interface slot responds to the recent data/address bus transaction. The 48 address+command lines are used as follows. The top 16 lines indicate if the device in the corresponding slot is rejecting the address-bus transaction due to insufficient buffer space. Similarly, the middle 16 lines are used to possibly reject the data-bus transaction. The lowest 16 lines indicate the cache-state of the block (present vs. not-present) being transferred on the data-bus. These lines help determine the state in which the data block will be loaded in the requesting processor, e.g., valid-exclusive versus shared. Finally, in case one of the processors has not finished its snoop by this cycle, it indicates so by asserting the corresponding inhibit line. (The data-resource ID lines during the data acknowledgment and arbitration cycles are called inhibit lines.) It continues to assert this line until it has finished the snoop. If the snoop indicates a clean cache block, the snooping node simply drops the inhibit line, and allows the requesting node to accept memory's response. If the snoop indicated a dirty block, the node re-arbitrates for the data bus and supplies the latest copy of the data, and only then drops the inhibit line.

For data-bus transactions, once a slot becomes the master, the 128 bytes of cache-block data is transferred in four consecutive cycles over the 256-bit wide data path. This four-cycle sequence begins with the data acknowledgment cycle and ends at the address cycle of the following transaction. Since the 256-bit wide data path is used only for four out of five cycles, the maximum possible efficiency of these data lines is 80%. In some sense though, this is the best that could be done; the signalling technology used in the Powerpath-2 bus requires one cycle turn-around time between different masters driving the lines. Figure 6-13 shows the cycles during which various bus lines are driven and their semantics.

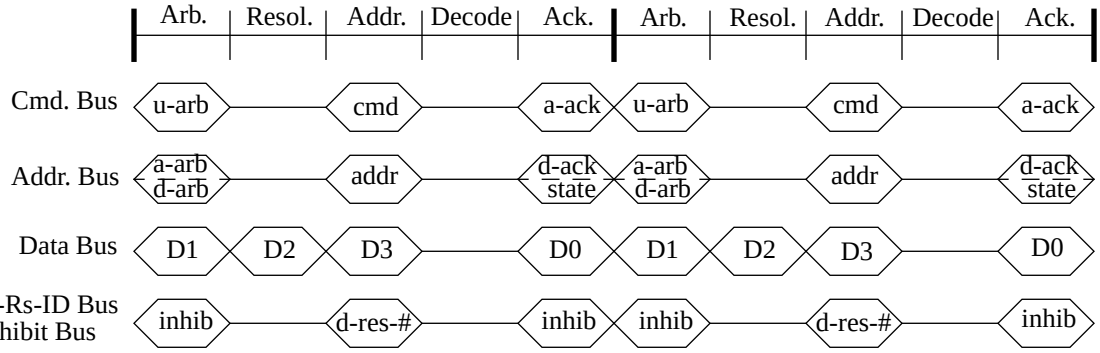

Figure 6-14 Powerpath-2 bus timing diagram.

During the arbitration cycle, the 48-bits of the address + command bus indicate requests from the 16-bus slots for data transaction, address transaction, and urgent transactions. Each bus interface determines the results of the arbitration independently following a common algorithm. If an address request is granted, the address +command are transferred in the address cycle, and the requests can be NACK'ed in the acknowedgment cycle. Similarly, if a data request is granted, the tag associated with it (data-resource-id) is transferred in the address cycle, it can be NACK'ed in the ack cycle, and the data is transferred in the following D0-D3 cycles.

6.6.2 SGI Processor and Memory Subsystems

In the Challenge system, each board can contain 4 MIPS R4400 processors. Furthermore, to reduce the cost of interfacing to the bus, many of the bus-interface chips are shared between the processors. Figure 6-13 shows the high-level organization of the processor board.

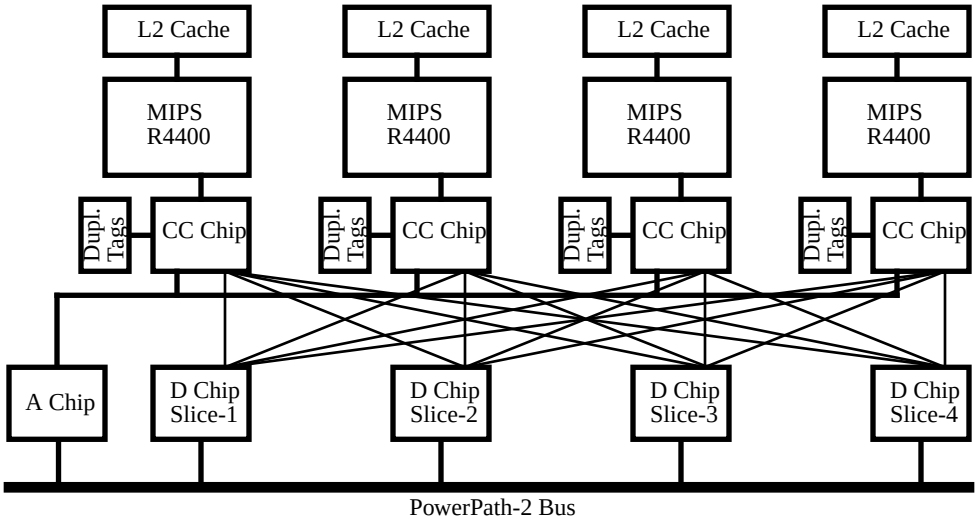

Figure 6-15 Organization and chip partitioning of the SGI Challenge processor board.

To minimize number of bus slots required to support thirty-six processors, four processors are put on each board. To maintain coherence and to interface to the bus, there is one cache-coherence chip per processor, there is one shared A-chip that keeps track of requests from all four processors, and to interface to the 256-bit wide data bus, there are four shared bit-sliced D-chips.

The processor board uses three different types of chips to interface to the bus and to support cache coherence. There is a single A-chip for all four processors that interfaces to the address bus. It contains logic for distributed arbitration, the eight-entry request-table storing currently outstanding transactions on the bus (see Section 6.5 for details), and other control logic for deciding when transactions can be issued on the bus and how to respond to them. It passes on requests observed on the bus to the CC-chip (one for each processor), which uses a duplicate set of tags to determine the presence of that memory block in the local cache, and communicates the results back to the A-chip. All requests from the processor also flow through the CC-chip to the A-chip, which then presents them on the bus. To interface to the 256-bit wide data bus, four bit-sliced D-chips are used. The D-chips are quite simple and are shared among the processors; they provide limited buffering capability and simply pass data between the bus and the CC-chip associated with each processor.

The Challenge main memory subsystem uses high-speed buffers to fan out addresses to a 576-bit wide DRAM bus. The 576 bits consist of 512 bits of data and 64 bits of ECC, allowing for single bit in-line correction and double bit error detection. Fast page-mode access allows an entire 128 byte cache block to be read in two memory cycles, while data buffers pipeline the response to the 256-bit wide data bus. Twelve clock cycles (~250ns) after the address appears on the bus, the response data appears on the data bus. Given current technology, a single memory board can hold 2 Gbytes of memory and supports a 2-way interleaved memory system that can saturate the 1.2 Gbytes/sec system bus.

Given the raw latency of 250ns that the main-memory subsystem takes, it is instructive to see the overall latency experienced by the processor. On the Challenge this number is close to 1µs or 1000ns. It takes approximately 300ns for the request to first appear on the bus; this includes time taken for the processor to realize that it has a first-level cache miss, a second-level cache miss, and then to filter through the CC-chip down to the A-chip. It takes approximately another 400ns for the complete cache block to be delivered to the D-chips across the bus. These include the 3 bus cycles until the address stage of the request transaction, 12 cycles to access the main memory, and another 5 cycles for the data transaction to deliver the data over the bus. Finally, it takes another 300ns for the data to flow through the D-chips, through the CC-chip, through the 64-bit wide interface onto the processor chip (16 cycles for the 128 byte cache block), loading the data into the primary cache and restart of the processor pipeline.[1]

To maintain cache coherence, by default the SGI Challenge used the Illinois MESI protocol as discussed earlier. It also supports update transactions. Interactions of the cache coherence protocol and the split-transaction bus interact are as described in Section 6.5.

6.6.3 SGI I/O Subsystem

To support the high computing power provided by multiple processors, in a real system, careful attention needs to be devoted to providing matching I/O capability. To provide scalable I/O performance, the SGI Challenge allows for multiple I/O cards to be placed on the system bus, each card providing a local 320 Mbytes/sec proprietary I/O bus. Personality ASICs are provided to act as an interface between the I/O bus and standards conforming (e.g., ethernet, VME, SCSI, HPPI) and non-standards conforming (e.g., SGI Graphics) devices. Figure 6-13 shows a block-level diagram of the SGI Challenge's PowerChannel-2 I/O subsystem.

As shown in the figure, the proprietary HIO input/output bus is at the core of the I/O subsystem. It is a 64-bit wide multiplexed address/data bus that runs off the same clock as the system bus. It supports split read-transactions, with up to four outstanding transactions per device. In contrast to the main system bus, it uses centralized arbitration as latency is much less of a concern. However, arbitration is pipelined so that bus bandwidth is not wasted. Furthermore, since the HIO bus supports several different transaction lengths (it does not require every transaction to handle a full cache block of data), at time of request transactions are required to indicate their length. The arbiter uses this information to ensure more efficient utilization of the bus. The narrower HIO bus allows the personality ASICs to be cheaper than if they were to directly interface to the very wide system bus. Also, common functionality needed to interface to the system bus can be shared by the multiple personality ASICs.

1. Note that on both the outgoing and return paths, the memory request passes through an asynchronous boundary. This adds a double synchronizer delay in both directions, about 30ns on average in each direction. The benefit of decoupling is that the CPU can run at a different clock rate than the system bus, thus allowing for migration to higher clock-rate CPUs in the future while keeping the same bus clock rate. The cost, of course, is the extra latency.

The newer generation of processor, the MIPS R10000, allows the processor to restart after only the needed word has arrived, without having to wait for the complete cache block to arrive. This early restart option reduces the miss latency.

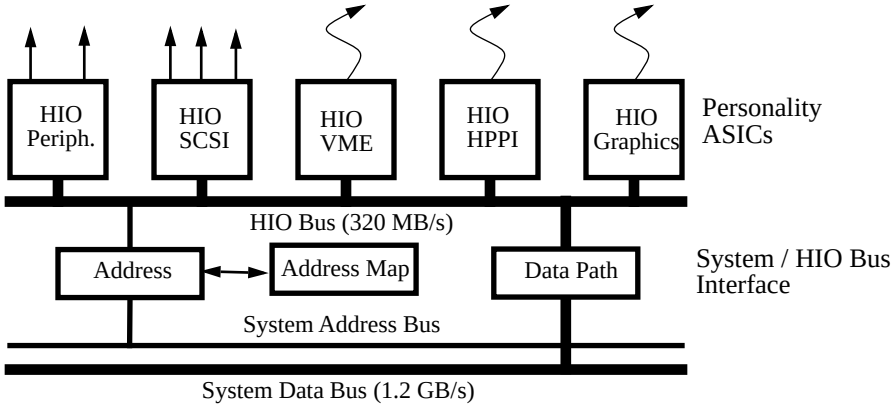

Figure 6-16 High-level organization of the SGI Challenge Powerchannel-2 I/O subsystem.

Each I/O board provides an interface to the Powerpath-2 system bus, and an internal 64-bits wide "HIO" I/O bus with peak bandwidth of 320 Mbytes/sec. The narrower HIO bus lowers the cost of interfacing to it, and it supports a number of personality ASICs which, in turn, support standard buses and peripherals.

HIO interface chips can request DMA read/write to system memory using the full 40-bit system address, make a request for address translation using the mapping resource in the system interface (e.g., for 32-bit VME), request interrupting the processor, or respond to processor I/O (PIO) reads. The system bus interface provides DMA read responses, results of address translation, and passes on PIO reads to devices.

To the rest of the system (processor boards and main-memory boards), the system bus interface on the I/O board provides a clean interface; it essentially acts like another processor board. Thus, when a DMA read makes it through the system-bus interface onto the system bus, it becomes a Powerpath-2 read, just like one that a processor would issue. Similarly, when a full-cache-block DMA write goes out, it becomes a special block write transaction on the bus that invalidates copies in all processors' caches. Note that even if a processor had the block dirty in its local cache, we do not want it to write it back, and hence the need for the special transaction.

To support partial block DMA writes, special care is needed, because data must be merged coherently into main memory. To support these partial DMA writes, the system-bus interface includes a fully associative, four block cache, that snoops on the Powerpath-2 bus in the usual fashion. The cache blocks can be in one of only two states: (i) invalid or (ii) modified exclusive. When a partial DMA write is first issued, the block is brought into this cache in modified exclusive state, invalidating its copy in all processors' caches. Subsequent partial DMA writes need not go to the Powerpath-2 bus if they hit in this cache, thus increasing the system bus efficiency. This modified exclusive block goes to the invalid state and supplies its contents on the system bus: (i) on any Powerpath-2 bus transaction accessing this block; (ii) when another partial DMA write causes this cache block to be replaced; and (iii) on any HIO bus read transaction that accesses this block. While DMA reads could have also used this four-block cache, the designers felt that partial DMA reads were rare, and the gains from such an optimization would have been minimal.

The mapping RAM in the system-bus interface provides general-purpose address translation for I/O devices. For example, it may be used to map small address spaces such as VME-24 or VME-32 into the 40-bit physical address space of the Powerpath-2 bus. Two types of mapping are sup-

ported: one level and two level. One-level mappings simply return one of the 8K entries in the mapping RAM, where by convention each entry maps 2 Mbytes of physical memory. In the two-level scheme, the map entry points to the page tables in main memory. However, each 4 KByte page has its own entry in the second-level table, so virtual pages can be arbitrarily mapped to physical pages. Note that PIOs face a similar translation problem, when going down to the I/O devices. Such translation is not done using the mapping RAM, but is directly handled by the personality ASIC interface chips.

The final issue that we explore for I/O is flow control. All requests proceeding from the I/O interfaces/devices to the Powerpath-2 system bus are implicitly flow controlled. For example, the HIO interface will not issue a read on the Powerpath-2 bus unless it has buffer space reserved for the response. Similarly, the HIO arbiter will not grant the bus to a requestor unless the system interface has room to accept the transaction. In the other direction, from the processors to I/O devices, however, PIOs can arrive unsolicited and they need to be explicitly flow controlled.

The flow control solution used in the Challenge system is to make the PIOs be solicited. After reset, HIO interface chips (e.g., HIO-VME, HIO-HPPI) signal their available PIO buffer space to the system-bus interface using special requests called IncPIO. The system-bus interface maintains this information in a separate counter for each HIO device. Every time a PIO is sent to a particular device, the corresponding count is decremented. Every time that device retires a PIO, it issues another IncPIO request to increment its counter. If the system bus interface receives a PIO for a device that has no buffer space available, it rejects (NACKs) that request on the Powerpath-2 bus and it must be retried later.

6.6.4 SGI Challenge Memory System Performance

The access time for various levels of the SGI Challenge memory system can be determined using the simple read microbenchmark from Chapter 3. Recall, the microbenchmark measures the average access time in reading elements of an array of a given size with a certain stride. Figure 6-17 shows the read access time for a range of sizes and strides. Each curve shows the average access time for a given size as a function of the stride. Arrays smaller than 32 KB fit entirely in the first level cache. Level two cache accesses have an access time of roughly 75 ns, and the inflection point shows that the transfer size is 16 bytes. The second bump shows the additional penalty of roughly 140 ns for a TLB miss, and the page size in 8 KB. With a 2 MB array accesses miss in the L2 cache, and we see that the combination of the L2 controller, Powerpath bus protocol and DRAM access result in an access time of roughly 1150 ns. The minimum bus protocol of 13 cycles at 50 MHz accounts for 260 ns of this time. TLB misses add roughly 200 ns. The simple ping-pong microbenchmark, in which a pair of nodes each spin on a flag until it indicates

their turn and then set the flag to signal the other shows a round-trip time of 6.2µs, a little less than four memory accesses.

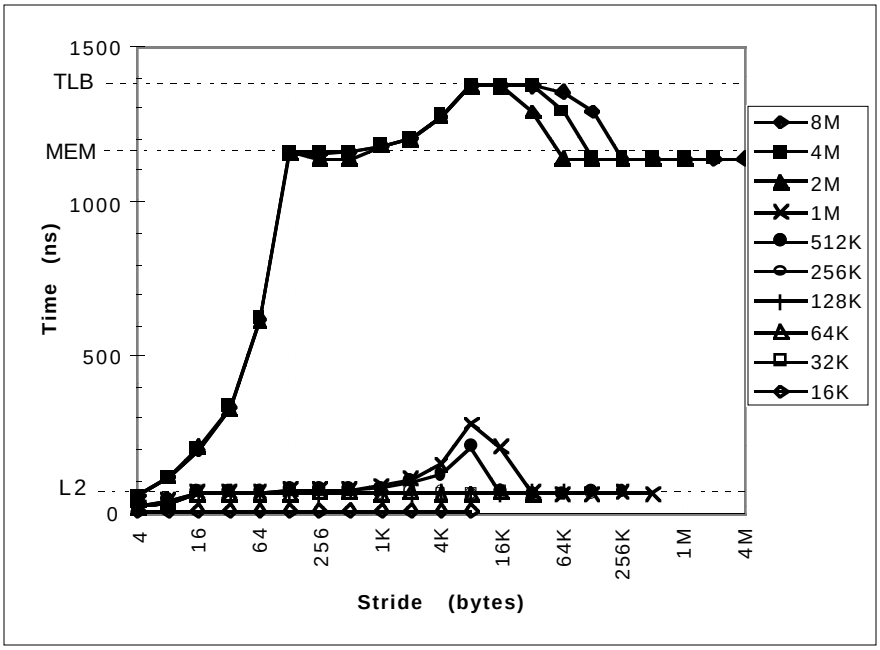

Figure 6-17 Read microbenchmark results for the SGI Challenge.

6.6.5 Sun Gigaplane System Bus

The Sun Gigaplane is a non-multiplexed, split-phase (or packet switched) bus with 256-bit data lines and 41-bit physical addresses, clocked at 83.5 MHz. It is a *centerplane design*, rather than a backplane, so cards plug into both sides of it. The total length of the bus is 18", so eight boards can plug into each side with 2" of cooling space between boards and 1" spacing between connectors. In sharp contrast to the SGI Challenge Powerpath-2 bus, the bus can support up to 112 outstanding transactions, including up to 7 from each board, so it is designed for devices that can sustain multiple outstanding transactions, such as lock-up free caches. The electrical and mechanical design allows for live insertion (hot plug) of processing and I/O modules.

Looking at the bus in more detail, it consists of 388 signals: 256 data, 32 ECC, 43 address (with parity), 7 id tag, 18 arbitration, and a number of configuration signals. The electrical design allows for turn-around with no dead cycles. Emphasis is placed on minimizing the latency of operations, and the protocol (illustrated in Figure 6-18) is quite different from that on the SGI Challenge. A novel collision-based arbitration technique is used to avoid the cost of bus arbitration. When a requestor arbitrates for the address bus, if the address bus is not scheduled to be in use from the previous cycle, it speculatively drives its request on the address bus. If there are no other requestors in that cycle, it wins arbitration and has already passed the address, so it can continue with the remainder of the transaction. If there is an address collision, the requestor that wins arbitration simply drives the address again in the next cycle, as it would with conventional arbitration. The 7-bit tag associated with the request is presented in the following cycle. The snoop state is associated with the address phase, not the data phase. Five cycles after the address, all

boards assert their snoop signals (shared, owned, mapped, and ignore). In the meantime, the board responsible for the memory address (the home board) can request the data bus three cycles after the address, before the snoop result. The DRAM access can be started speculatively, as well. When home board wins arbitration, it must assert the tag two cycles later, informing all devices of the approaching data transfer. Three cycles after driving the tag and two cycles before the data, the home board drives a status signal, which will indicate that the data transfer is cancelled if some cache owns the block (as detected in the snoop state). The owner places the data on the bus by arbitrating for the data bus, driving the tag, and driving the data. Figure 6-18 shows a second read transaction, which experiences a collision in arbitration, so the address is supplied in the conventional slot, and cache ownership, so the home board cancels its data transfer.

Like the SGI Challenge, invalidations are ordered by the BusRdX transactions on the address bus and handled in FIFO fashion by the cache subsystems, thus no acknowledgment of invalidation completion is required. To maintain sequential consistency, is still necessary to gain arbitration for the address bus before allowing the writing processor to proceed with memory operations past the write[1].

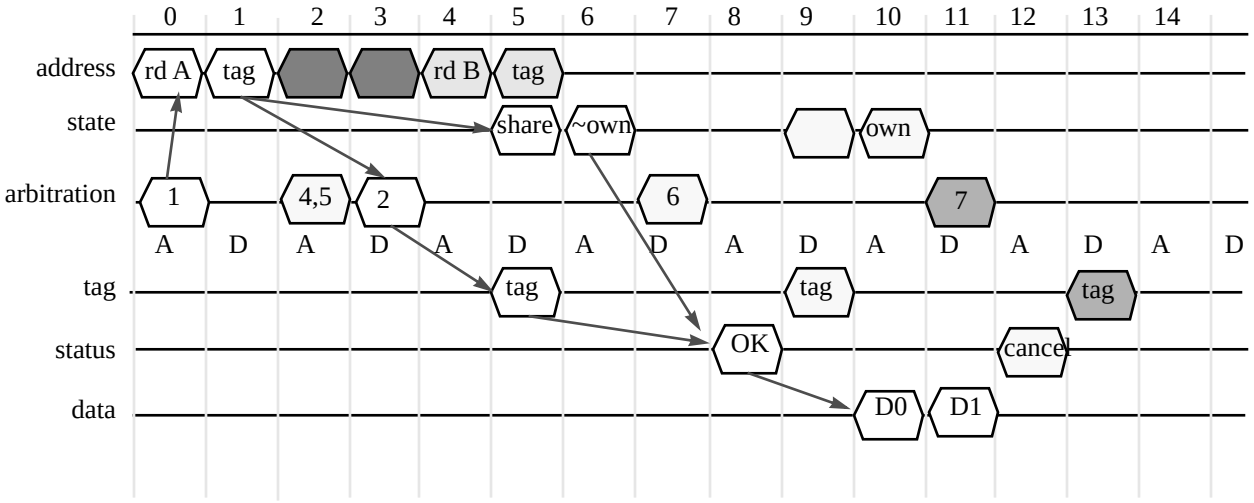

Figure 6-18 Sun Gigaplane Signal Timing for BusRd with fast address arbitration

Board 1 initiates a read transaction with fast arbitration, which is responded to by home board 2. Boards 4 and 5 collide during arbitration, board 4 wins, and initiates a read transaction. Home board 6 arbitrates for the data bus and then cancels its response. Eventually the owning cache, board 7, responds with the data. Board 5 retry is not shown.

6.6.6 Sun Processor and Memory Subsystem

In the Sun Enterprise, each processing board has two processors, each with external L2 caches, and two banks of memory connected through a cross-bar, as shown in Figure 6-19. Data lines within the UltraSPARC module are buffered to drive an internal bus, called the UPA (universal port architecture) with an internal bandwidth of 1.3 GB/s. A very wide path to memory is pro-

1. The SPARC V9 specification weakens the consistency model in this respect to allow the processor to employ write buffers, which we discuss in more depth in Chapter 6.

vided so that a full 64 byte cache block can be read in a singe memory cycle, which is two bus cycles in length. The address controller adapts the UPA protocol to the Gigaplane protocol, realizes the cache coherency protocol, provides buffering, and tracks the potentially large number of outstanding transactions. It maintains a set of duplicate tags (state and address, but no data) for the L2 cache. Although the UltraSPARC implements a 5-state MOESI protocol, the D-tags maintain an approximation to the state: owned, shared, invalid. It essentially combines states which are handled identically at the Gigaplane level. In particular, it needs to know if the L2 cache has a block and if that block is the only block in a cache. It does not need to know if that block is clean or dirty. For example, on a BusRd the block will need to be flushed onto the bus if it is in the L2 cache as modified, owned (flushed since last modified), or exclusive (not shared when read and not modified since), thus the D-tags represent only 'owned'. This has the advantage that the address controller need not be informed when the UltraSPARC elevates a block from exclusive to modified. It will be informed of a transition from invalid, shared, or owned to modified, because it needs to initiate bus transaction.

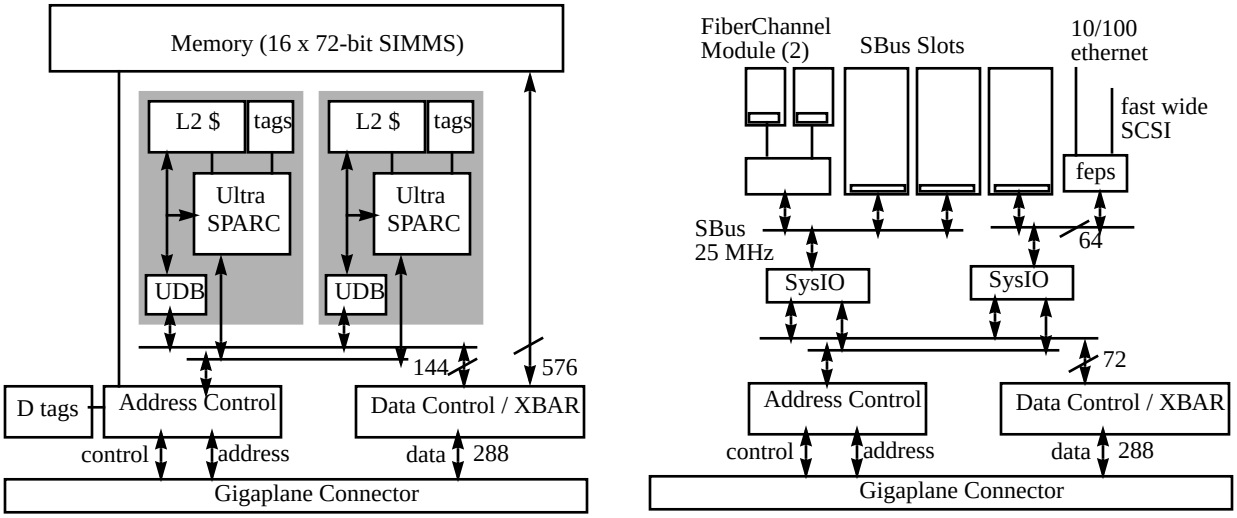

Figure 6-19 Organization of the Sun Enterprise Processing and I/O Board

Processing board contains two UltraSPARC modules with L2 caches on an internal bus, and two wide memory banks interfaced to the system bus through two ASICs. The Address Controller adapts between the two bus protocols and implements the cache coherency protocol. The Data Controller is essentially a cross bar. The I/O board uses the same two ASICs to interface to two I/O controllers. The SysIO asics essentially appear to the bus as a one block cache. On the other side, they support independent I/O busses and interfaces to FiberChannel, ethernet, and SCSI.

6.6.7 Sun I/O Subsystem

The Enterprise I/O board uses the same bus interface ASICS as the processing board. The internal bus is only half as wide and there is no memory path. The I/O boards only do cache block transactions, just like the processing boards, in order to simplify the design of the main bus. The SysI/O ASICs implement a single block cache on behalf of the I/O devices. Two independent 64-bit 25 MHz SBus are supported. One of these supports two dedicated FiberChannel modules providing a redundant, high bandwidth interconnect to large disk storage arrays. The other provides dedicated ethernet and fast wide SCSI connections. In addition, three SBUS interface cards can be plugged into the two busses to support arbitrary peripherals, including a 622 Mb/s ATM inter-

face. The I/O bandwidth, the connectivity to peripherals, and the cost of the I/O subsystem scales with the number of I/O cards.

6.6.8 Sun Enterprise Memory System Performance

The access time for various level of the Sun Enterprise via the read microbenchmark is shown in Figure 6-20. Arrays of 16 KB or less fit entirely in the first level cache. Level two cache accesses have an access time of roughly 40 ns, and the inflection point shows that the transfer size is 16 bytes. With a 1 MB array accesses miss the L2 cache, and we see that the combination of the L2 controller, bus protocol and DRAM access result in an access time of roughly 300 ns. The minimum bus protocol of 11 cycles at 83.5 MHz accounts for 130 ns of this time. TLB misses add roughly 340 ns. The machine has a software TLB handler. The simple ping-pong microbenchmark, in which a pair of nodes each spin on a flag until it indicates their turn and then set the flag to signal the other shows a round-trip time of 1.7 µs, roughly five memory accesses.

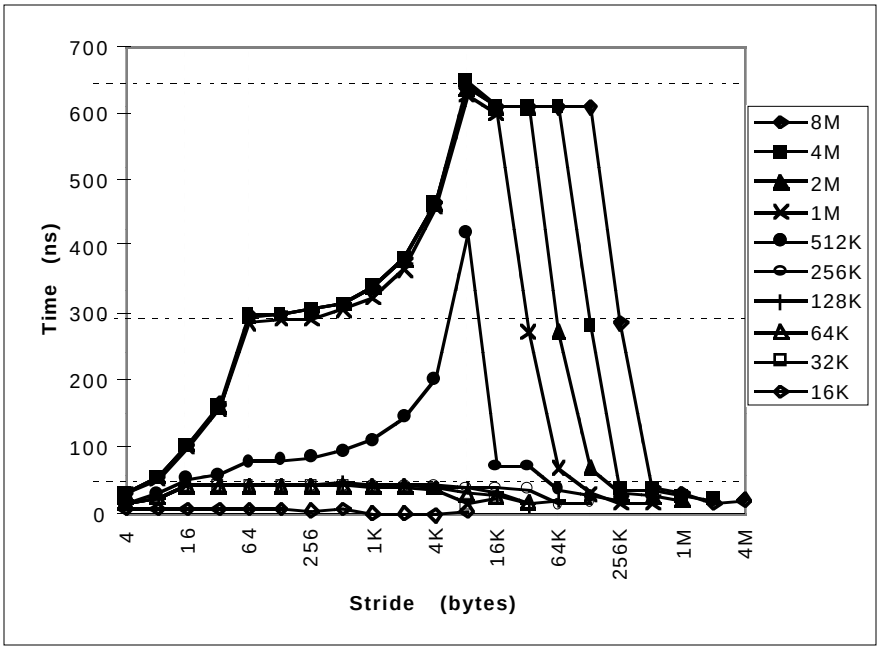

Figure 6-20 Read Microbenchmark results for the Sun Enterprise.

6.6.9 Application Performance

Having understood the machines and their microbenchmark performance, let us examine the performance obtained on our parallel applications. Absolute performance for commercial machines is not presented in this book; instead, the focus is on performance improvements due to parallelism. Let us first look at application speedups and then at scaling, using only the SGI Challenge for illustration.

Application Speedups

Figure 6-21 shows the speedups obtained on our six parallel programs, for two data set sizes each. We can see that the speedups are quite good for most of the programs, with the exception of the Radix sorting kernel. Examining the breakdown of execution time for the sorting kernel shows that the vast majority of the time is spent stalled on data access. The shared bus simply gets swamped with the data and coherence traffic due to the permutation phase of the sort, and the resulting contention destroys performance. The contention also leads to severe load imbalances in data access time and hence time spent waiting at global barriers. It is unfortunately not alleviated much by increasing the problem size, since the communication to computation ratio in the permutation phase is independent of problem size. The results shown are for a radix value of 256, which delivers the best performance over the range of processor counts for both problem sizes. Barnes-Hut, Raytrace and Radiosity speed up very well even for the relatively small input problems used. LU does too, and the bottleneck for the smaller problem at 16 processors is primarily load imbalance as the factorization proceeds along the matrix. Finally, the bottleneck for the small Ocean problem size is both the high communication to computation ratio and the imbalance this generates since some partitions have fewer neighbors than others. Both problems are alleviated by running larger data sets.

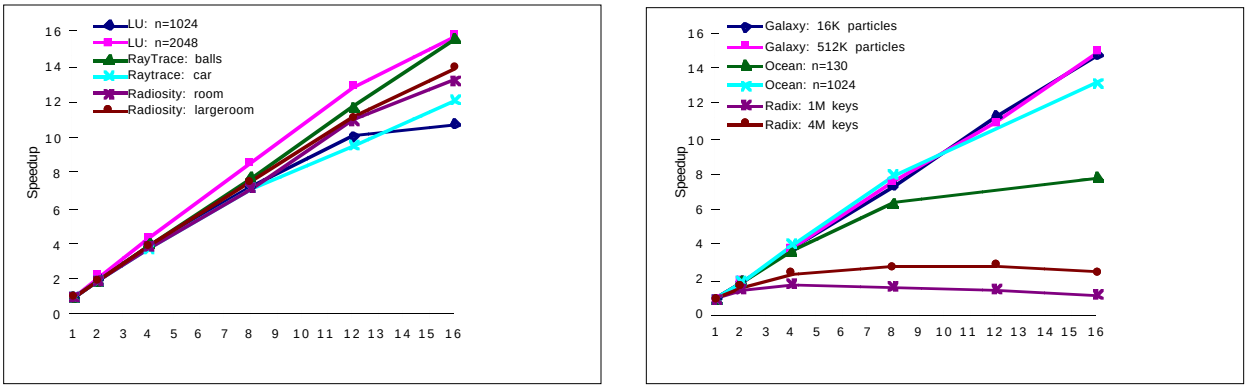

Figure 6-21 Speedups for the parallel applications on the SGI Challenge.

The block size for the blocked LU factorization is 32-by-32.

Scaling

Let us now examine the impact of scaling for a few of the programs. Following the discussion of Chapter 4, we look at the speedups under the different scaling models, as well as at how the work done and the data set size used change. Figure 6-22 shows the results for the Barnes-Hut and Ocean applications. “Naive” time-constrained (TC) or memory-constrained (MC) refers to scaling on the number of particles or grid length (n) without changing the other application parameters (accuracy or the number of time steps). It is clear that the work done under realistic MC scaling grows much faster than linearly in the number of processors in both applications, so the parallel execution time grows very quickly. The number of particles or the grid size that can be simulated under TC scaling grows much more slowly than under MC scaling, and also much more slowly than under naive TC where n is the only application parameter scaled. Scaling the

other application parameters causes the work done and execution time to increase, leaving much less room to grow n. Data set size is controlled primarily by *n*, which explains its scaling trends.

The speedups under different scaling models are measured as described in Chapter 4. Consider the Barnes-Hut galaxy simulation, where the speedups are quite good for this size of machine under all scaling models. The differences can be explained by examining the major performance factors. The communication to computation ratio in the force calculation phase depends primarily on the number of particles. Another important factor that affects performance is the relative amount of work done in the force calculation phase, which speeds up well, to the amount done in the tree-building phase which does not. This tends to increase with greater accuracy in force computation, i.e. smaller θ. However, smaller θ (and to a lesser extent greater n) increase the working set size [SHG93], so the scaled problem that changes θ may have worse first-level cache behavior than the baseline problem (with a smaller n and larger θ) on a uniprocessor. These factors can be used to explain why naive TC scaling yields better speedups than realistic TC scaling. The working sets behave better, and the communication to computation ratio is more favorable since n grows more quickly when θ and Δt are not scaled.

The speedups for Ocean are quite different under different models. Here too, the major controlling factors are the communication to computation ratio, the working set size, and the time spent in different phases. However, all the effects are much more strongly dependent on the grid size relative to number of processors. Under MC scaling, the communication to computation ratio does not change with the number of processors used, so we might expect the best speedups. However, as we scale two effects become visible. First, conflicts across grids in the cache increase as a processor's partitions of the grids become further apart in the address space. Second, more time is spent in the higher levels of the multigrid hierarchy in the solver, which have worse parallel performance. The latter effect turns out to be alleviated when accuracy and time-step interval are refined as well, so realistic MC scales a little better than naive MC. Under naive TC scaling, the growth in grid size is not fast enough to cause major conflict problems, but good enough that communication to computation ratio diminishes significantly, so speedups are very good. Realistic TC scaling has a slower growth of grid size and hence improvement in communication to computation ratio, and hence lower speedups. Clearly, many effects play an important role in performance under scaling, and which scaling model is most appropriate for an application affects the results of evaluating a machine.

6.7 Extending Cache Coherence

The techniques for achieving cache coherence extend in many directions. This examines a few important directions: scaling down with shared caches, scaling in functionality with virtually indexed caches and translation lookaside buffers (TLBs), and scaling up with non-bus interconnects.

6.7.1 Shared-Cache Designs

Grouping processors together to share a level of the memory hierarchy (e.g., the first or the second-level cache) is a potentially attractive option for shared-memory multiprocessors, especially as we consider designs with multiple processors on a chip. Compared with each processor having its own memory at that level of the hierarchy, it has several potential benefits. The benefits—like

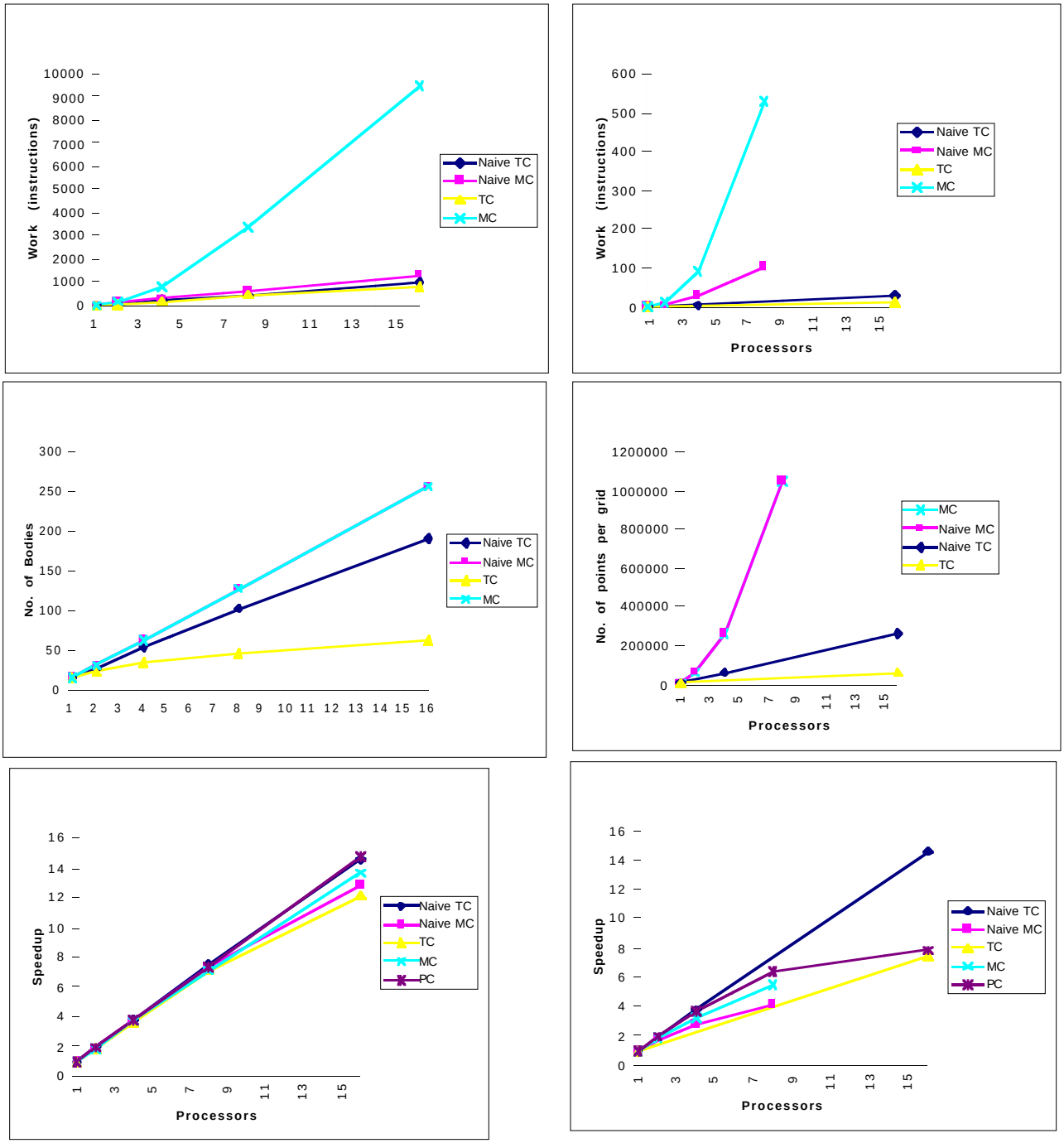

Figure 6-22 Scaling results for Barnes-Hut (left) and Ocean (right) on the SGI Challenge.

The graphs show the scaling of work done, data set size, and speedups under different scaling models. PC, TC, and MC refer to problem constrained, time constrained and memory constrained scaling, respectively. The top set of graphs shows that the work needed to solve the problem grows very quickly under realistic MC scaling for both applications. The middle set of graphs shows that the data set size that can be run grows much more quickly under MC or naive TC scaling than under realistic TC scaling. The impact of scaling model on speedup is much larger for Ocean than for Barnes-Hut, primarily because the communication to computation ratio is much more strongly dependent on problem size and number of processors in Ocean.

the later drawbacks—are encountered when sharing at any level of the hierarchy, but are most extreme when it is the first-level cache that is shared among processors. The benefits of sharing a cache are:

- It eliminates the need for cache-coherence at this level. In particular, if the first-level cache is shared then there are no multiple copies of a cache block and hence no coherence problem whatsoever.
- It reduces the latency of communication that is satisfied within the group. The latency of communication between processors is closely related to the level in the memory hierarchy where they meet. When sharing the first-level cache, communication latency can be as low as 2-10 clock cycles. The corresponding latency when processors meet at the main-memory level is usually many times larger (see the Challenge and Enterprise case studies). The reduced latency enables finer-grained sharing of data between tasks running on the different processors.
- Once one processor misses on a piece of data and brings it into the shared cache, other processors in the group that need the data may find it already there and will not have to miss on it at that level. This is called prefetching data across processors. With private caches each processor would have to incur a miss separately. The reduced number of misses reduces the bandwidth requirements at the next level of the memory and interconnect hierarchy.
- It allows more effective use of long cache blocks. Spatial locality is exploited even when different words on a cache block are accessed by different processors in a group. Also, since there is no cache coherence within a group at this level there is also no false sharing. For example, consider a case where two processors P1 and P2 read and write every alternate word of a large array, and think about the differences when they share a first-level cache and when they have private first-level caches.
- The working sets (code or data) of the processors in a group may overlap significantly, allowing the size of shared cache needed to be smaller than the combined size of the private caches if each had to hold its processor's entire working set.
- It increases the utilization of the cache hardware. The shared cache does not sit idle because one processor is stalled, but rather services other references from other processors in the group.
- The grouping allows us to effectively use emerging packaging technologies, such as multi-chip-modules, to achieve higher computational densities (computation power per unit area).

The extreme form of cache sharing is the case in which all processors share a first level cache, below which is a shared main memory subsystem. This completely eliminates the cache coherence problem. Processors are connected to the shared cache by a switch. The switch could even be a bus but is more likely a crossbar to allow cache accesses from different processors to proceed in parallel. Similarly, to support the high bandwidth imposed by multiple processors, both the cache and the main memory system are interleaved.

An early example of a shared-cache architecture is the Alliant FX-8 machine, designed in the early 1980s. An Alliant FX-8 contained up to 8 custom processors. Each processor was a pipelined implementation of the 68020 instruction set, augmented with vector instructions, and had a clock cycle of 170ns. The processors were connected using a crossbar to a 512Kbyte, 4-way interleaved cache. The cache had 32byte blocks, and was writeback, direct-mapped, and lock-up free allowing each processor to have two outstanding misses. The cache bandwidth was eight 64-bit words per instruction cycle. The interleaved main memory subsystem had a peak bandwidth of 192 Mbytes/sec.

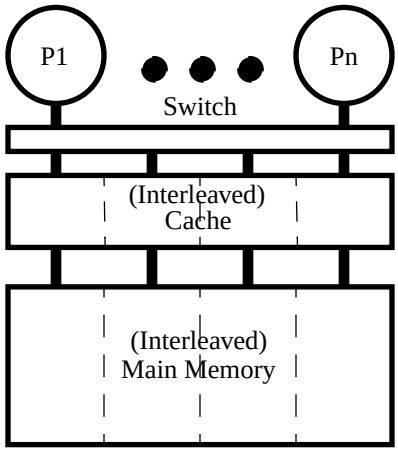

Figure 6-23 Generic architecture for a shared-cache multiprocessor.

The interconnect is placed between the processors and the first-level cache. Both the cache and the memory system may be interleaved to provide high bandwidth.

A somewhat different early use of the shared-cache approach was exemplified by the Encore Multimax, a contemporary of the FX-8. The Multimax was a snoopy cache-coherent multiprocessor, but each private cache supported two processors instead of one (with no need for coherence within a pair). The motivation for Encore at the time was to lower the cost of snooping hardware, and to increase the utilization of the cache given the very slow, multiple-CPI processors.

Today, shared first-level caches are being investigated for single-chip multiprocessors, in which four-to-eight multiprocessors share an on-chip first-level cache. These can be used in themselves as multiprocessors, or as the building-blocks for larger systems that maintain coherence among the single-chip shared-cache groups. As technology advances and the number of transistors on a chip reaches several tens or hundreds of millions, this approach becomes increasingly attractive. Workstations using such chips will be able to offer very high-performance for workloads requiring either fine-grain or coarse-grain parallelism. The question is whether this is a more effective approach or one that uses the hardware resources to build more complex processors.

However, sharing caches, particularly at first level, has several disadvantages and challenges:

- The shared cache has to satisfy the bandwidth requirements from multiple processors, restricting the size of a group. The problem is particularly acute for shared first-level caches, which are therefore limited to very small numbers of processors. Providing the bandwidth needed is one of the biggest challenges of the single-chip multiprocessor approach.
- The hit latency to a shared cache is usually higher than to a private cache at the same level, due to the interconnect in between. This too is most acute for shared first-level caches, where the imposition of a switch between the processor and the first-level cache means that either the machine clock cycle is elongated or that additional delay-slots are added for load instructions in the processor pipeline. The slow down due to the former is obvious. While compilers have some capability to schedule independent instructions in load delay slots, the success depends on the application. Particularly for programs that don’t have a lot of instruction-level

parallelism, some slow down is inevitable. The increased hit latency is aggravated by contention at the shared cache, and correspondingly the miss latency is also increased by sharing.

- For the above reasons, the design complexity for building an effective system is higher.
- Although a shared cache need not be as large as the sum of the private caches it replaces, it is still much larger and hence slower than an individual private cache. For first-level caches, this too will either elongate the machine clock cycle or lead to multiple processor-cycle cache access times.
- The converse of overlapping working sets (or constructive interference) is the performance of the shared cache being hurt due to cache conflicts across processor reference streams (destructive interference). When a shared-cache multiprocessor is used to run workloads with little data sharing, for example a parallel compilation or a database/transaction processing workload, the interference in the cache between the data sets needed by the different processors can hurt performance substantially. In scientific computing where performance is paramount, many programs try to manage their use of the per-processor cache very carefully, so that the many arrays they access do not interfere in the cache. All this effort by the programmer or compiler can easily be undone in a shared-cache system.
- Finally, shared caches today do not meet the trend toward using commodity microprocessor technology to build cost-effective parallel machines, particularly shared first-level caches.

Since many microprocessors already provide snooping support for first-level caches, an attractive approach may be to have private first-level caches and a shared second-level cache among groups of processors. This will soften both the benefits and drawbacks of shared first-level caches, but may be a good tradeoff overall. The shared cache will likely be large to reduce destructive interference. In practice, packaging considerations will also have a very large impact on decisions to share caches.

6.7.2 Coherence for Virtually Indexed Caches

Recall from uniprocessor architecture the tradeoffs between physically and virtually indexed caches. With physically indexed first-level caches, for cache indexing to proceed in parallel with address translation requires that the cache be either very small or very highly associative, so the bits that do not change under translation (log_2*(page_size)* bits or a few more if page coloring is used) are sufficient to index into it [HeP90]. As on-chip first-level caches become larger, virtually indexed caches become more attractive. However, these have their own, familiar problems. First, different processors may use the same virtual address to refer to unrelated data in different address spaces. This can be handled by flushing the whole cache on a context switch or by associating address space identifier (ASID) tags with cache blocks in addition to virtual address tags. The more serious problem for cache coherence is synonyms: distinct virtual pages, from the same or different processes, pointing to the same physical page for sharing purposes. With virtually addressed caches, the same physical memory block can be fetched into two distinct blocks at different indices in the cache. As we know, this is a problem for uniprocessors, but the problem extends to cache coherence in multiprocessors as well. If one processor writes the block using one virtual address synonym and another reads it using a different synonym, then by simply putting virtual addresses on the bus and snooping them the write to the shared physical page will not become visible to the latter processor. Putting virtual addresses on the bus also has another drawback, requiring I/O devices and memory to do virtual to physical translation since they deal with physical addresses. However, putting physical addresses on the bus seems to require reverse

translation to look up the caches during a snoop, and this does not solve the synonym coherence problem anyway.

There are two main software solutions to avoiding the synonym problem: forcing synonyms to be the same in the bits used to index the cache if these are more than $log_2(page\_size)$ (i.e. forcing them to have the same page color), and forcing processes to use the same shared virtual address when referring to the same page (as in the SPUR research project [HEL+86]).

Sophisticated cache designs have also been proposed to solve the synonym coherence problem in hardware [Goo87]. The idea is to use virtual addresses to look up the cache on processor accesses, and to put physical addresses on the bus for other caches and devices to snoop. This requires mechanisms to be provided for the following: (i) if a lookup with the virtual address fails, then to look up the cache with the physical address (which is by now available) as well in case it was brought in by a synonym access, (ii) to ensure that the same physical block is never in the same cache under two different virtual addresses at the same time, and (iii) to convert a snooped physical address to a effective virtual address to look up the snooping cache. One way to accomplish these goals is for caches to maintain both virtual and physical tags (and states) for their cached blocks, indexed by virtual and physical addresses respectively, and for the two tags for a block to point to each other (i.e. to store the corresponding physical and virtual indices, respectively, see Figure 6-24). The cache data array itself is indexed using the virtual index (or the pointer from the physical tag entry, which is the same). Let's see at a high level how this provides the above mechanisms.

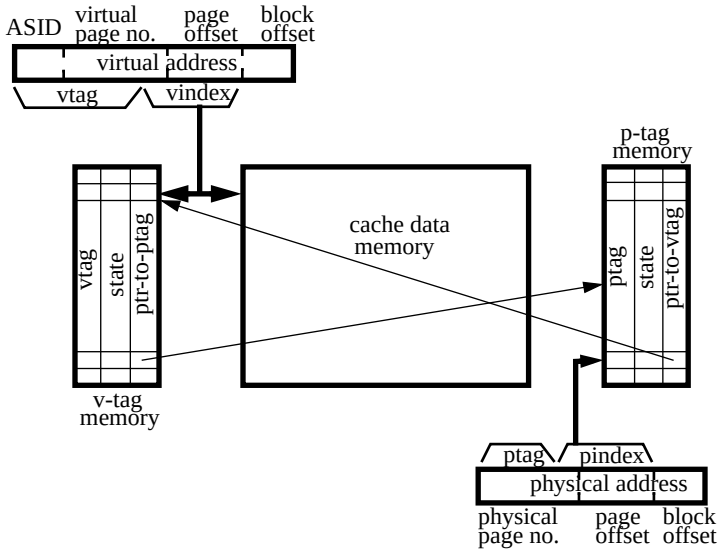

Figure 6-24 Organization of a dual-tagged virtually-addressed cache.

The v-tag memory on the left services the CPU and is indexed by virtual addresses. The p-tag memory on the right is used for bus snooping, and is indexed by physical addresses. The contents of the memory block are stored based on the index of the v-tag. Corresponding p-tag and v-tag entries point to each other for handling updates to cache.

A processor looks up the cache with its virtual address, and at the same time the virtual to physical translation is done by the memory management unit in case it is needed. If the lookup with the virtual address succeeds, all is well. If it fails, the translated physical address is used to look

up the physical tags, and if this hits the block is found through the pointer in the physical tag. This achieves the first goal. A virtual miss but physical hit detects the possibility of a synonym, since the physical block may have been brought in via a different virtual address. In a direct-mapped cache, it must have, and let us assume a direct-mapped cache for concreteness. The pointer contained in the physical tags now points to a different block in the cache array (the synonym virtual index) than does the virtual index of the current access. We need to make the current virtual index point to this physical data, and reconcile the virtual and physical tags to remove the synonym. The physical block, which is currently pointed to by the synonym virtual index, is copied over to replace the block pointed to by the current virtual index (which is written back if necessary), so references to the current virtual index will hereafter hit right away. The former cache block is rendered invalid or inaccessible, so the block is now accessible only through the current virtual index (or through the physical address via the pointer in the physical tag), but not through the synonym. A subsequent access to the synonym will miss on its virtual address lookup and will have to go through this procedure. Thus, a given physical block is valid only in one (virtually indexed) location in the cache at any given time, accomplishing the second goal. Note that if both the virtual and physical address lookups fail (a true cache miss), we may need up to two write-backs. The new block brought into the cache will be placed at the index determined from the virtual (not physical) address, and the virtual and physical tags and states will be suitably updated to point to each other.

The address put on the bus, if necessary, is always a physical address, whether for a writeback, a read miss, or a read-exclusive or upgrade. Snooping with physical addresses from the bus is easy. Explicit reverse translation is not required, since the information needed is already there. The physical tags are looked up to check for the presence of the block, and the data are found from the pointer (corresponding virtual index) it contains. If action must be taken, the state in the virtual tag pointed to by the physical tag entry is updated as well. Further details of how such a cache system operates can be found in [Goo87]. This approach has also been extended to multi-level caches, where it is even more attractive: the L1 cache is virtually-tagged to speed cache access, while the L2 cache is physically tagged [WBL89].

6.7.3 Translation Lookaside Buffer Coherence

A processor's *translation lookaside buffer* (TLB) is a cache on the page table entries (PTEs) used for virtual to physical address translation. A PTE can come to reside in the TLBs of multiple processors, due to actual sharing or process migration. PTEs may be modified—for example when the page is swapped out or its protection is changed—leading to direct analog of the cache coherence problem.

A variety of solutions have been used for TLB coherence. Software solutions, through the operating system, are popular, since TLB coherence operations are much less frequent than cache coherence operations. The exact solutions used depend on whether PTEs are loaded into the TLB directly by hardware or through software, and on several other of the many variables in how TLBs and operating systems are implemented. Hardware solutions are also used by some systems, particularly when TLB operations are not visible to software. This section provides a brief overview of four approaches to TLB coherence: virtually addressed caches, software TLB shootdown, address space identifiers (ASIDs), and hardware TLB coherence. Further details can be found in [TBJ+99,Ros89,Tel90] and the papers referenced therein.

TLBs, and hence the TLB coherence problem, can be avoided by using virtually addressed caches. Address translation is now needed only on cache misses, so particularly if the cache miss rate is small we can use the page tables directly. Page table entries are brought into the regular data cache when they are accessed, and are therefore kept coherent by the cache coherence mechanism. However, when a physical page is swapped out or its protection changed, this is not visible to the cache coherence hardware, so they must be flushed from the virtually addressed caches of all processors by the operating system. Also, the coherence problem for virtually addressed caches must be solved. This approach was explored in the SPUR research project [HEL+86, WEG+86].

A second approach is called TLB shootdown. There are many variants that rely on different (but small) amounts of hardware support, usually including support for interprocessor interrupts and invalidation of TLB entries. The TLB coherence procedure is invoked on a processor, called the initiator, when it makes changes to PTEs that may be cached by other TLBs. Since changes to PTEs must be made by the operating system, it knows which PTEs are being changed and which other processors might be caching them in their TLBs (conservatively, since entries may have been replaced). The OS kernel locks the PTEs being changed (or the relevant page table sections, depending on the granularity of locking), and sends interrupts to other processors that it thinks have copies. The recipients disable interrupts, look at the list of page-table entries being modified (which is in shared memory) and locally invalidate those entries from their TLB. The initiator waits for them to finish, perhaps by polling shared memory locations, and then unlocks the page table sections. A different, somewhat more complex shootdown algorithm is used in the Mach operating system [BRG+89].

Some processor families, most notably the MIPS family from Silicon Graphics, used software-loaded rather than hardware-loaded TLBs, which means that the OS is involved not only in PTE modifications but also in loading a PTE into the TLB on a miss [Mip91]. In these cases, the coherence problem for process-private pages due to process migration can be solved using a third approach, that of ASIDs, which avoids interrupts and TLB shootdown. Every TLB entry has an ASID field associated with it, used just like in virtually addressed caches to avoid flushing the entire cache on a context switch. ASIDs here are like tags allocated dynamically by the OS, using a free pool to which they are returned when TLB entries are replaced; they are not associated with processes for their lifetime. One way to use the ASIDs, used in the Irix 5.2 operating system, is as follows. The OS maintains an array for each process that tracks the ASID assigned to that process on each of the processors in the system. When a process modifies a PTE, the ASID of that process for all other processors is set to zero. This ensures that when the process is migrated to another processor, it will find its ASID to be zero there so and the kernel will allocate it a new one, thus preventing use of stale TLB entries. TLB coherence for pages truly shared by processes is performed using TLB shootdown.

Finally, some processor families provide hardware instructions to invalidate other processors' TLBs. In the PowerPC family [WeS94] the "TLB invalidate entry" (`tlbie`) instruction broadcasts the page address on the bus, so that the snooping hardware on other processors can automatically invalidate the corresponding TLB entries without interrupting the processor. The algorithm for handling changes to PTEs is simple: the operating system first makes changes to the page table, and then issues a `tlbie` instruction for the changed PTEs. If the TLB is not software-loaded (it is not in the PowerPC) then the OS does not know which other TLBs might be caching the PTE so the invalidation must be broadcast to all processors. Broadcast is well-suited to a bus, but undesirable for the more scalable systems with distributed networks that will be discussed in subsequent chapters.

6.7.4 Cache coherence on Rings

Since the scale of bus-based cache coherent multiprocessors is fundamentally limited by the bus, it is natural to ask how it could be extended to other less limited interconnects. One straightforward extension of a bus is a ring. Instead of a single set of wires onto which all modules are attached, each module is attached to two neighboring modules. A ring is an interesting interconnection network from the perspective of coherence, since it inherently supports broadcast-based communication. A transaction from one node to another traverses link by link down the ring, and since the average distance of the destination node is half the length of the ring, it is simple and natural to let the acknowledgment simply propagate around the rest of the ring and return to the sender. In fact, the natural way to structure the communication in hardware is to have the sender place the transaction on the ring, and other nodes inspect (snoop) it as it goes by to see if it is relevant to them. Given this broadcast and snooping infrastructure, we can provide snoopy cache coherence on a ring even with physically distributed memory. The ring is a bit more complicated than a bus, since multiple transactions may be in progress around the ring simultaneously, and the modules see the transactions at different times and potentially in different order.

The potential advantage of rings over busses is that the short, point-to-point nature of the links allows them to be driven at very high clock rates. For example, the IEEE Scalable Coherent Interface (SCI) [Gus92] transport standard is based on 500 MHz 16-bit wide point-to-point links. The linear, point-to-point nature also allows the links to be extensively pipelined, that is, new bits can be pumped onto the wire by the source before the previous bits have reached the destination. This latter feature allows the links to be made long without affecting their throughput. A disadvantage of rings is that the communication latency is high, typically higher than that of buses, and grows linearly with the number of processors in the ring (on average, *p/2* hops need to be traversed before getting to the destination on a unidirectional ring, and half that on a bidirectional ring).

Since rings are a broadcast media snooping cache coherence protocols can be implemented quite naturally on them. An early ring-based snoopy cache-coherent machine was the KSR1 sold by Kendall Square Research. [FBR93]. More recent commercial offerings use rings as the second level interconnect to connect together multiprocessor nodes, such as the Sequent NUMA-Q and Convex's Exemplar family [Con93,TSS+96]. (Both of these systems use a "directory protocol" rather than snooping on the ring interconnect, so we will defer discussion of them until ** Chapter 9**, when these protocols are introduced. Also, in the Exemplar, the interconnect within a node is not a bus or a ring, but a richly-connected, low-latency crossbar, as discussed below.) The University of Toronto's Hector system [VSL+91, FVS92] is a ring-based research prototype.

Figure 6-25 illustrates the organization of a ring-connected multiprocessor. Typically rings are used with physically distributed memory, but the memory may still be logically shared. Each node consists of a processor, its private cache, a portion of the global main memory, and a ring interface. The interface to the ring consists of an input link from the ring, a set of latches organized as a FIFO, and an output link to the ring. At each ring clock cycle the contents of the latches are shifted forward, so the whole ring acts as a circular pipeline. The main function of the latches is to hold a passing transaction long enough so that the ring-interface can decide whether to forward the message to the next node or not. A transaction may be taken out of the ring by storing the contents of the latch in local buffer memory and writing an empty-slot indicator into that latch instead. If a node wants to put something on the ring, it waits for a an opportunity to fill a passing empty slot and fills it. Of course, it is desirable to minimize the number of latches in each interface, to reduce the latency of transactions going around the ring.

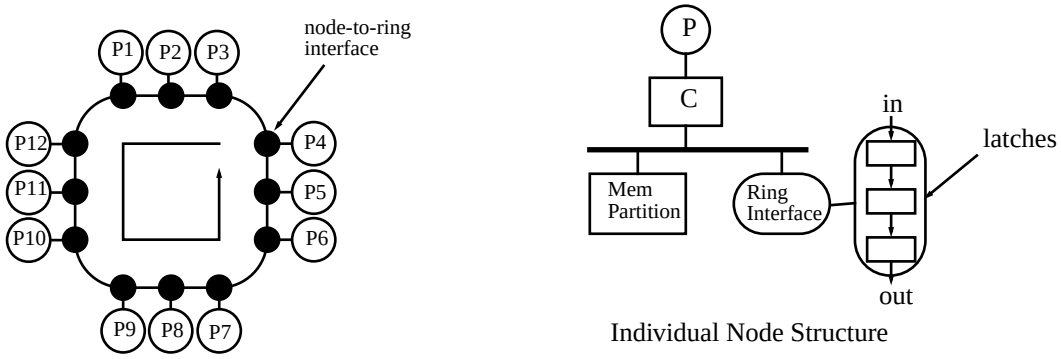

Figure 6-25 Organization of a single-ring multiprocessor.

The mechanism that determines when a node can insert a transaction on the ring, called the ring access control mechanism, is complicated by the fact that the data path of the ring is usually much narrower than size of the transactions being transferred on it. As a result, transactions need multiple consecutive slots on the ring. Furthermore, transactions (messages) on the ring can themselves have different sizes. For example, *request* messages are short and contain only the command and address, while *data reply* messages contain the contents of the complete memory block and are longer. The final complicating factor is that arbitration for access to the ring must be done in a distributed manner, since unlike in a bus there are no global wires.

Three main options have been used for access control (i.e., arbitration): token-passing rings, register-insertion rings, and slotted rings. In *token-passing rings* a special bit pattern, called a token, is passed around the ring, and only the node currently possessing it is allowed to transmit on the ring. Arbitration is easy, but the disadvantage is that only one node may initiate a transaction at a time even though there may be empty slots on the ring passing by another nodes, resulting in wasted bandwidth. *Register-insertion rings* were chosen for the IEEE SCI standard. Here, a bypass FIFO between the input and output stages of the ring is used to buffer incoming transactions (with backward flow-control to avoid overloading), while the local node is transmitting. When the local node finishes, the contents of bypass FIFO are forwarded on the output link, and the local node is not allowed to transmit until the bypass FIFO is empty. Multiple nodes may be transmitting at a time, and parts of the ring will stall when they are overloaded. Finally, in *slotted rings*, the ring is divided into transaction slots with labelled types (for different sized transactions such as requests and data replies), and these slots keep circulating around the ring. A processor ready to transmit a transaction waits until an empty slot of the required type comes by (indicated by a bit in the slot header), and then it inserts its message. A “slot” here really means a sequence of empty time slots, the length of the sequence depending on the type of message. In theory, the slotted ring restricts the utilization of the ring bandwidth by hard-wiring the mixture of available slots of different types, which may not match the actual traffic pattern for a given workload. However, for a given coherence protocol the mix of message types is reasonably well known and little bandwidth is wasted in practice [BaD93, BaD95].

While it may seem at first that broadcast and snooping wastes bandwidth on an interconnect such as a ring, in reality it is not so. A broadcast takes only twice as the average point-to-point message on a ring, since the latter between two randomly chosen nodes will traverse half the ring on

average. Also, broadcast is needed only for request messages (read-miss, write-miss, upgrade requests) which are all short; data reply messages are put on the ring by the source of the data and stop at the requesting node.

Consider a read miss in the cache. If the home of the block is not local, the read request is placed on the ring. If the home is local, we must determine if the block is dirty in some other node, in which case the local memory should not respond and a request should be placed on the ring. A simple solution is to place all misses on the ring, as on a bus-based design with physically distributed memory, such as the Sun Enterprise. Alternatively, a *dirty bit*, can be maintained for each block in home memory. This bit is turned ON if a block is cached in dirty state in some node other than the home. If the bit is on, the request goes on the ring. The read request now circles the ring. It is snooped by all nodes, and either the home or the dirty node will respond (again, if the home were not local the home node uses the dirty bit to decide whether or not it should respond to the request). The request and response transactions are removed from the ring when they reach the requestor. Write miss and write upgrade transactions also appear on the ring as requests, if the local cache state warrants that an invalidation request be sent out. Other nodes snoop these requests and invalidate their blocks if necessary. The return of the request to the requesting node serves as an acknowledgment. When multiple nodes attempt to write to the same block concurrently, the winner is the one that reaches the current *owner* of the block (home node if block is clean, else the dirty node) first; the other nodes are implicitly/explicitly sent negative acknowledgments (NAKs), and they must retry.

From an implementation perspective, a key difficulty with snoopy protocols on rings is the real-time constraints imposed: The snooper on the ring-interface must examine and react to all passing messages without excessive delay or internal queuing. This can be difficult for register-insertion rings, since many short request messages may be adjacent to each other in the ring. With rings operating at high speeds, the requests can be too close together for the snooper to respond to in a fixed time. The problem is simplified in slotted rings, where careful choice and placement of short request messages and long data response messages (the data response messages are point-to-point and do not need snooping) can ensure that request-type messages are not too close together [BaD95]. For example, slots can be grouped together in frames, and each frame can be organized to have request slots followed by response slots. Nonetheless, as with busses ultimately bandwidth on rings is limited by snoop bandwidth rather than raw data transfer bandwidth.

Coherence issues are handled as follows in snoopy ring protocols: (a) Enough information about the state in other nodes to determine whether to place a transaction on the ring is obtained from the location of the home, and from the dirty bit in memory if the block is locally allocated; (b) other copies are found through broadcast; and (c) communication with them happens simultaneously with finding them, through the same broadcast and snooping mechanism. Consistency is a bit trickier, since there is the possibility that processors at different points on the ring will see a pair of transactions on the ring in different orders. Using invalidation protocols simplifies this problem because writes only cause read-exclusive transactions to be placed on the ring and all nodes but the home node will respond simply by invalidating their copy. The home node can determine when conflicting transactions are on the ring and take special action, but this does increase the number of transient states in the protocol substantially.

6.7.5 Scaling Data Bandwidth and Snoop Bandwidth

There are several alternative ways to increase the bandwidth of SMP designs that preserve much of the simplicity of bus-based approaches. We have seen that with split-transaction busses, the arbitration, the address phase, and the data phase are pipelined, so each of them can go on simultaneously. Scaling data bandwidth is the easier part, the real challenge is scaling the snoop bandwidth.

Let's consider first scaling data bandwidth. Cache blocks are large compared to the address that describes them. The most straightforward to increase the data bandwidth is simple to make the data bus wider. We see this, for example, in the Sun Enterprise design which uses a 128-bit wide data bus. With this approach, a 32-byte block is transferred in only two cycles. The down side of this approach is cost; as the bus get wider it uses a larger connector, occupies more space on the board, and draws more power. It certainly pushes the limit of this style of design, since it means that a snoop operations, which needs to be observed by all the caches and acknowledged, must complete in only two cycles. A more radical alternative is to replace the data bus with a cross-bar, directly connecting each processor-memory module to every other one. It is only the address portion of the transaction that needs to be broadcast to all the nodes in order to determine the coherence operation and the data source, i.e., memory or cache. This approach is followed in the IBM PowerPC based RS6000 G30 multiprocessor. A bus is used for addresses and snoop results, but a cross-bar is used to move the actual data. The individual paths in the cross-bar need not be extremely wide, since multiple transfers can occur simultaneously.

A brute force way to scale bandwidth in a bus based system is simply to use multiple busses. In fact, this approach offers a fundamental contribution. In order to scale the snoop bandwidth beyond one coherence result per address cycle, there must be multiple simultaneous snoop operations. Once there are multiple address busses, the data bus issues can be handled by multiple data busses, cross-bars, or whatever. Coherence is easy. Different portions of the address space use different busses, typically each bus will serve specific memory banks, so a given address always uses the same bus. However, multiple address busses would seem to violate the critical mechanism used to ensure memory consistency – serialized arbitration for the address bus. Remember, however, that sequential consistency requires that there be a logical total order, not that the address events be in strict chronological order. A static ordering is assigned logically assigned to the sequence of busses. An address operation i logically preceeds j if it occurs before j in time or if they happen on the same cycle but i takes place on a lower numbered bus. This multiple bus approach is used in Sun SparcCenter 2000, which provided two split-phase (or packet switched) XDB busses, each identical to that used in the SparcStation 1000, and scales to 30 processors. The Cray CS6400 used four such busses and scales to 64 processors. Each cache controller snoops all of the busses and responds according to the cache coherence protocol. The Sun Enterprise 10000 combines the use of multiple address busses and data cross bars to scale to 64 processors. Each board consists of four 250 MHz processors, four banks of memory (up to 1 GB each), and two independent SBUS I/O busses. Sixteeen of these boards are connected by a 16x16 cross bar with paths 144 bits wide, as well as four address busses associated with the four banks on each board. Collectively this provide 12.6 GB/s of data bandwidth and a snoop rate of 250 MHz.

6.8 Concluding Remarks

The design issues that we have explored in this chapter are fundamental, and will remain important with progress in technology. This is not to say that the optimal design choices will not change. For example, while shared-cache architectures are not currently very popular, it is possible that sharing caches at some level of the hierarchy may become quite attractive when multi-chip-module packaging technology becomes cheap or when multiple processors appear on a single chip, as long as destructive interference does not dominate in the workloads of interest.

A shared bus interconnect clearly has bandwidth limitations as the number of processors or the processor speed increases. Architects will surely continue to find innovative ways to squeeze more data bandwidth and more snoop bandwidth out of these designs, and will continue to exploit the simplicity of a broadcast-based approach. However, the general solution in building scalable cache-coherent machines is to distribute memory physically among nodes and use a scalable interconnect, together with coherence protocols that do not rely on snooping. This direction is the subject of the subsequent chapters. It is likely to find its way down to even the small scale as processors become faster relative to bus and snoop bandwidth. It is difficult to predict what the future holds for busses and the scale at which they will be used, although they are likely to have an important role for some time to come. Regardless of that evolution, the issues discussed here in the context of busses—placement of the interconnect within the memory hierarchy, the cache coherence problem and the various coherence protocols at state transition level, and the correctness and implementation issues that arise when dealing with many concurrent transactions—are all largely independent of technology and are crucial to the design of all cache-coherent shared-memory architectures regardless of the interconnect used. Morevover, these designs provide the basic building block for larger scale design presented in the remainder of the book.

6.9 References

[AdG96] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. IEEE Computer, vol. 29, no. 12, December 1996, pp. 66-76.

[AgG88] Anant Agarwal and Anoop Gupta. Memory-reference Characteristics of Multiprocessor Applications Under MACH. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 215-225, May 1988.

[ArB86] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[BaW88] Jean-Loup Baer and Wen-Hann Wang. On the Inclusion Properties for Multi-level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 73--80, May 1988.

[BJS88] F. Baskett, T. Jermoluk, and D. Solomon, The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. *Proc of the 33rd IEEE Computer Society Int. Conf. - COMPCOM 88*, pp. 468-71, Feb. 1988.

[BRG+89] David Black, Richard Rashid, David Golub, Charles Hill, Robert Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, April 1989.

[CAH+93] Ken Chan, et al. Multiprocessor Features of the HP Corporate Business Servers. In *Proceedings of COMPCON*, pp. 330-337, Spring 1993.

[CoF93] Alan Cox and Robert Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 98-108, May 1993.

[DSB86] Michel Dubois, Christoph Scheurich and Faye A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 434-442.

[DSR+93] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenstrom. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 88-97, May 1993.

[DuL92] C. Dubnicki and T. LeBlanc. Adjustable Block Size Coherent Caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 170-180, May 1992.

[EgK88] Susan Eggers and Randy Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 373-382, May 1988.

[EgK89a] Susan Eggers and Randy Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257-270, May 1989.

[EgK89b] Susan Eggers and Randy Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 2-15, May 1989.

[FCS+93] Jean-Marc Frailong, et al. The Next Generation SPARC Multiprocessing System Architecture. In *Proceedings of COMPCON*, pp. 475-480, Spring 1993.

[GaW93] Mike Galles and Eric Williams. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. In Proceedings of 27th Annual Hawaii International Conference on Systems Sciences, January 1993.

[Goo83] James Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 124-131, June 1983.

[Goo87] James Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, October 1987.

[GSD95] H. Grahn, P. Stenstrom, and M. Dubois, Implementation and Evaluation of Update-based Protocols under Relaxed Memory Consistency Models. In Future Generation Computer Systems, 11(3): 247-271, June 1995.

[HeP90] John Hennessy and David Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 1990.

[Her91] Maurice Herlihy. Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems, vol. 13, no. 1, January 1991, pp. 124-149.

[HiS87] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. IEEE Transactions on Computers, vol. C-38, no. 12, December 1989, pp. 1612-1630.

[HEL+86] Mark Hill et al. Design Decisions in SPUR. *IEEE Computer*, 19(10):8-22, November 1986.

[JeE91] Tor E. Jeremiassen and Susan J. Eggers. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pp. 377-381.

[KMR+86] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator. Competitive Snoopy Caching. In Pro-

ceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, 1986.

[Kro81] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of 8th International Symposium on Computer Architecture*, pp. 81-87, May 1981.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, vol. C-28, no. 9, September 1979, pp. 690-691.

[Lau94] James Laudon. Architectural and Implementation Tradeoffs for Multiple-Context Processors. Ph.D. Thesis, Computer Systems Laboratory, Stanford University, 1994.

[Lei92] C. Leiserson, et al. The Network Architecture of the Connection Machine CM-5. Symposium of Parallel Algorithms and Architectures, 1992.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. IEEE Transactions on Parallel and Distributed Systems, 4(1):41-61, January 1993.

[McC84] McCreight, E. The Dragon Computer System: An Early Overview. Technical Report, Xerox Corporation, September 1984.

[Mip91] MIPS R4000 User's Manual. MIPS Computer Systems Inc. 1991.

[PaP84] Mark Papamarcos and Janak Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 348-354, June 1984.

[Ros89] Bryan Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. In *Proceedings of the Symposium on Operating Systems Principles*, December 1989.

[ScD87] Christoph Scheurich and Michel Dubois. Correct Memory Operation of Cache-based Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, June 1987, pp. 234-243.

[SFG+93] Pradeep Sindhu, et al. XDBus: A High-Performance, Consistent, Packet Switched VLSI Bus. In *Proceedings of COMPCON*, pp. 338-344, Spring 1993.

[SHG93] Jaswinder Pal Singh, John L. Hennessy and Anoop Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, vol. 26, no. 7, July 1993.

[Smi82] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473-530, September 1982.

[SwS86] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 414-423, May 1986.

[TaW97] Andrew S. Tanenbaum and Albert S. Woodhull, Operating System Design and Implementation (Second Edition), Prentice Hall, 1997.

[Tel90] Patricia Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6):26-36, June 1990.

[TBJ+88] M. Thompson, J. Barton, T. Jermoluk, and J. Wagner. Translation Lookaside Buffer Synchronization in a Multiprocessor System. In *Proceedings of USENIX Technical Conference*, February 1988.

[TLS88] Charles Thacker, Lawrence Stewart, and Edwin Satterthwaite, Jr. Firefly: A Multiprocessor Workstation, IEEE Transactions on Computers, vol. 37, no. 8, Aug. 1988, pp. 909-20.

[TLH94] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651-663, June 1994.

[WBL89] Wen-Hann Wang, Jean-Loup Baer and Henry M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In Proceedings of the 16th Annual International Symposium

on Computer Architecture, pp. 140-148, June 1989.

[WeS94] Shlomo Weiss and James Smith. Power and PowerPC. Morgan Kaufmann Publishers Inc. 1994.

[WEG+86] David Wood, et al. An In-cache Address Translation Mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 358-365, May 1986.

6.10 Exercises

6.1 shared cache versus private caches. Consider two machines M1 and M2. M1 is a four-processor shared-cache machine, while M2 is a four-processor bus-based snoopy cache machine. M1 has a single shared 1 Mbyte two-way set-associative cache with 64-byte blocks, while each processor in M2 has a 256 Kbyte direct-mapped cache with 64-byte blocks. M2 uses the Illinois MESI coherence protocol. Consider the following piece of code:

```
double A[1024,1024]; /* row-major; 8-byte elems */
double C[4096];
double B[1024,1024];

for (i=0; i<1024; i+=1) /* loop-1 */
     for (j=myPID; j<1024; j+=numPEs)
          {
           B[i,j] = (A[i+i,j] + A[i-1,j] +
                          A[i,j+1] + A[i,j-1]) / 4.0;
          }
for (i=myPID; i<1024; i+=numPEs) /* loop-2 */
     for (j=0; j<1024; j+=1)
          {
           A[i,j] = (B[i+i,j] + B[i-1,j] +
                          B[i,j+1] + B[i,j-1]) / 4.0;
          }
```

a. Assume that the array A starts at address 0x0, array C at 0x300000, and array B at 0x308000. Assume that all caches are initially empty. Assume each processor executes the above code, and that `myPID` varies from 0-3 for the four processors. Compute misses for M1, separately for loop-1 and loop-2. Do the same for M2, stating any assumptions that you make.

b. Briefly comment on how your answer to part **a.** would change if the array C were not present. State any other assumptions that you make.

c. What can be learned about advantages and disadvantages of shared-cache architecture from this exercise?

d. Given your knowledge about the Barnes-Hut, Ocean, Raytrace, and Multiprog workloads from previous chapters and data in Section 5.5, comment on how each of the applications would do on a four-processor shared-cache machine with a 4 Mbyte cache versus a four processor snoopy-bus-based machine with1 Mbyte caches. It might be useful to verify your intuition using simulation.

e. Compared to a shared first-level cache, what are the advantages and disadvantages of having private first-level caches, but a shared second-level cache? Comment on how modern microprocessors, for example, MIPS R10000 and IBM/Motorola PowerPC 620, encourage or discourage this trend. What would be the impact of packaging technology on such designs?

6.2 Cache inclusion.

a. Using terminology in Section 6.4, assume both L1 and L2 are 2-way, and n2 > n1, and b1=b2, and replacement policy is FIFO instead of LRU. Does inclusion hold? What if it is random, or based on a ring counter.

b. Give an example reference stream showing inclusion violation for the following situations:

(i) L1 cache is 32 bytes, 2-way set-associative, 8-byte cache blocks, and LRU replacement. L2 cache is 128 bytes, 4-way set-associative, 8-byte cache blocks, and LRU replacement.

(ii) L1 cache is 32 bytes, 2-way set-associative, 8-byte cache blocks, and LRU replacement. L2 cache is 128 bytes, 2-way set-associative, 16-byte cache blocks, and LRU replacement.

c. For the following systems, state whether or not the caches provide for inclusion: If not, state the problem or give an example that violates inclusion.

(i) Level 1: 8KB Direct mapped primary instruction cache, 32 byte line size
8KB Direct mapped primary data cache, write through, 32 byte line size
Level 2: 4MB 4 way set associative unified secondary cache, 32 byte line size

(ii) Level 1: 16KB Direct Mapped unified primary cache, write-through, 32 byte line size
Level 2: 4MB 4 way set associative unified secondary cache, 64 byte line size

d. The discussion of the inclusion property in Section 6.4 stated that in a common case inclusion is satisfied quite naturally. The case is when the L1 cache is direct-mapped (a_1 = 1), L2 can be direct-mapped or set associative (a_2 >= 1) with any replacement policy (e.g., LRU, FIFO, random) as long as the new block brought in is put in both L1 and L2 caches, the block-size is the same ($b_1 = b_2$), and the number of sets in the L1 cache is equal to or smaller than in L2 cache (n_1 =< n_2). Show or argue why this is true.

6.3 Cache tag contention <> Assume that each processor has separate instruction and data caches, and that there are no instruction misses. Further assume that, when active, the processor issues a cache request every 3 clock cycles, the miss rate is 1%, miss latency is 30 cycles. Assume that tag reads take one clock cycle, but modifications to the tag take two clock cycles.

a. Quantify the performance lost to tag contention if a single-level data cache with only one set of cache tags is used. Assume that the bus transactions requiring snoop occur every 5 clock cycles, and that 10% of these invalidate a block in the cache. Further assume that snoops are given preference over processor accesses to tags. Do back-of-the-envelope calculations first, and then check the accuracy of your answer by building a queuing model or writing a simple simulator.

b. What is the performance lost to tag contention if separate sets of tags for processor and snooping are used?

c. In general, would you decide to give priority in accessing tags to processor references or bus snoops?

6.4 Protocol and Memory System Implementation: SGI Challenge.

a. The designers of the SGI Challenge multiprocessor considered the following bus controller optimization to make better use of interleaved memory and bus bandwidth. If the controller finds that a request is already outstanding for a given memory bank (which can be determined from the request table), it does not issue that request until the previous one for that bank is satisfied. Discuss potential problems with this optimization and what features in the Challenge design allow this optimization.

b. The Challenge multiprocessor's Powerpath-2 bus allows for eight outstanding transactions. How do you think the designers arrived at that decision? In general, how would you

determine how many outstanding transactions should be allowed by a split-transaction bus. State all your assumptions clearly.

c. Although the Challenge supports the MESI protocol states, it does not support the cache-to-cache transfer feature of the original Illinois MESI protocol.

(i) Discuss the possible reasons for this choice.

(ii) Extend the Challenge implementation to support cache-to-cache transfers. Discuss extra signals needed on bus, if any, and keep in mind the issue of fairness.

d. Although the Challenge MESI protocol has four states, the tags stored with the cache controller chip keep track of only three states (I, S, and E+M). Explain why this is still works correctly. Why do you think that they made this optimization?

e. The main memory on the Challenge speculatively initiates fetching the data for a read request, even before it is determined if it is dirty in some processor's cache. Using data in Table 5-3 estimate the fraction of useless main memory accesses. Based on the data, are you in favor of the optimization? Are these data methodologically adequate? Explain.

f. The bus interfaces on the SGI Challenge support request merging. Thus, if multiple processors are stalled waiting for the same memory block, then when the data appears on the bus, all of them can grab that data off the bus. This feature is particularly useful for implementing spin-lock based synchronization primitives. For a test-test&set lock, show the minimum traffic on the bus with and without this optimization. Assume that there are four processors, each acquiring lock once and then doing an unlock, and that initially no processor had the memory block containing the lock variable in its cache.

g. Discuss the cost, performance, implementation, and scalability tradeoffs between the multiple bus architecture of the SparcCenter versus the single fast-wide bus architecture of the SGI Challenge, as well as any implications for program semantics and deadlock.

6.5 Split Transaction Busses.

a. The SGI Challenge bus allows for eight outstanding transactions. How did the designers arrive at that decision? To answer that, suggest a general formula to indicate how many outstanding transactions should be supported given the parameters of the bus. Use the following parameters:

P	Number of processors
M	Number of memory banks
L	Average memory latency (cycles)
B	Cache block size (bytes)
W	Data bus width (bytes)

Define any other parameters you think are essential. *Keep your formula simple*, clearly state any assumptions, and justify your decisions.

a. To improve performance of the alternative design for supporting coherence on a split-transaction bus (discussed at the end of Section 6.5), a designer lays down the following design objectives: (i) the snoop results are generated in order, (ii) the data responses, however, may be out of order, and (iii) there *can* be multiple pending requests to the same memory block. Discuss in detail the implementation issues for such an option, and how it compares to the base implementation discussed in Section 6.5.

b. In the split-transaction solution we have discussed in Section 6.5, depending on the processor-to-cache interface, it is possible that an invalidation request comes immediately after the data response, so that the block is invalidated before the processor has had a chance to actually access that cache block and satisfy its request. Why might this be a problem and how can you solve it?

c. When supporting lock-up-free caches, a designer suggests that we also add more entries to the request-table sitting on the bus-interface of the split-transaction bus. Is this a good idea and do you expect the benefits to be large?

d. <<The ways to preserve SC with multiple outstanding transactions and commit versus complete. Apply them to Example 6-3 on page 381 and convince yourself that they work. Under what conditions is one solution better than the other? Answer: For SC, preserving order may be better without the optimizations since need to check/flush frequently in second case. Particularly with multi-level caches.

6.6 Computing bandwidth needs using data provided. Assume a system bus similar to Powerpath2, as discussed in Section 6.6. Assuming 200-MIPS/200-MFLOPS processors with 1 Mbyte caches and 64-byte cache blocks, for each of the applications in Table 5-1 compute the bus bandwidth when using:

a. The Illinois MESI protocol.

b. The Dragon protocol.

c. The Illinois MESI protocol assuming 256-byte cache blocks.

For each of the above parts, compute the utilization of the address+command bus separately from the utilization of the data bus. State all assumptions clearly.

d. Do parts a, and b for a single SparcCenter XDBus, which has 64-bit wide multiplexed address and data signals. Assume that the bus runs at 100MHz, and that transmitting address information takes 2 cycles on the bus, and that 64bytes of data takes 9 cycles on the bus.

6.7 Multi-level Caches

a. One deadlock solution proposed for multi-level caches in Section 6.5.8 is to make all queues 9 deep. Can the queues be smaller? If so, why? Discuss, why it may be beneficial to have deeper queues than the size required by deadlock considerations.

b. [Section 6.4 presents coherence protocols assuming 2-level caches. What if there are three or more levels in the cache hierarchy. Extend the Illinois MESI protocol for the middle cache in a 3-level hierarchy. List any additional states or actions needed, and present the state-transition diagram. Discuss the implementation details, including the number and nature of intervening queues.

6.8 TLB Shootdown. Figure 6-26 shows the details of the TLB shootdown algorithm used in the Mach operating system [BRG+89]. The basic data structures are as follows. For each processor, the following data structures are maintained: (i) an *active* flag indicating whether the processor is actively using any page tables; (ii) a queue of TLB flush notices indicating the range of virtual addresses whose mappings are to be changed; and (iii) a list indicating currently active page tables, i.e., processes whole PTEs may be cached in the TLB. For every page table, there is: (i) a spinlock that processor must hold while making changes to that page table, and (ii) a set of processors on which this page table is currently active. While the basic shootdown approach is simple, practical implementations require careful sequencing of steps and locking of data structures.

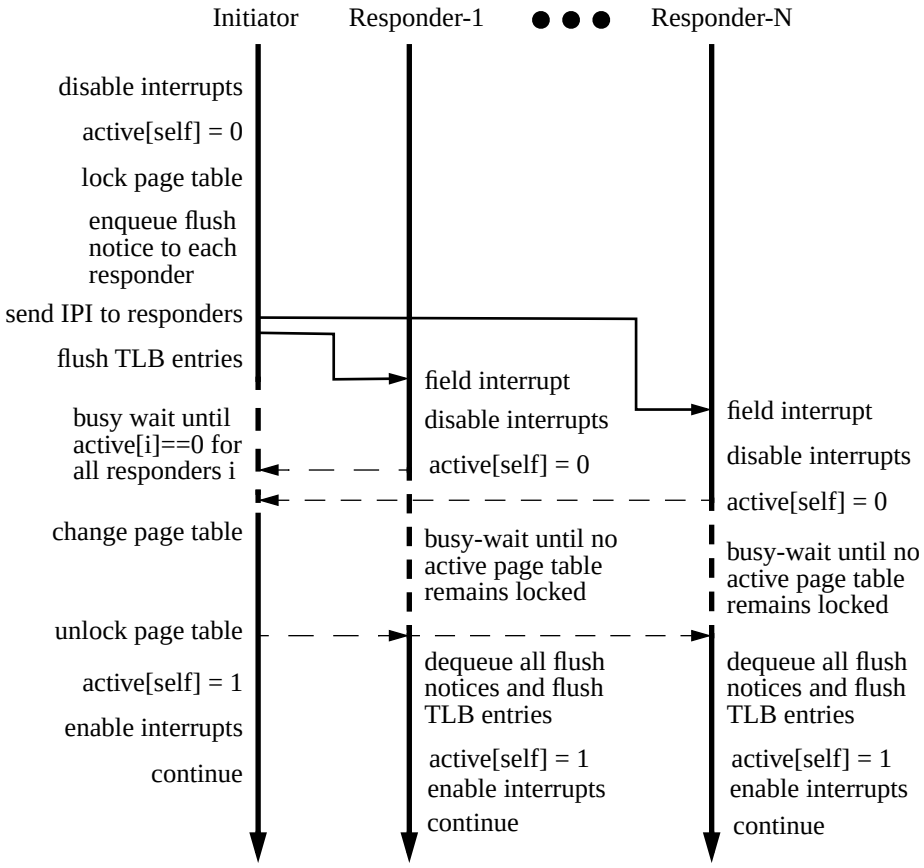

Figure 6-26 The Mach TLB shootdown algorithm.

The initiator is the processor making changes to the page-table, while the responders are all other processors that may have entries from that page-table cached.

a. Why are page table entries modified before sending interrupts or invalidate messages to other processors in TLB coherence?

b. Why must the initiator of the shootdown in Figure 6-26 mask out inter-processor-interrupts (IPIs) before acquiring the page table lock, and to clear its own active flag before acquiring the page table lock? Can you think of any deadlock conditions that exist in the figure, and if so how would you solve them?

c. A problem with the Mach algorithm is that it makes all responders busy-wait while the initiator makes changes to the page table. The reason is that it was designed for use with microprocessors that autonomously wrote back the entire TLB entry into the corresponding PTE whenever the usage/dirty bit was set. Thus, for example, if other processors were allowed to use the page table while the initiator was modifying it, an autonomous write-back from those processors could overwrite the new changes. How would you design the TLB hardware and/or algorithm so that responders do not have to busy-wait? [One solution was used in the IBM RP3 system [Ros89].

d. For MACH shootdown we say it would be better to update the use and dirty information for pages in software. (Machines based on the MIPS processor architecture actually do all of this in software.) Lookup the operating system of a MIPS-based machine or suggest how you would write the TLB fault handlers so that the usage and dirty information was made available to the OS page replacement algorithms and for writeback of dirty pages.

e. Under what circumstances would it be better to flush the whole TLB versus selectively trying to invalidate TLB entries?