Advanced Topics in Computer Architecture

Lecture 5

Multiprocessors and Thread-Level Parallelism

Marenglen Biba
Department of Computer Science
University of New York Tirana

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

The Trend

- We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini,

describing Intel's future direction at the

Intel Developers Forum in 2005

Introduction

- As we have discussed so far, in the previous lectures, the slowdown in uniprocessor performance arising from diminishing returns in exploiting ILP, combined with growing concern over power, is leading to a new era in computer architecture — an era where multiprocessors play a major role.

Other factors

This trend toward more reliance on multiprocessing is reinforced by other factors:

- A growing interest in servers and server performance
- A growth in data-intensive applications
- The insight that increasing performance on the desktop is less important (outside of graphics, at least)
- An improved understanding of how to use multiprocessors effectively, especially in server environments where there is significant natural thread-level parallelism
- The advantages of leveraging a design investment by replication rather than unique design — all multiprocessor designs provide such leverage

A Taxonomy of Parallel Architectures

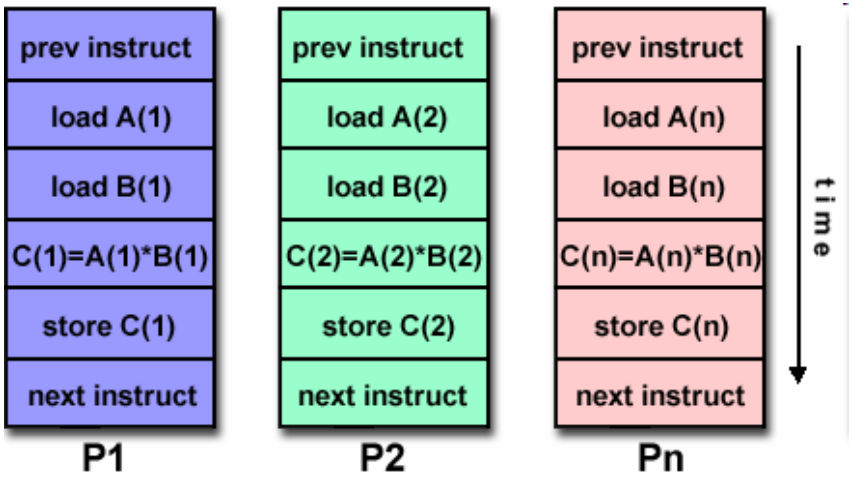
- The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 40 years ago, Flynn [1966] proposed a simple model of categorizing all computers that is still useful today.
- He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor, and placed all computers into one of four categories:

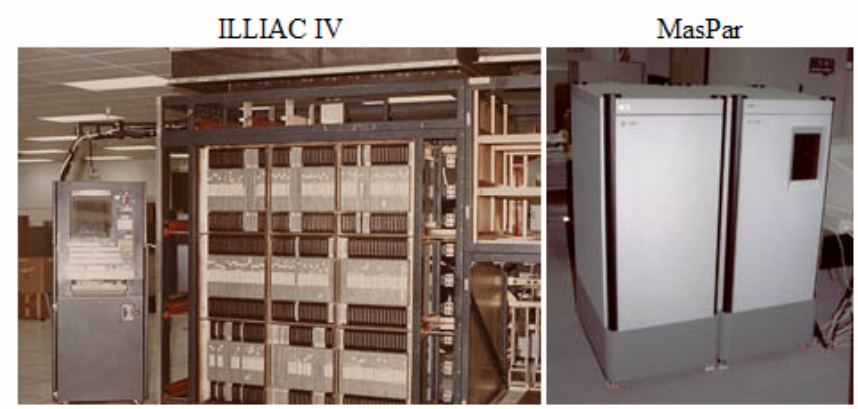
- 1. *Single instruction stream*, *single data stream* (SISD) — This category is the uniprocessor.

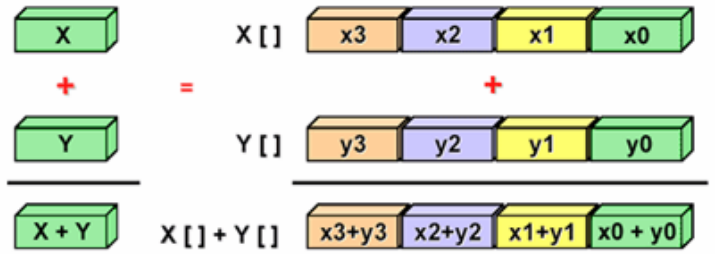
SIMD: Single instruction stream, multiple data streams

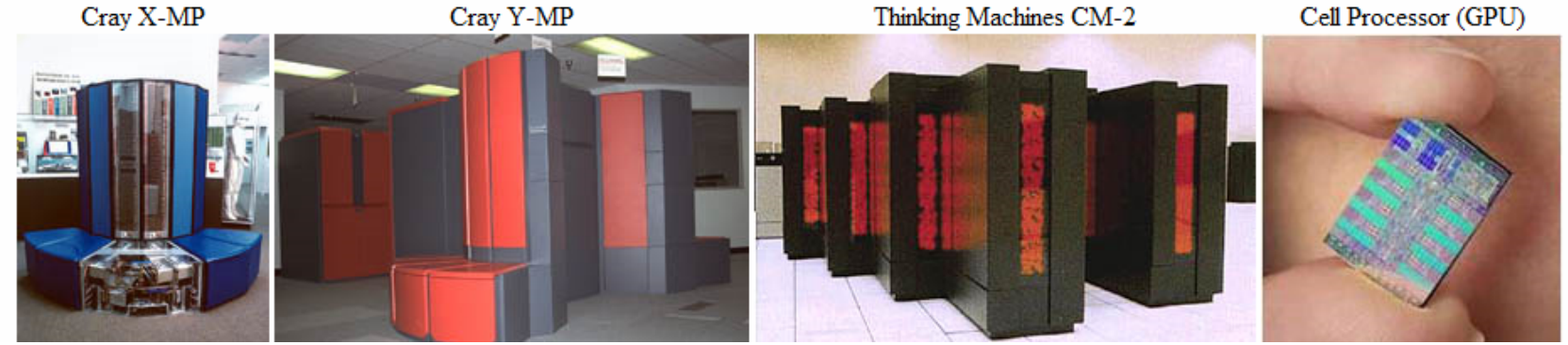
- The same instruction is executed by multiple processors using different data streams.
- SIMD computers exploit *data-level parallelism* by applying the same operations to multiple items of data in parallel.
- Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions.
- For applications that display significant data-level parallelism, the SIMD approach can be very efficient.
- Vector architectures, are the largest class of SIMD architectures.
- SIMD approaches have experienced a rebirth in the last few years with the growing importance of graphics performance, especially for the game market.
- SIMD approaches are the favored method for achieving the high performance needed to create realistic threedimensional, real-time virtual environments.

SIMD


prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
load A(2)
load B(2)
C(2)=A(2)*B(2)
store C(2)
next instruct
P2
prev instruct
load A(n)
load B(n)
C(n)=A(n)*B(n)
store C(n)
next instruct
Pn
time


ILLIAC IV
MasPar


X
X [ ]
x3
x2
x1
x0
+
=
+
Y
Y [ ]
y3
y2
y1
y0
X + Y
X [ ] + Y [ ]
x3+y3
x2+y2
x1+y1
x0 + y0


Cray X-MP
Cray Y-MP
Thinking Machines CM-2
Cell Processor (GPU)

MISD and MIMD

3. *Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date.
4. *Multiple instruction streams, multiple data streams* (MIMD)
   - Each processor fetches its own instructions and operates on its own data.
   - MIMD computers exploit *thread-level parallelism*, since multiple threads operate in parallel.

- In general, thread-level parallelism is more flexible than data-level parallelism and thus more generally applicable.

MIMD Advantages

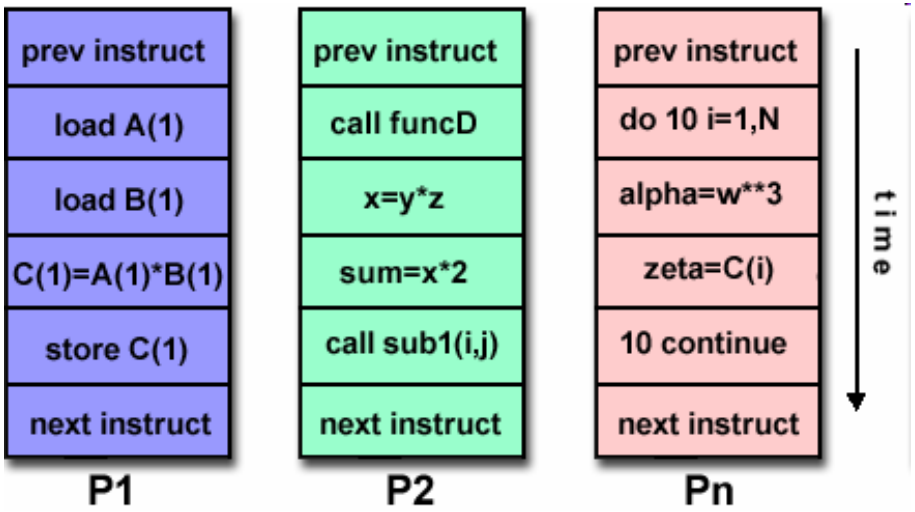
- Because the MIMD model can exploit thread-level parallelism, it is the architecture of choice for general-purpose multiprocessors and our focus in this Lecture.
- Two other factors have also contributed to the rise of the MIMD multiprocessors:

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.
2. MIMDs can build on the cost-performance advantages of off-the-shelf processors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers. Furthermore, multicore chips leverage the design investment in a single processor core by replicating it.

MIMD as Clusters

- One popular class of MIMD computers are *clusters* , which often use standard components and often standard network technology, so as to leverage as much commodity technology as possible.
- We distinguish two different types of clusters:
  - *commodity clusters*, which rely entirely on third-party processors and interconnection technology
  - *custom clusters*, in which a designer customizes either the detailed node design or the interconnection network, or both.
- In a commodity cluster, the nodes of a cluster are often blades or rack-mounted servers (including small-scale multiprocessor servers).
- Applications that focus on throughput and require almost no communication among threads, such as Web serving, multiprogramming, and some transaction-processing applications, can be accommodated inexpensively on a cluster.
- Commodity clusters are often assembled by users or computer center directors, rather than by vendors.

MIMD


prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
call funcD
x=y*z
sum=x*2
call sub1(i,j)
next instruct
P2
prev instruct
do 10 i=1,N
alpha=w**3
zeta=C(i)
10 continue
next instruct
Pn
time


IBM POWER5
HP/Compaq Alphaserver
Intel IA32
AMD Opteron
Cray XT3
IBM BG/L

Multicore

- Starting in the 1990s, the increasing capacity of a single chip allowed designers to place multiple processors on a single die.
- This approach, initially called *onchip multiprocessing* or *single-chip multiprocessing*, has come to be called *multicore*, a name arising from the use of multiple processor cores on a single die.
- In such a design, the multiple cores typically share some resources, such as a second- or third-level cache or memory and I/O buses.
- Recent processors, including the IBM Power5, the Sun T1, and the Intel Pentium D and Xeon-MP, are multicore and multithreaded.
- Just as using multiple copies of a microprocessor in a multiprocessor leverages a design investment through replication, a multicore achieves the same advantage relying more on replication than the alternative of building a wider superscalar.

Execution in MIMD

- With an MIMD, each processor is executing its own instruction stream.
- In many cases, each processor executes a different process.
- A *process* is a segment of code that may be run independently; the state of the process contains all the information necessary to execute that program on a processor.
- In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of other processes.

MIMD and Threads

- It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space.
- When multiple processes share code and data in this way, they are often called *threads*.
- Today, the term *thread* is often used in a casual way to refer to multiple sequences of execution that may run on different processors, even when they do not share an address space.
- For example, a multithreaded architecture actually allows the simultaneous execution of multiple processes, with potentially separate address spaces, as well as multiple threads that share the same address space.

MIMD and Threads

- 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 compiler.
- The threads may come from large-scale, independent processes scheduled and manipulated by the operating system. 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 and that the threads consist of hundreds to millions of instructions that may be executed in parallel.

Classes of MIMD

- Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy.
  - *centralized shared-memory architectures*
  - *multiprocessors with physically distributed memory*

Centralized shared-memory architectures

- These have at most a few dozen processor chips (and less than 100 cores).
- For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory.
- With large caches, a single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors.
- By using multiple point-to-point connections, or a switch, and adding additional memory banks, a centralized shared-memory design can be scaled to a few dozen processors.
- Although scaling beyond that is technically conceivable, sharing a centralized memory becomes less attractive as the number of processors sharing it increases.

Centralized shared-memory architectures

- Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are most often called *symmetric (shared-memory) multiprocessors* (SMPs).
- This style of architecture is sometimes called *uniform memory access* (UMA), arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks.
- This type of symmetric shared-memory architecture is currently by far the most popular organization.

Centralized shared-memory architectures

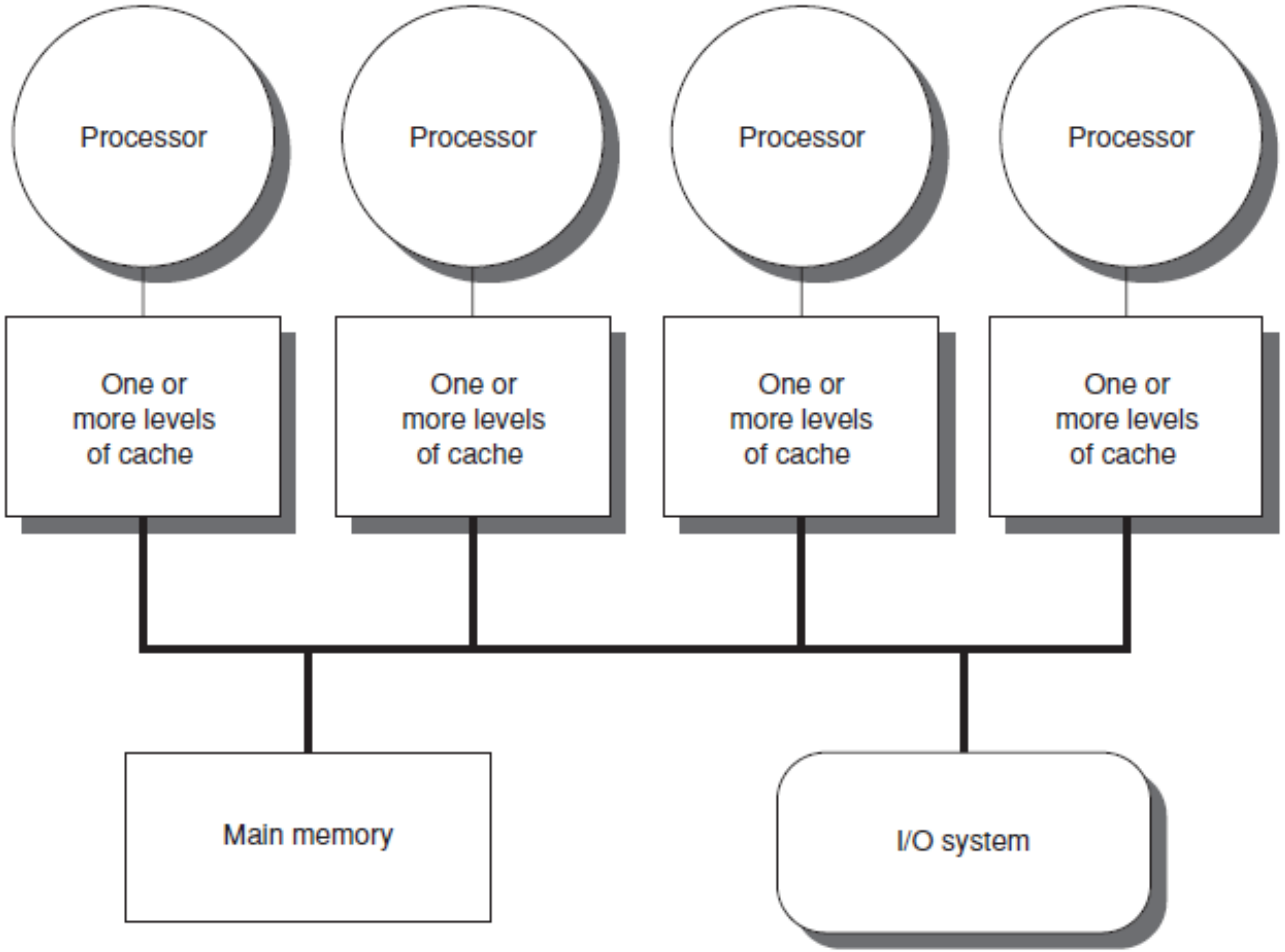


Figure 4.1 Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by one or more buses or a switch. The key architectural property is the uniform access time to all of memory from all the processors.

Multiprocessors with physically distributed memory

- 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.
- The larger number of processors also raises the need for a high-bandwidth interconnect.

Multiprocessors with physically distributed memory

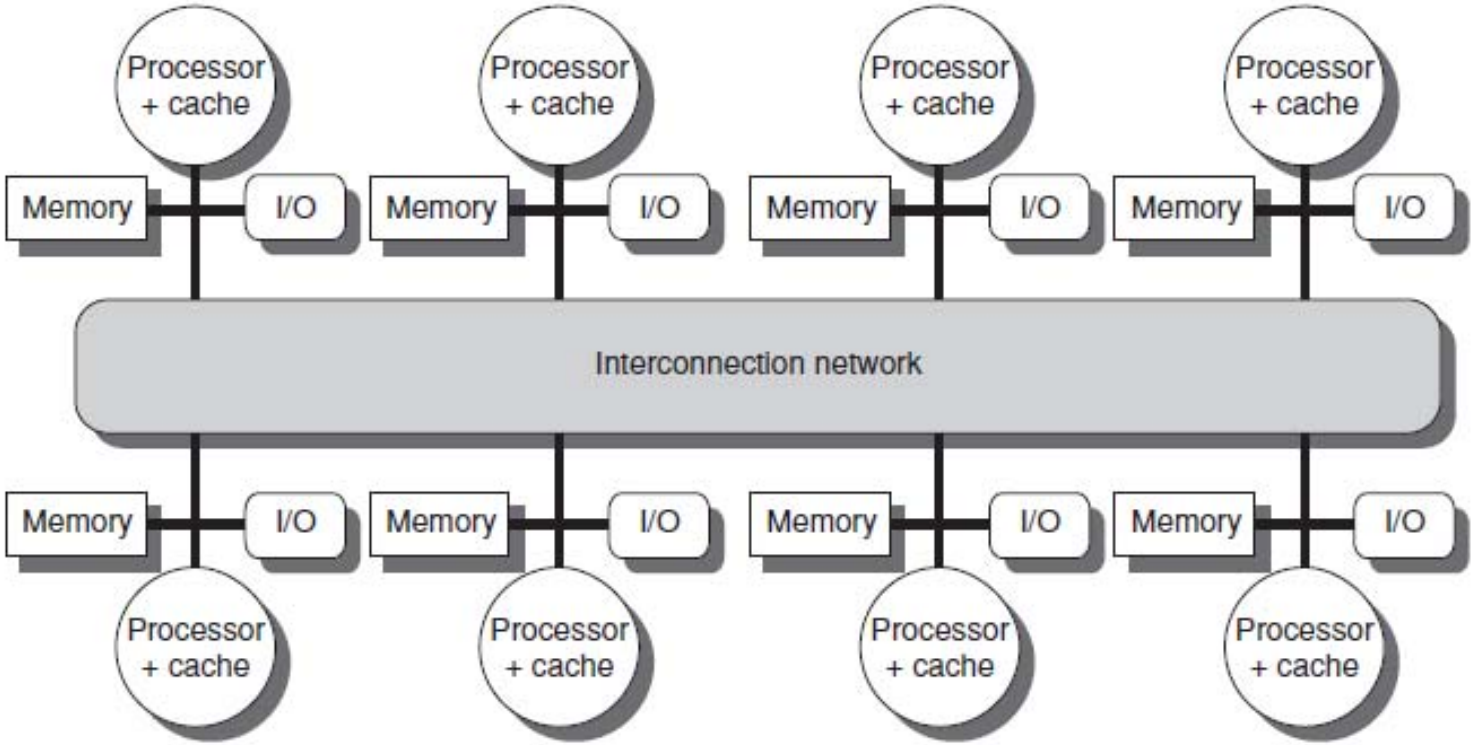


Figure 4.2 The basic architecture of a distributed-memory multiprocessor consists of individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes. Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

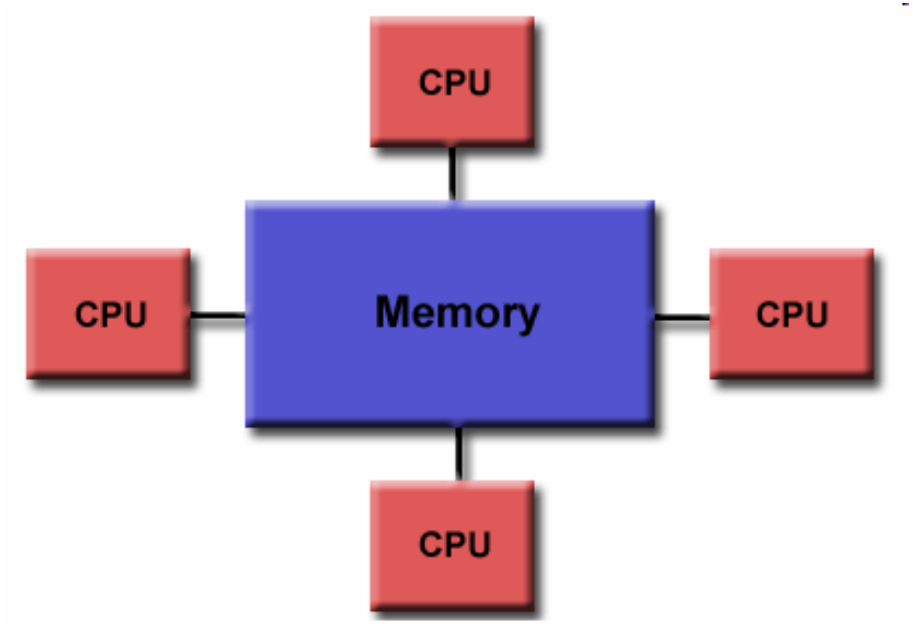
Advantages and disadvantages

- Distributing the memory among the nodes has two major benefits.
  - First, it is a cost-effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node.
  - Second, it reduces the latency for accesses to the local memory.
- These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency.
- The key disadvantages for a distributed-memory architecture are that communicating data between processors becomes somewhat more complex, and that it requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories.
- As we will see shortly, the use of distributed memory also leads to two different paradigms for interprocessor communication.

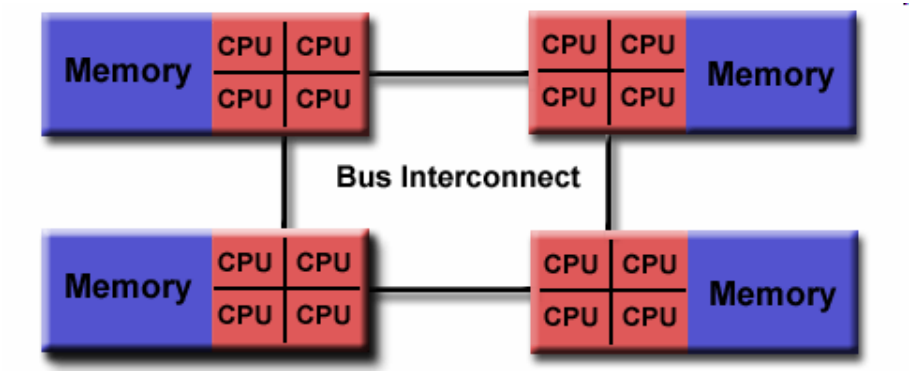
Models for Communication and Memory Architecture

- There are two alternative architectural approaches that differ in the method used for communicating data among processors.
- In the first method, communication occurs through a shared address space, as it does in a symmetric shared-memory architecture.
- The physically separate memories can be addressed as one logically 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. These multiprocessors are called *Distributed shared-memory* (DSM) architectures.
- The term *shared memory* refers to the fact that the *address space* is shared; that is, the same physical address on two processors refers to the same location in memory. Shared memory does *not* mean that there is a single, centralized memory.
- In contrast to the symmetric shared-memory multiprocessors, also known as UMAs (uniform memory access), the DSM multiprocessors are also called NUMAs (nonuniform memory access), since the access time depends on the location of a data word in memory.

UMA and NUMA



Shared Memory (UMA)



Shared Memory (NUMA)

Models for Communication and Memory Architecture

- Alternatively, the address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor.
- In such multiprocessors, the same physical address on two different processors refers to two different locations in two different memories.
- Each processor-memory module is essentially a separate computer.
- Initially, such computers were built with different processing nodes and specialized interconnection networks.
- Today, most designs of this type are actually clusters.

Message-passing multiprocessors

- With each of the organizations for the address space, there is an associated communication mechanism.
- For a multiprocessor with a shared address space, that address space can be used to communicate data implicitly via load and store operations — hence the name *shared memory* for such multiprocessors.
- For a multiprocessor with multiple address spaces, communication of data is done by explicitly passing messages among the processors.
- Therefore, these multiprocessors are often called *message-passing multiprocessors*.
- Clusters inherently use message passing.

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 the next example shows => next slide.

Example

- Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?
- Amdahl's Law is:

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

- 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:

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

Example

- Simplifying we have:

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

- Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of 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.

Latency of remote access

- The second major challenge in parallel processing involves the large latency of remote access in a parallel processor.
- In existing shared-memory multiprocessors, communication of data between processors may cost anywhere from 50 clock cycles (for multicores) to over 1000 clock cycles (for large-scale multiprocessors), depending on:
  - the communication mechanism
  - the type of interconnection network
  - the scale of the multiprocessor.
- The effect of long communication delays is clearly substantial. Let's consider a simple example => next slide

Example

- Suppose we have an application running on a 32-processor multiprocessor, which has a 200 ns time to handle reference to a remote memory.
- For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which is slightly optimistic.
- Processors are stalled on a remote request, and the processor clock rate is 2 GHz.
- If the base CPI (assuming that all references hit in the cache) is 0.5, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

Example

- It is simpler to first calculate the CPI. The effective CPI for the multiprocessor with 0.2% remote references is:

$$\begin{aligned} \text{CPI} &= \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost} \\ &= 0.5 + 0.2\% \times \text{Remote request cost} \end{aligned}$$

- The remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{200 \text{ ns}}{0.5 \text{ ns}} = 400 \text{ cycles}$$

- We can compute the CPI: CPI = 0.5+ 0.8 = 1.3

- The multiprocessor with all local references is 1.3/0.5 = 2.6 times faster.
- In practice, the performance analysis is much more complex, since some fraction of the noncommunication references will miss in the local hierarchy and the remote access time does not have a single constant value.

Attacking problems

- The two problems — insufficient parallelism and long-latency remote communication — are the two biggest performance challenges in using multiprocessors.
- The problem of inadequate application parallelism must be attacked primarily in software with new algorithms that can have better parallel performance.
- Reducing the impact of long remote latency can be attacked both by the architecture and by the programmer.
- For example, we can reduce the frequency of remote accesses with either hardware mechanisms, such as caching shared data, or software mechanisms, such as restructuring the data to make more accesses local.
- We can try to tolerate the latency by using multithreading or by using prefetching.

Attacking problems

- Here we will focuses on techniques for reducing the impact of long remote communication latency:
  - how caching can be used to reduce remote access frequency, while maintaining a coherent view of memory.
  - synchronization, which, because it inherently involves interprocessor communication and also can limit parallelism, is a major potential bottleneck.
  - latency-hiding techniques and memory consistency models for shared memory.

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

Symmetric Shared-Memory Architectures

- The use of large, multilevel caches can substantially reduce the memory bandwidth demands of a processor.
- If the main memory bandwidth demands of a single processor are reduced, multiple processors may be able to share the same memory.
- Starting in the 1980s, this observation, combined with the emerging dominance of the microprocessor, motivated many designers to create small-scale multiprocessors where several processors shared a single physical memory connected by a shared bus.
- Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors were extremely cost-effective, provided that a sufficient amount of memory bandwidth existed.

Private and shared data

- Symmetric shared-memory machines usually support the caching of both *shared and private data*.
- *Private data* are used by a single processor, while *shared data* are used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data.
- When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required.
- Since no other processor uses the data, the program behavior is identical to that in a uniprocessor.

  When shared data are cached, the shared value may be replicated in multiple caches.

Multiprocessor Cache Coherence

- Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values.
- Two different processors can have two different values for the same location. This difficulty is generally referred to as the *cache coherence problem.*

Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

Figure 4.3 The cache coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 1!

Cache coherence and consistency

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item.

- This definition, although intuitively appealing, is vague and simplistic; the reality is much more complex.
- This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs.
- The first aspect, called *coherence*, defines what values can be returned by a read.
- The second aspect, called *consistency*, determines when a written value will be returned by a read.

Cache coherence

A memory system is coherent if

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
3. Writes to the same location are *serialized;* that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

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.
- 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.

Cache migration and replication

- 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, small-scale multiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches.

Cache coherence protocols.

- 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, which use different techniques to track the sharing status, in use:

1. *Directory based* — The sharing status of a block of physical memory is kept in just one location, called the *directory*.
2. *Snooping* — Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept.
   - The caches are all accessible via some broadcast medium (a bus or switch), 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.

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.
- In the simple case of a bus-based multiprocessor, *the bus must support both the coherence traffic as well as normal memory traffic arising from the caches*.
- Likewise, if there is a single memory unit, it must accommodate all processor requests.
- As processors have increased in speed in the last few years, *the number of processors that can be supported on a single bus or by using a single physical memory unit has fallen*.

Increasing memory bandwidth

- 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
  - interconnection networks, such as crossbars
  - small point-to-point networks.
- In such designs, the memory system can be configured into multiple physical banks, so as to boost the effective memory bandwidth while retaining uniform access time to memory.

MP with UMA

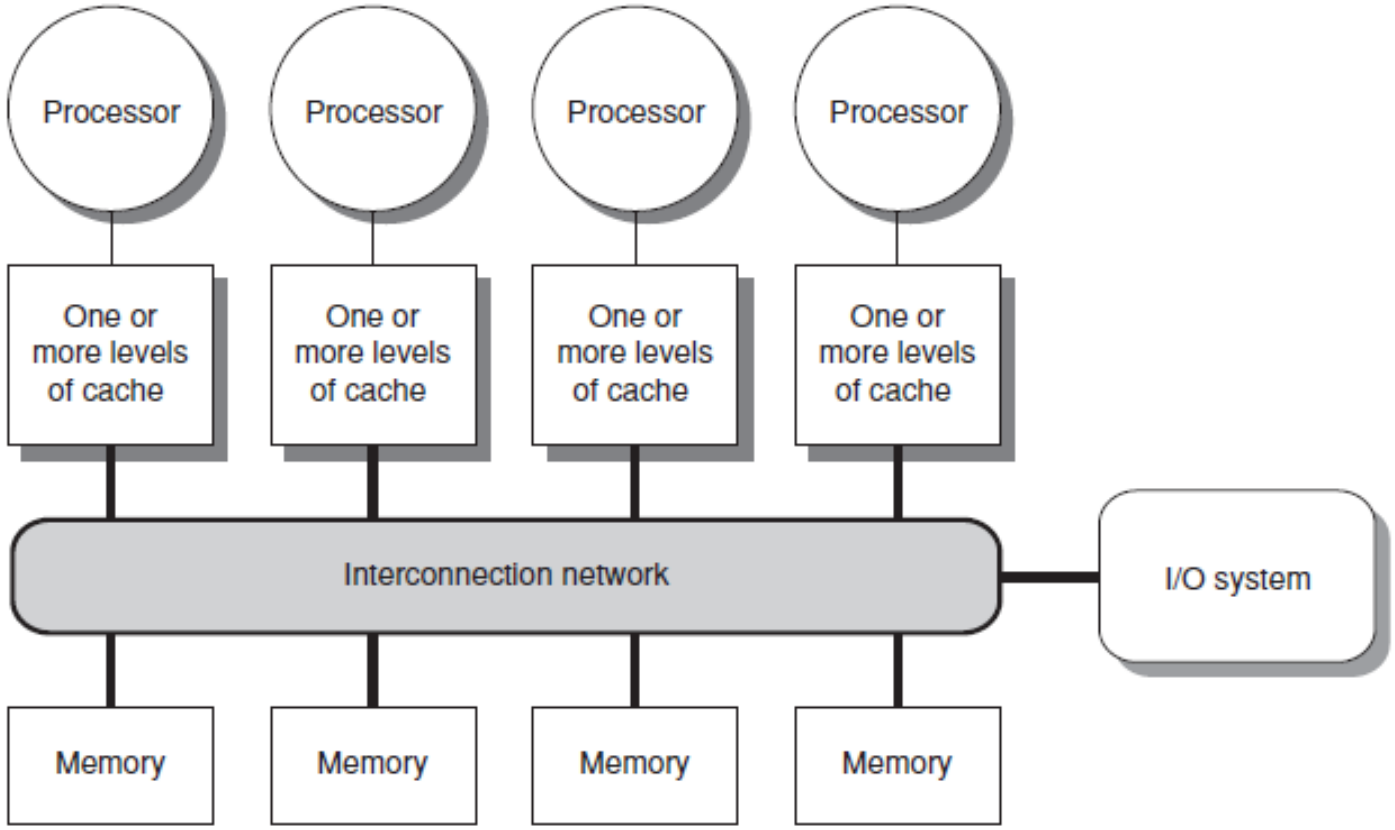


Figure 4.8 A multiprocessor with uniform memory access using an interconnection network rather than a bus.

Example: AMD Opteron

- The AMD Opteron represents another intermediate point in the spectrum between a snoopy and a directory protocol.
- Memory is directly connected to each dual-core processor chip, and up to four processor chips, eight cores in total, can be connected.
- 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 find potentially shared copies, like a snoopy protocol, but uses the acknowledgments to order operations, like a directory protocol.
- Interestingly, the remote memory latency and local memory latency are not dramatically different, allowing the operating system to treat an Opteron multiprocessor as having uniform memory access.

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

Performance of Symmetric Shared-Memory Multiprocessors

- In a multiprocessor using a snoopy coherence protocol, several different phenomena combine to determine performance.
- In particular, the overall cache performance is a combination of the behavior of uniprocessor cache miss traffic and the traffic caused by communication.
- Changing the processor count, cache size, and block size can affect these two components of the miss rate in different ways, leading to overall system behavior that is a combination of the two effects.

A Commercial Workload

- We examine here the memory system behavior of a four-processor shared-memory multiprocessor.
- The results were collected either on an Alpha-Server 4100 or using a configurable simulator modeled after the Alpha-Server 4100.
- Each processor in the Alpha-Server 4100 is an Alpha 21164, which issues up to four instructions per clock and runs at 300 MHz.
- Although the clock rate of the Alpha processor in this system is considerably slower than processors in recent systems, the basic structure of the system, consisting of a four-issue processor and a three-level cache hierarchy, is comparable to many recent systems.

A Commercial Workload

- Each processor has a three-level cache hierarchy:
  - L1 consists of a pair of 8 KB direct-mapped on-chip caches, one for instruction and one for data. The block size is 32 bytes, and the data cache is write through to L2, using a write buffer.
  - L2 is a 96 KB on-chip unified three-way set associative cache with a 32-byte block size, using write back.
  - L3 is an off-chip, combined, direct-mapped 2 MB cache with 64-byte blocks also using write back.

- The latency for an access to L2 is 7 cycles, to L3 it is 21 cycles, and to main memory it is 80 clock cycles.
- Cache-to-cache transfers, which occur on a miss to an exclusive block held in another cache, require 125 clock cycles.

A Commercial Workload

The workload used for this study consists of three applications:

1. An online transaction-processing workload (OLTP) modeled after TPC-B (which has similar memory behavior to its newer cousin TPC-C) 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.

A Commercial Workload

2. A decision support system (DSS) workload based on TPC-D and also using Oracle 7.3.2 as the underlying database.
   - The workload includes only 6 of the 17 read queries in TPC-D, 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.

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.

Distribution of execution times

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

Figure 4.9 The distribution of execution time in the commercial workloads. The OLTP benchmark has the largest fraction of both OS time and CPU 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 et al. [1998] on a four-processor AlphaServer 4100.

Performance Measurements of the Commercial Workload

- We start by looking at the overall CPU execution for these benchmarks on the four-processor system.
  - These benchmarks include substantial I/O time, which is ignored in the CPU 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.

Execution time breakdown

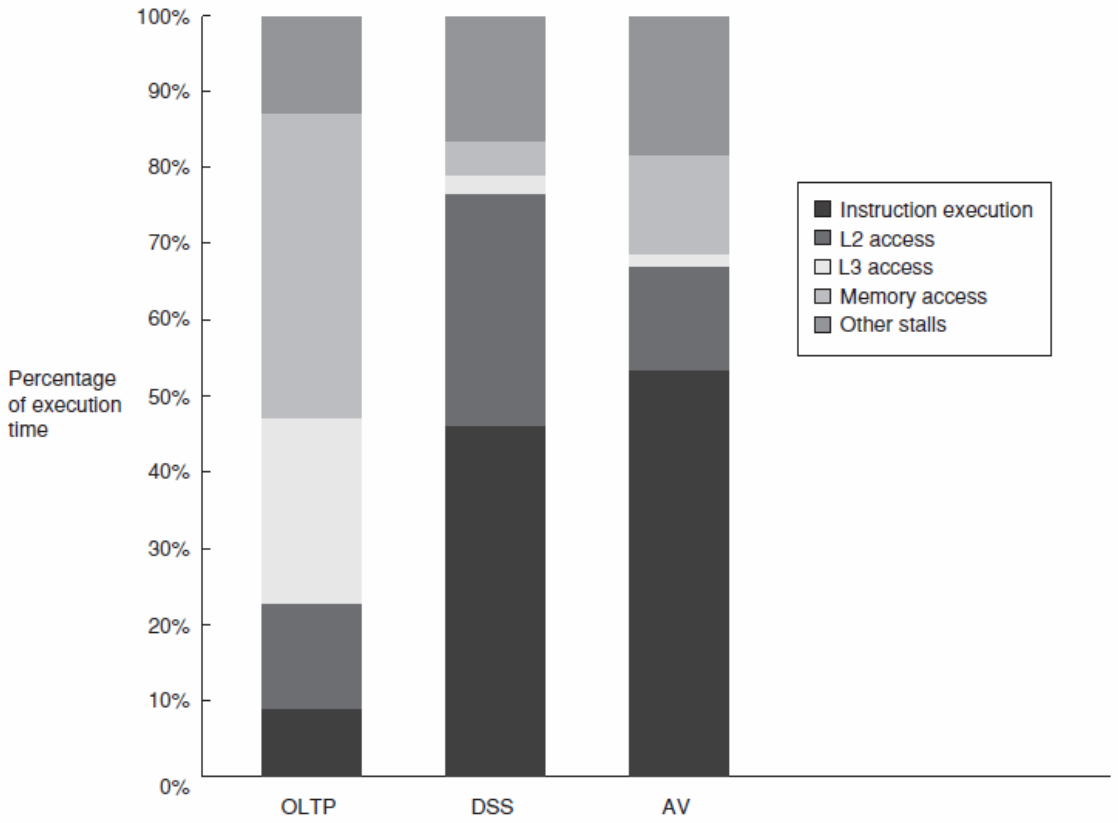


Figure 4.10 The execution time breakdown for the three programs (OLTP, DSS, and AltaVista) in the commercial workload. The DSS numbers are the average across six different queries. The CPI varies widely from a low of 1.3 for AltaVista, to 1.61 for the DSS queries, to 7.0 for OLTP. (Individually, the DSS queries show a CPI range of 1.3 to 1.9.) Other stalls includes resource stalls (implemented with replay traps on the 21164), branch mispredict, memory barrier, and TLB misses. For these benchmarks, resource-based pipeline stalls are the dominant factor. These data combine the behavior of user and kernel accesses. Only OLTP has a significant fraction of kernel accesses, and the kernel accesses tend to be better behaved than the user accesses! All the measurements shown in this section were collected by Barroso, Gharachorloo, and Bugnion [1998].

Performance of OLTP with L3 Cache growing

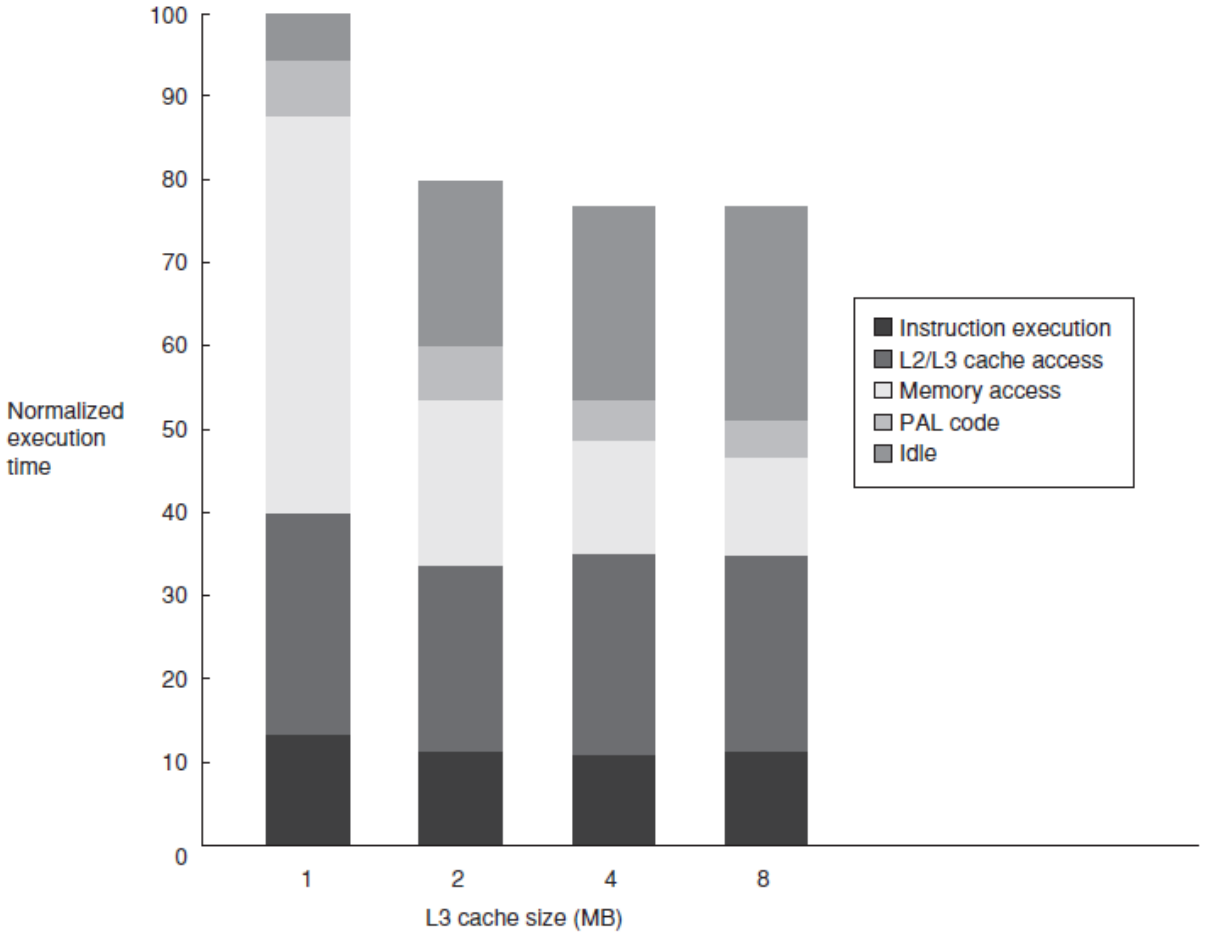


Figure 4.11 The relative performance of the OLTP workload as the size of the L3 cache, which is set as two-way set associative, grows from 1 MB to 8 MB. The idle time also grows as cache size is increased, reducing some of the performance gains. This growth occurs because, with fewer memory system stalls, more server processes are needed to cover the I/O latency. The workload could be retuned to increase the computation/communication balance, holding the idle time in check

Recollect some definitions

- Coherence misses occur when blocks of data are shared among multiple caches.
- *True sharing* cache misses occur whenever two processors access the same data word.
  - True sharing requires the processors involved to explicitly synchronize with each other to ensure program correctness.
- *False sharing* misses occur when independent data words accessed by different processors happen to be placed in the same cache block, and at least one of the accesses is a write.
  - Even if a processor re-uses a data item, the item may no longer be in the cache due to an intervening access by another processor to another word in the same cache line.

Recollect some definitions

- *Compulsory (Cold) misses* occur on the first reference to a memory block by a processor.

- *Capacity misses* occur when all the blocks that are referenced by a processor do not fit in the cache, so some are replaced and later accessed again.

Increasing cache size

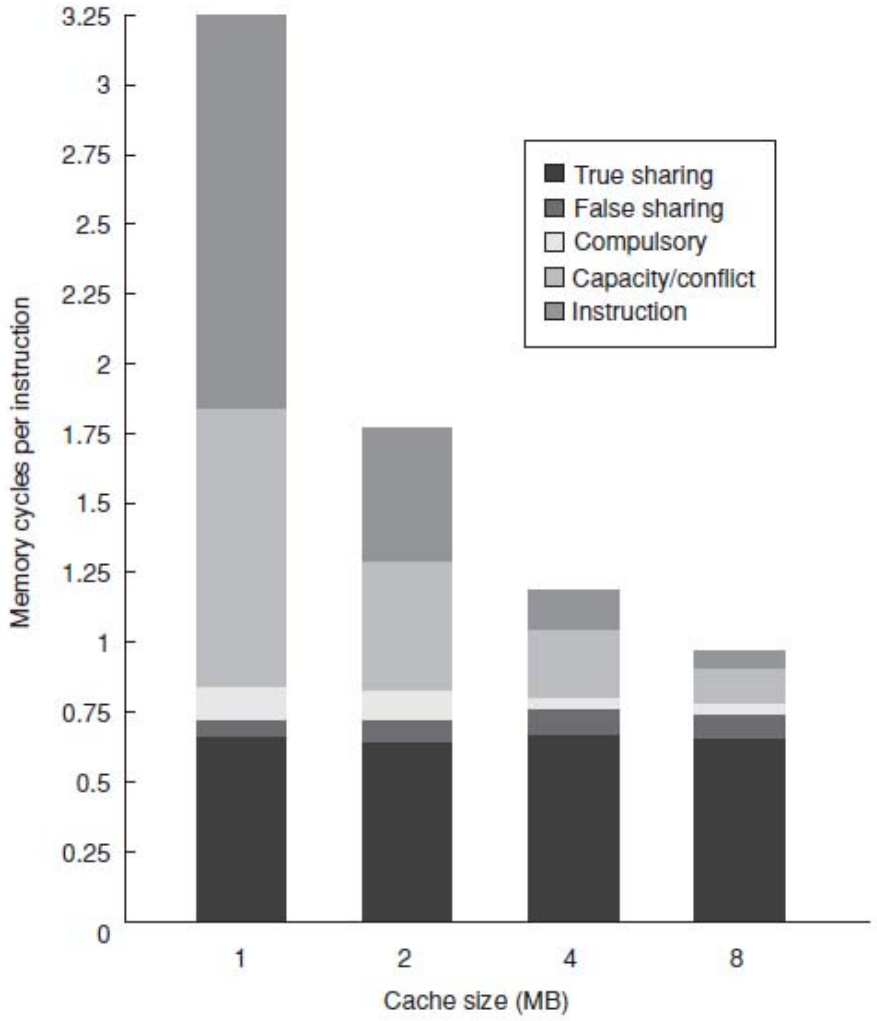


Figure 4.12 The contributing causes of memory access cycles shift as the cache size is increased. The L3 cache is simulated as two-way set associative.

Increasing processor count

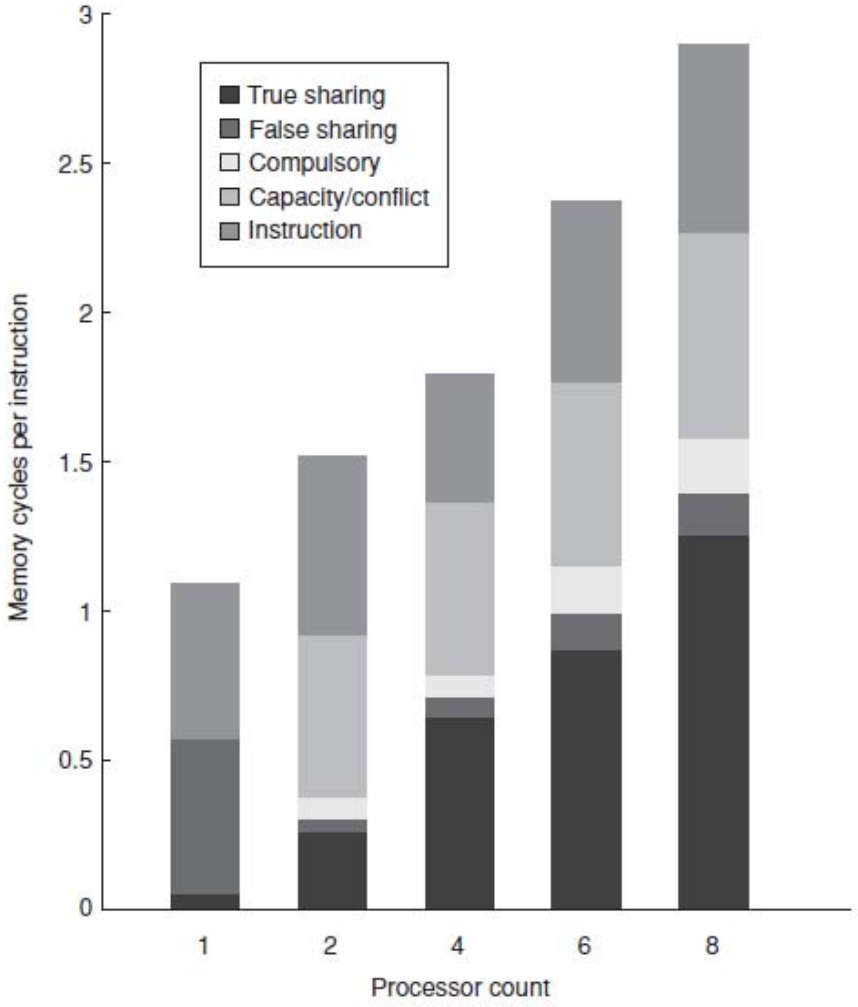


Figure 4.13 The contribution to memory access cycles increases as processor count increases primarily due to increased true sharing. The compulsory misses slightly increase since each processor must now handle more compulsory misses.

A Multiprogramming and OS Workload

- Our next study is a multiprogrammed workload consisting of both user activity and OS activity.
- The workload used is two independent copies of the compile phases of the Andrew benchmark, a benchmark that emulates a software development environment.
- The compile phase consists of a parallel make using eight processors.
- The workload runs for 5.24 seconds on eight processors, creating 203 processes and performing 787 disk requests on three different file systems.
- The workload is run with 128 MB of memory, and no paging activity takes place.

A Multiprogramming and OS Workload

- The workload has three distinct phases:
  - compiling the benchmarks, which involves substantial compute activity;
  - installing the object files in a library
  - removing the object files.
- The last phase is completely dominated by I/O and only two processes are active (one for each of the runs).
- In the middle phase, I/O also plays a major role and the processor is largely idle.
- The overall workload is much more system and I/O intensive than the highly tuned commercial workload.

A Multiprogramming and OS Workload: Components

- For the workload measurements, we assume the following memory and I/O systems:
  - *Level 1 instruction cache*—32 KB, two-way set associative with a 64-byte block, 1 clock cycle hit time.
  - *Level 1 data cache*—32 KB, two-way set associative with a 32-byte block, 1 clock cycle hit time. We vary the L1 data cache to examine its effect on cache behavior.
  - *Level 2 cache*—1 MB unified, two-way set associative with a 128-byte block, hit time 10 clock cycles.
  - *Main memory*—Single memory on a bus with an access time of 100 clock cycles.
  - *Disk system*—Fixed-access latency of 3 *ms* (less than normal to reduce idle time)

Breaking of execution time

	User execution	Kernel execution	Synchronization wait	CPU idle (waiting for I/O)
% instructions executed	27	3	1	69
% execution time	27	7	2	64

Figure 4.15 The distribution of execution time in the multiprogrammed parallel make workload. The high fraction of idle time is due to disk latency when only one of the eight processors is active. These data and the subsequent measurements for this workload were collected with the SimOS system [Rosenblum et al. 1995]. The actual runs and data collection were done by M. Rosenblum, S. Herrod, and E. Bugnion of Stanford University.

Execution time is broken into four components:

1. *Idle*—Execution in the kernel mode idle loop
2. *User*—Execution in user code
3. *Synchronization*—Execution or waiting for synchronization variables
4. *Kernel*—Execution in the OS that is neither idle nor in synchronization access

Cache miss

- This multiprogramming workload has a significant instruction cache performance loss, at least for the OS.
- The instruction cache miss rate in the OS for a 64-byte block size, two-way set-associative cache varies from 1.7% for a 32 KB cache to 0.2% for a 256 KB cache.
- User-level instruction cache misses are roughly one-sixth of the OS rate, across the variety of cache sizes.
- This partially accounts for the fact that although the user code executes nine times as many instructions as the kernel, those instructions take only about four times as long as the smaller number of instructions executed by the kernel.

Performance of the Multiprogramming and OS Workload

- We examine here the cache performance of the multiprogrammed workload as the cache size and block size are changed.
- Because of differences between the behavior of the kernel and that of the user processes, we keep these two components separate.
- Remember, though, that the user processes execute more than eight times as many instructions, so that the overall miss rate is determined primarily by the miss rate in user code, which, as we will see, is often one-fifth of the kernel miss rate.

Data miss rates for the user and kernel components

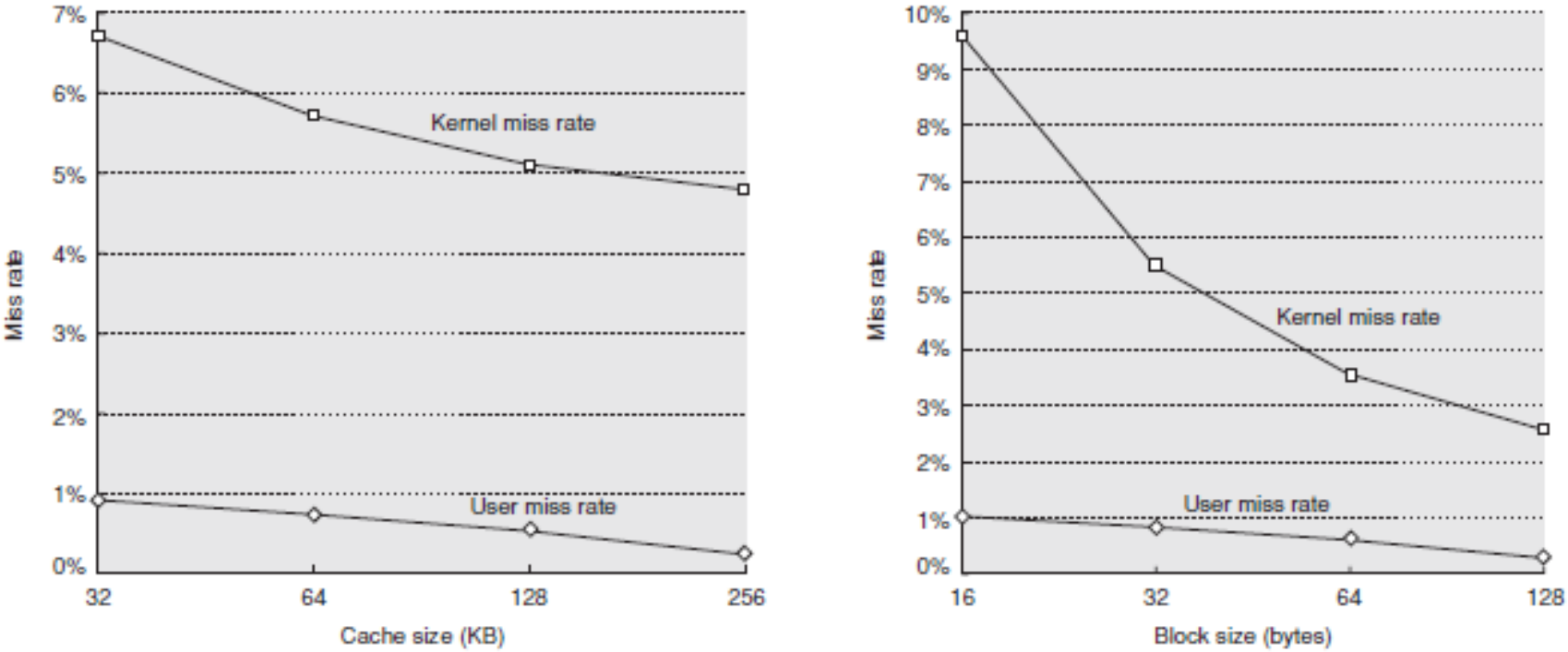


Figure 4.16 The data miss rates for the user and kernel components behave differently for increases in the L1 data cache size (on the left) versus increases in the L1 data cache block size (on the right). Increasing the L1 data cache from 32 KB to 256 KB (with a 32-byte block) causes the user miss rate to decrease proportionately more than the kernel miss rate: the user-level miss rate drops by almost a factor of 3, while the kernel-level miss rate drops only by a factor of 1.3. The miss rate for both user and kernel components drops steadily as the L1 block size is increased (while keeping the L1 cache at 32 KB). In contrast to the effects of increasing the cache size, increasing the block size improves the kernel miss rate more significantly (just under a factor of 4 for the kernel references when going from 16-byte to 128-byte blocks versus just under a factor of 3 for the user references).

Increasing cache size

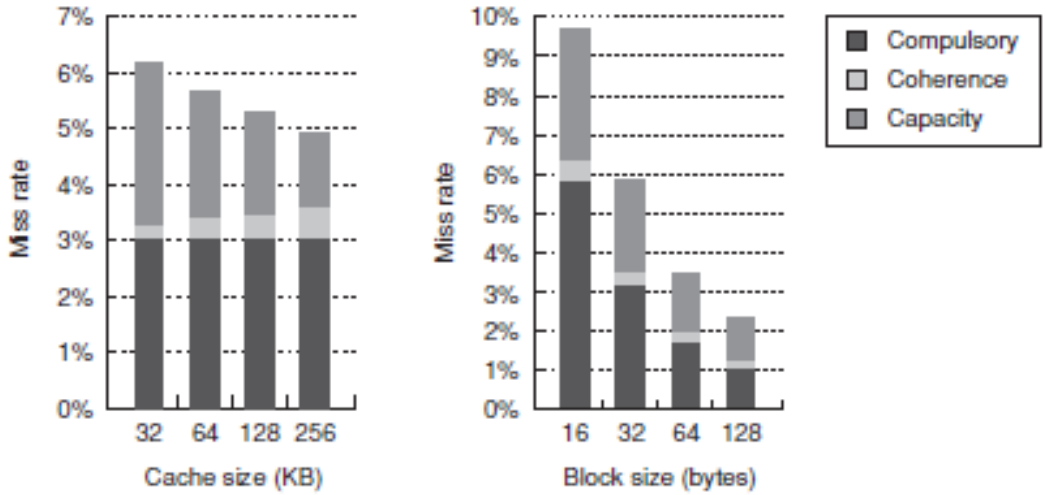


Figure 4.17 The components of the kernel data miss rate change as the L1 data cache size is increased from 32 KB to 256 KB, when the multiprogramming workload is run on eight processors. The compulsory miss rate component stays constant, since it is unaffected by cache size. The capacity component drops by more than a factor of 2, while the coherence component nearly doubles. The increase in coherence misses occurs because the probability of a miss being caused by an invalidation increases with cache size, since fewer entries are bumped due to capacity. As we would expect, the increasing block size of the L1 data cache substantially reduces the compulsory miss rate in the kernel references. It also has a significant impact on the capacity miss rate, decreasing it by a factor of 2.4 over the range of block sizes. The increased block size has a small reduction in coherence traffic, which appears to stabilize at 64 bytes, with no change in the coherence miss rate in going to 128-byte lines. Because there are not significant reductions in the coherence miss rate as the block size increases, the fraction of the miss rate due to coherence grows from about 7% to about 15%.

Bytes needed for data reference

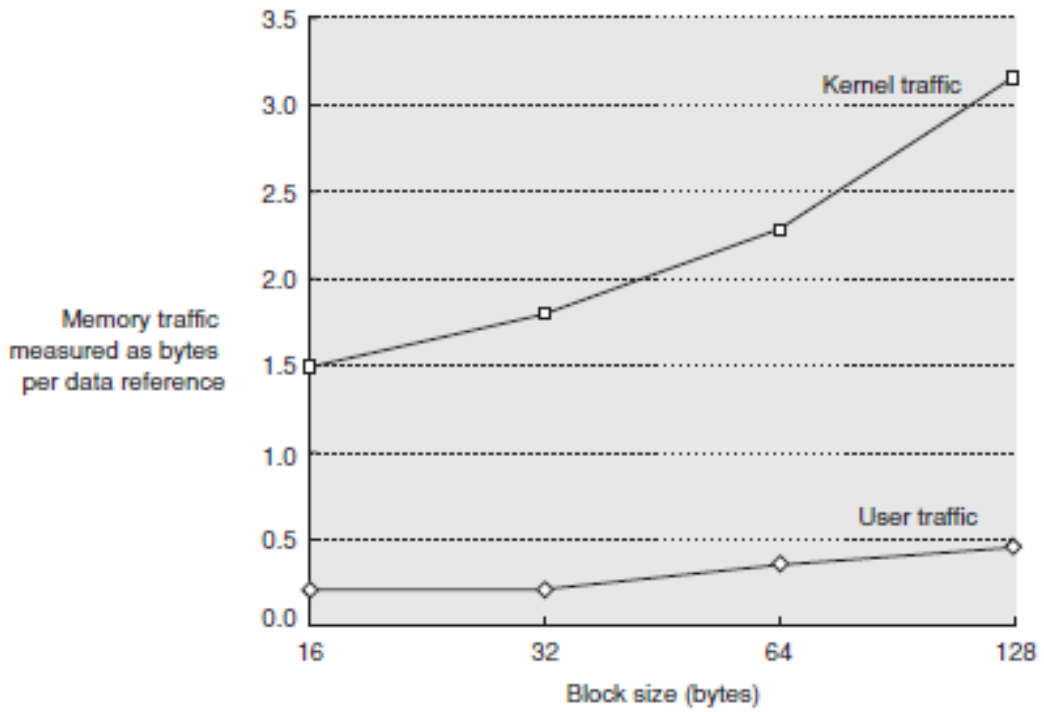


Figure 4.18 The number of bytes needed per data reference grows as block size is increased for both the kernel and user components. It is interesting to compare this chart against the data on scientific programs shown in Appendix H.

Conclusions regarding OS workload

- For the multiprogrammed workload, the OS is a much more demanding user of the memory system.
- If more OS or OS-like activity is included in the workload, and the behavior is similar to what was measured for this workload, it will become very difficult to build a sufficiently capable memory system.
- One possible route to improving performance is to make the OS more cache aware, through either better programming environments or through programmer assistance.

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

Directory coherence protocol

- A directory keeps the state of every block that may be cached.
- Information in the directory includes which caches have copies of the block, whether it is dirty, and so on.
- A directory protocol also can be used to reduce the bandwidth demands in a centralized shared-memory machine, as the Sun T1 design does.
- We explain a directory protocol as if it were implemented with a distributed memory, but the same design also applies to a centralized memory organized into banks.

Directory Implementations

- The simplest directory implementations associate an entry in the directory with each memory block.
- In such implementations, the amount of information is proportional to the product of the number of memory blocks (where each block is the same size as the level 2 or level 3 cache block) and the number of processors.
- This overhead is not a problem for multiprocessors with less than about 200 processors because the directory overhead with a reasonable block size will be tolerable.
- For larger multiprocessors, we need methods to allow the directory structure to be efficiently scaled.
  - The methods that have been used either try to keep information for fewer blocks (e.g., only those in caches rather than all memory blocks) or try to keep fewer bits per entry by using individual bits to stand for a small collection of processors.

Distributed directory

- To prevent the directory from becoming the bottleneck, the directory is distributed along with the memory (or with the interleaved memory banks in an SMP), so that different directory accesses can go to different directories, just as different memory requests go to different memories.

Distributed Directory

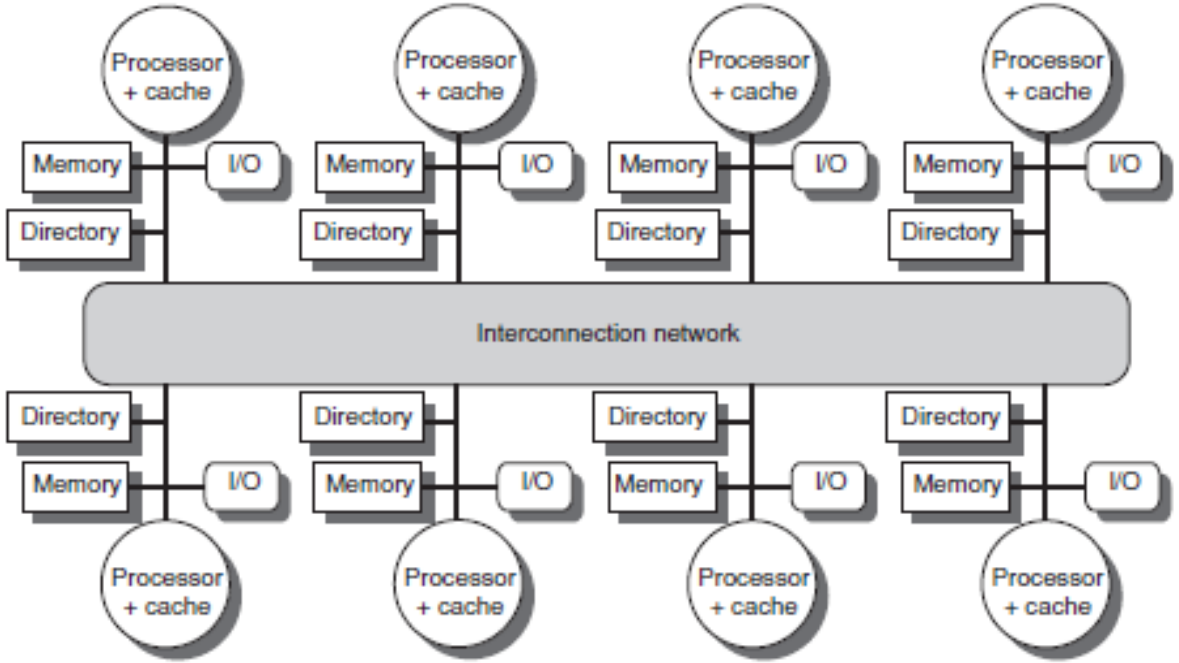


Figure 4.19 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The directory may communicate with the processor and memory over a common bus, as shown, or it may have a separate port to memory, or it may be part of a central node controller through which all intranode and internode communications pass.

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

Synchronization mechanisms

- Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions.
- For smaller multiprocessors or low-contention situations, the key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value.
- Software synchronization mechanisms are then constructed using this capability.
- Lock and unlock operation can be used straightforwardly to create mutual exclusion, as well as to implement more complex synchronization mechanisms.

Basic Hardware Primitives

- The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location.
- Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases.
- These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers.
- In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky.

Basic Hardware Primitives: Atomic Exchange

- One typical operation for building synchronization operations is the *atomic exchange*, which interchanges a value in a register for a value in memory.
- To see how to use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and 1 is used to indicate that the lock is unavailable.
- A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.
- The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise.
- In the latter case, the value is also changed to 1, preventing any competing exchange from also retrieving a 0.

Basic Hardware Primitives: test-and-set

- There are a number of other atomic primitives that can be used to implement synchronization.
- They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically.
- One operation, present in many older multiprocessors, is *test-and-set*, which tests a value and sets it if the value passes the test.
- Another atomic synchronization primitive is *fetch-and-increment:* It returns the value of a memory location and atomically increments it.

Load Linked and Store Conditional

- An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instructions were atomic.
- Thus, when an instruction pair is effectively atomic, no other processor can change the value between the instruction pair.
- The pair of instructions includes a special load called a *load linked* or *load locked* and a special store called a *store conditional*.

Spin Locks

- Once we have an atomic operation, we can use the coherence mechanisms of a multiprocessor to implement *spin locks* — locks that a processor continuously tries to acquire, spinning around a loop until it succeeds.
- Spin locks are used when programmers expect the lock to be held for a very short amount of time and when they want the process of locking to be low latency when the lock is available.
- Because spin locks consume the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

Models of Memory Consistency:

- Cache coherence ensures that multiple processors see a consistent view of memory.
- It does not answer the question of *how consistent the view of memory must be*.
- By “how consistent” we mean, *when must a processor see a value that has been updated by another processor?*
- Since processors communicate through shared variables (used both for data values and for synchronization), the question boils down to this: *In what order must a processor observe the data writes of another processor?*
- Since the only way to “observe the writes of another processor” is through reads, the question becomes,

  What properties must be enforced among reads and writes to different locations by different processors?

Sequential consistency

- The most straightforward model for memory consistency is called *sequential consistency*.
- Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved.

Relaxed Consistency Models

- The key idea in relaxed consistency models is to allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent.

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

Crosscutting Issues

- Because multiprocessors redefine many system characteristics (e.g., performance assessment, memory latency, and the importance of scalability), they introduce interesting design problems that cut across the spectrum, affecting both hardware and software.

- Here we discuss about issues of memory consistency.

Compiler Optimization and the Consistency Model

- Another reason for defining a model for memory consistency is to specify the range of legal compiler optimizations that can be performed on shared data.
- In explicitly parallel programs, unless the synchronization points are clearly defined and the programs are synchronized, the compiler could not interchange a read and a write of two different shared data items because such transformations might affect the semantics of the program.
- This prevents even relatively simple optimizations, such as register allocation of shared data, because such a process usually interchanges reads and writes.
- In implicitly parallelized programs — for example, those written in High Performance FORTRAN (HPF)— programs must be synchronized and the synchronization points are known, so this issue does not arise.

Using Speculation to Hide Latency in Strict Consistency Models

- Speculation can be used to hide memory latency.
- It can also be used to hide latency arising from a strict consistency model, giving much of the benefit of a relaxed memory model.
- The key idea is for the processor to use dynamic scheduling to reorder memory references, letting them possibly execute out of order.
- Executing the memory references out of order may generate violations of sequential consistency, which might affect the execution of the program.
  - This possibility is avoided by using the delayed commit feature of a speculative processor.

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

Sun T1

- T1 is a multicore multiprocessor introduced by Sun in 2005 as a server processor.
- What makes T1 especially interesting is that it is almost totally focused on exploiting thread-level parallelism (TLP) rather than instruction-level parallelism (ILP).
- Indeed, it is the only single-issue desktop or server microprocessor introduced in more than five years.
- Instead of focusing on ILP, T1 puts all its attention on TLP, using both multiple cores and multithreading to produce throughput.

Sun T1

- Each T1 processor contains eight processor cores, each supporting four threads.
- Each processor core consists of a simple six-stage, single-issue pipeline (a standard five-stage RISC pipeline like that of Appendix A, with one stage added for thread switching).
- T1 uses fine-grained multithreading, switching to a new thread on each clock cycle, and threads that are idle because they are waiting due to a pipeline delay or cache miss, are bypassed in the scheduling.

  The processor is idle only when all four threads are idle or stalled.
- Both loads and branches incur a 3-cycle delay that can only be hidden by other threads.
- A single set of floating-point functional units is shared by all eight cores, as floating-point performance was not a focus for T1.

Sun T1 Architecture

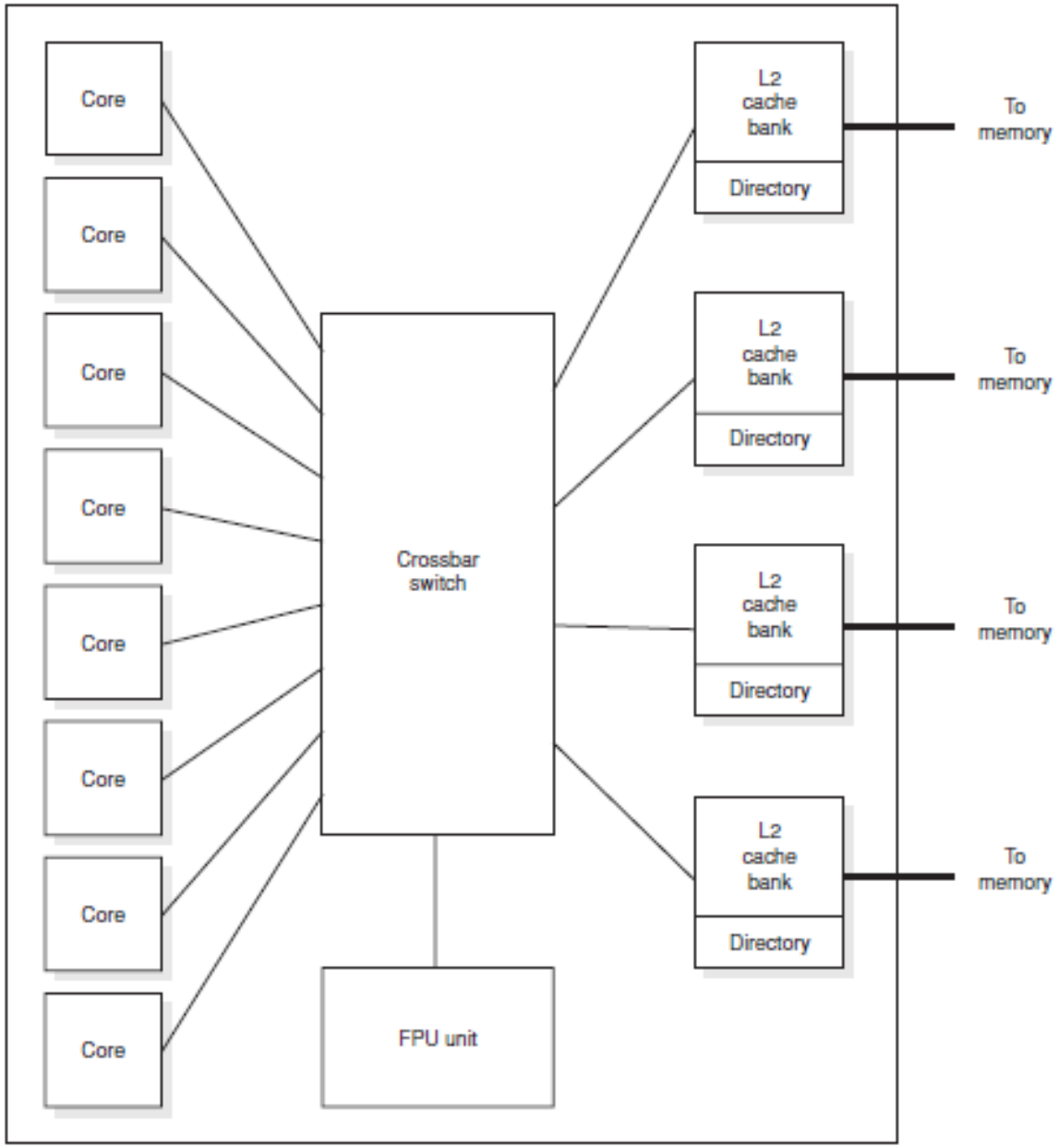


Figure 4.24 The T1 processor. Each core supports four threads and has its own level 1 caches (16 KB for instructions and 8 KB for data). The level 2 caches total 3 MB and are effectively 12-way associative. The caches are interleaved by 64-byte cache lines.

Coherency is enforced among the L1 caches by a directory associated with each L2 cache block.

Summary of T1

Characteristic	Sun T1
Multiprocessor and multithreading support	Eight cores per chip; four threads per core. Fine-grained thread scheduling. One shared floating-point unit for eight cores. Supports only on-chip multiprocessing.
Pipeline structure	Simple, in-order, six-deep pipeline with 3-cycle delays for loads and branches.
L1 caches	16 KB instructions; 8 KB data. 64-byte block size. Miss to L2 is 23 cycles, assuming no contention.
L2 caches	Four separate L2 caches, each 750 KB and associated with a memory bank. 64-byte block size. Miss to main memory is 110 clock cycles assuming no contention.
Initial implementation	90 nm process; maximum clock rate of 1.2 GHz; power 79 W; 300M transistors, 379 mm^2 die.

Figure 4.25 A summary of the T1 processor.

T1 Performance

- We look at the performance of T1 using three server-oriented benchmarks: TPCC, SPECJBB (the SPEC Java Business Benchmark), and SPECWeb99.
- The SPECWeb99 benchmark is run on a four-core version of T1 because it cannot scale to use the full 32 threads of an eight-core processor; the other two benchmarks are run with eight cores and 4 threads each for a total of 32 threads.

Miss Rate: Varying cache size and block size

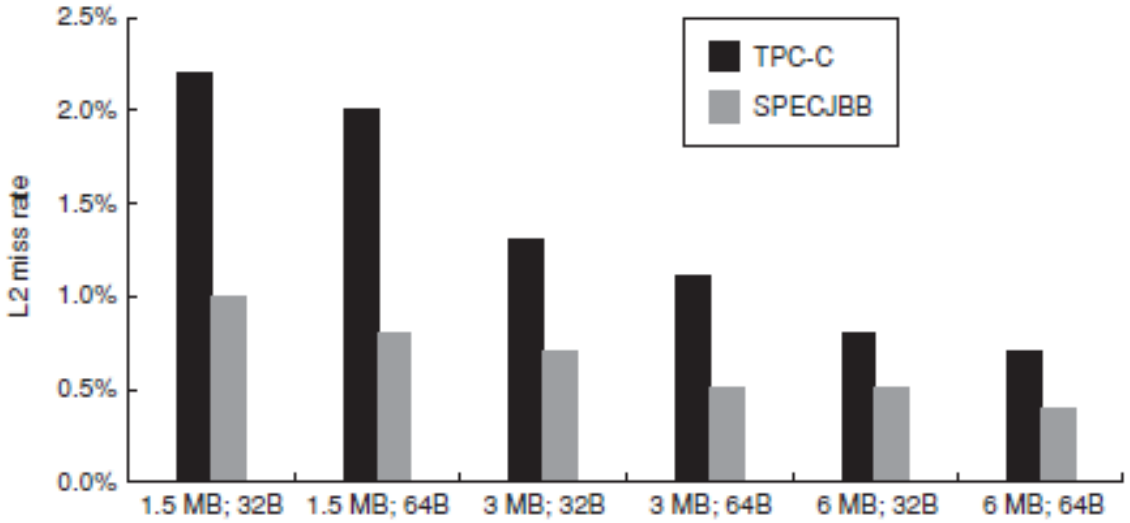


Figure 4.27 Change in the L2 miss rate with variation in cache size and block size. Both TPC-C and SPECJBB are run with all eight cores and four threads per core. Recall that T1 has a 3 MB L2 with 64-byte lines.

Miss Latency: Varying cache size and block size

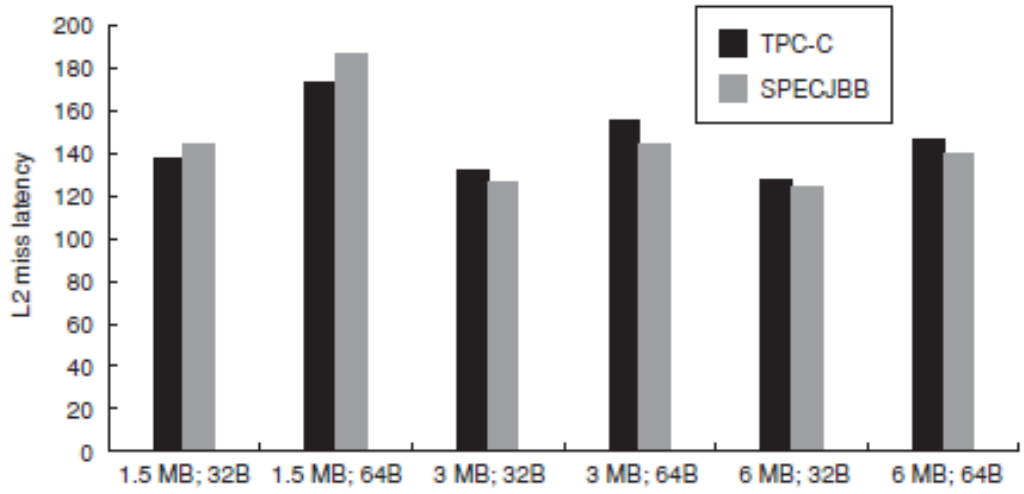


Figure 4.28 The change in the miss latency of the L2 cache as the cache size and block size are changed. Although TPC-C has a significantly higher miss rate, its miss penalty is only slightly higher. This is because SPECJBB has a much higher dirty miss rate, requiring L2 cache lines to be written back with high frequency. Recall that T1 has a 3 MB L2 with 64-byte lines.

As we can see, for either a 3 MB or 6 MB cache, the larger block size results in a smaller L2 cache miss time.

How can this be if the miss rate changes much less than a factor of 2?

Reply: Modern DRAMs provide a block of data for only slightly more time than needed to provide a single word; thus, the miss penalty for the 32-byte block is only slightly less than the 64-byte block.

T1 Overall Performance

Benchmark	Per-thread CPI	Per core CPI	Effective CPI for eight cores	Effective IPC for eight cores
TPC-C	7.2	1.8	0.225	4.4
SPECJBB	5.6	1.40	0.175	5.7
SPECWeb99	6.6	1.65	0.206	4.8

Figure 4.29 The per-thread CPI, the per-core CPI, the effective eight-core CPI, and the effective IPC (inverse of CPI) for the eight-core T1 processor.

At first glance, one might react that T1 is not very efficient, since the effective throughout is between 56% and 71% of the ideal on these three benchmarks.
But, consider the comparative performance of a wide-issue superscalar.
Processors such as the Itanium 2 (higher transistor count, much higher power, comparable silicon area) would need to achieve incredible instruction throughput sustaining 4.5–5.7 instructions per clock, well more than double the acknowledged IPC.
It appears quite clear that, at least for integer-oriented server applications with thread-level parallelism, a *multicore approach is a much better alternative than a single very wide issue processor*.

Interaction between multithreading and parallel processing

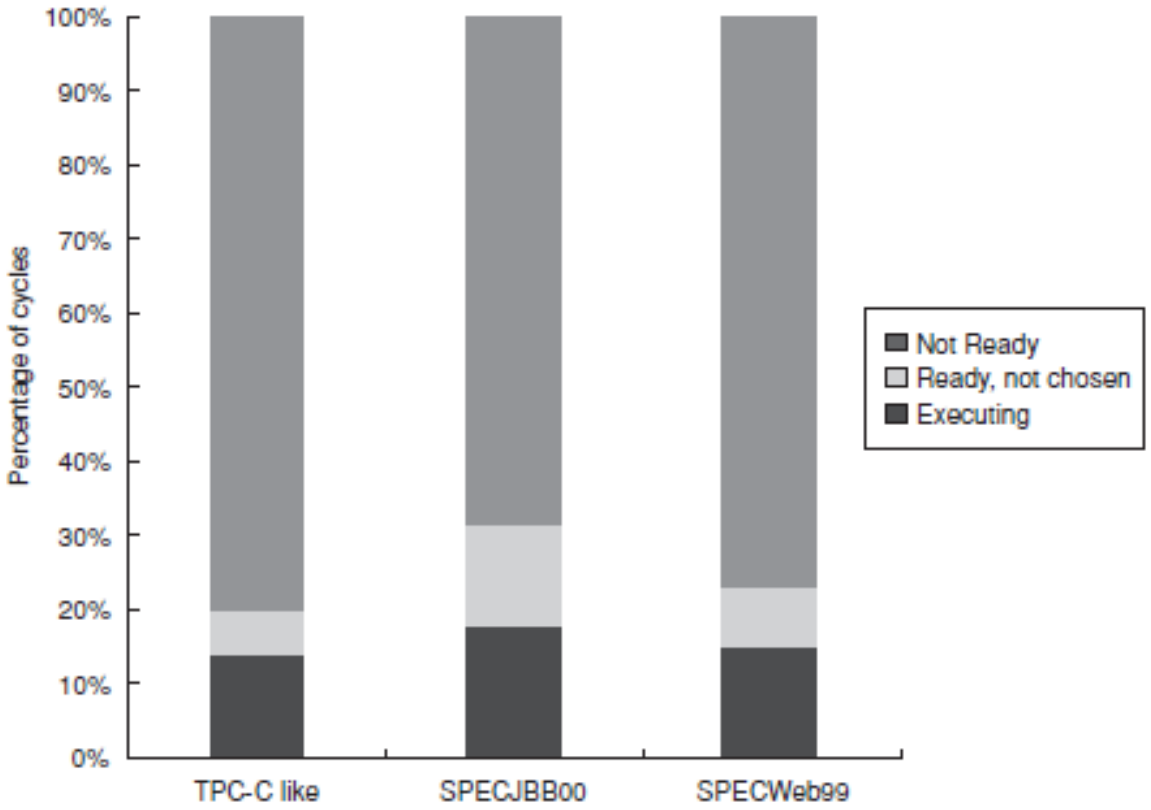


Figure 4.30 Breakdown of the status on an average thread. Executing indicates the thread issues an instruction in that cycle. Ready but not chosen means it could issue, but another thread has been chosen, and *not ready* indicates that the thread is awaiting the completion of an event (a pipeline delay or cache miss, for example).

Why a thread is not ready?

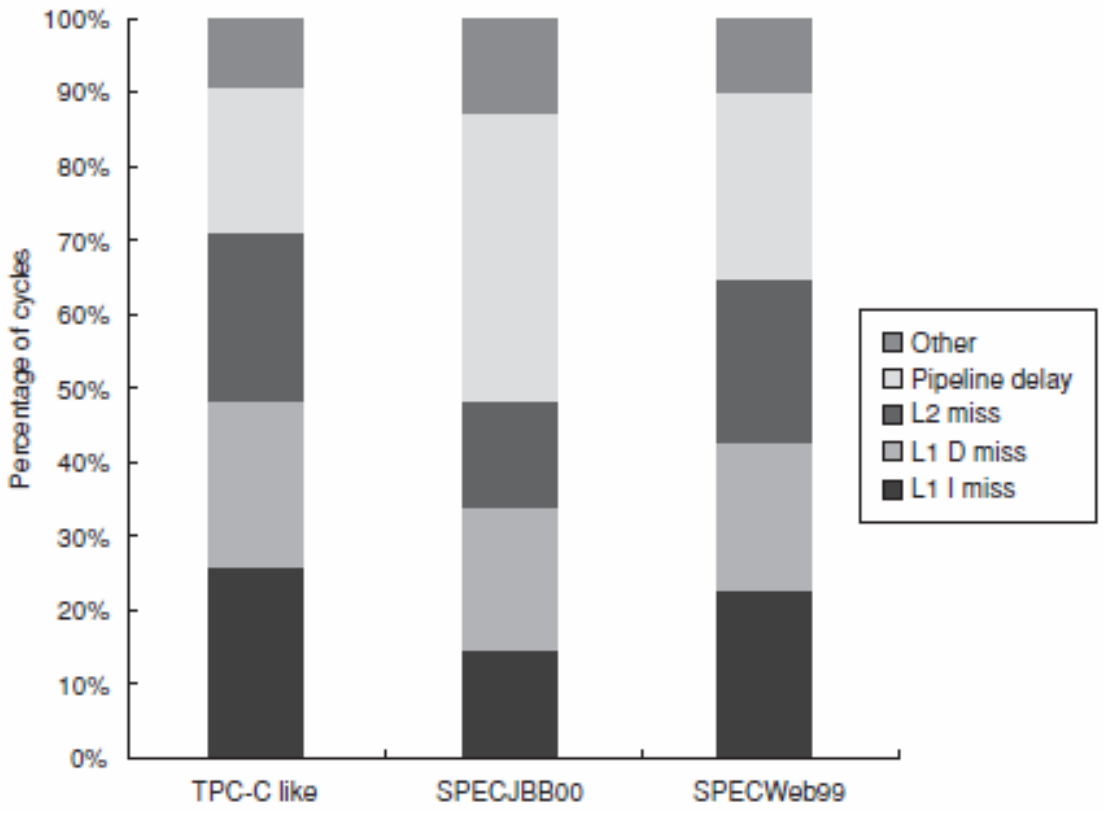


Figure 4.31 The breakdown of causes for a thread being not ready. The contribution to the "other" category varies. In TPC-C, store buffer full is the largest contributor; in SPEC-JBB, atomic instructions are the largest contributor; and in SPECWeb99, both factors contribute.

Performance of Multicore Processors on SPEC Benchmarks

- Among recent processors, T1 is uniquely characterized by an intense focus on thread-level parallelism versus instruction-level parallelism.
- It uses multithreading to achieve performance from a simple RISC pipeline, and it uses multiprocessing with eight cores on a die to achieve high throughput for server applications.
- In contrast, the dual-core Power5, Opteron, and Pentium D use both multiple issue and multicore.
- Of course, exploiting significant ILP requires much bigger processors, with the result being that fewer cores fit on a chip in comparison to T1.

Four multicore processors

Characteristic	SUN T1	AMD Opteron	Intel Pentium D	IBM Power5
Cores	8	2	2	2
Instruction issues per clock per core	1	3	3	4
Multithreading	Fine-grained	No	SMT	SMT
Caches L1 I/D in KB per core L2 per core/shared L3 (off-chip)	16/8 3 MB shared	64/64 1 MB/core	12K uops/16 1 MB/core	64/32 L2: 1.9 MB shared L3: 36 MB
Peak memory bandwidth (DDR2 DRAMs)	34.4 GB/sec	8.6 GB/sec	4.3 GB/sec	17.2 GB/sec
Peak MIPS FLOPS	9600 1200	7200 4800 (w. SSE)	9600 6400 (w. SSE)	7600 7600
Clock rate (GHz)	1.2	2.4	3.2	1.9
Transistor count (M)	300	233	230	276
Die size (mm^2)	379	199	206	389
Power (W)	79	110	130	125

Figure 4.32 Summary of the features and characteristics of four multicore processors.

Significant other differences exits => next slide

Differences: Floating-point support

1. There are significant differences in floating-point support and performance.
   - The Power5 puts a major emphasis on floating-point performance, the Opteron and Pentium allocate significant resources, and the T1 almost ignores it.
   - As a result, Sun is unlikely to provide any benchmark results for floating-point applications.
   - A comparison that included only integer programs would be unfair to the three processors that include significant floating-point hardware

Other differences

- The multiprocessor expandability of these systems differs and that affects the memory system design and the use of external interfaces. Power5 is designed for the most expandability. The Pentium and Opteron design offer limited multiprocessor support. The T1 is not expandable to a larger system.
- The implementation technologies vary, making comparisons based on die size and power more difficult.
- There are significant differences in the assumptions about memory systems and the memory bandwidth available. For benchmarks with high cache miss rates, such as TPC-C and similar programs, the processors with larger memory bandwidth have a significant advantage.

Nonetheless, given the importance of the trade-off between ILP-centric and TLP-centric designs, it would be useful to try to quantify the performance differences as well as the efficacy of the approaches.

Performance comparison

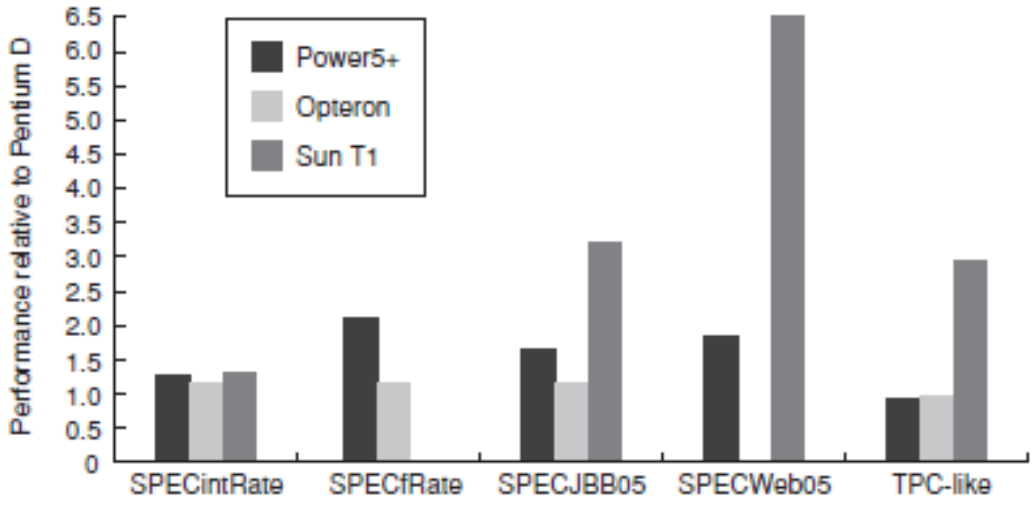


Figure 4.33 Four dual-core processors showing their performance on a variety of SPEC benchmarks and a TPC-C-like benchmark. All the numbers are normalized to the Pentium D (which is therefore at 1 for all the benchmarks). Some results are estimates from slightly larger configurations (e.g., four cores and two processors, rather than two cores and one processor), including the Opteron SPECJBB2005 result, the Power5 SPECWeb05 result, and the TPC-C results for the Power5, Opteron, and Pentium D. At the current time, Sun has refused to release SPECRate results for either the integer or FP portion of the suite.

Performance comparison

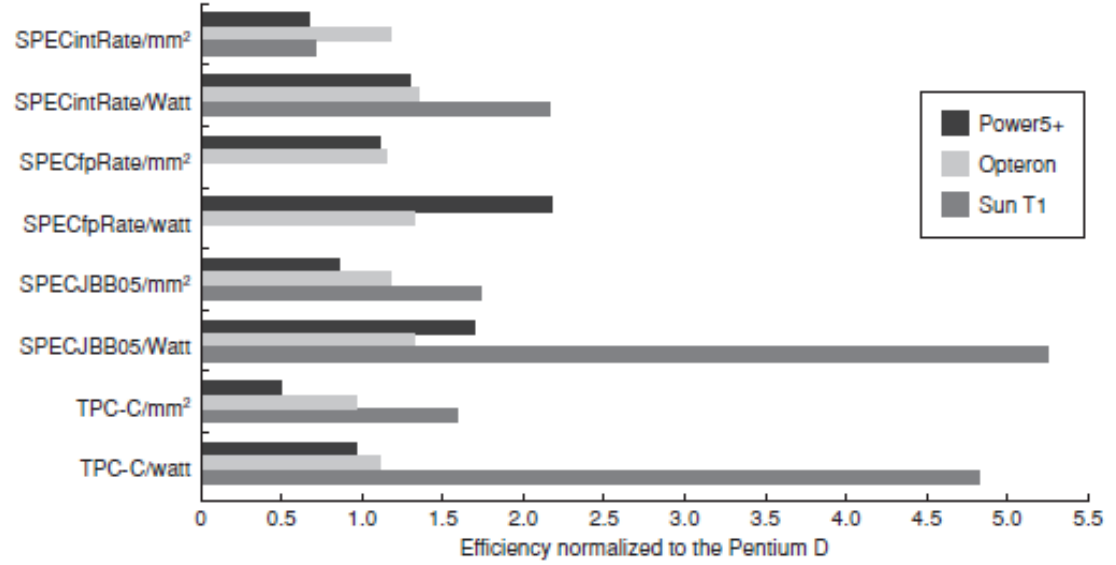


Figure 4.34 Performance efficiency on SPECRate for four dual-core processors, normalized to the Pentium D metric (which is always 1).

- Efficiency measures in terms of performance per unit die area and per watt for the four dual-core processors, with the results normalized to the measurement on the Pentium D.
- There is significant advantage in terms of performance/watt for the Sun T1 processor on the TPC-C-like and SPECJBB05 benchmarks.
- These measurements clearly demonstrate that for multithreaded applications, a TLP approach may be much more power efficient than an ILP-intensive approach.
- This is the strongest evidence to date that the TLP route may provide a way to increase performance in a power-efficient fashion.

Conclusions for TLP

- It is too early to conclude whether the TLP-intensive approaches will win across the board.
- If typical server applications *have enough threads to keep T1 busy and the per-thread performance is acceptable, the T1 approach will be tough to beat.*
- If single-threaded performance remains important in server or desktop environments, then we may see the market further fracture with significantly different processors for throughput-oriented environments and environments where higher single-thread performance remains important.

Outline

- Introduction
- Symmetric Shared-Memory Architectures
- Performance of Symmetric Shared-Memory Multiprocessors
- Distributed Shared Memory and Directory-Based Coherence
- Synchronization: The Basics
- Models of Memory Consistency: An Introduction
- Crosscutting Issues
- Putting It All Together: The Sun T1 Multiprocessor
- Fallacies and Pitfalls

Pitfall: Measuring performance of multiprocessors by linear speedup versus execution time.

- Although speedup is one facet of a parallel program, it is not a direct measure of performance.
  - A program that linearly improves performance to equal 100 Intel 486s may be slower than the sequential version on a Pentium 4.
- Comparing execution times is fair only if you are comparing the best algorithms on each computer.
  - Comparing the identical code on two computers may seem fair, but it is not; the parallel program may be slower on a uniprocessor than a sequential version.
- Developing a parallel program will sometimes lead to algorithmic improvements, so that comparing the previously best-known sequential program with the parallel code—which seems fair — will not compare equivalent algorithms.
- To reflect this issue, the terms *relative speedup* (same program) and *true speedup* (best program) are sometimes used.

Fallacy: Amdahl's Law doesn't apply to parallel computers.

- In 1987, the head of a research organization claimed that Amdahl's Law had been broken by an MIMD multiprocessor.
- This statement hardly meant, however, that the law has been overturned for parallel computers;
  - the neglected (not parallelizable) portion of the program will still limit performance.

Fallacy: Linear speedups are needed to make multiprocessors cost-effective.

- It is widely recognized that one of the major benefits of parallel computing is to offer a “shorter time to solution” than the fastest uniprocessor.
- Many people, however, also hold the view that parallel processors cannot be as cost-effective as uniprocessors unless they can achieve perfect linear speedup.
- This argument says that because the cost of the multiprocessor is a linear function of the number of processors, anything less than linear speedup means that the ratio of performance/cost decreases, making a parallel processor less cost-effective than using a uniprocessor.

Linear Speedup

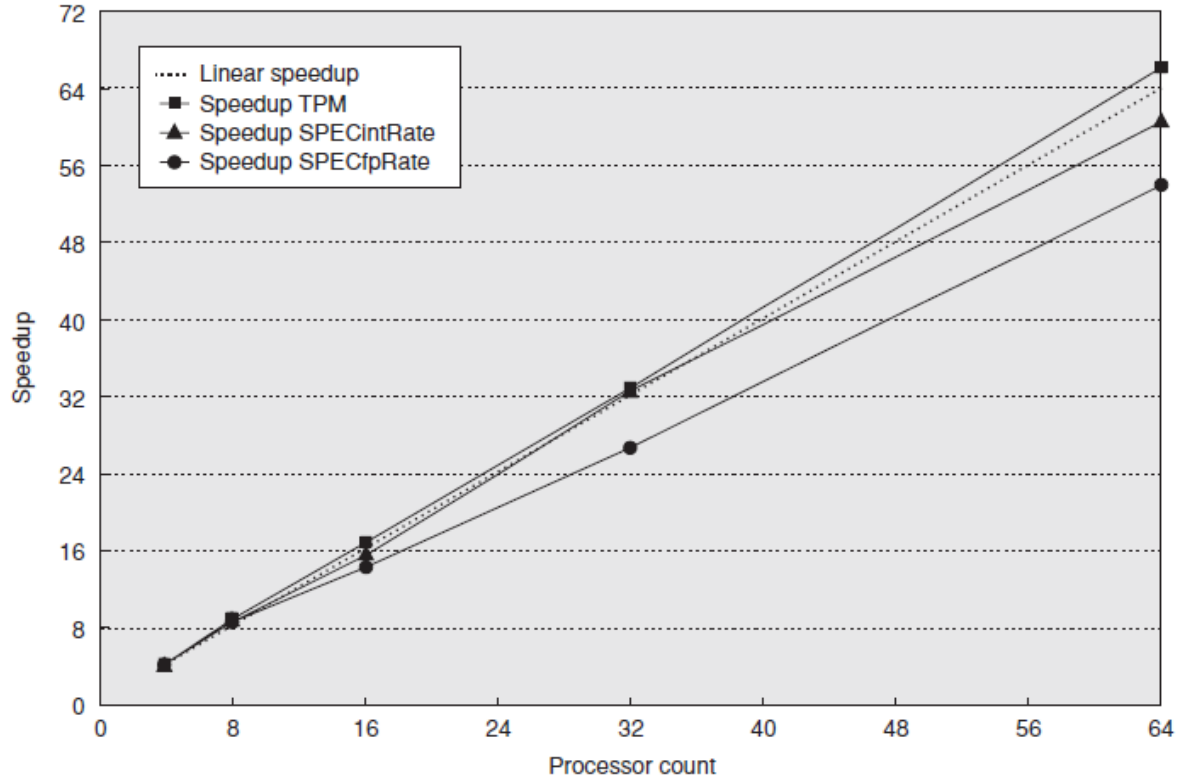


Figure 4.35 Speedup for three benchmarks on an IBM eserver p5 multiprocessor when configured with 4, 8, 16, 32, and 64 processors. The dashed line shows linear speedup.

For SPECintRate and SPECfpRate, speedup is less than linear, but so is the cost, since unlike TPC-C the amount of main memory and disk required both scale less than linearly.

Linear Speedup

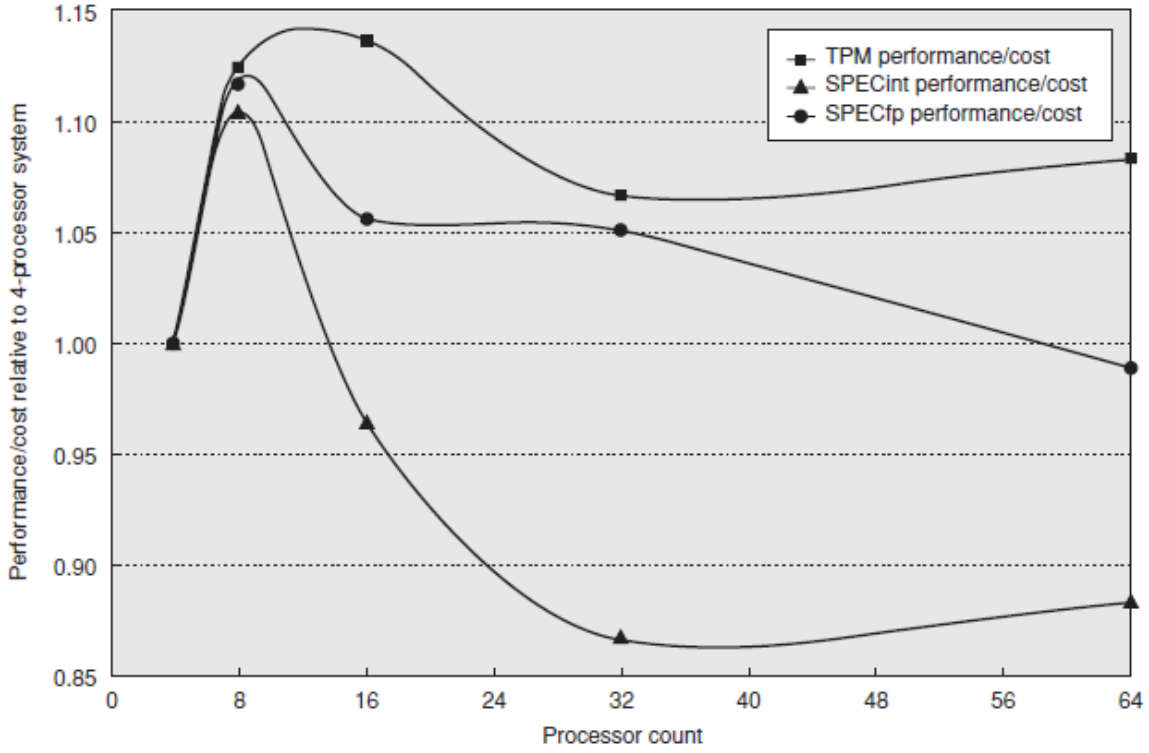


Figure 4.36 The performance/cost relative to a 4-processor system for three benchmarks run on an IBM eserver p5 multiprocessor containing from 4 to 64 processors shows that the larger processor counts can be as cost-effective as the 4-processor configuration. For TPC-C the configurations are those used in the official runs, which means that disk and memory scale nearly linearly with processor count, and a 64-processor machine is approximately twice as expensive as a 32-processor version. In contrast, the disk and memory are scaled more slowly (although still faster than necessary to achieve the best SPECRate at 64 processors). In particular the disk configurations go from one drive for the 4-processor version to four drives (140 GB) for the 64-processor version. Memory is scaled from 8 GB for the 4-processor system to 20 GB for the 64-processor system.

- As Figure 4.36 shows, larger processor counts can actually be more cost-effective than the four-processor configuration.
- In the future, as the cost of multiple processors decreases compared to the cost of the support infrastructure (cabinets, power supplies, fans, etc.), the performance/cost ratio of larger processor configurations will improve further.

Conclusions about speedup

- In comparing the cost-performance of two computers, we must be sure to include accurate assessments of both total system cost and what performance is achievable.
- For many applications with larger memory demands, such a comparison can dramatically increase the attractiveness of using a multiprocessor.

Fallacy: Scalability is almost free

- The goal of scalable parallel computing was a focus of much of the research and a significant segment of the high-end multiprocessor development from the mid- 1980s through the late 1990s.
- In the first half of that period, it was widely held that you could build scalability into a multiprocessor and then simply offer the multiprocessor at any point on the scale from a small to large number of processors without sacrificing cost-effectiveness.
- The difficulty with this view is that multiprocessors that scale to larger processor counts require substantially more investment (in both dollars and design time) in:
  - the interprocessor communication network
  - operating system support
  - reliability
  - reconfigurability.

Pitfall: Not developing the software to take advantage of, or optimize for, a multiprocessor architecture.

- This pitfall indicates the kind of subtle but significant performance bugs that can arise when software runs on multiprocessors.
- Like many other key software components, the OS algorithms and data structures must be rethought in a multiprocessor context.
- Similar problems exist in memory structures, which increases the coherence traffic in cases where no sharing is actually occurring.

End of Lecture 5

- Readings
  - Book: Chapter 4