

SHARED-MEMORY MULTIPROCESSORS - SYMMETRIC MULTIPROCESSING HARDWARE*

Charles Severance

Kevin Dowd

This work is produced by OpenStax-CN X and licensed under the
Creative Commons Attribution License 3.0[†]

In Figure 1 (A shared-memory multiprocessor), we viewed an ideal shared-memory multiprocessor. In this section, we look in more detail at how such a system is actually constructed. The primary advantage of these systems is the ability for any CPU to access all of the memory and peripherals. Furthermore, the systems need a facility for deciding among themselves who has access to what, and when, which means there will have to be hardware support for arbitration. The two most common architectural underpinnings for symmetric multiprocessing are *buses* and *crossbars*. The bus is the simplest of the two approaches. Figure 2 (A typical bus architecture) shows processors connected using a bus. A bus can be thought of as a set of parallel wires connecting the components of the computer (CPU, memory, and peripheral controllers), a set of protocols for communication, and some hardware to help carry it out. A bus is less expensive to build, but because all traffic must cross the bus, as the load increases, the bus eventually becomes a performance bottleneck.

*Version 1.3: Aug 25, 2010 11:12 am -0500

[†]<http://creativecommons.org/licenses/by/3.0/>

A shared-memory multiprocessor

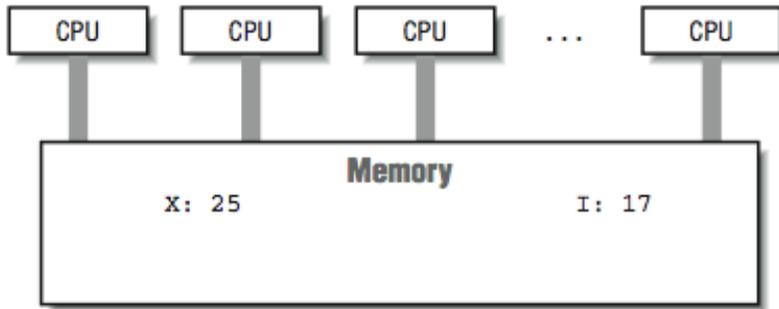


Figure 1

A typical bus architecture

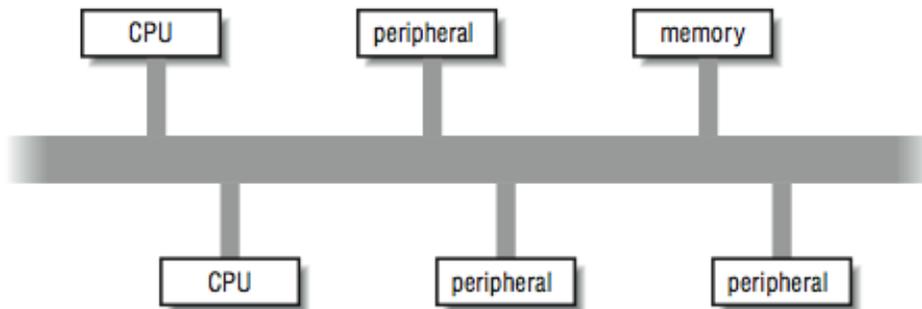


Figure 2

A crossbar is a hardware approach to eliminate the bottleneck caused by a single bus. A crossbar is like several buses running side by side with attachments to each of the modules on the machine — CPU, memory, and peripherals. Any module can get to any other by a path through the crossbar, and multiple

paths may be active simultaneously. In the 4×5 crossbar of Figure 3 (A crossbar), for instance, there can be four active data transfers in progress at one time. In the diagram it looks like a patchwork of wires, but there is actually quite a bit of hardware that goes into constructing a crossbar. Not only does the crossbar connect parties that wish to communicate, but it must also actively arbitrate between two or more CPUs that want access to the same memory or peripheral. In the event that one module is too popular, it's the crossbar that decides who gets access and who doesn't. Crossbars have the best performance because there is no single shared bus. However, they are more expensive to build, and their cost increases as the number of ports is increased. Because of their cost, crossbars typically are only found at the high end of the price and performance spectrum.

Whether the system uses a bus or crossbar, there is only so much memory bandwidth to go around; four or eight processors drawing from one memory system can quickly saturate all available bandwidth. All of the techniques that improve memory performance (as described in) also apply here in the design of the memory subsystems attached to these buses or crossbars.

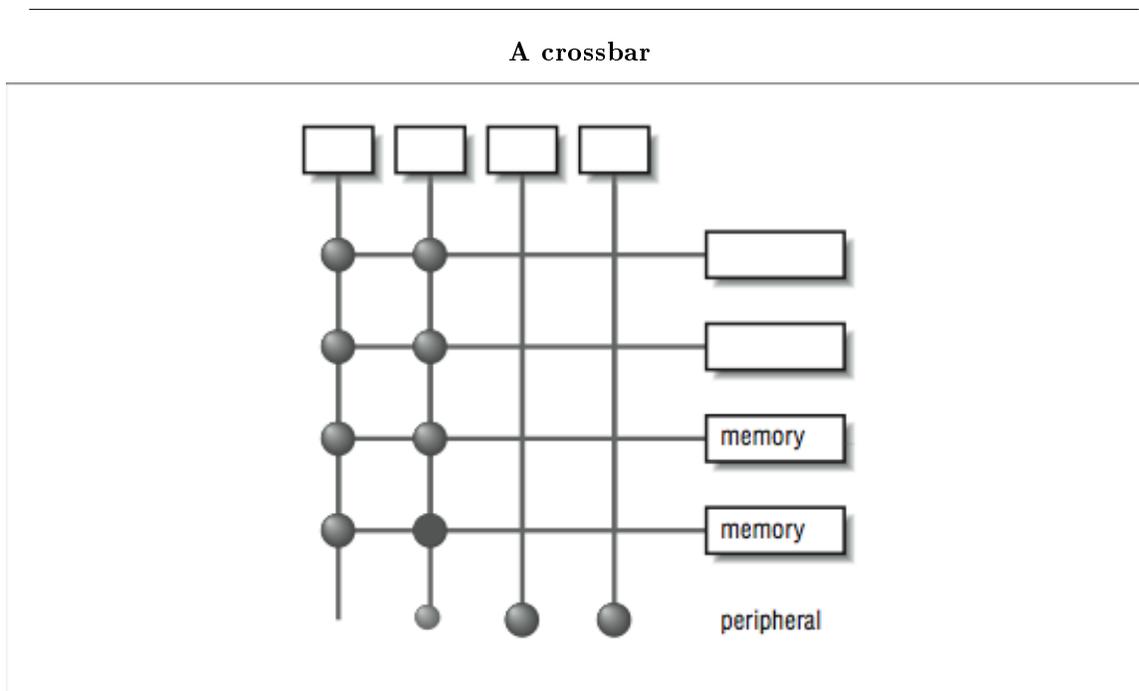


Figure 3

1 The Effect of Cache

The most common multiprocessing system is made up of commodity processors connected to memory and peripherals through a bus. Interestingly, the fact that these processors make use of cache somewhat mitigates the bandwidth bottleneck on a bus-based architecture. By connecting the processor to the cache and viewing the main memory through the cache, we significantly reduce the memory traffic across the bus. In this architecture, most of the memory accesses across the bus take the form of cache line loads and flushes. To understand why, consider what happens when the cache hit rate is very high. In Figure 4 (High cache hit rate

reduces main memory traffic), a high cache hit rate eliminates some of the traffic that would have otherwise gone out across the bus or crossbar to main memory. Again, it is the notion of “locality of reference” that makes the system work. If you assume that a fair number of the memory references will hit in the cache, the equivalent attainable main memory bandwidth is more than the bus is actually capable of. This assumption explains why multiprocessors are designed with less bus bandwidth than the sum of what the CPUs can consume at once.

Imagine a scenario where two CPUs are accessing different areas of memory using unit stride. Both CPUs access the first element in a cache line at the same time. The bus arbitrarily allows one CPU access to the memory. The first CPU fills a cache line and begins to process the data. The instant the first CPU has completed its cache line fill, the cache line fill for the second CPU begins. Once the second cache line fill has completed, the second CPU begins to process the data in its cache line. If the time to process the data in a cache line is longer than the time to fill a cache line, the cache line fill for processor two completes before the next cache line request arrives from processor one. Once the initial conflict is resolved, both processors appear to have conflict-free access to memory for the remainder of their unit-stride loops.

High cache hit rate reduces main memory traffic

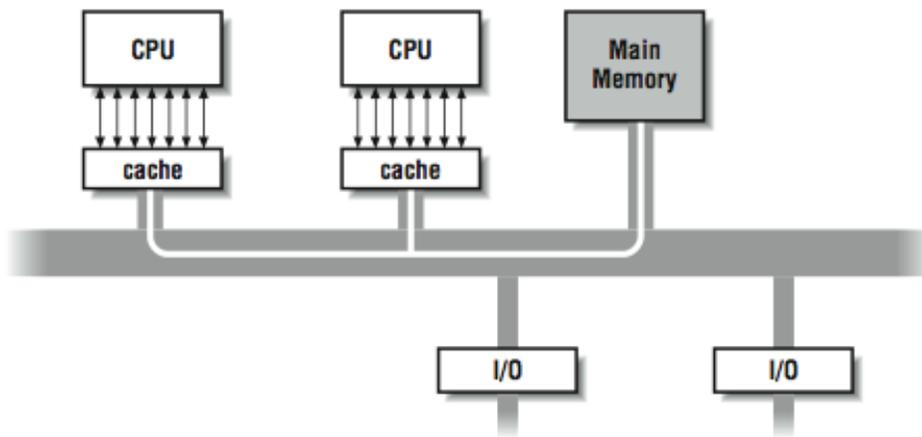


Figure 4

In actuality, on some of the fastest bus-based systems, the memory bus is sufficiently fast that up to 20 processors can access memory using unit stride with very little conflict. If the processors are accessing memory using non-unit stride, bus and memory bank conflict becomes apparent, with fewer processors.

This bus architecture combined with local caches is very popular for general-purpose multiprocessing loads. The memory reference patterns for database or Internet servers generally consist of a combination of time periods with a small working set, and time periods that access large data structures using unit stride. Scientific codes tend to perform more non-unit-stride access than general-purpose codes. For this reason, the most expensive parallel-processing systems targeted at scientific codes tend to use crossbars connected to multibanked memory systems.

The main memory system is better shielded when a larger cache is used. For this reason, multiprocessors sometimes incorporate a two-tier cache system, where each processor uses its own small on-chip local cache, backed up by a larger second board-level cache with as much as 4 MB of memory. Only when neither can satisfy a memory request, or when data has to be written back to main memory, does a request go out over the bus or crossbar.

2 Coherency

Now, what happens when one CPU of a multiprocessor running a single program in parallel changes the value of a variable, and another CPU tries to read it? Where does the value come from? These questions are interesting because there can be multiple copies of each variable, and some of them can hold old or stale values.

For illustration, say that you are running a program with a shared variable A. Processor 1 changes the value of A and Processor 2 goes to read it.

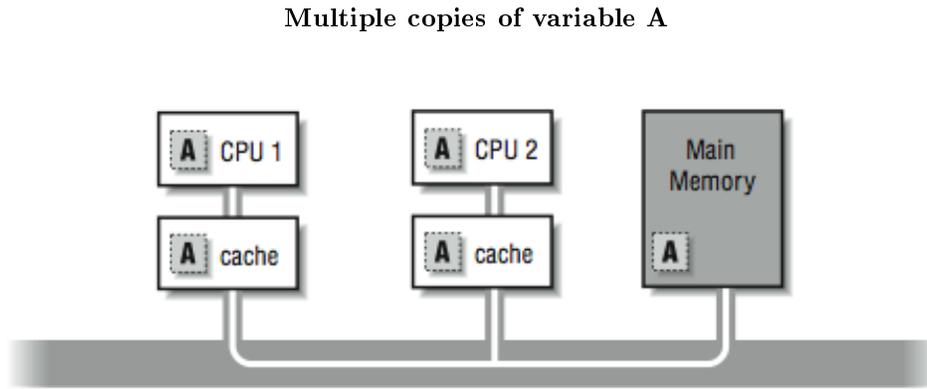


Figure 5

In Figure 5 (Multiple copies of variable A), if Processor 1 is keeping A as a register-resident variable, then Processor 2 doesn't stand a chance of getting the correct value when it goes to look for it. There is no way that 2 can know the contents of 1's registers; so assume, at the very least, that Processor 1 writes the new value back out. Now the question is, where does the new value get stored? Does it remain in Processor 1's cache? Is it written to main memory? Does it get updated in Processor 2's cache?

Really, we are asking what kind of *cache coherency protocol* the vendor uses to assure that all processors see a uniform view of the values in "memory." It generally isn't something that the programmer has to worry about, except that in some cases, it can affect performance. The approaches used in these systems are similar to those used in single-processor systems with some extensions. The most straight-forward cache coherency approach is called a *write-through policy*: variables written into cache are simultaneously written into main memory. As the update takes place, other caches in the system see the main memory reference being performed. This can be done because all of the caches continuously monitor (also known as *snooping*) the traffic on the bus, checking to see if each address is in their cache. If a cache "notices" that it contains a copy of the data from the locations being written, it may either *invalidate* its copy of the variable or obtain new values (depending on the policy). One thing to note is that a write-through cache demands a

fair amount of main memory bandwidth since each write goes out over the main memory bus. Furthermore, successive writes to the same location or bank are subject to the main memory cycle time and can slow the machine down.

A more sophisticated cache coherency protocol is called *copyback* or *writeback*. The idea is that you write values back out to main memory only when the cache housing them needs the space for something else. Updates of cached data are coordinated between the caches, by the caches, without help from the processor. Copyback caching also uses hardware that can monitor (snoop) and respond to the memory transactions of the other caches in the system. The benefit of this method over the write-through method is that memory traffic is reduced considerably. Let's walk through it to see how it works.

3 Cache Line States

For this approach to work, each cache must maintain a state for each line in its cache. The possible states used in the example include:

Modified: This cache line needs to be written back to memory.

Exclusive: There are no other caches that have this cache line.

Shared: There are read-only copies of this line in two or more caches.

Empty/Invalid: This cache line doesn't contain any useful data.

This particular coherency protocol is often called *MESI*. Other cache coherency protocols are more complicated, but these states give you an idea how multiprocessor writeback cache coherency works.

We start where a particular cache line is in memory and in none of the writeback caches on the systems. The first cache to ask for data from a particular part of memory completes a normal memory access; the main memory system returns data from the requested location in response to a cache miss. The associated cache line is marked *exclusive*, meaning that this is the only cache in the system containing a copy of the data; it is the owner of the data. If another cache goes to main memory looking for the same thing, the request is intercepted by the first cache, and the data is returned from the first cache — not main memory. Once an interception has occurred and the data is returned, the data is marked *shared* in both of the caches.

When a particular line is marked shared, the caches have to treat it differently than they would if they were the exclusive owners of the data — especially if any of them wants to modify it. In particular, a write to a shared cache entry is preceded by a broadcast message to all the other caches in the system. It tells them to invalidate their copies of the data. The one remaining cache line gets marked as *modified* to signal that it has been changed, and that it must be returned to main memory when the space is needed for something else. By these mechanisms, you can maintain cache coherence across the multiprocessor without adding tremendously to the memory traffic.

By the way, even if a variable is not shared, it's possible for copies of it to show up in several caches. On a symmetric multiprocessor, your program can bounce around from CPU to CPU. If you run for a little while on this CPU, and then a little while on that, your program will have operated out of separate caches. That means that there can be several copies of seemingly unshared variables scattered around the machine. Operating systems often try to minimize how often a process is moved between physical CPUs during context switches. This is one reason not to overload the available processors in a system.

4 Data Placement

There is one more pitfall regarding shared memory we have so far failed to mention. It involves data movement. Although it would be convenient to think of the multiprocessor memory as one big pool, we have seen that it is actually a carefully crafted system of caches, coherency protocols, and main memory. The problems come when your application causes lots of data to be traded between the caches. Each reference that falls out of a given processor's cache (especially those that require an update in another processor's cache) has to go out on the bus.

Often, it's slower to get memory from another processor's cache than from the main memory because of the protocol and processing overhead involved. Not only do we need to have programs with high locality of reference and unit stride, we also need to minimize the data that must be moved from one CPU to another.