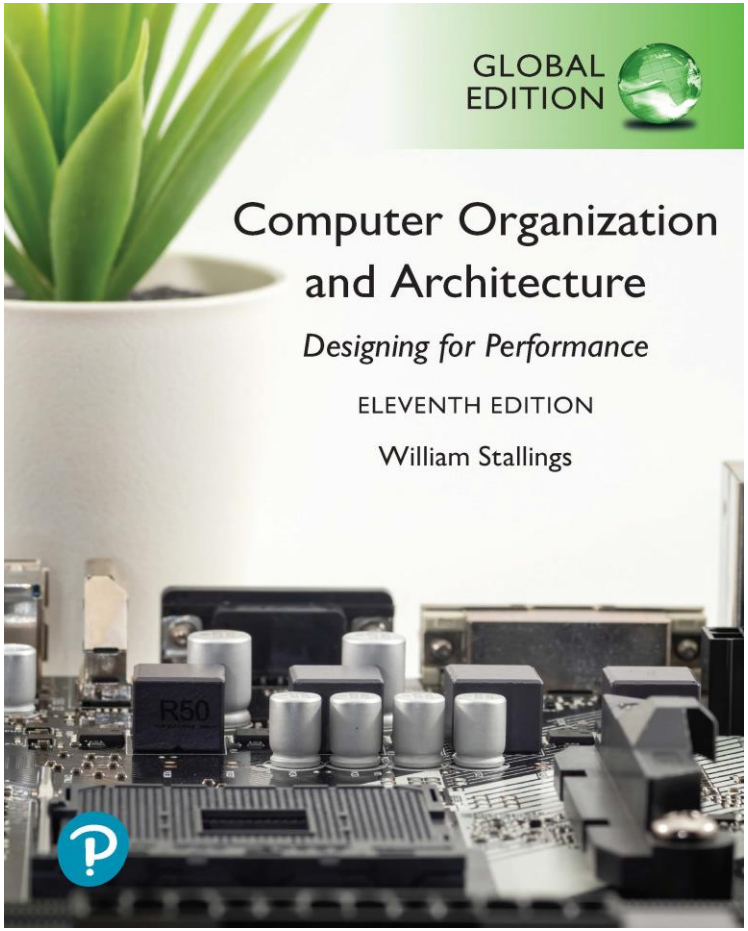
Computer Organization and Architecture

Designing for Performance

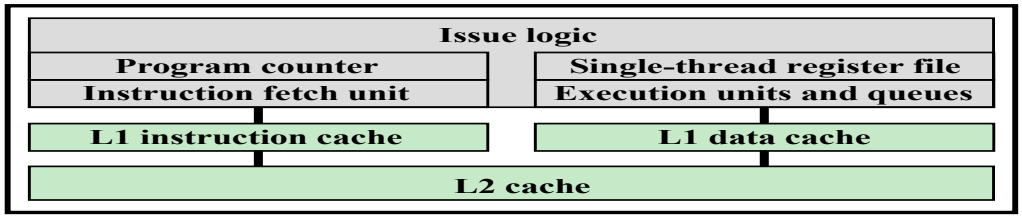
11th Edition, Global Edition



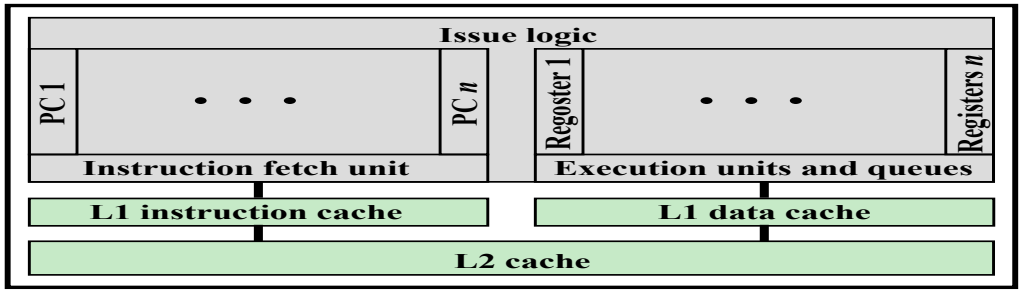
Chapter 21

Multicore Computers

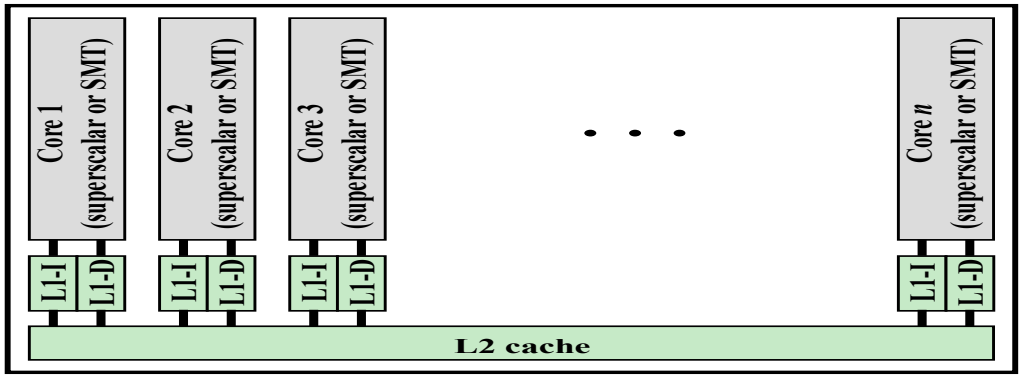
Figure 21.1 Alternative Chip Organizations



(a) Superscalar



(b) Simultaneous multithreading



(c) Multicore

The organizational changes in processor design have primarily been focused on exploiting ILP, so that more work is done in each clock cycle. These changes include, in chronological order (Figure 21.1):

* **Pipelining:** Individual instructions are executed through a pipeline of stages so that while one instruction is executing in one stage of the pipeline, another instruction is executing in another stage of the pipeline.

* **Superscalar:** Multiple pipelines are constructed by replicating execution resources. This enables parallel execution of instructions in parallel pipelines, so long as hazards are avoided.

- **Simultaneous multithreading (SMT):** Register banks are replicated so that multiple threads can share the use of pipeline resources.

Figure 21.2
Power and Memory Considerations

To maintain the trend of higher performance as the number of transistors per chip

rises, designers have resorted to more elaborate processor designs (pipelining, superscalar,

SMT) and to high clock frequencies. Unfortunately, power requirements have

grown exponentially as chip density and clock frequency have risen. This was shown

in Figure 2.2.

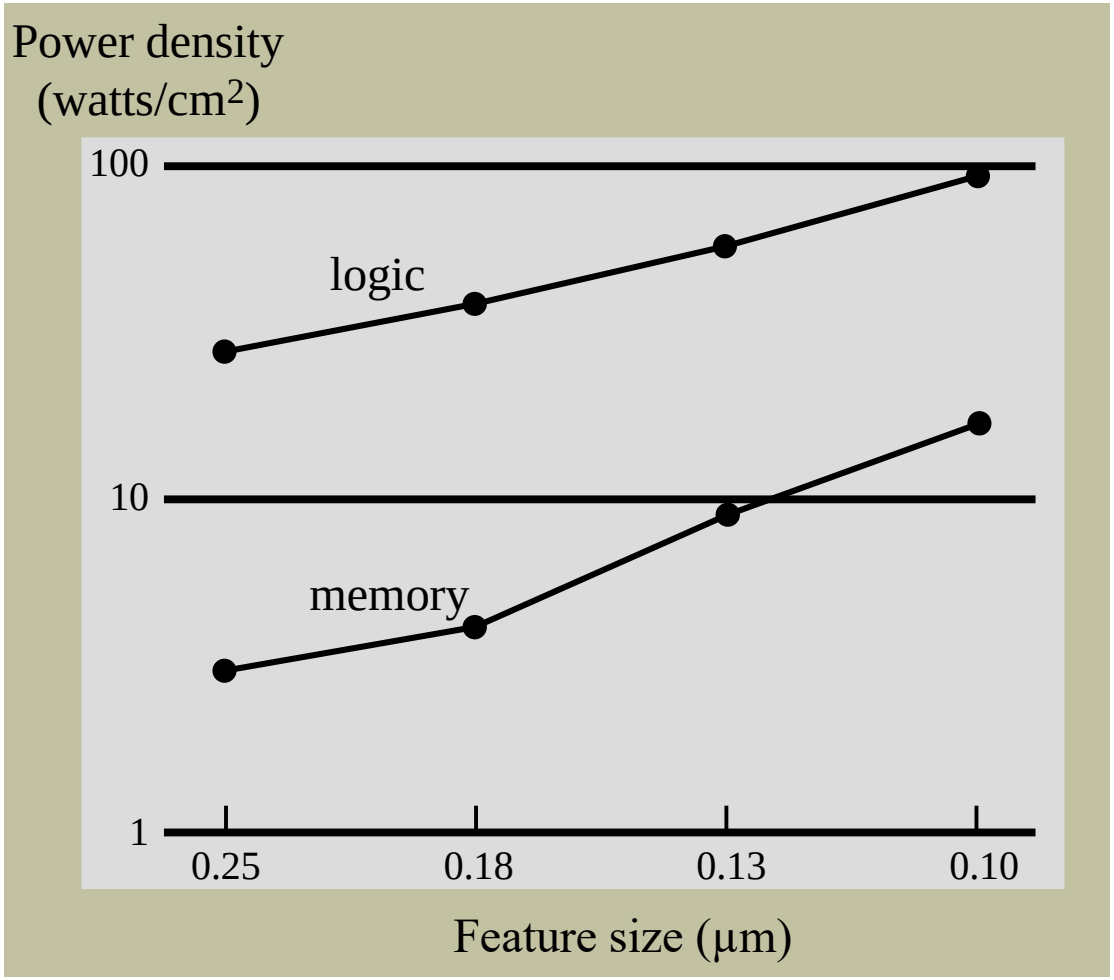
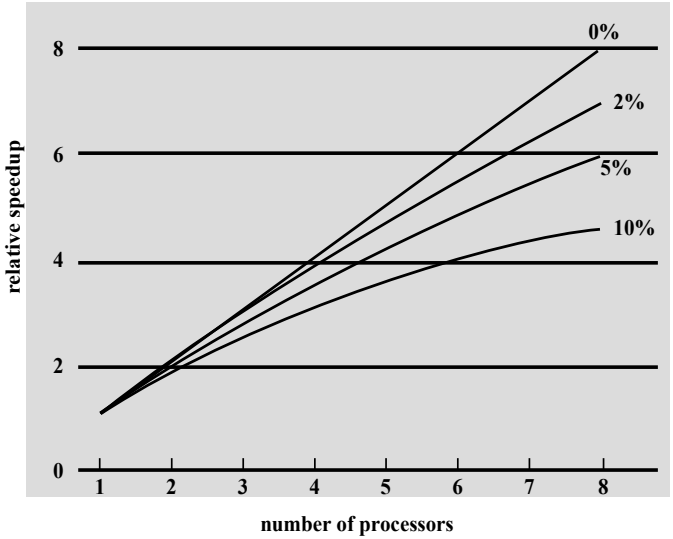


Figure 21.3
Performance Effect of Multiple Cores

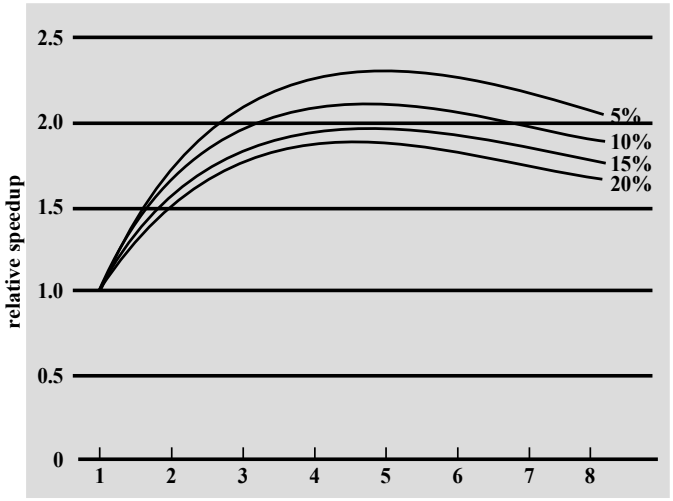
The potential performance benefits of a multicore organization depend on the ability to effectively exploit the parallel resources available to the application. Let us focus first on a single application running on a multicore system.

The law assumes a program in which a fraction $(1 - f)$ of the execution time involves code that is inherently serial and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead.

This law appears to make the prospect of a multicore organization attractive. But as Figure 21.3a shows, even a small amount of serial code has a noticeable impact. If only 10% of the code is inherently serial (f = 0.9), running the program on a multi- core system with 8 processors yields a performance gain of only a factor of 4.7. In addition, software typically incurs overhead as a result of communication and distribution of work among multiple processors and as a result of cache coherence overhead. This results in a curve where performance peaks and then begins to degrade because of the increased burden of the overhead of using multiple processors (e.g., coordination and OS management). Figure 21.3b, from [MCDO05], is a representative example.

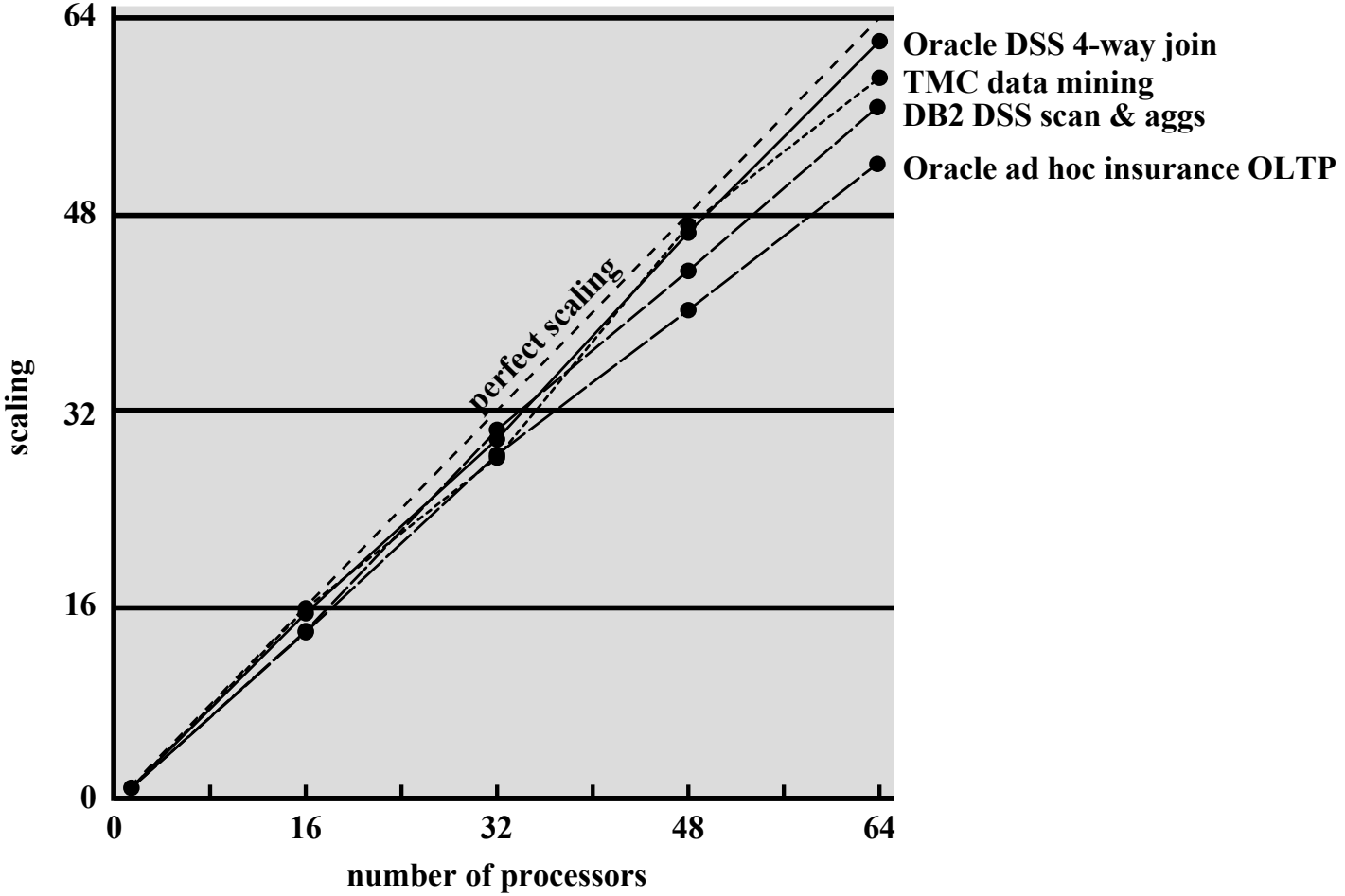


(a) Speedup with 0%, 2%, 5%, and 10% sequential portions



(b) Speedup with overheads

Figure 21.4
Scaling of Database Workloads on Multiple-Processor Hardware



Effective Applications for Multicore Processors

- **Multi-threaded native applications**
  - Thread-level parallelism
  - Characterized by having a small number of highly threaded processes
- **Multi-process applications**
  - Process-level parallelism
  - Characterized by the presence of many single-threaded processes
- **Java applications**
  - Embrace threading in a fundamental way
  - Java Virtual Machine is a multi-threaded process that provides scheduling and memory management for Java applications
- **Multi-instance applications**
  - If multiple application instances require some degree of isolation, virtualization technology can be used to provide each of them with its own separate and secure environment

Threading Granularity

- The minimal unit of work that can be beneficially parallelized
- The finer the granularity the system enables, the less constrained is the programmer in parallelizing a program
- Finer grain threading systems allow parallelization in more situations than coarse-grained ones
- The choice of the target granularity of an architecture involves an inherent tradeoff
  - The finer grain systems are preferable because of the flexibility they afford to the programmer
  - The finer the threading granularity, the more significant part of the execution is taken by the threading system overhead

Figure 21.5
Hybrid Threading for Rendering Module

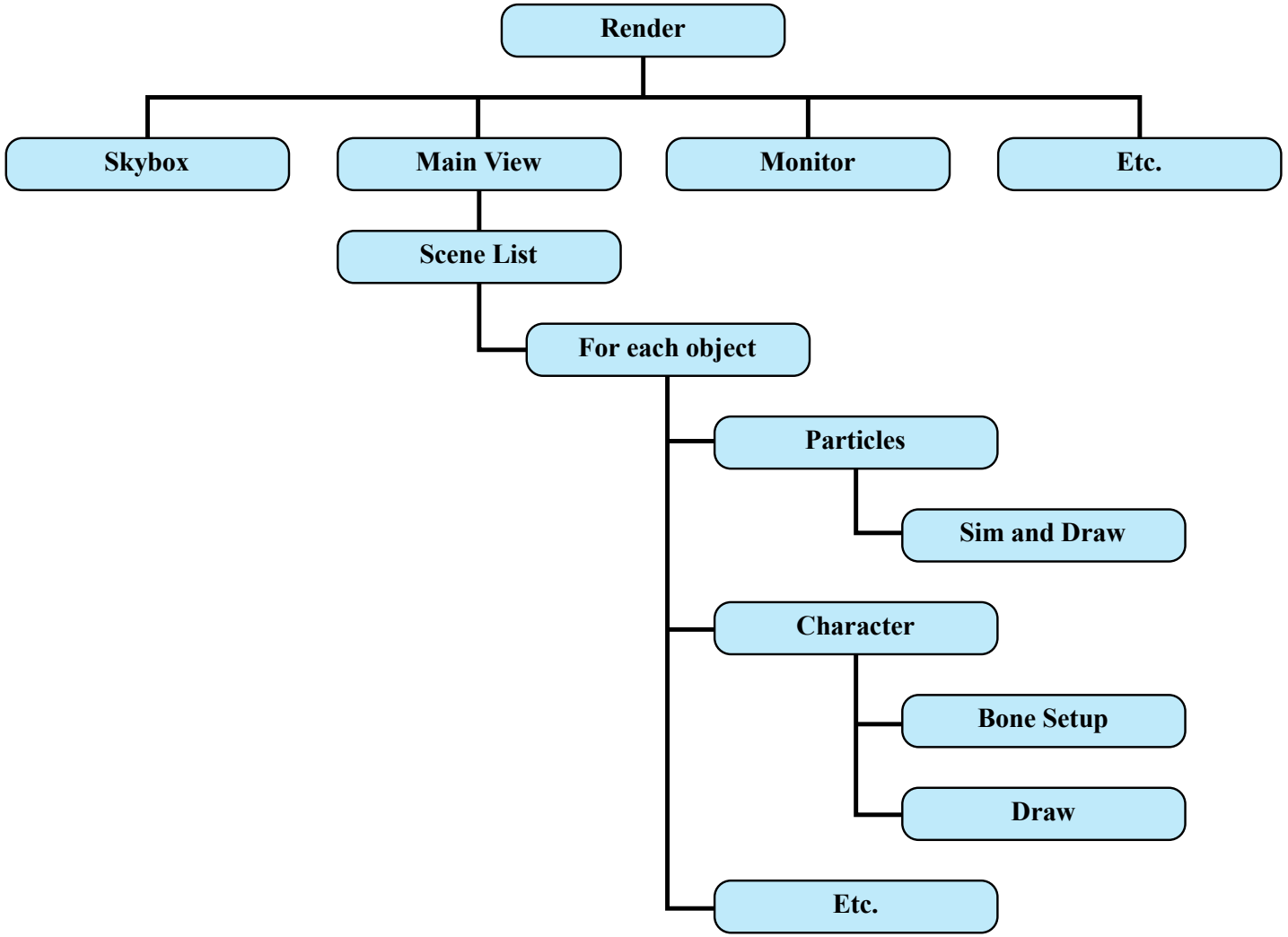
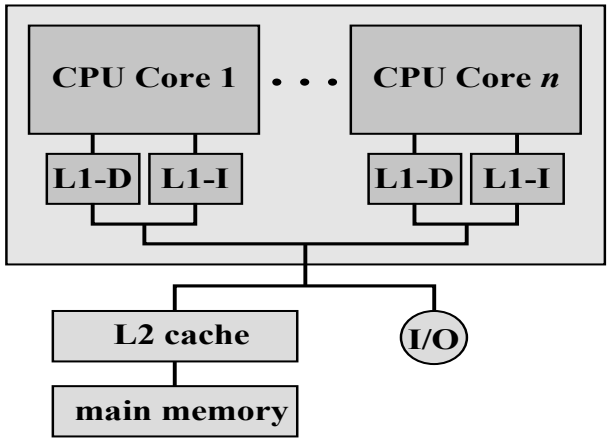
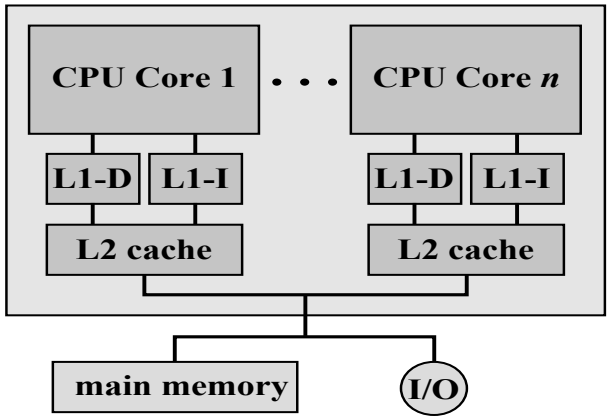


Figure 21.6
Multicore Organization Alternatives

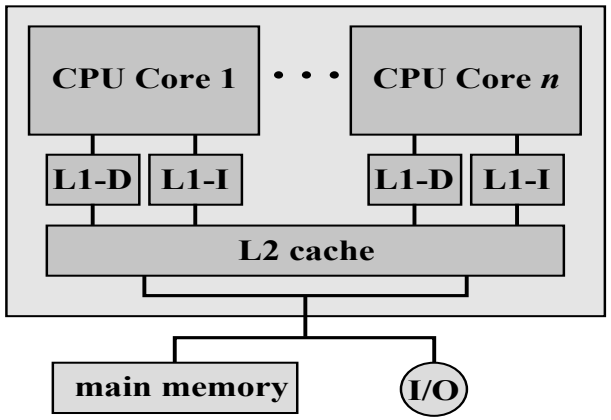
Figure 21.6 shows four general organizations for multicore systems.



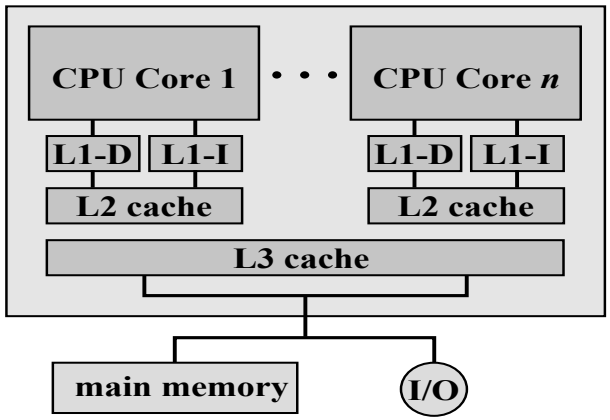
(a) Dedicated L1 cache



(b) Dedicated L2 cache



(c) Shared L2 cache



(d) Shared L3 cache

Heterogeneous Multicore Organization

Refers to a processor chip that includes more than one kind of core

The most prominent trend is the use of both CPUs and graphics processing units (GPUs) on the same chip

- This mix however presents issues of coordination and correctness

GPUs are characterized by the ability to support thousands of parallel execution trends

Thus, GPUs are well matched to applications that process large amounts of vector and matrix data

Figure 21.7
Heterogenous Multicore Chip Elements

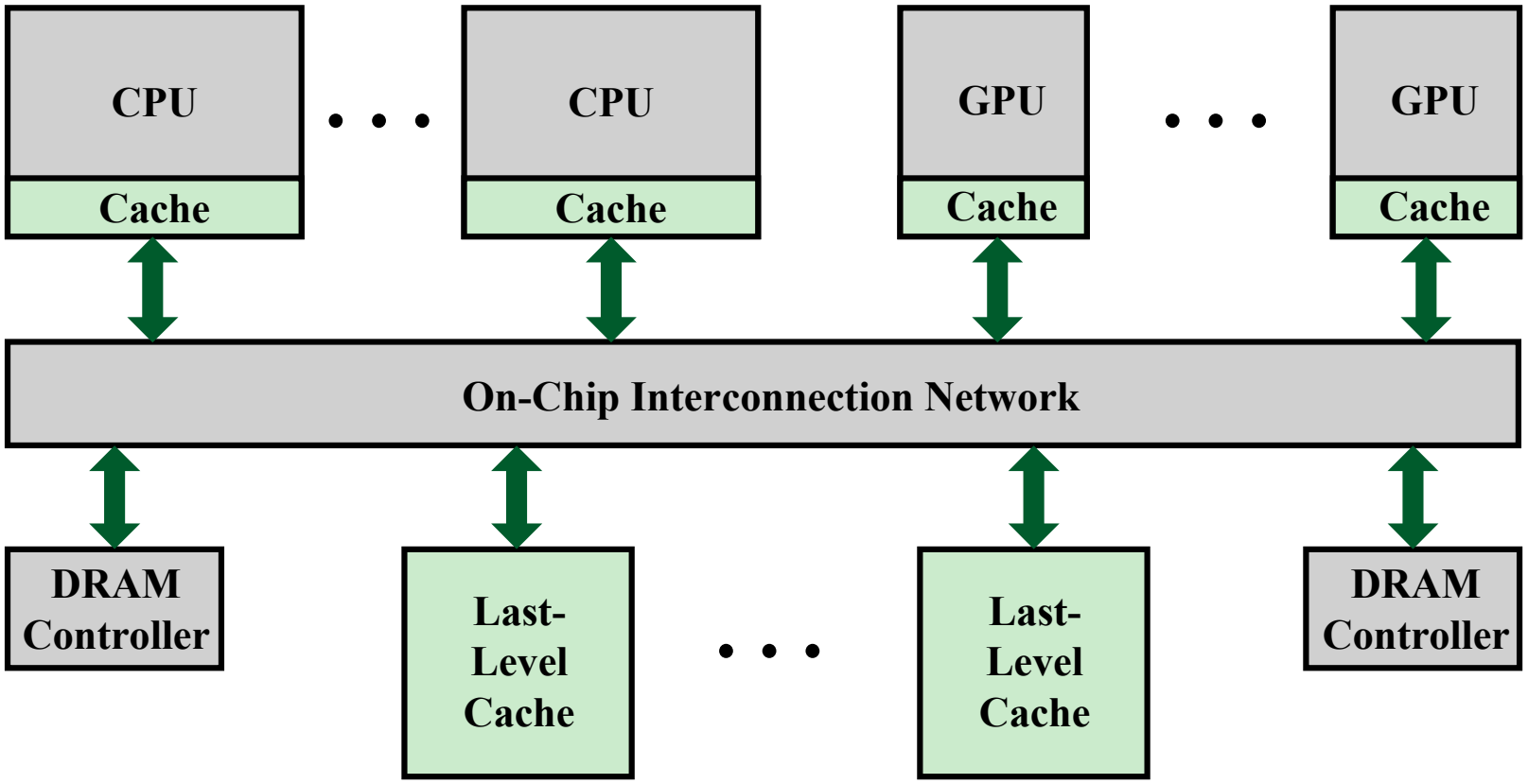


Table 21.1
Operating Parameters of AMD 5100K Heterogeneous Multicore Processor

	CPU	GPU
Clock frequency (GHz)	3.8	0.8
Cores	4	384
FLOPS/core	8	2
GFLOPS	121.6	614.4

FLOPS = floating point operations per second.
FLOPS/core = number of parallel floating point operations that can be performed.

Table 21.1 illustrates the potential performance benefit of combining CPUs and GPUs for scientific applications

Heterogeneous System Architecture (HSA)

- Key features of the HSA approach include:
  - The entire virtual memory space is visible to both CPU and GPU
  - The virtual memory system brings in pages to physical main memory as needed
  - A coherent memory policy ensures that CPU and GPU caches both see an up-to-date view of data
  - A unified programming interface that enables users to exploit the parallel capabilities of the GPUs within programs that rely on CPU execution as well
- The overall objective is to allow programmers to write applications that exploit the serial power of CPUs and the parallel-processing power of GPUs seamlessly with efficient coordination at the OS and hardware level

Figure 21.8 Texas Instruments 66AK2H12 Heterogenous Multicore Chip

Another common example of a heterogeneous multicore chip is a mixture of CPUs and digital signal processors (DSPs). A DSP provides ultra-fast instruction sequences (shift and add; multiply and add), which are commonly used in math-intensive digital signal processing applications. DSPs are used to process analog data from sources such as sound, weather satellites, and earthquake monitors.

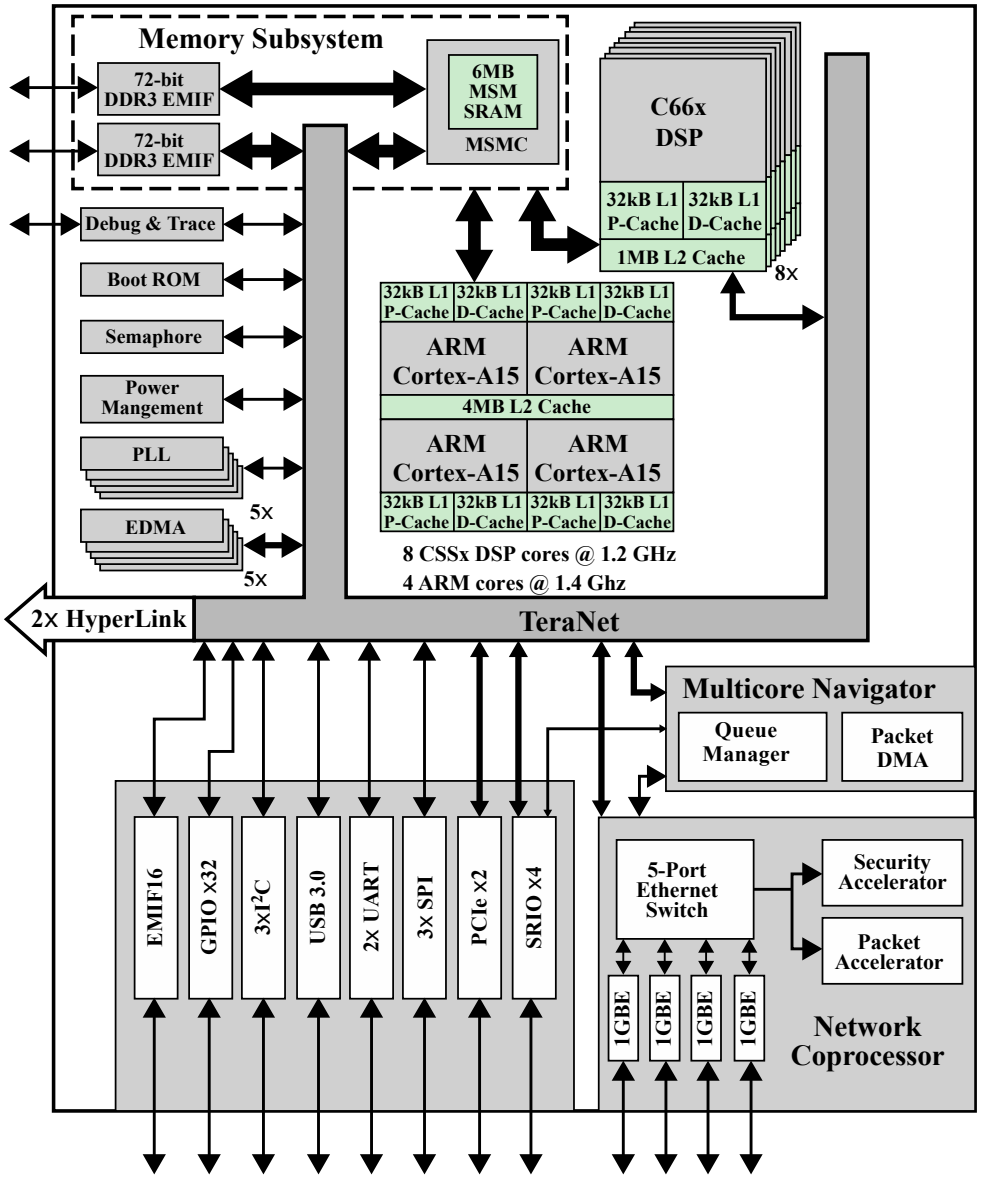
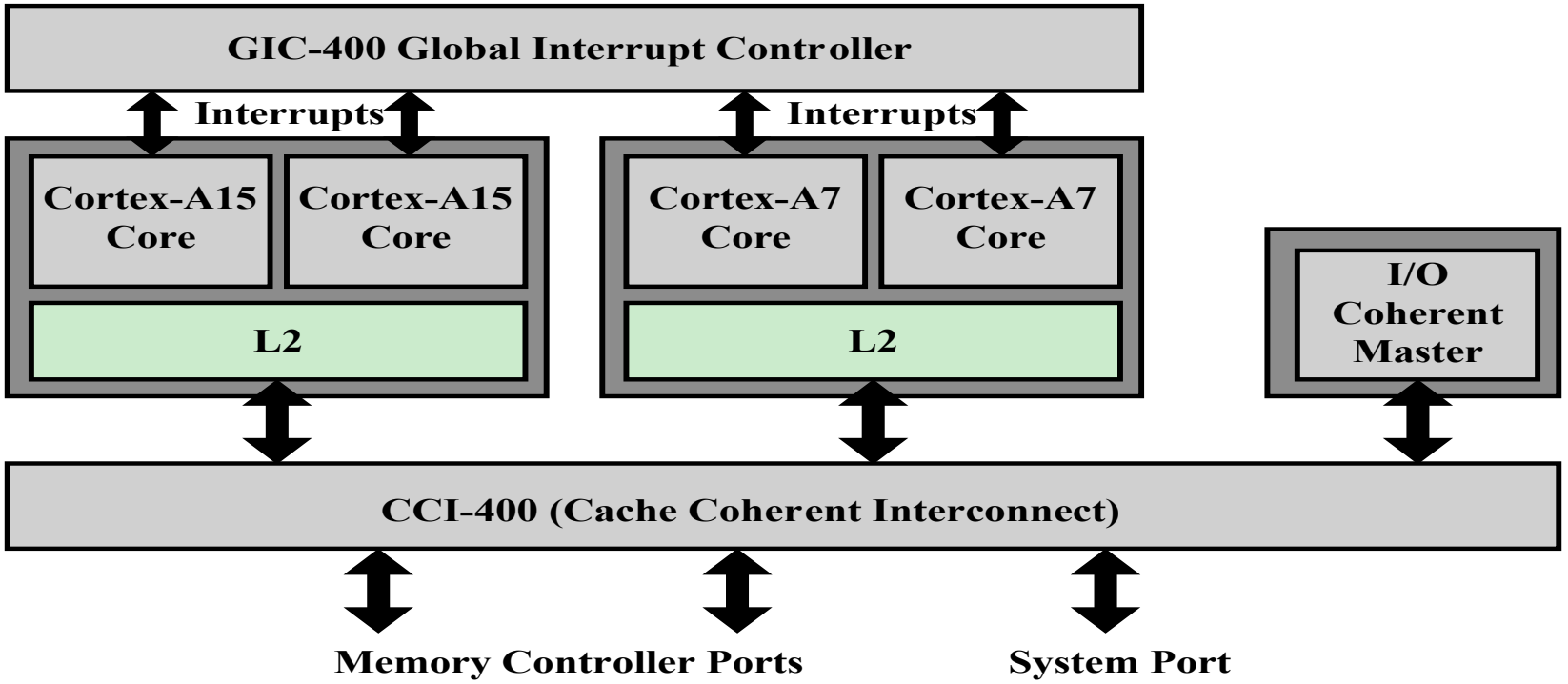
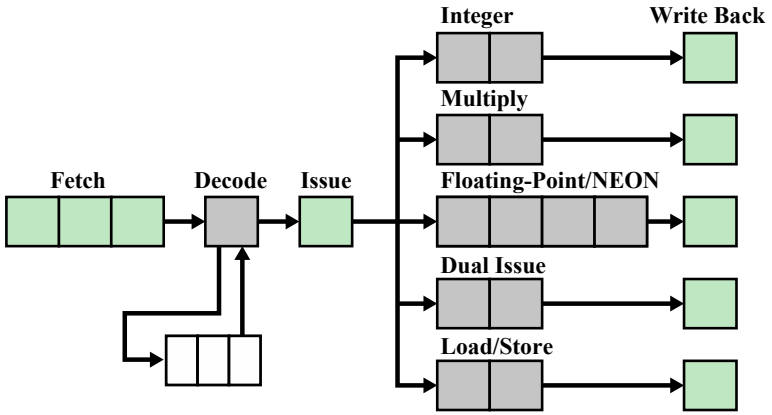


Figure 21.9
big.Little Chip Components

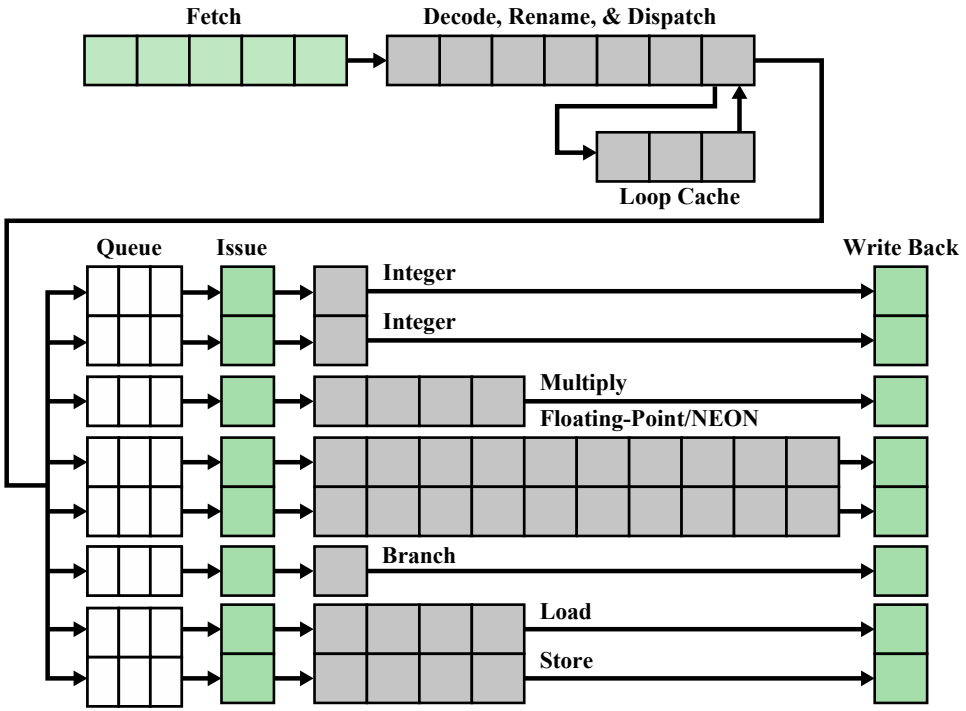


Another recent approach to heterogeneous multicore organization is the use of multiple cores that have equivalent ISAs but vary in performance or power efficiency. The leading example of this is ARM's big.Little architecture, which we examine in this section.

Figure 21.10 Cortex A-7 and A-15 Pipelines



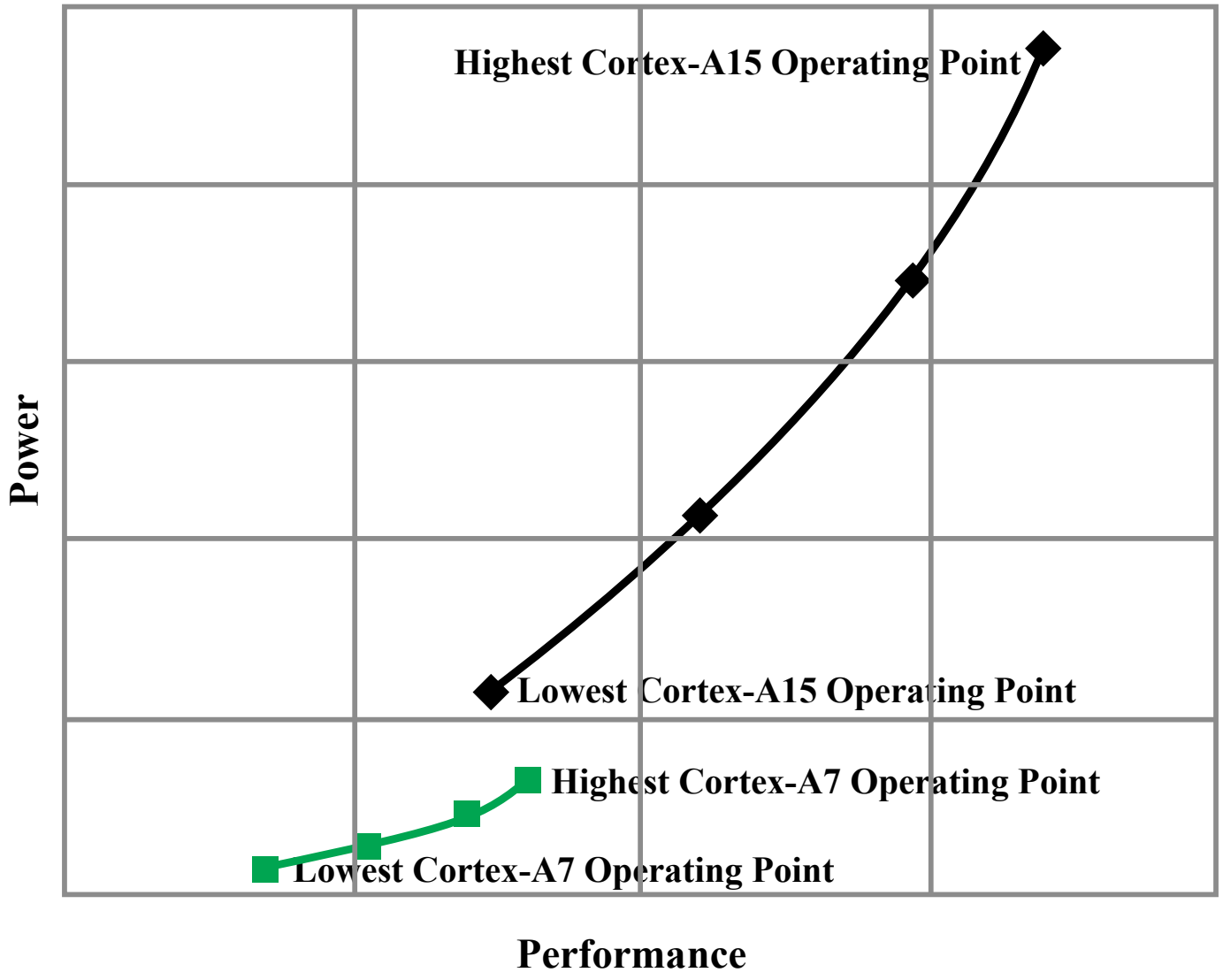
(a) Cortex A-7 Pipeline



(b) Cortex A-15 Pipeline

The A7 is far simpler and less powerful than the A15. But its simplicity requires far fewer transistors than does the A15's complexity—and fewer transistors require less energy to operate. The differences between the A7 and A15 cores are seen most clearly by examining their instruction pipelines, as shown in Figure 21.10.

Figure 21.11
Cortex-A7 and A15 Performance Comparison



Cache Coherence

- May be addressed with software-based techniques
  - Software burden consumes too many resources in a SoC chip
- When multiple caches exist there is a need for a cache-coherence scheme to avoid access to invalid data
- There are two main approaches to hardware implemented cache coherence
  - Directory protocols
  - Snoopy protocols
- ACE (Advanced Extensible Interface Coherence Extensions)
  - Hardware coherence capability developed by ARM
  - Can be configured to implement whether directory or snoopy approach
  - Has been designed to support a wide range of coherent masters with differing capabilities
  - Supports coherency between dissimilar processors enabling ARM big.Little technology
  - Supports I/O coherency for un-cached masters, supports masters with differing cache line sizes, differing internal cache state models, and masters with write-back or write-through caches

Figure 21.12
ARM ACE Cache Line States

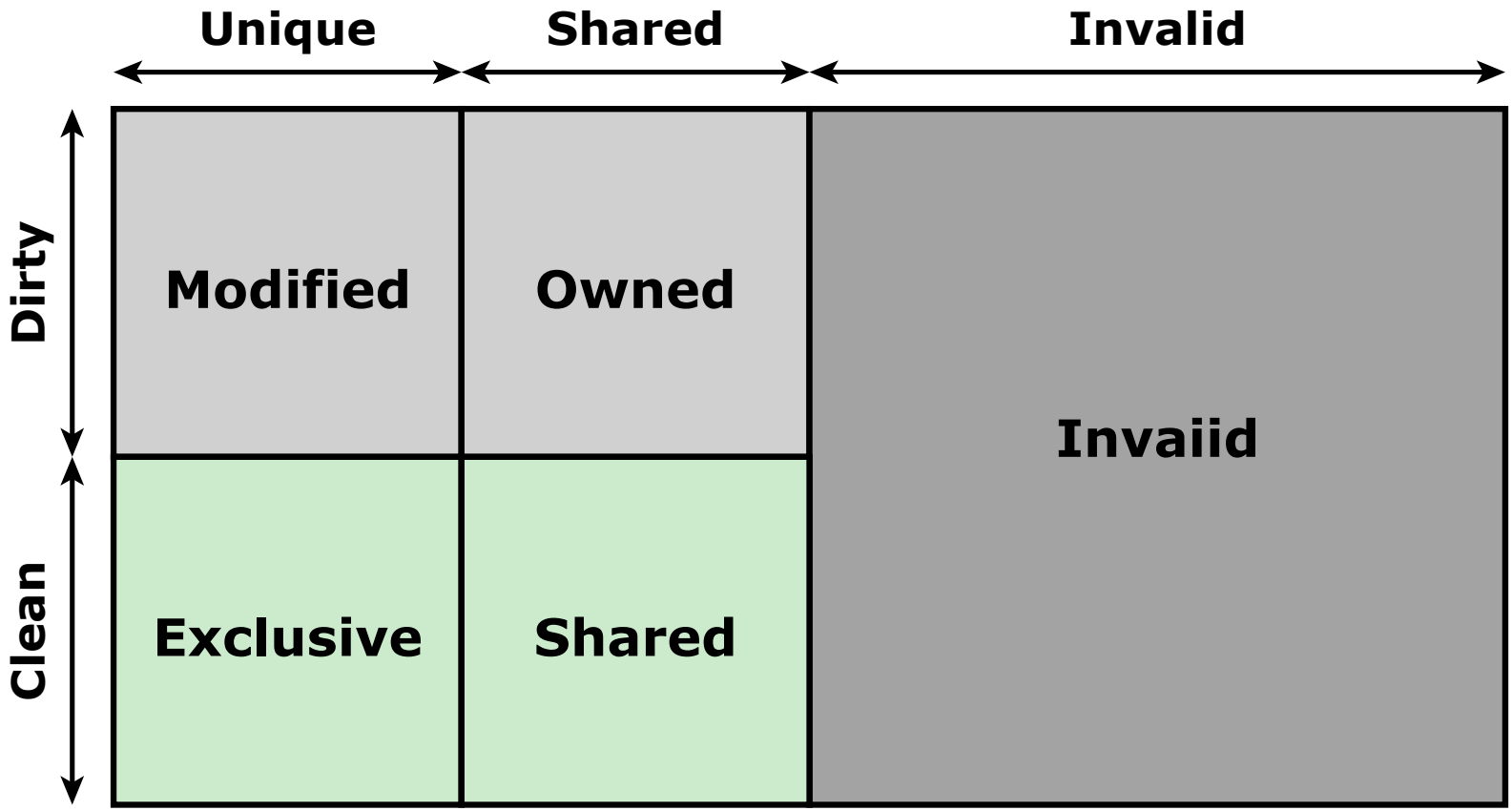


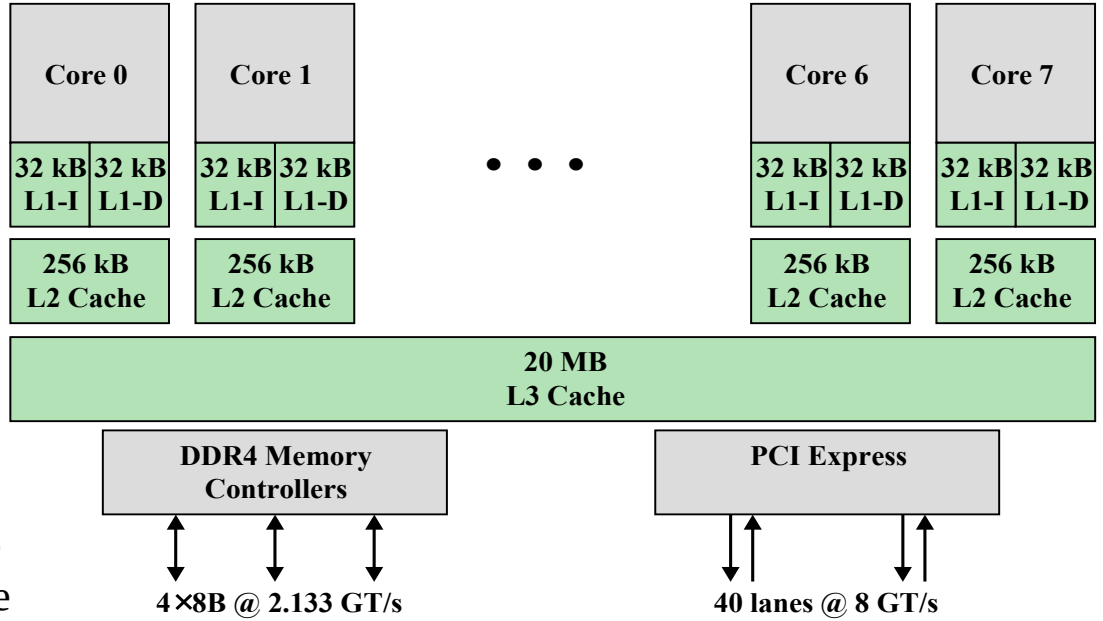
Table 21.2
Comparison of States in Snoop Protocols

(a) MESIM				
	Modified	**Exclusive**	**Shared**	**Invalid**
Clean/Dirty	Dirty	Clean	Clean	N/A
Unique?	Yes	Yes	No	N/A
Can write?	Yes	Yes	No	N/A
Can forward?	Yes	Yes	Yes	N/A
Comments	Must write back to share or replace	Transitions to M on write	Shared implies clean, can forward	Cannot read

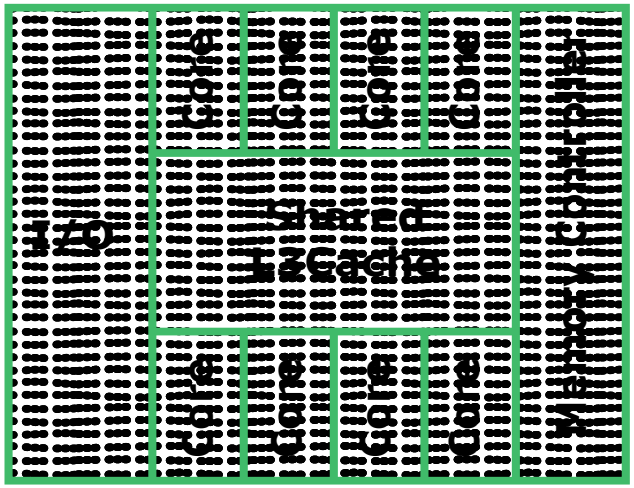
(b) MOISI					
	Modified	**Owned**	**Exclusive**	**Shared**	**Invalid**
Clean/Dirty	Dirty	Dirty	Clean	Either	N/A
Unique?	Yes	Yes	Yes	Yes	N/A
Can write?	Yes	Yes	Yes	Yes	N/A
Can forward?	Yes	Yes	Yes	Yes	N/A
Comments	Can share without write back	Must write back to transition	Transitions to M on write	Shared, can be dirty or clean	Cannot read

(Table can be found on page 756 in the textbook.)

Figure 21.13 Intel Core i7-5960X Block Diagram



(a) Block diagram



(b) Physical layout on chip

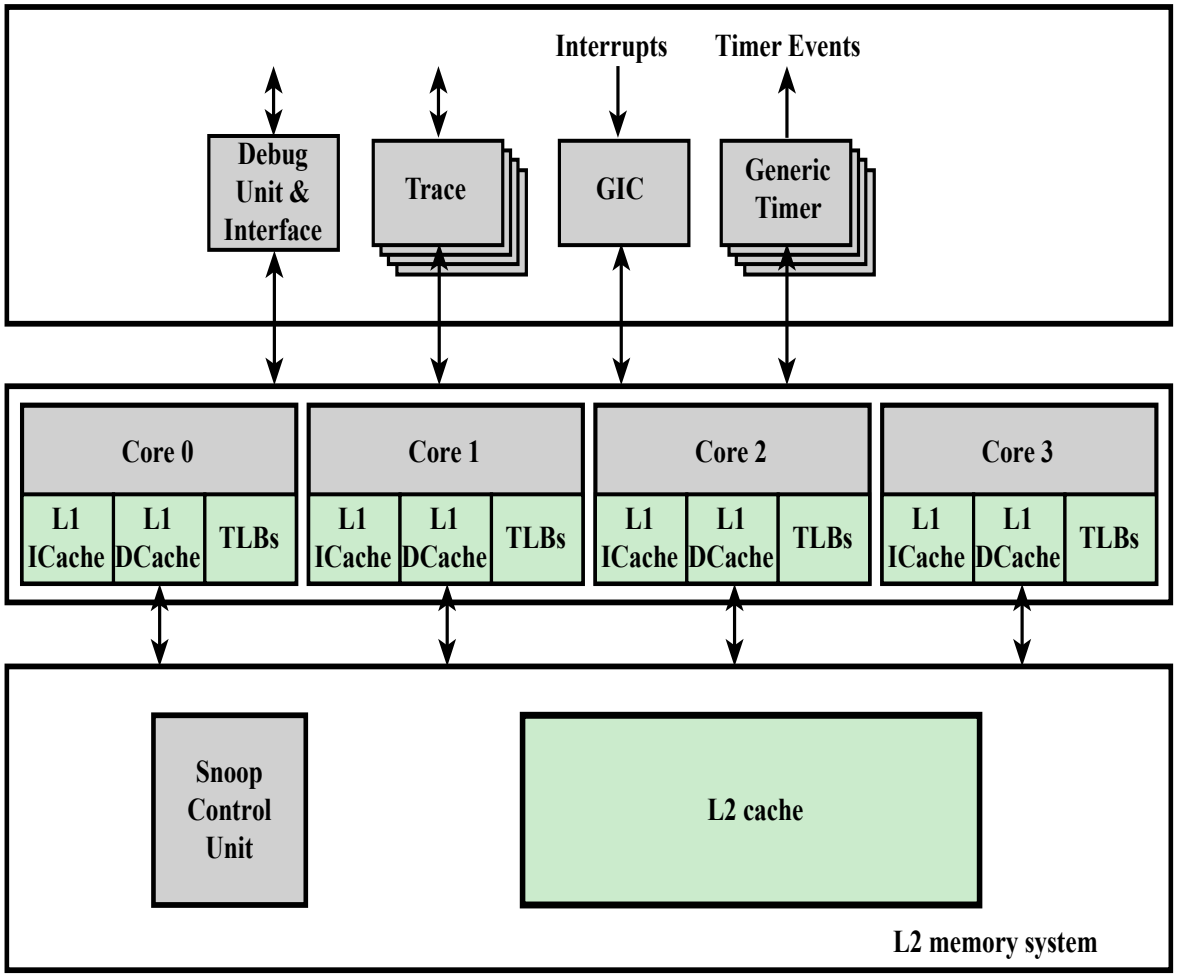
Intel has introduced a number of multicore products in recent years. In this section, we look at the Intel Core i7-5960X.

The general structure of the Intel Core i7-5960X is shown in Figure 21.13. Each core has its own **dedicated L2 cache** and the eight cores share a 20-MB **L3 cache** . One mechanism Intel uses to make its caches more effective is prefetching, in which the hardware examines memory access patterns and attempts to fill the caches speculatively with data that's likely to be requested soon.

Figure 21.14 ARM Cortex-A15 MPCore Chip Block Diagram

In this section, we introduce the Cortex-A15 MPCore multicore chip, which is a homogeneous multicore processor using multiple A15 cores. The A15 MPCore is a high-performance chip targeted at applications including mobile computing, high-end digital home servers, and wireless infrastructure.

Generic interrupt controller (GIC): Handles interrupt detection and interrupt prioritization. The GIC distributes interrupts to individual cores.



Interrupt Handling

Generic interrupt controller (GIC) provides:

- Masking of interrupts
- Prioritization of the interrupts
- Distribution of the interrupts to the target A15 cores
- Tracking the status of interrupts
- Generation of interrupts by software

GIC

- Is memory mapped
- Is a single functional unit that is placed in the system alongside A15 cores
- This enables the number of interrupts supported in the system to be independent of the A15 core design
- Is accessed by the A15 cores using a private interface through the SCU

GIC

Designed to satisfy two functional requirements:

- Provide a means of routing an interrupt request to a single CPU or multiple CPUs as required
- Provide a means of interprocessor communication so that a thread on one CPU can cause activity by a thread on another CPU

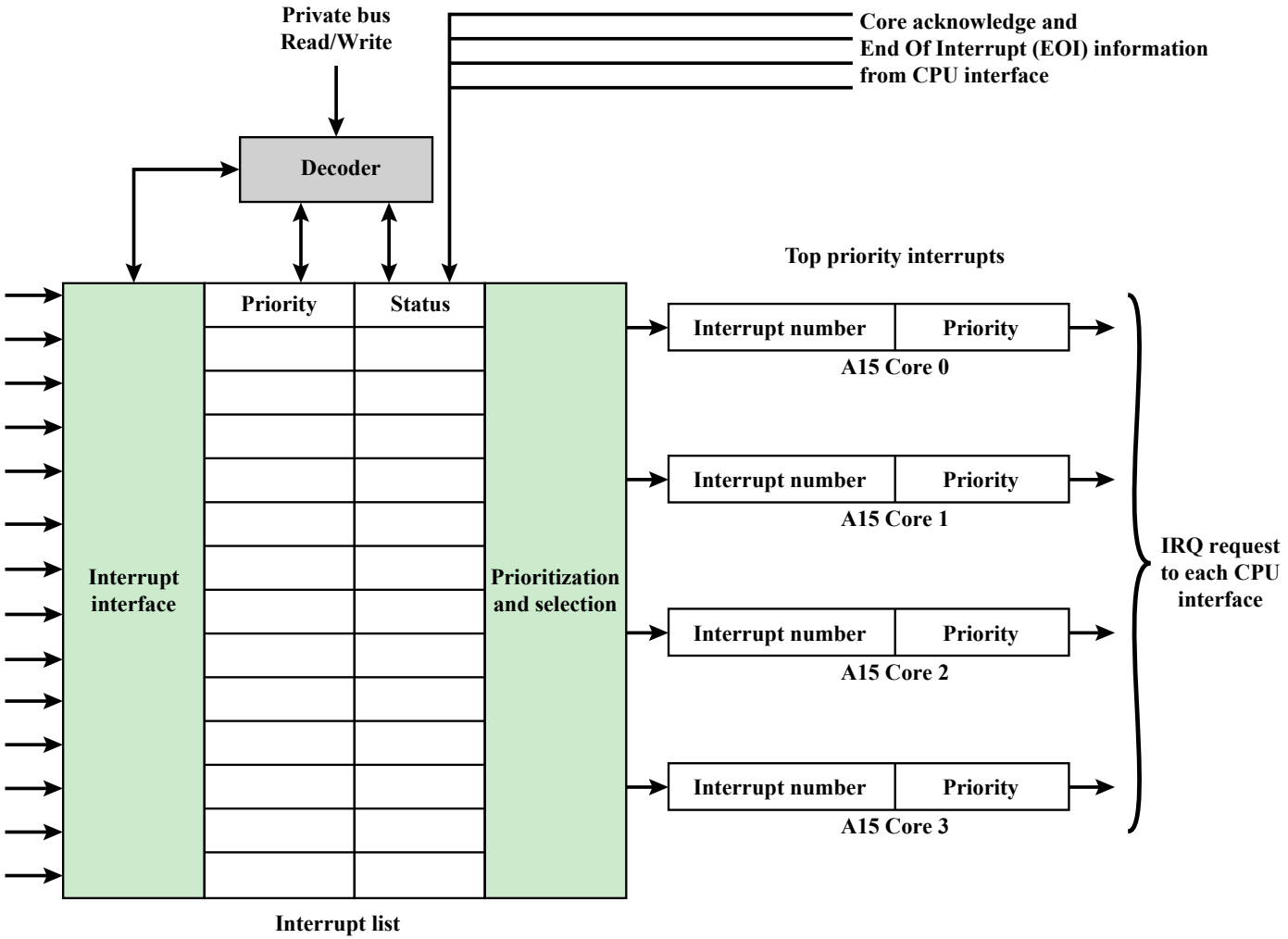
Can route an interrupt to one or more CPUs in the following three ways:

- An interrupt can be directed to a specific processor only
- An interrupt can be directed to a defined group of processors
- An interrupt can be directed to all processors

Interrupts can be:

- Inactive
  - One that is nonasserted, or which in a multiprocessing environment has been completely processed by that CPU but can still be either Pending or Active in some of the CPUs to which it is targeted, and so might not have been cleared at the interrupt source
- Pending
  - One that has been asserted, and for which processing has not started on that CPU
- Active
  - One that has been started on that CPU, but processing is not complete
  - Can be pre-empted when a new interrupt of higher priority interrupts A15 core interrupt processing
- Interrupts come from the following sources:
  - Interprocessor interrupts (IPIs)
  - Private timer and/or watchdog interrupts
  - Legacy FIQ lines
  - Hardware interrupts

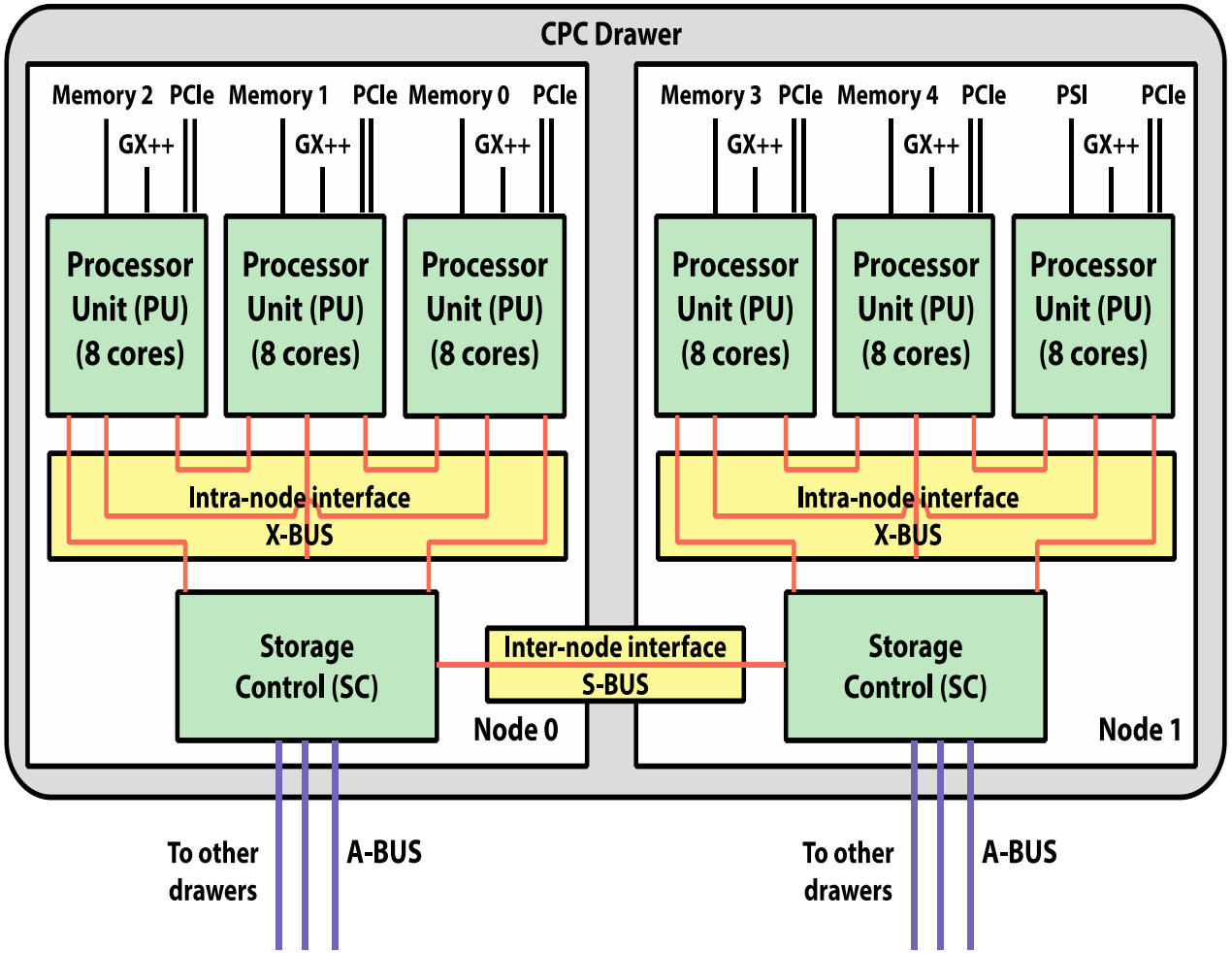
Figure 21.15
Generic Interrupt Controller Block Diagram



Cache Coherency

- Snoop Control Unit (SCU) resolves most of the traditional bottlenecks related to access to shared data and the scalability limitation introduced by coherence traffic
- L1 cache coherency scheme is based on the MESI protocol
- Direct Data Intervention (DDI)
  - Enables copying clean data between L1 caches without accessing external memory
  - Reduces read after write from L1 to L2
  - Can resolve local L1 miss from remote L1 rather than L2
- Duplicated tag RAMs
  - Cache tags implemented as separate block of RAM
  - Same length as number of lines in cache
  - Duplicates used by SCU to check data availability before sending coherency commands
  - Only send to CPUs that must update coherent data cache
- Migratory lines
  - Allows moving dirty data between CPUs without writing to L2 and reading back from external memory

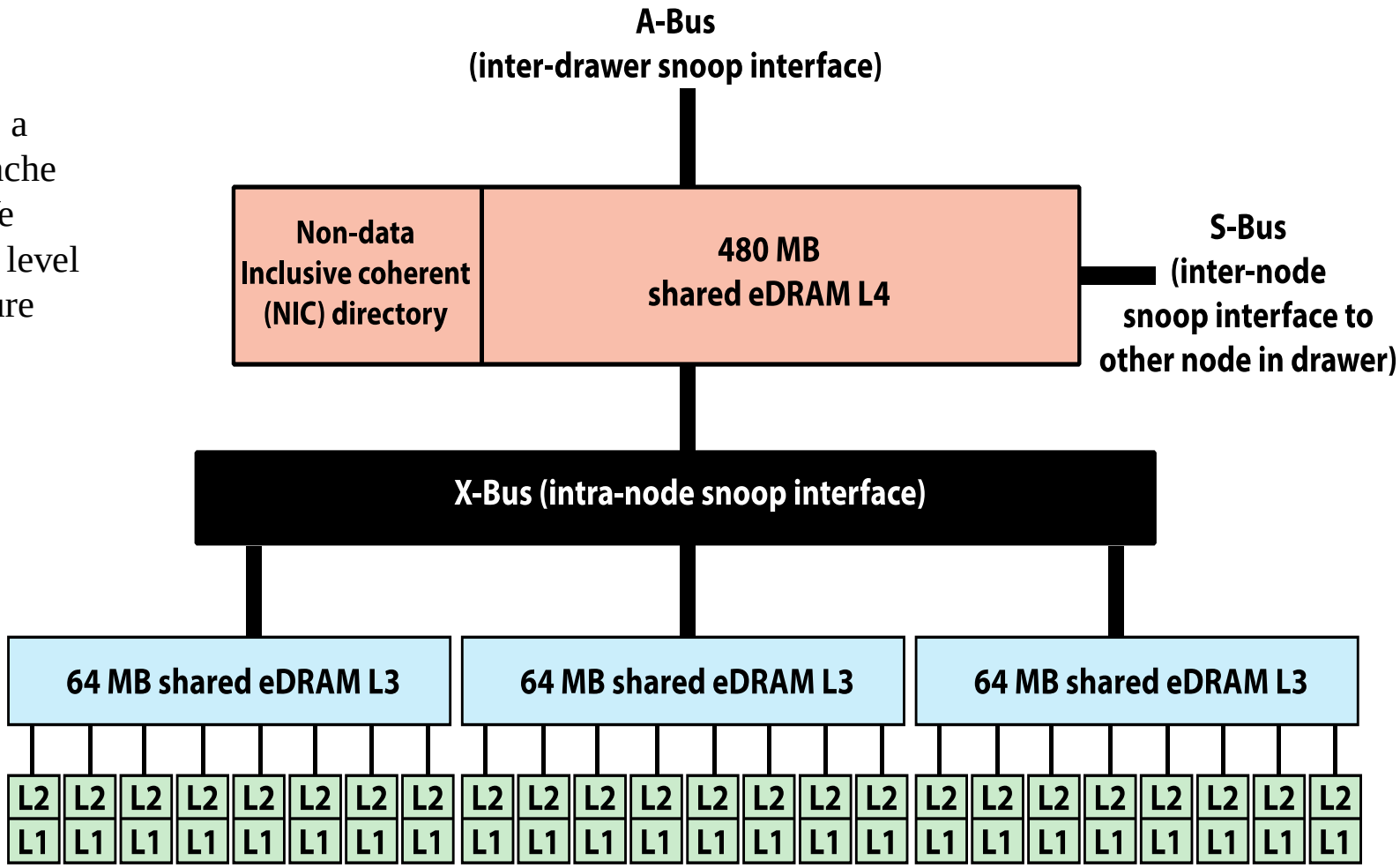
Figure 21.16
IBM z13 Drawer Structure



The principal building block of the z13 is the processor node. Two nodes are connected together with an inter-node S-Bus and housed in a drawer that fits into a slot of the mainframe cabinet. A-Bus interfaces connect these two nodes with nodes in other drawers. A z13 configuration can have up to four drawers.

Figure 21.17
IBM z13 Cache Hierarchy in Single Node

The zEC12 incorporates a four level cache structure. We look at each level in turn (Figure 21.17).



Summary

Chapter 21

- Hardware performance issues
  - Increase in parallelism and complexity
  - Power consumption
- Software performance issues
  - Software on multicore
  - Valve game software example
- Intel Core i7-5960X
- IBM z13 mainframe
  - Organization
  - Cache structure

Multicore Computers

- Multicore organization
  - Levels of cache
  - Simultaneous multithreading
- Heterogeneous multicore organization
  - Different instruction set architectures
  - Equivalent instruction set architectures
  - Cache coherence and the MOESI model
- ARM Cortex-A15 MPCore
  - Interrupt handling
  - Cache coherency
  - L2 cache coherency