CS 451 / 551 / ECE 541

ADVANCED COMPUTER ARCHITECTURE

SESSION no. 26

from memory, even if the memory is organized into multiple banks. Figure 5.1 shows what these multiprocessors look like. The architecture of SMPs is the topic of Section 5.2, and we explain the approach in the context of a multicore.

The alternative design approach consists of multiprocessors with physically distributed memory, called *distributed shared memory* (DSM). Figure 5.2 shows what these multiprocessors look like. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise, the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the size of a multiprocessor for which distributed memory is preferred continues to shrink. The introduction of multicore processors has meant that even two-chip multiprocessors use distributed memory. The larger number of processors also raises the need for a high-bandwidth interconnect, of which we will see examples in Appendix F. Both

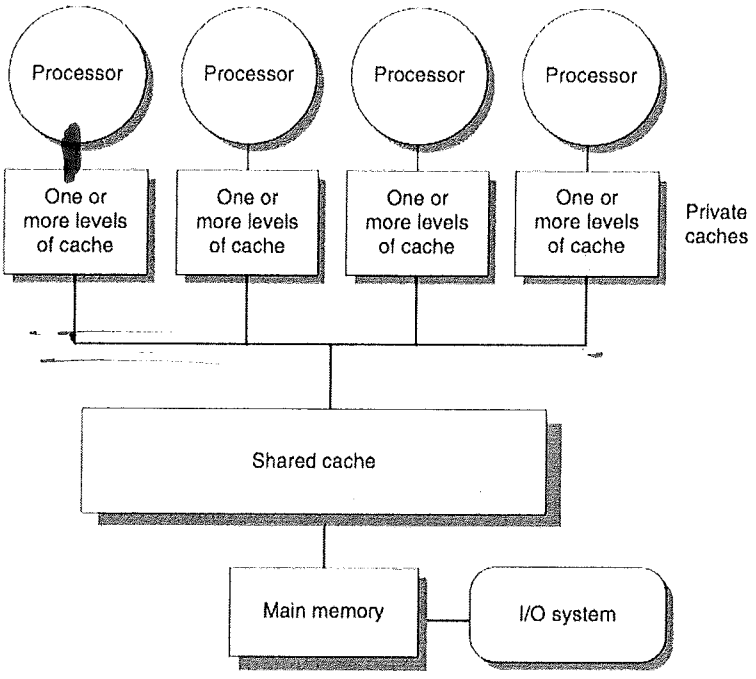


Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor–cache subsystems share the same physical memory, typically with one level of shared cache, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip version the shared cache would be omitted and the bus or interconnection network connecting the processors to memory would run between chips as opposed to within a single chip.

Thus, our focus will be on multiprocessors with a small to moderate number of processors (2 to 32). Such designs vastly dominate in terms of both units and dollars. We will pay only slight attention to the larger-scale multiprocessor design space (33 or more processors), primarily in Appendix I, which covers more aspects of the design of such processors, as well as the behavior performance for parallel scientific workloads, a primary class of applications for large-scale multiprocessors. In large-scale multiprocessors, the interconnection networks are a critical part of the design; Appendix F focuses on that topic.

Multiprocessor Architecture: Issues and Approach

To take advantage of an MIMD multiprocessor with n processors, we must usually have at least n threads or processes to execute. The independent threads within a single process are typically identified by the programmer or created by the operating system (from multiple independent requests). At the other extreme, a thread may consist of a few tens of iterations of a loop, generated by a parallel compiler exploiting data parallelism in the loop. Although the amount of computation assigned to a thread, called the *grain size*, is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction from instruction-level parallelism is that thread-level parallelism is identified at a high level by the software system or programmer and that the threads consist of hundreds to millions of instructions that may be executed in parallel.

Threads can also be used to exploit data-level parallelism, although the overhead is likely to be higher than would be seen with an SIMD processor or with a GPU (see Chapter 4). This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. For example, although a vector processor or GPU may be able to efficiently parallelize operations on short vectors, the resulting grain size when the parallelism is split among many threads may be so small that the overhead makes the exploitation of the parallelism prohibitively expensive in an MIMD.

Existing shared-memory multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization because what constitutes a small or large number of processors is likely to change over time.

The first group, which we call *symmetric (shared-memory) multiprocessors* (SMPs), or *centralized shared-memory multiprocessors*, features small numbers of cores, typically eight or fewer. For multiprocessors with such small processor counts, it is possible for the processors to share a single centralized memory that all processors have equal access to, hence the term *symmetric*. In multicore chips, the memory is effectively shared in a centralized fashion among the cores, and all existing multicores are SMPs. When more than one multicore is connected, there are separate memories for each multicore, so the memory is distributed rather than centralized.

SMP architectures are also sometimes called *uniform memory access* (UMA) multiprocessors, arising from the fact that all processors have a uniform latency

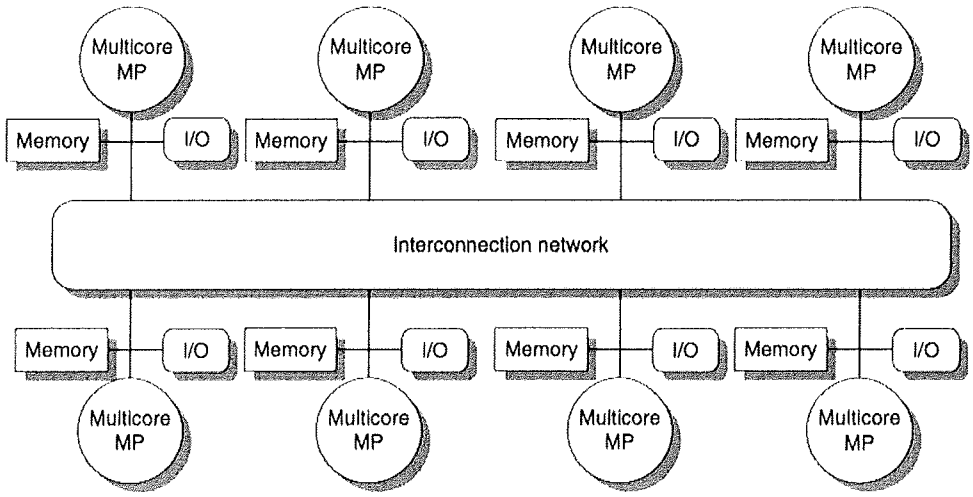


Figure 5.2 The basic architecture of a distributed-memory multiprocessor in 2011 typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes. Each processor core shares the entire memory, although the access time to the lock memory attached to the core's chip will be much faster than the access time to remote memories.

directed networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used.

Distributing the memory among the nodes both increases the bandwidth and reduces the latency to local memory. A DSM multiprocessor is also called a *NUMA* (nonuniform memory access), since the access time depends on the location of a data word in memory. The key disadvantages for a DSM are that communicating data among processors becomes somewhat more complex, and a DSM requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories. Because all multicore-based multiprocessors with more than one processor chip (or socket) use distributed memory, we will explain the operation of distributed memory multiprocessors from this viewpoint.

In both SMP and DSM architectures, communication among threads occurs through a shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. The term *shared memory* associated with both SMP and DSM refers to the fact that the *address space* is shared.

In contrast, the clusters and warehouse-scale computers of the next chapter look like individual computers connected by a network, and the memory of one processor cannot be accessed by another processor without the assistance of software protocols running on both processors. In such designs, message-passing protocols are used to communicate data among processors.

Challenges of Parallel Processing

The application of multiprocessors ranges from running independent tasks with essentially no communication to running parallel programs where threads must communicate to complete the task. Two important hurdles, both explainable with Amdahl's law, make parallel processing challenging. The degree to which these hurdles are difficult or easy is determined both by the application and by the architecture.

The first hurdle has to do with the limited parallelism available in programs, and the second arises from the relatively high cost of communications. Limitations in available parallelism make it difficult to achieve good speedups in any parallel processor, as our first example shows.

Example Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

Answer Recall from Chapter 1 that Amdahl's law is

$$\text{Speedup} = \frac{1}{\dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

For simplicity in this example, assume that the program operates in only two modes: parallel with all processors fully used, which is the enhanced mode, or serial with only one processor in use. With this simplification, the speedup in enhanced mode is simply the number of processors, while the fraction of enhanced mode is the time spent in parallel mode. Substituting into the previous equation:

$$80 = \frac{1}{\dfrac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields:

$$\begin{aligned} 0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) &= 1 \\ 80 - 79.2 \times \text{Fraction}_{\text{parallel}} &= 1 \\ \text{Fraction}_{\text{parallel}} &= \frac{80 - 1}{79.2} \\ \text{Fraction}_{\text{parallel}} &= 0.9975 \end{aligned}$$

Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of the original computation can be sequential. Of course, to achieve linear speedup (speedup of n with n processors), the entire program must usually be parallel with no serial portions. In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors when running in parallel mode.

Basic Schemes for Enforcing Coherence

The coherence problem for multiprocessors and I/O, although similar in origin, has different characteristics that affect the appropriate solution. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will normally have copies of the same data in several caches. In a coherent multiprocessor, the caches provide both *migration* and *replication* of shared data items.

Coherent caches provide migration, since a data item can be moved to a local cache and used there in a transparent fashion. This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.

Coherent caches also provide replication for shared data that are being simultaneously read, since the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item. Supporting this migration and replication is critical to performance in accessing shared data. Thus, rather than trying to solve the problem by avoiding it in software, multiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches.

The protocols to maintain coherence for multiple processors are called *cache coherence protocols*. Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block. There are two classes of protocols in use, each of which uses different techniques to track the sharing status:

- *Directory based*—The sharing status of a particular block of physical memory is kept in one location, called the *directory*. There are two very different types of directory-based cache coherence. In an SMP, we can use one centralized directory, associated with the memory or some other single serialization point, such as the outermost cache in a multicore. In a DSM, it makes no sense to have a single directory, since that would create a single point of contention and make it difficult to scale to many multicore chips given the memory demands of multicores with eight or more cores. Distributed directories are more complex than a single directory, and such designs are the subject of Section 5.4.
- *Snooping*—Rather than keeping the state of sharing in a single directory, every cache that has a copy of the data from a block of physical memory could track the sharing status of the block. In an SMP, the caches are typically all accessible via some broadcast medium (e.g., a bus connects the per-core caches to the shared cache or memory), and all cache controllers monitor or *snoop* on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access. Snooping can also be used as the coherence protocol for a multichip multiprocessor, and some designs support a snooping protocol on top of a directory protocol within each multicore!

Snooping protocols became popular with multiprocessors using microprocessors (single-core) and caches attached to a single shared memory by a bus.

The bus provided a convenient broadcast medium to implement the snooping protocols. Multicore architectures changed the picture significantly, since all multicores share some level of cache on the chip. Thus, some designs switched to using directory protocols, since the overhead was small. To allow the reader to become familiar with both types of protocols, we focus on a snooping protocol here and discuss a directory protocol when we come to DSM architectures.

Snooping Coherence Protocols

There are two ways to maintain the coherence requirement described in the prior subsection. One method is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a write. It is by far the most common protocol. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: All other cached copies of the item are invalidated.

Figure 5.4 shows an example of an invalidation protocol with write-back caches in action. To see how this protocol ensures coherence, consider a write followed by a read by another processor: Since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence, the protocol name). Thus, when the read occurs, it misses in the cache and is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

Figure 5.4 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The processor and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, processor A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it. If a multicore uses a shared cache (e.g., L3), then all memory is seen through the shared cache; L3 acts like the memory in this example, and coherency must be handled for the private L1 and L2 for each core. It is this observation that led some designers to opt for a directory protocol within the multicore. To make this work the L3 cache must be inclusive (see page 397).

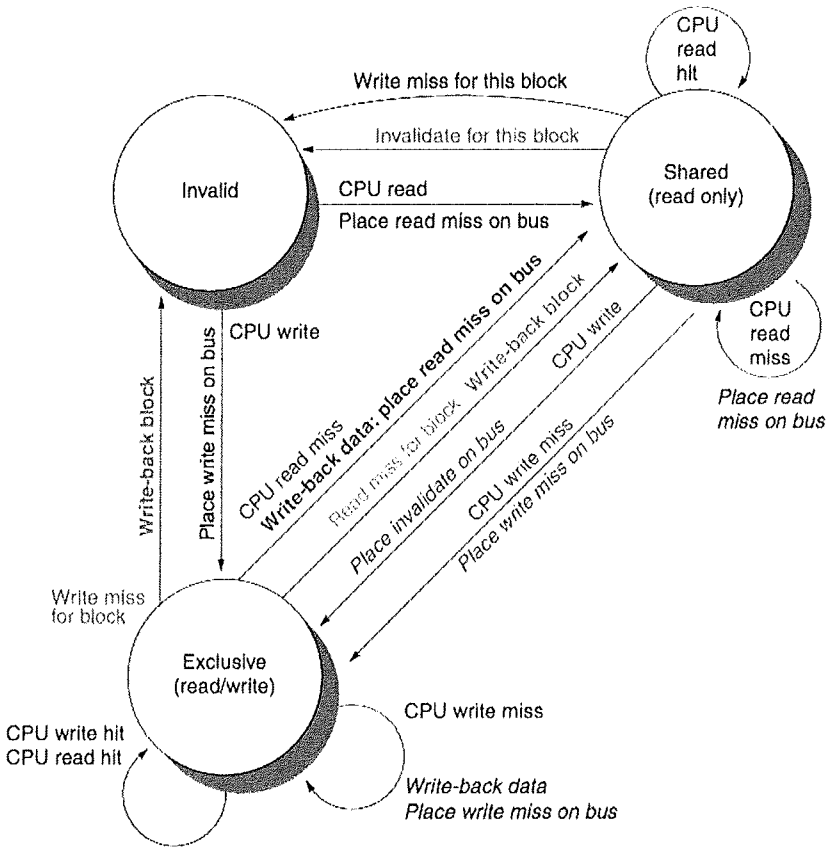


Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. As in Figure 5.6, the activities on a transition are shown in bold.

receive a response as a single atomic action. In reality this is not true. In fact, even a read miss might not be atomic; after detecting a miss in the L2 of a multicore, the core must arbitrate for access to the bus connecting to the shared L3. Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. We will explore these complications later in this section and when we examine DSM designs.

With multicore processors, the coherence among the processor cores is all implemented on chip, using either a snooping or simple central directory protocol. Many dual-processor chips, including the Intel Xeon and AMD Opteron, supported multichip multiprocessors that could be built by connecting a high-speed interface (called Quickpath or Hypertransport, respectively). These next-level interconnects are not just extensions of the shared bus, but use a different approach for interconnecting multicores.

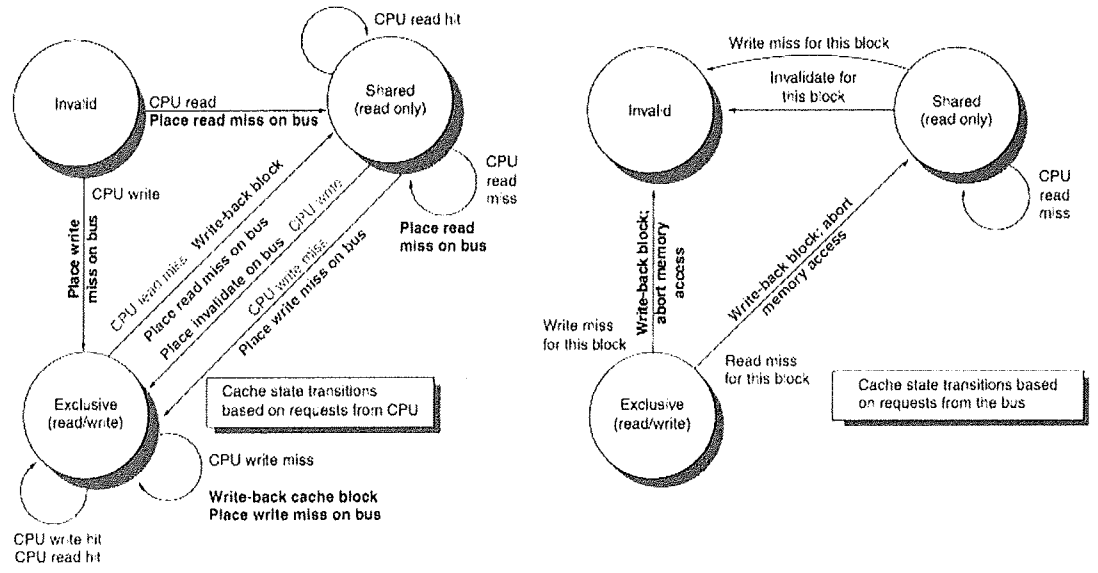


Figure 5.6 A write invalidate, cache coherence protocol for a private write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles, with any access permitted by the local processor without a state transition shown in parentheses under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the private cache, not to a specific address in the cache. Hence, a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the local cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state generates an invalidate. Whenever a bus transaction occurs, all private caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory (or a shared cache) provides data on a read miss for a block that is clean in all local caches. In actual implementations, these two sets of state diagrams are combined. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss. In a multicore chip, the shared cache (usually L3, but sometimes L2) acts as the equivalent of memory, and the bus is the bus between the private caches of each core and the shared cache, which in turn interfaces to the memory.

or memory if there is no shared cache), which simplifies the implementation. In fact, it does not matter whether the level out from the private caches is a shared cache or memory; the key is that all accesses from the cores go through that level.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and

Although the clock rate of the Alpha processor in this system is considerably slower than processors in systems designed in 2011, the basic structure of the system, consisting of a four-issue processor and a three-level cache hierarchy, is very similar to the multicore Intel i7 and other processors, as shown in Figure 5.9. In particular, the Alpha caches are somewhat smaller, but the miss times are also lower than on an i7. Thus, the behavior of the Alpha system should provide interesting insights into the behavior of modern multicore designs.

The workload used for this study consists of three applications:

1. An online transaction-processing (OLTP) workload modeled after TPC-B (which has memory behavior similar to its newer cousin TPC-C, described in Chapter 1) and using Oracle 7.3.2 as the underlying database. The workload consists of a set of client processes that generate requests and a set of servers that handle them. The server processes consume 85% of the user time, with the remaining going to the clients. Although the I/O latency is hidden by careful tuning and enough requests to keep the processor busy, the server processes typically block for I/O after about 25,000 instructions.
2. A decision support system (DSS) workload based on TPC-D, the older cousin of the heavily used TPC-E, which also uses Oracle 7.3.2 as the underlying database. The workload includes only 6 of the 17 read queries in TPC-D,

VAX

Cache level	Characteristic	Alpha 21164	Intel i7
L1	Size	8 KB I/8 KB D	32 KB I/32 KB D
	Associativity	Direct mapped	4-way I/8-way D
	Block size	32 B	64 B
	Miss penalty	7	10
L2	Size	96 KB	256 KB
	Associativity	3-way	8-way
	Block size	32 B	64 B
	Miss penalty	21	35
L3	Size	2 MB	2 MB per core
	Associativity	Direct mapped	16-way
	Block size	64 B	64 B
	Miss penalty	80	~100

Figure 5.9 The characteristics of the cache hierarchy of the Alpha 21164 used in this study and the Intel i7. Although the sizes are larger and the associativity is higher on the i7, the miss penalties are also higher, so the behavior may differ only slightly. For example, from Appendix B, we can estimate the miss rates of the smaller Alpha L1 cache as 4.9% and 3% for the larger i7 L1 cache, so the average L1 miss penalty per reference is 0.34 for the Alpha and 0.30 for the i7. Both systems have a high penalty (125 cycles or more) for a transfer required from a private cache. The i7 also shares its L3 among all the cores.

Benchmark	% Time user mode	% Time kernel	% Time processor idle
OLTP	71	18	11
DSS (average across all queries)	87	4	9
AltaVista	>98	<1	<1

Figure 5.10 The distribution of execution time in the commercial workloads. The OLTP benchmark has the largest fraction of both OS time and processor idle time (which is I/O wait time). The DSS benchmark shows much less OS time, since it does less I/O, but still more than 9% idle time. The extensive tuning of the AltaVista search engine is clear in these measurements. The data for this workload were collected by Barroso, Gharachorloo, and Bugnion [1998] on a four-processor AlphaServer 4100.

although the 6 queries examined in the benchmark span the range of activities in the entire benchmark. To hide the I/O latency, parallelism is exploited both within queries, where parallelism is detected during a query formulation process, and across queries. Blocking calls are much less frequent than in the OLTP benchmark; the 6 queries average about 1.5 million instructions before blocking.

3. A Web index search (AltaVista) benchmark based on a search of a memory-mapped version of the AltaVista database (200 GB). The inner loop is heavily optimized. Because the search structure is static, little synchronization is needed among the threads. AltaVista was the most popular Web search engine before the arrival of Google.

Figure 5.10 shows the percentages of time spent in user mode, in the kernel, and in the idle loop. The frequency of I/O increases both the kernel time and the idle time (see the OLTP entry, which has the largest I/O-to-computation ratio). AltaVista, which maps the entire search database into memory and has been extensively tuned, shows the least kernel or idle time.

Performance Measurements of the Commercial Workload

We start by looking at the overall processor execution for these benchmarks on the four-processor system; as discussed on page 367, these benchmarks include substantial I/O time, which is ignored in the processor time measurements. We group the six DSS queries as a single benchmark, reporting the average behavior. The effective CPI varies widely for these benchmarks, from a CPI of 1.3 for the AltaVista Web search, to an average CPI of 1.6 for the DSS workload, to 7.0 for the OLTP workload. Figure 5.11 shows how the execution time breaks down into instruction execution, cache and memory system access time, and other stalls (which are primarily pipeline resource stalls but also include translation lookaside buffer (TLB) and branch mispredict stalls). Although the performance of the DSS

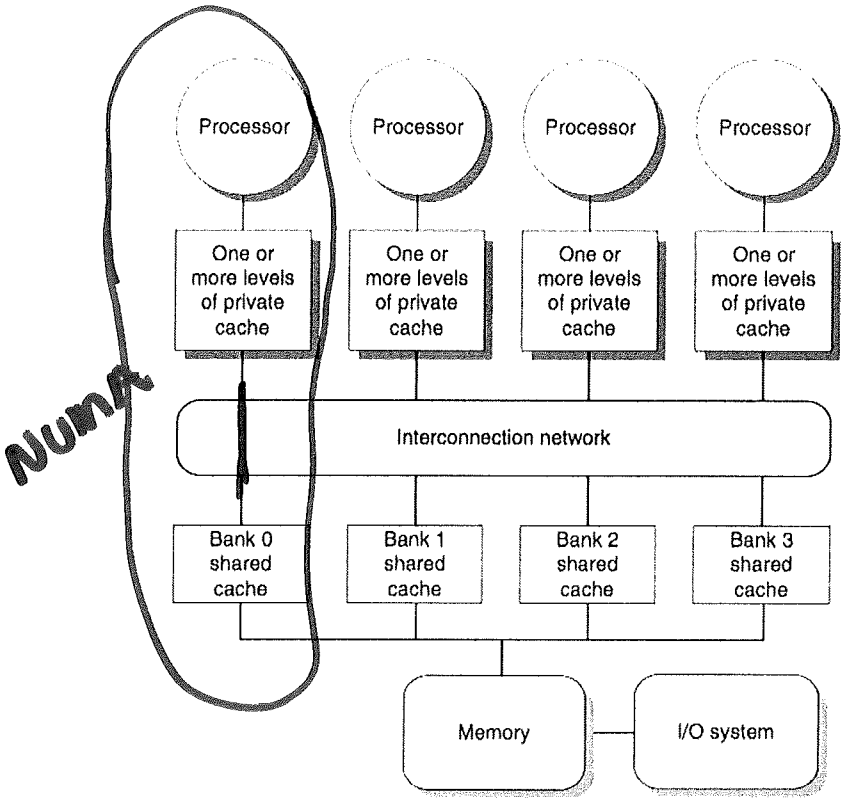


Figure 5.8 A multicore single-chip multiprocessor with uniform memory access through a banked shared cache and using an interconnection network rather than a bus.

find potentially shared copies, like a snooping protocol, but uses the acknowledgments to order operations, like a directory protocol. Because local memory is only somewhat faster than remote memory in the Opteron implementation, some software treats an Opteron multiprocessor as having uniform memory access.

A snooping cache coherence protocol can be used without a centralized bus, but still requires that a broadcast be done to snoop the individual caches on every miss to a potentially shared cache block. This cache coherence traffic creates another limit on the scale and the speed of the processors. Because coherence traffic is unaffected by larger caches, faster processors will inevitably overwhelm the network and the ability of each cache to respond to snoop requests from *all* the other caches. In Section 5.4, we examine directory-based protocols, which eliminate the need for broadcast to all caches on a miss. As processor speeds and the number of cores per processor increase, more designers are likely to opt for such protocols to avoid the broadcast limit of a snooping protocol.

Implementing Snooping Cache Coherence

The devil is in the details.

Classic proverb

When we wrote the first edition of this book in 1990, our final "Putting It All Together" was a 30-processor, single-bus multiprocessor using snoop-based coherence; the bus had a capacity of just over 50 MB/sec, which would not be enough bus bandwidth to support even one core of an Intel i7 in 2011! When we wrote the second edition of this book in 1995, the first cache coherence multiprocessors with more than a single bus had recently appeared, and we added an appendix describing the implementation of snooping in a system with multiple buses. In 2011, most multicore processors that support only a single-chip multiprocessor have opted to use a shared bus structure connecting to either a shared memory or a shared cache. In contrast, *every* multicore multiprocessor system that supports 16 or more cores uses an interconnect other than a single bus, and designers must face the challenge of implementing snooping without the simplification of a bus to serialize events.

As we said earlier, the major complication in actually implementing the snooping coherence protocol we have described is that write and upgrade misses are not atomic in any recent multiprocessor. The steps of detecting a write or upgrade miss, communicating with the other processors and memory, getting the most recent value for a write miss and ensuring that any invalidates are processed, and updating the cache cannot be done as if they took a single cycle.

In a single multicore chip, these steps can be made effectively atomic by arbitrating for the bus to the shared cache or memory first (before changing the cache state) and not releasing the bus until all actions are complete. How can the processor know when all the invalidates are complete? In some multicores, a single line is used to signal when all necessary invalidates have been received and are being processed. Following that signal, the processor that generated the miss can release the bus, knowing that any required actions will be completed before any activity related to the next miss. By holding the bus exclusively during these steps, the processor effectively makes the individual steps atomic.

In a system without a bus, we must find some other method of making the steps in a miss atomic. In particular, we must ensure that two processors that attempt to write the same block at the same time, a situation which is called a *race*, are strictly ordered: One write is processed and precedes before the next is begun. It does not matter which of two writes in a race wins the race, just that there be only a single winner whose coherence actions are completed first. In a snooping system, ensuring that a race has only one winner is accomplished by using broadcast for all misses as well as some basic properties of the interconnection network. These properties, together with the ability to restart the miss handling of the loser in a race, are the keys to implementing snooping cache coherence without a bus. We explain the details in Appendix I.

A multiprocessor built with multiple multicore chips will have a distributed memory architecture and will need an interchip coherency mechanism above and beyond the one within the chip. In most cases, some form of directory scheme is used.

Extensions to the Basic Coherence Protocol

The coherence protocol we have just described is a simple three-state protocol and is often referred to by the first letter of the states, making it a MSI (Modified, Shared, Invalid) protocol. There are many extensions of this basic protocol, which we mentioned in the captions of figures in this section. These extensions are created by adding additional states and transactions, which optimize certain behaviors, possibly resulting in improved performance. Two of the most common extensions are

1. *MESI* adds the state Exclusive to the basic MSI protocol to indicate when a cache block is resident only in a single cache but is clean. If a block is in the E state, it can be written without generating any invalidates, which optimizes the case where a block is read by a single cache before being written by that same cache. Of course, when a read miss to a block in the E state occurs, the block must be changed to the S state to maintain coherence. Because all subsequent accesses are snooped, it is possible to maintain the accuracy of this state. In particular, if another processor issues a read miss, the state is changed from exclusive to shared. The advantage of adding this state is that a subsequent write to a block in the exclusive state by the same core need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this local cache; the processor merely changes the state to modified. This state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a bock is modified. The popular MESI protocol, which is named for the four states it includes (Modified, Exclusive, Shared, and Invalid), uses this structure. The Intel i7 uses a variant of a MESI protocol, called MESIF, which adds a state (Forward) to designate which sharing processor should respond to a request. It is designed to enhance performance in distributed memory organizations.
2. *MOESI* adds the state Owned to the MESI protocol to indicate that the associated block is owned by that cache and out-of-date in memory. In MSI and MESI protocols, when there is an attempt to share a block in the Modified state, the state is changed to Shared (in both the original and newly sharing cache), and the block must be written back to memory. In a MOESI protocol, the block can be changed from the Modified to Owned state in the original cache without writing it to memory. Other caches, which are newly sharing the block, keep the block in the Shared state; the O state, which only the original cache holds, indicates that the main memory copy is out of date and that the designated cache is the owner. The owner of the block must supply it on a miss, since memory is not up to date and must write the block back to memory if it is replaced. The AMD Opteron uses the MOESI protocol.

The next section examines the performance of these protocols for our parallel and multiprogrammed workloads; the value of these extensions to a basic protocol will be clear when we examine the performance. But, before we do that, let's take a brief look at the limitations on the use of a symmetric memory structure and a snooping coherence scheme.

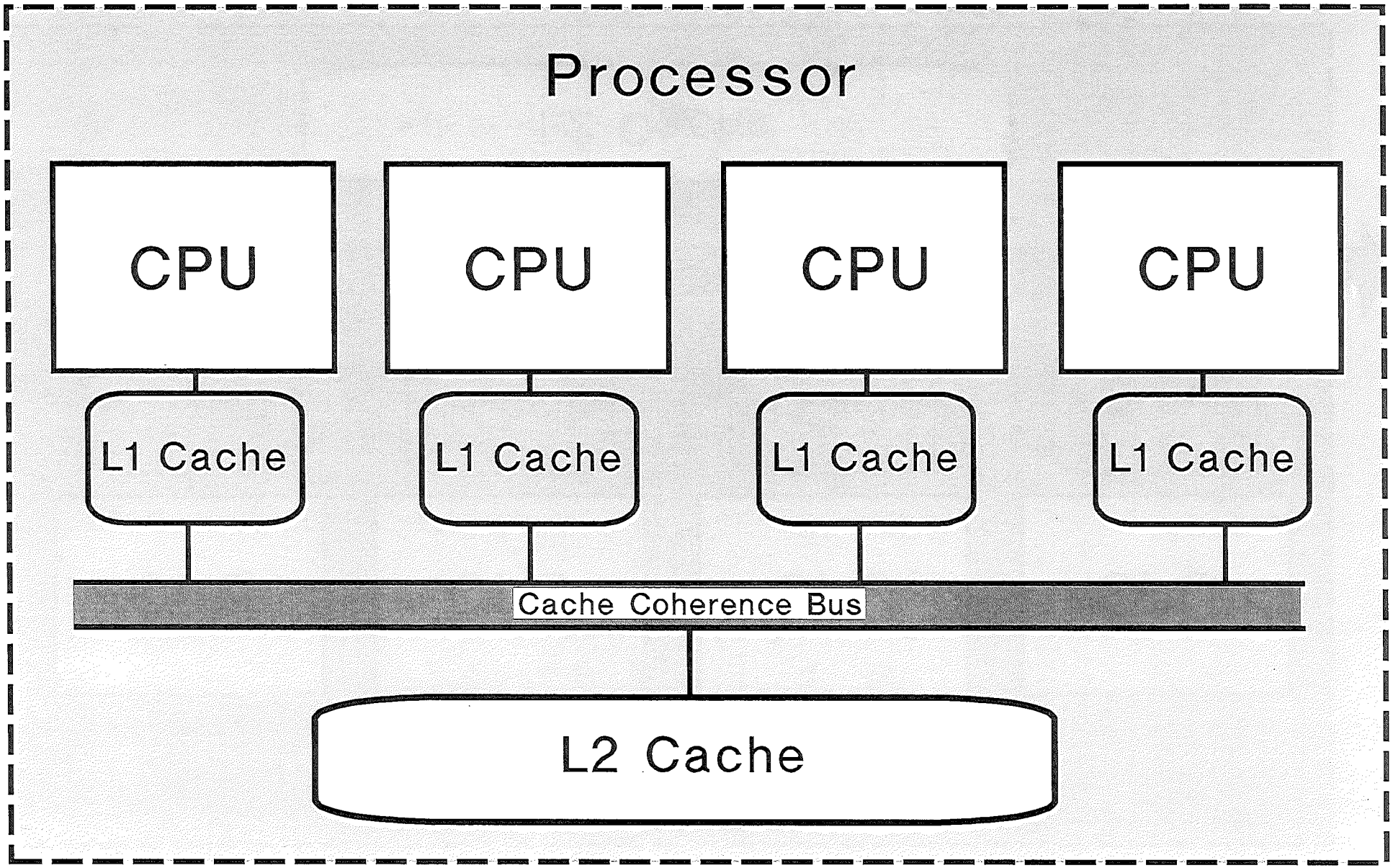
Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

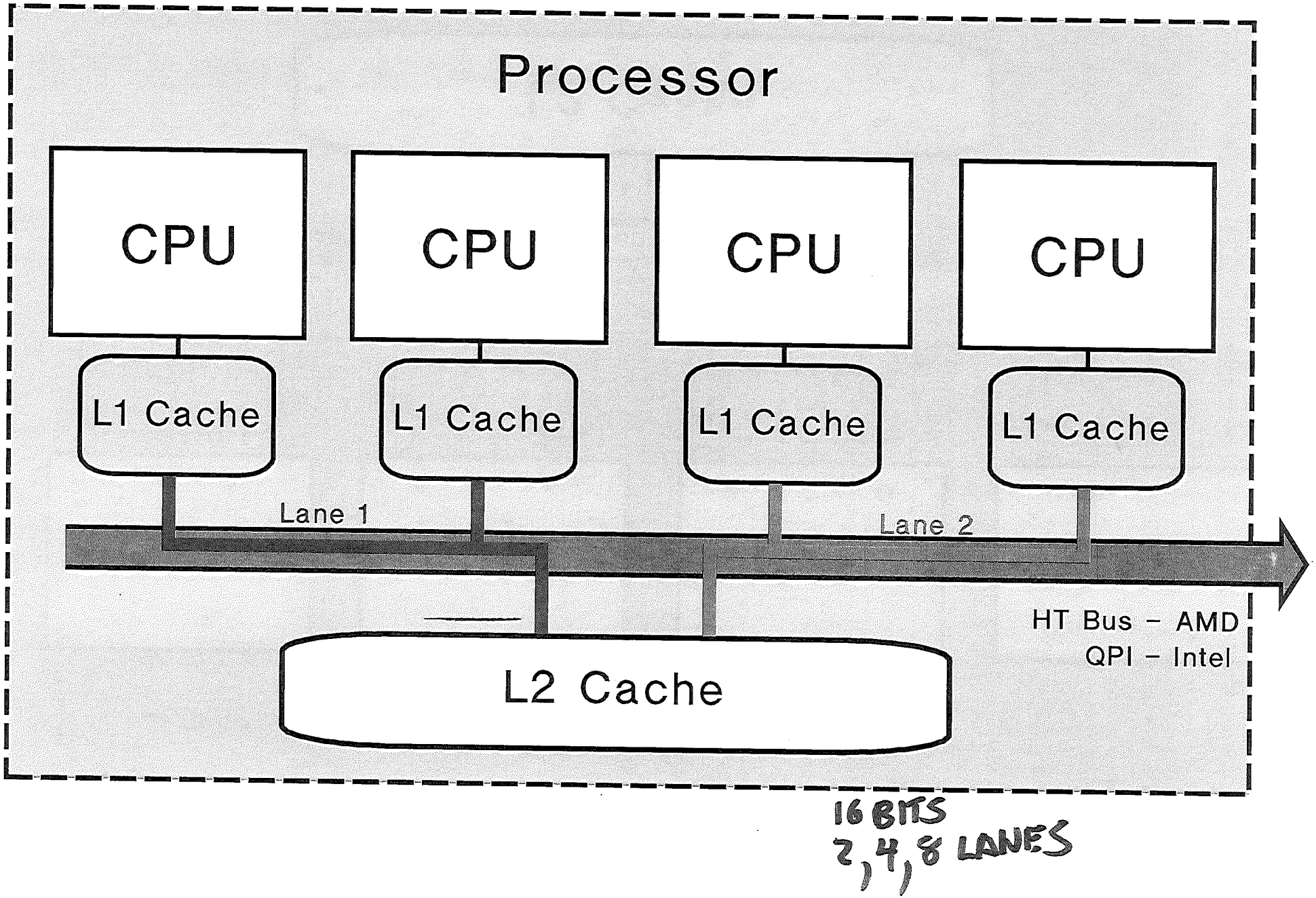
As the number of processors in a multiprocessor grows, or as the memory demands of each processor grow, any centralized resource in the system can become a bottleneck. Using the higher bandwidth connection available on-chip and a shared L3 cache, which is faster than memory, designers have managed to support four to eight high-performance cores in a symmetric fashion. Such an approach is unlikely to scale much past eight cores, and it will not work once multiple multicores are combined.

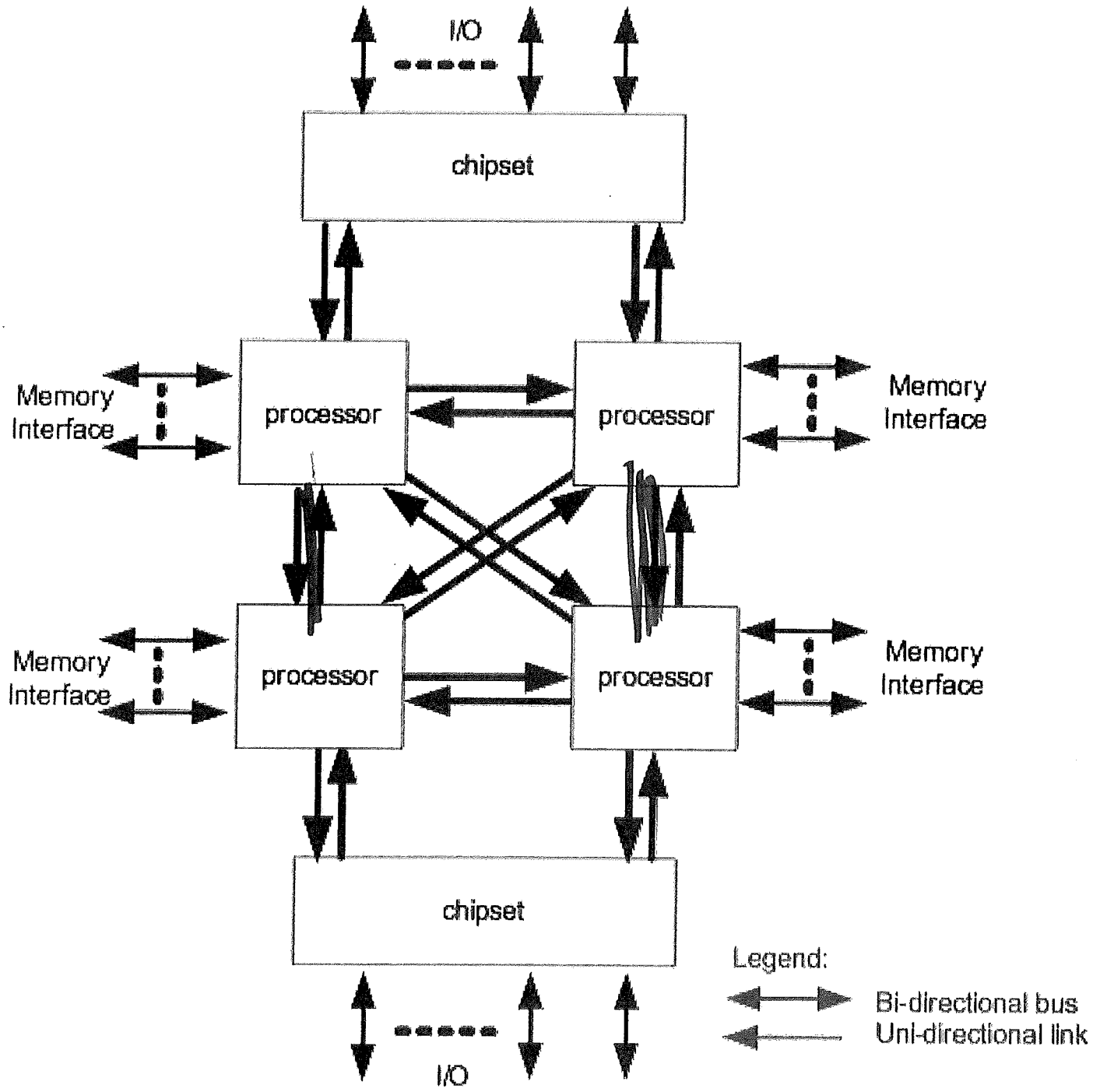
Snooping bandwidth at the caches can also become a problem, since every cache must examine every miss placed on the bus. As we mentioned, duplicating the tags is one solution. Another approach, which has been adopted in some recent multicores, is to place a directory at the level of the outermost cache. The directory explicitly indicates which processor's caches have copies of every item in the outermost cache. This is the approach Intel uses on the i7 and Xeon 7000 series. Note that the use of this directory does not eliminate the bottleneck due to a shared bus and L3 among the processors, but it is much simpler to implement than the distributed directory schemes that we will examine in Section 5.4.

How can a designer increase the memory bandwidth to support either more or faster processors? To increase the communication bandwidth between processors and memory, designers have used multiple buses as well as interconnection networks, such as crossbars or small point-to-point networks. In such designs, the memory system (either main memory or a shared cache) can be configured into multiple physical banks, so as to boost the effective memory bandwidth while retaining uniform access time to memory. Figure 5.8 shows how such a system might look if it where implemented with a single-chip multicore. Although such an approach might be used to allow more than four cores to be interconnected on a single chip, it does not scale well to a multichip multiprocessor that uses multicore building blocks, since the memory is already attached to the individual multicore chips, rather than centralized.

The AMD Opteron represents another intermediate point in the spectrum between a snooping and a directory protocol. Memory is directly connected to each multicore chip, and up to four multicore chips can be connected. The system is a NUMA, since local memory is somewhat faster. The Opteron implements its coherence protocol using the point-to-point links to broadcast up to three other chips. Because the interprocessor links are not shared, the only way a processor can know when an invalid operation has completed is by an explicit acknowledgment. Thus, the coherence protocol uses a broadcast to


Processor
CPU
CPU
CPU
CPU
L1 Cache
L1 Cache
L1 Cache
L1 Cache
Cache Coherence Bus
L2 Cache


Processor
CPU
CPU
CPU
CPU
L1 Cache
L1 Cache
L1 Cache
L1 Cache
Lane 1
Lane 2
HT Bus – AMD
QPI – Intel
L2 Cache
16 BITS
2, 4, 8 LANES


I/O
chipset
Memory Interface
processor
processor
Memory Interface
Memory Interface
processor
processor
Memory Interface
chipset
I/O
Legend:
Bi-directional bus
Uni-directional link

HYPERTRANSPORT - HT - SER/PAR

~~I/O~~

MEMORY

PCIe - SER

QPI