Multiprocessor Memory Issues

Norman Matloff
Department of Computer Science
University of California at Davis

November 1, 2003

Contents

1 Overview

There is an ever-increasing appetite among computer users for faster and faster machines. This was epitomized by a statement by Steve Jobs, current CEO of Pixar (the computer graphics company which made the movie *Toy Story*), and founder/former CEO of Apple. He noted that when he was at Apple, he was always worried that some other company would come out with a faster machine than his. But now at Pixar, whose graphics work requires extremely fast computers, he is always hoping someone produces faster machines, so that he can use them!

A major source of speedup is the parallelizing of operations. Parallel operations can be either within-processor, such as with pipelining or having several ALUs within a processor, or between-processor, in which many processor work on different parts of a problem in parallel. Our focus here is on between-processor operations.

For example, the Registrar's Office at UC Davis uses a Sequent machine for processing its on-line registration work. The earlier version of the Sequent consists of several (typically 8-12) Intel processors, connected to the same bus. (More recent versions use an Omega network, to be described later.) All processors then access the same memory chips connected to the bus; this is called a **shared-memory multiprocessor** system. The software performs an enormous amount of database computation. In order to handle this computation reasonably quickly, the program partitions the work to be done, assigning different portions of the database to different processors. Currently database work comprises the biggest use of parallel machines. It is due to the database field that such machines are now so successful commercially.

As the Pixar example shows, highly computation-intensive applications like computer graphics also have a need for these fast parallel computers. No one wants to wait hours just to generate a single image, and the use of parallel processing machines can speed things up considerably. For example, consider **ray tracing** operations. Here our code follows the path of a ray of light in a scene, accounting for reflection and absorbtion of the light by various objects. Suppose the image is to consist of 1,000 rows of pixels, with 1,000 pixels per row. In order to attack this problem in a parallel processing manner with, say, 25 processors, we could divide the image into 25 squares of size 200x200, and have each processor do the computations for its square.[1]

Remember, the reason for buying a parallel processing machine is *speed*. You need to have a program that runs as fast as possible. That means that in order to write good parallel processing software, you must have a good knowledge of the underlying hardware. You must find also think of clever tricks for **load balancing,** i.e. keeping all the processors busy as much as possible. In the graphics ray-tracing application, for instance, suppose a ray is coming from the "northeast" section of the image, and is reflected by a solid object. Then the ray won't reach some of the "southwest" portions of the image, which then means that the processors assigned to those portions will not have any work to do which is associated with this ray. What we need to do is then try to give these processors some other work to do; the more they are idle, the slower our system will be.

[1] It may be more challenging than this implies. If one really wants a good speedup, one may need to take into account the fact that some squares require more computation work than others. Moreover, there are interactions between the squares to take into account, so in some cases a processor assigned to one square must wait for information from the processor assigned to another square, causing a delay and thus loss of speed.

2 Shared-Memory Multiprocessors

The term **shared memory** means that the processors all share a common address space. If this is occurring at the hardware level, then if processor P3 issues a memory-read instruction for location 200, and processor P4 does the same, they both will be referring to the same physical memory cell. In non-shared-memory machines, each processor has its own private memory, and each one will then have its own location 200, completely independent of the locations 200 at the other processors' memories.

Say a program contains a global variable X and a local variable Y on share-memory hardware (and we use shared-memory software). If for example the compiler assigns location 200 to the variable X, i.e. `&X = 200`, then the point is that all of the processors will have that variable in common, because any processor which issues a memory operation on location 200 will access the same physical memory cell.

On the other hand, each processor will have its own separate run-time stack,[2] and thus each processor will have its own independent copy of the local variable Y.[3]

To make the meaning of "shared memory" more concrete, suppose we have a bus-based system, with all the processors and memory attached to the bus. Let us compare the above variables X and Y here. Suppose again that the compiler assigns X to memory location 200. Then in the machine language code for the program, every reference to X will be there as 200. Every time an instruction involving X is executed by a CPU, that CPU will put 200 into its Memory Address Register (MAR), from which the 200 flows out on the address lines in the bus, and goes to memory. This will happen in the same way no matter which CPU it is. Thus the same physical memory location will end up being accessed, no matter which CPU generated the reference.

By contrast, say the compiler assigns Y to something like SP+2, the third item on the stack.[4] Each CPU will have its own current value for SP, so the stacks of the various CPUs will be separate.[5]

3 Memory Modules

The biggest obstacle to high performance in a shared-memory multiprocessor system is that the processors will often try to access memory at the same time, causing delays as some wait for others to finish their memory access. For this reason, memory is broken down into modules. As long as different processors access different modules, the access can be simultaneous.

High-order interleaving is used in design of the modules. Here consecutive addresses are in the same module (except at boundaries). For example, suppose for simplicity that our memory consists of addresses 0 through 1023, and that there are four modules.[6] Then module M0 would contain addresses 0-255, M1 would have

[2] Still in shared memory, but a separate stack for each processor, since each CPU has a different value in its SP register.

[3] Compare this to the situation in which several copies of the same program are running simultaneously on a single-CPU machine. They would typically share instructions but not data, even global data.

[4] SP will point to the saved return address for the call, SP+1 to something else and SP+2 to Y.

[5] The stacks will be in the physical shared memory, and thus P3, say, could theoretically access P8's stack, say if there were an erroneous pointer value. But even that would not occur if we are using virtual memory and thus have protections against this.

[6] By the way, it is typical that there will be the same number of modules as processors, but it does not have to be this way.

256-511, M2 would have 512-767, and M3 would have 768-1023.

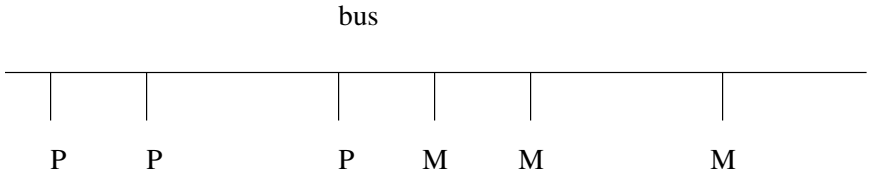
Of course, even with multiple memory modules, we will sometimes have clashes, in which two or more processors need to access the same module (even if not the same word in the module) simultaneously. Thus careful design of the software is crucial, in order to minimize the frequency with which clashes occur.

For example, in a database application, the data structures in our application program might be set up so that different processors usually access different parts of the database, which in turn would be stored in different memory modules.[7]

4 Interconnecting the CPUs and Memory Modules

4.1 Bus Interconnect

Let's first consider the following structure:



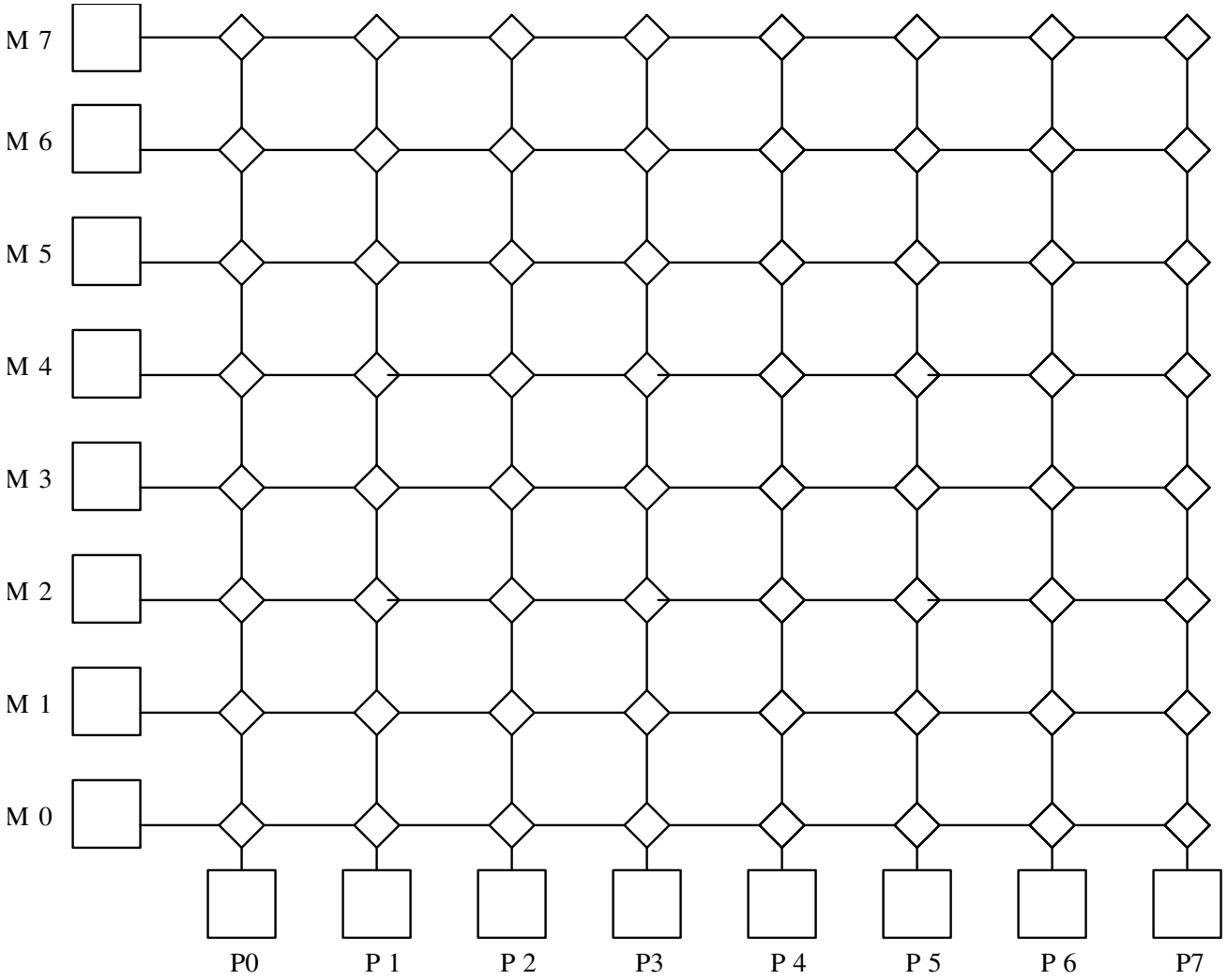
- The Ps are processors, e.g. off-the-shelf chips such as Pentium, and the Ms are memory modules.
- To make sure only one P uses the bus at a time, standard bus arbitration signals and/or arbitration devices can be used, say the Intel 8259A chip.
- There may also be **coherent caches** at the Ps, which we will discuss later.

The problem with a bus connection, of course, is that there is only one pathway for communication. After perhaps two dozen processors are on the bus, the bus becomes saturated, even if traffic-reducing methods such as adding caches are used. Thus multipathway topologies are used for all but the smallest systems.

4.2 Crossbar Interconnects

Consider a shared-memory system with n processors and n memory modules. A crossbar connection would provide n^2 pathways. E.g. for n = 8:

[7] Databases are primarily stored on disk, but for fast access one usually has copies of portions of them in memory.



Communication from node to node consists of a packet containing information on both source address (i.e. processor number) and destination address (memory module number). E.g. if P2 wants to read from M5, the source and destination will be 3-bit strings in the packet, coded as 010 and 101, respectively. Note that the source address is needed, since M5 needs to know to which processor it should send its reply.

Each diamond-shaped node has two inputs (bottom and right) and two outputs (left and top), with buffers at the two inputs.[8] If the packets at the heads of the two buffers both need to go out the same output, the one (say) from the bottom input will be given priority.

In this setting, there would also be a return network of the same type, with this one being memory → processor, to return the results of the read requests.[9]

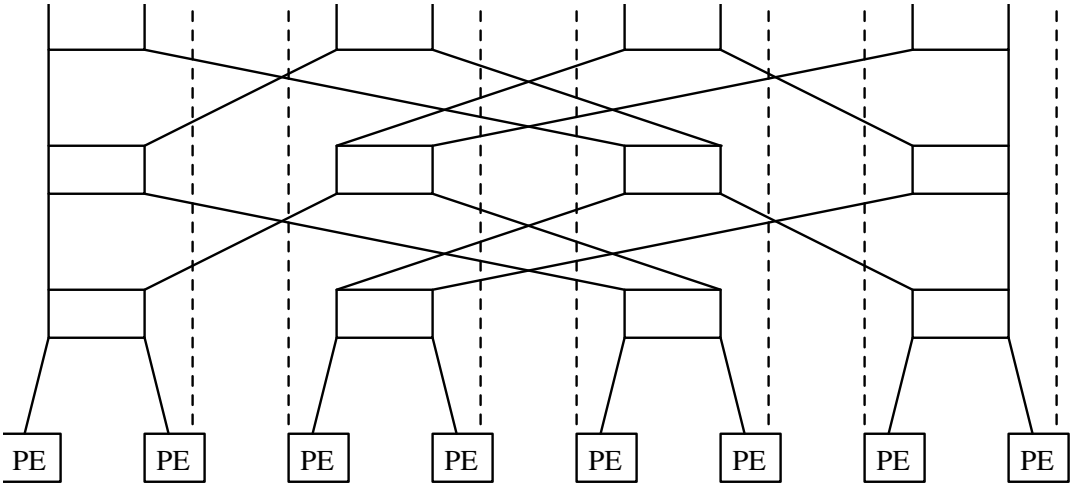
Crossbar switches are too expensive for large-scale systems, but are useful in some small systems. The 16-CPU Sun Microsystems Enterprise 10000 system includes a 16x16 crossbar.

[8] If a buffer fills, there are two design options: (a) Have the node from which the input comes block at that output. (b) Have the node from which the input comes discard the packet, and retry later, possibly outputting some other packet for now.

[9] For safety's sake, i.e. fault tolerance, even writes are typically acknowledged in multiprocessor systems.

4.3 Omega Interconnects

These are multistage networks similar to crossbars, but with fewer paths. Here is an example of an 8x8 system:



The PEs are processing elements, consisting of a CPU and a memory module. Thus each CPU has a "favorite" memory module, one that it can access quite quickly relative to accessing the other memory modules.[10] But the memory modules still collectively define the address space shared by all the processors, and if a CPU needs to access a "remote" module, it will send a packet through the network.

At each network node (the nodes are the three rows of rectangles), the output routing is done by destination bit. Let's number the stages here 0, 1 and 2, starting from the bottom stage, number the nodes within a stage 0, 1, 2 and 3 from left to right, number the PEs from 0 to 7, left to right, and number the bit positions in a destination address 0, 1 and 2, starting from the most significant bit. Then at stage i, bit i of the destination address is used to determine routing, with a 0 meaning routing out the left output, and 1 meaning the right one.

Note that at the third stage of the network (top of picture), the outputs are routed back to the PEs, each of which consists of a processor and a memory module. PE3, for instance, consists of P3 and M3.

Say P2 wishes to read from M5. It sends a read-request packet, including 5 = 101 as its destination address, to the switch in stage 0, node 1. Since the first bit of 101 is 1, that means that this switch will route the packet out its right-hand output, sending it to the switch in stage 1, node 3. The latter switch will look at the next bit in 101, a 0, and thus route the packet out its left output, to the switch in stage 2, node 2. Finally, that switch will look at the last bit, a 1, and output out its right-hand output, sending it to PE5, as desired. M5 will process the read request, and send a packet back to PE2, along the same

Again, if two packets at a node want to go out the same output, one must get priority (let's say it is the one from the left input).

Here is how the more general case of N = 2^n PEs works. Again number the rows of switches, and switches

[10] This is not an inherent property of Omega networks, but it is the natural way to arrange things.

within a row, as above. So, S_{ij} will denote the switch in the i-th row from the bottom and j-th column from the left (starting our numbering with 0 in both cases). Row i will have a total of N input ports I_{ik} and N output ports O_{ik}, where k = 0 corresponds to the leftmost of the N in each case. Then if row i is not the last row ($i < n-1$), O_{ik} will be connected to I_{jm}, where j = i+1 and

$$m = (2k + \lfloor (2k)/N \rfloor) mod N$$

where $\lfloor \rfloor$ denotes the "floor" function, which extracts the integer part of a floating-point number. For example, $\lfloor 2.78 \rfloor = 2$. If row i is the last row, then O_{ik} will be connected to, PE k.

4.4 Comparative Analysis

Multiprocessor architects use the terms **latency** and **bandwidth** to provide a rough description of the performance potential of an interconnect. There are no official definitions for these terms, but generally latency means access time (in network cycles) in the absence of other traffic, and bandwidth means the number, out of n packets to n distinct destinations, which can be sent simultaneously. Another factor is cost, measured by the number of components.

In the world of parallel architectures, a key criterion for a proposed feature is **scalability**, meaning how well the feature performs as we go to larger and larger systems. Let n be the system size, either the number of processors and memory modules, or the number of PEs. Then we are interested in how fast the latency, bandwidth and cost grow with n:

criterion	bus	Omega	crossbar
latency	O(1)	O($log_2 n$)	O(n)
bandwidth	O(1)	O(n)	O(n)
cost	O(1)	O(n $log_2 n$)	O(n^2)

Let us see where these expressions come from, beginning with a bus: No matter how large n is, the time to get from, say, a processor to a memory module will be the same, thus O(1). Similarly, no matter how large n is, only one communication can occur at a time, thus again O(1).[11]

Again, we are interested only in "O( )" measures, because we are only interested in growth rates as the system size n grows. For instance, if the system size doubles, the cost of a crossbar will quadruple; the $O(n^2)$ cost measure tells us this, with any multiplicative constant being irrelevant.

For Omega networks, it is clear that $log_2 n$ network rows are needed, hence the latency value given. Also, each row will have n/2 switches, so the number of network nodes will be O(n $log_2 n$). This figure then gives the cost (in terms of switches, the main expense here). It also gives the bandwidth, since the maximum number of simultaneous transmissions will occur when all switches are sending at once.

Similar considerations hold for the crossbar case.

[11] Note that the '1' in "O(1)" does not refer to the fact that only one communication can occur at a time. If we had, for example, a two-bus system, the bandwidth would still be O(1), since multiplicative constants do not matter. What O(1) means, again, is that as n grows, the bandwidth stays at a multiple of 1, i.e. stays constant.

The crossbar's big advantage is that it is guaranteed that n packets can be sent simultaneously, providing they are to distinct destinations.[12]

That is not true for Omega-networks. If for example, PE0 wants to send to PE3, and at the same time PE4 wishes to sent to PE2, the two packets will clash at the leftmost node of stage 1, where the packet from PE0 will get priority.

On the other hand, a crossbar is very expensive, and thus is dismissed out of hand in most modern systems. Note, though, that an equally problematic aspect of crossbars is their high latency value; this is a big drawback when the system is not heavily loaded.

The bottom line is that Omega-networks amount to a compromise between buses and crossbars, and for this reason have become popular.

5 Interprocessor Synchronization on Shared-Memory Hardware

Consider a bus-based system. In addition to whatever read and write instructions the processor included, say a load LD and a store ST, there would also be a TAS ("test and set")instruction. This instruction would control a TAS pin on the processor chip, and the pin in turn would be connected to a TAS line in the bus.

Applied to a location L in memory and a register R, say, TAS does the following:

```
copy L to R
if R is 0 then write 1 to L
```

And most importantly, these operations are done in an **atomic** manner; no bus transactions by other processors may occur between the two steps.

The TAS operation is applied to variables used as **locks**. Let's say that 1 means locked and 0 unlocked. Then the guarding of a critical section C by a lock variable L would be done by having the following code in the program being run:

```
TRY:  TAS R,L
      JNZ TRY
  C:  ...   ; start of critical section
      ...
      ...   ; end of critical section
      MOV L,0  ; unlock
```

where of course JNZ is a jump-if-nonzero instruction, and we are assuming that the copying from the Memory Data Register to R results in the processor N and Z flags (condition codes) being affected.

[12] If two or more go to the same destination, they couldn't be satisfied simultaneously anyway, unless dual-port memory were used.

The LOCK() library function in MulSim consists of a loop like that pictured above, with a TAS instruction comprising its core. The UNLOCK() function merely puts 0 into L, as in the MOV above.

A **critical section** is a portion of a program in which we cannot have more than one processor execute at a time. For instance, consider an airline reservation system. If a flight has only one seat left, we want to avoid giving it to two different customers who might be talking to two agents at the same time. The lines of code in which the seat is finally assigned (the **commit** phase, in database terminology) is then a critical section.

In crossbar or Ω-network systems, some 2-bit field in the packet must be devoted to transaction type, say 00 for Read, 01 for Write and 10 for TAS. But note that the atomicity here is best done at the memory, i.e. some hardware should be added at the memory so that TAS can be done; otherwise, an entire processor-to-memory path would have to be locked up for a fairly long time, obstructing even the packets which go to other memory modules.

6 Cache Coherency

Consider, for example, a bus-based system. Relying purely on TAS for interprocessor synchronization would be unthinkable: As each processor contending for a lock variable spins in the loop shown above, it is adding tremendously to bus traffic.

An answer is to have caches at each processor, to store values of lock variables.[13] The point is this: Why keep looking at a lock variable L again and again, using up the bus bandwidth? L may not change value for a while, so why not keep a copy in the cache, avoiding use of the bus?

The answer of course is that eventually L will change value, and this causes some delicate problems. Say for example that processor P5 wishes to enter a critical section guarded by L, and that processor P2 is already in there. During the time P2 is in the critical section, P5 will spin around, always getting the same value for L (1) from C5, P5's cache. When P2 leaves the critical section, P2 will set L to 0—and now C5's copy of L will be incorrect. This is the **cache coherency problem**, inconsistency between caches.

A number of solutions have been devised for this problem. For bus-based systems, **snoopy** protocols of various kinds are used, with the word "snoopy" referring to the fact that all the caches monitor ("snoop on") the bus, watching for transactions made by other caches.

The most common protocols are the **invalidate** and **update** types. This relation between these two is somewhat analogous to the relation between **write-back** and **write-through** protocols for caches in uniprocessor systems:

- Under an invalidate protocol, when a processor writes to a variable in a cache, it first (i.e. before actually doing the write) tells all other caches to mark their copies of the variable[14] as invalid. Their caches will updated only later, the next time their processors need to access this cache line.
- For an update protocol, the processor which writes to the variable tells all other caches to immediately update their copies of that variable with the new value.

[13] Of course, non-lock variables are stored too. However, the discussion here will focus on effects on lock variables.

[14] Actually, mark their cache lines containing such copies.

Let's look at how one implementation (many variations exist) of an invalidate protocol would operate:[15]

In the scenario outlined above, when P2 leaves the critical section, it will write the new value 0 to L. Under the invalidate protocol, P2 will post an invalidation message on the bus. All the other caches will notice, as they have been monitoring the bus. They then mark their cached copies of the line containing L as invalid.

Now, the next time P5 executes the TAS instruction—which will be very soon, since it is in the loop shown above—P5 will find that the copy of L in C5 is invalid. It will respond to this cache miss by going to the bus, and requesting P2 to supply the "real" (and valid) copy of the line containing L.

Suppose that all this time P6 had also been executing the loop shown above. Then P5 and P6 may have to contend with each other; whoever manages to grab possession of the bus first[16] will be the one who ends up finding that L = 0. Let's say that that one is P6. Then when P6 relinquishes the bus, P5 tries its execution of the TAS again. P5 acquires a valid copy of L now, but L will be 1 at this time, so P5 must resume executing the loop. P5 will then continue to use its valid local copy of L each time it does the TAS, until P6 leaves the critical section, writes 0 to L, and causes another cache miss at P5, etc.

At first the update approach seems obviously superior, and actually, if our shared, cacheable[17] variables were only lock variables, this might be true.

But consider a shared, cacheable vector. Suppose the vector fits into one block, and that we write to each vector element sequentially. Under an update policy, we would have to send a new message on the bus/network for each component, while under an invalidate policy, only one message (for the first component) would be needed. If during this time the other processors do not need to access this vector, all those update messages, and the bandwidth they use, would be wasted.

Suppose for example we have code like

```
Sum += X[I];
```

in the middle of a **for** loop. Under an update protocol, we would have to write the value of Sum back many times, even though the other processors may only be interested in the final value when the loop ends. (This would be true, for instance, if the code above were part of a critical section.)

Now, how is cache coherency handled in non-bus shared-memory systems? Here the problem is more complex. The very feature which was the biggest negative feature of bus systems—the fact that there was only one path between components made bandwidth very limited—was a very positive feature in terms of cache coherency, because it made broadcast very easy: Since everyone is attached to that single pathway, sending a message to all of them costs no more than sending it to just one—we get the others for free. That's no longer the case for multipath systems. In such systems, extra copies of the message must be created for each path, adding to overall traffic.

A solution is to send messages only to "interested parties." In **directory-based** protocols, a list would be kept of all caches which currently have valid copies of all blocks. In one common implementation, for

[15] Note that this is just an outline. The full details, covering all possible settings and timings, are much more complex. For example, the SCI (Scalar Coherent Interface) protocol formal specification takes up 278 pages!

[16] Again, remember that ordinary bus arbitration methods would be used.

[17] Many modern processors, including Pentium and MIPS, allow the programmer to mark some blocks as being noncacheable.

example, while P2 is in the critical section above, it would be the **owner** of the block containing L. (Whoever is the latest node to write to L would be considered its current owner.) It would maintain a directory of all caches having valid copies of that block, say C5 and C6 in our story here. As soon as P2 wrote to L, it would then send either invalidate or update packets (depending on which type was being used) to C5 and C6 (and not to other caches which didn't have valid copies).

There would also be a directory at the memory, listing the current owners of all blocks. If for example P0 now wishes to "join the club," i.e. it tries to access L and gets a cache miss due to not having a valid copy of that block, P0 must consult the home of L, say P14. (The home is determined by the location in main memory according to high-order interleaving; it is the place where the main-memory version of L resides.) A table at P14 will inform P0 that P2 is the current owner of that block. P0 will then send a message to P2 to add C0 to the list of caches having valid copies of that block.[18]

Example: the MESI Cache Coherency Protocol Many types of cache coherency protocols have been proposed and used, some of them quite complex. A relatively simple one for snoopy bus systems which is widely used is MESI, which stands for the four states a given cache line can be in for a given CPU:

- Modified
- Exclusive
- Shared
- Invalid

It is the protocol used in the Pentium I, for example.

Here is a summary of the meanings of the states:

state	meaning
M	written to more than once; no other copy valid
E	valid; no other cache copy valid; memory copy valid
S	valid; at least one other cache copy valid
I	invalid

Following is a summary of MESI state changes. (See *Pentium Processor System Architecture*, by D. Anderson and T. Shanley, Addison-Wesley, 1995. We have simplified the presentation here, by eliminating certain programmable options.) When reading it, keep in mind that there is a separate state for each CPU/cache line combination. Also, in addition to the terms **read hit**, **read miss**, **write hit**, **write miss**, which you are already familiar with, there are also **read snoop** and **write snoop**. These refer to the case in which our CPU observes a read or write by another CPU on the bus.

Our CPU does a read:

[18] Similarly, a cache might "resign" from the club, due that cache line being replaced, e.g. in a LRU setting, when some other cache miss occurs.

present state	event	new state
M	read hit	M
E	read hit	E
S	read hit	S
I	read miss; no valid cache copy	E
I	read miss; at least one valid cache copy	S

Our CPU does a memory write:

present state	event	new state
M	write hit; do not put invalidate signal on bus; do not update memory	M
E	same as M above	M
S	write hit; put invalidate signal on bus; update memory	E
I	write miss; update memory but do nothing else	I

Our CPU does a read snoop (i.e. observes another CPU's cache do a read action on the bus) or write snoop (i.e. observes another CPU's cahce to a write action on the bus):

present state	event	newstate
M	read snoop; write line back to memory, picked up by other CPU	S
M	write snoop; write line back to memory	I
E	read snoop; put shared signal on bus; no memory action	S
E	write snoop; no memory action	I
S	read snoop	S
S	write snoop	I
I	any snoop	I