Thread Level Paralleism (Week#16)

Distributed Shared-Memory and Directory-Based Coherence

- ✓ 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.
- ✓ Within a multicore with a shared outermost cache (say, L3), it is easy to implement a directory scheme:
- ✓ simply keep a bit vector of size equal to the number of cores for each L3 block.
- ✓ Bit vector indicates which private L2 caches may have copies of a block in L3 and invalidations are sent only to those caches.
- ✓ works perfectly for a single multicore and this scheme is used in Intel i7.
- ✓ Solution of a single directory used in a multicore is not scalable, even though it avoids broadcast.
- ✓ Directory must be distributed in a way that coherence protocol knows where to find directory information for any cached block of memory.
- ✓ Solution is to distribute directory along with memory so that different coherence requests can go to different directories.
- ✓ If information is maintained at an outer cache L3,directory information can be distributed with different cache banks, effectively increasing bandwidth.

Distributed Shared-Memory and Directory-Based Coherence

- ✓ Figure 5.18 shows how our distributed-memory multiprocessor looks with directories added to each node.

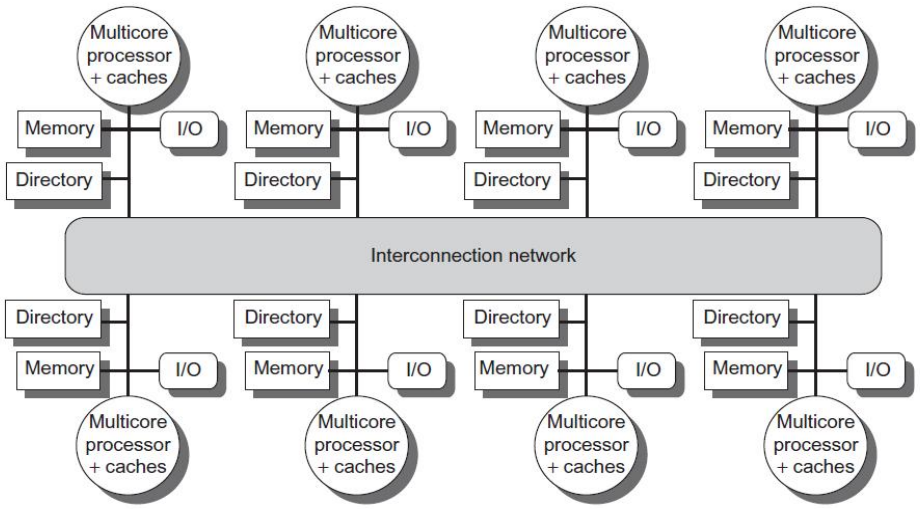


Figure 5.18 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. In this case, a node is shown as a single multicore chip, and the directory information for the associated memory may reside either on or off the multicore. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The coherence mechanism will handle both the maintenance of the directory information and any coherence actions needed within the multicore node.

Directory-Based Cache Coherence Protocols: The Basics

- ✓ In a simple protocol, these states could be following:
- ✓ **Shared:** One or more nodes have the block cached, and value in memory is up to date.
- ✓ **Uncached:** No node has a copy of the cache block.
- ✓ **Modified:** Exactly one node has a copy of the cache block, and it has written the block, so the memory copy is out of date. The processor is called the owner of the block.
- ✓ Figure 5.19 shows types of messages sent among nodes.
- ✓ **Local node:** is the node where a request originates.
- ✓ **Home node:** is the node where the memory location and the directory entry of an address reside.
- ✓ For example, the high-order bits may provide the node number, whereas the low-order bits provide the offset within the memory on that node.
- ✓ **Remote node:** is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared.

Directory-Based Cache Coherence Protocols: The Basics

Figure 5.19 shows types of messages sent among nodes.

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/ invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write back a data value for address A.

Figure 5.19 The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting node number, A = requested address, and D = data contents), and the function of the message. The first three messages are requests sent by the local node to the home. The fourth through sixth messages are messages sent to a remote node by the home when the home needs the data to satisfy a read or write miss request. Data value replies are used to send a value from the home node back to the requesting node. Data value write-backs occur for two reasons: when a block is replaced in a cache and must be written back to its home memory, and also in reply to fetch or fetch/invalidate messages from the home. Writing back the data value whenever the block becomes shared simplifies the number of states in the protocol because any dirty block must be exclusive and any shared block is always available in the home memory.

An Example Directory Protocol

Figure 5.20 shows protocol actions to which an individual cache responds. The state transitions for an individual cache are caused by read misses, write misses, invalidates, and data fetch requests.

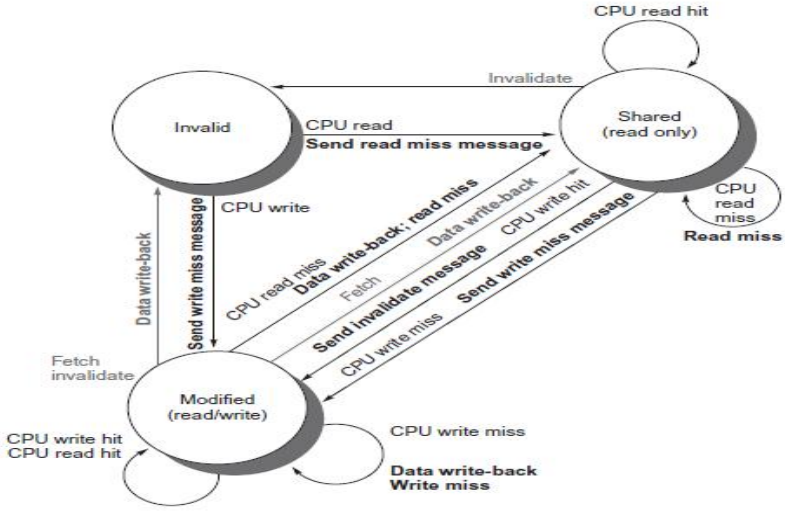


Figure 5.20 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in *black*, and those from the home directory are shown in *gray*. The states are identical to those in the snooping case, and the transactions are very similar, with explicit invalidate and write-back requests replacing the write misses that were formerly broadcast on the bus. As we did for the snooping controller, we assume that an attempt to write a shared cache block is treated as a miss; in practice, such a transaction can be treated as an ownership request or upgrade request and can deliver ownership without requiring that the cache block be fetched.

An Example Directory Protocol

- ✓ A message sent to a directory causes two different types of actions: updating the directory state and sending additional messages to satisfy the request.
- ✓ Directory state indicates the state of all the cached copies of a memory block, rather than for a single cache block.
- ✓ Memory block may be uncached by any node, cached in multiple nodes and readable (shared) or cached exclusively and writable in exactly one node.
- ✓ Directory must track the set of nodes that have a copy of a block; use a set called Sharers to perform this function.
- ✓ In multiprocessors with fewer than 64 nodes, directory requests need to update set Sharers and read the set to perform invalidations.

An Example Directory Protocol

Figure 5.21 shows actions taken at the directory in response to messages received.

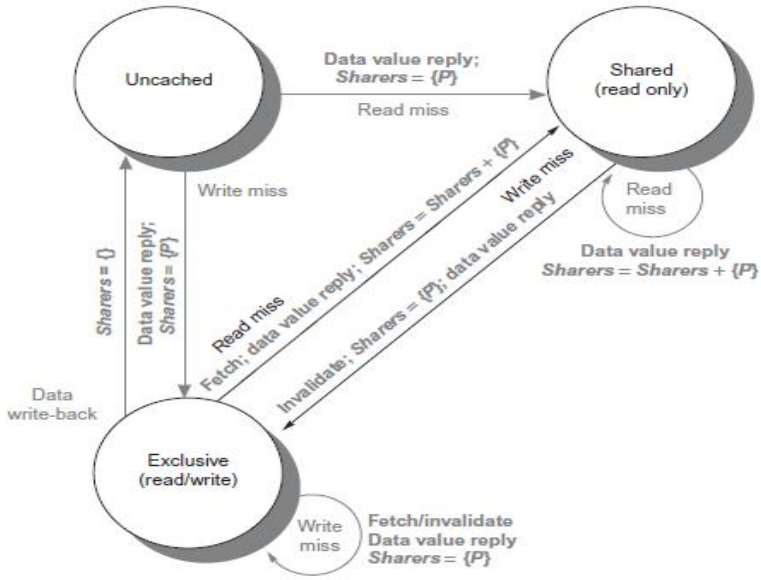


Figure 5.21 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. *Bold* indicates the action taken by the directory in response to the request.

An Example Directory Protocol

When a block is in the uncached state, the copy in memory is the current value, so the only possible requests for that block are:

- ✓ **Read miss:** requesting node is sent requested data from memory and requester is made the only sharing node. State of the block is made shared.
- ✓ **Write miss:** requesting node is sent the value and becomes the sharing node. Block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state, the memory value is up to date, so the same two requests can occur:

- ✓ **Read miss:** requesting node is sent the requested data from memory and requesting node is added to sharing set.
- ✓ **Write miss:** requesting node is sent the value. All nodes in the set Sharers are sent invalidate messages and Sharers set is to contain the identity of requesting node.

An Example Directory Protocol

When block is in exclusive state, current value of block is held in a cache on node identified by set Sharers (the owner), so there are three possible directory requests:

- ✓ **Read miss:** Owner is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes owner to send data to directory, where it is written to memory and sent back to requesting processor. Requesting node is added to the set Sharers, which contains the identity of processor that was owner.
- ✓ **Data write-back:** Owner is replacing the block and must write it back. This write-back makes the memory copy up to date,block is now uncached and Sharers set is empty.
- ✓ **Write miss:** Block has a new owner. A message is sent to old owner causing cache to invalidate the block and send value to directory. it is sent to requesting node, which becomes new owner. Sharers is set to identity of new owner and state of block remains exclusive.

Synchronization: The Basics

- ✓ 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 can be used straightforwardly to create mutual exclusion, as well as to implement more complex synchronization mechanisms.

Synchronization: The Basics

Basic Hardware Primitives:

- ✓ Synchronization operations is atomic exchange, which interchanges a value in a register for a value in memory.
- ✓ Assume 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.
- ✓ For example, consider two processors that each try to do the exchange simultaneously: This race is broken because exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange.
- ✓ Using exchange (or swap) primitive to implement synchronization is that the operation is atomic: the exchange is indivisible, and two simultaneous exchanges will be ordered by the write serialization mechanisms.
- ✓ **test-and-set operation:** which tests a value and sets it if the value passes the test. For example,an operation that tested for 0 and set the value to 1.
- ✓ Atomic synchronization primitive is fetch-and-increment: it returns the value of a memory location and atomically increments it.

Synchronization: The Basics

- ✓ Implementing a single atomic memory operation introduces some challenges because it requires both a memory read and a write in a single, uninterruptible instruction.
- ✓ This requirement complicates implementation of coherence because hardware cannot allow any other operations between read and write.
- ✓ Pair of instructions is atomic if it appears as all other operations executed by any processor occurred before or after the pair.
- ✓ when an instruction pair is atomic, no other processor can change the value between instruction pair. Used in MIPS processors and in RISC V.
- ✓ In RISC V, pair of instructions includes load called a load reserved and a store called a store conditional.

These instructions are used in sequence, and because the load reserved returns the initial value and the store conditional returns 0 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of x1 with the value in x4:

```
try:    mov    x3,x4       ;mov exchange value
        lr     x2,x1       ;load reserved from
        sc     x3,0(x1)    ;store conditional
        bnez   x3,try      ;branch store fails
        mov    x4,x2       ;put load value in x4
```

At the end of this sequence, the contents of x4 and the memory location specified by x1 have been atomically exchanged. Anytime a processor intervenes and modifies the value in memory between the lr and sc instructions, the sc returns 0 in x3, causing the code sequence to try again.

Synchronization: The Basics

- ✓ An advantage of load reserved/store conditional mechanism is that it can be used to build other synchronization primitives. For example,an atomic fetch-and-increment:

  try: lr x2,x1 ;load reserved 0(x1)

  addi x3,x2,1 ;increment

  sc x3,0(x1) ;store conditional

  bnez x3,try ;branch store fails

- ✓ These instructions are implemented by keeping track of address specified in the lr instruction in a register called reserved register.
- ✓ If an interrupt occurs or if cache block matching the address in the link register is invalidated, link register is cleared.
- ✓ sc instruction checks that its address matches in reserved register. If so, the sc succeeds; otherwise fails.
- ✓ store conditional will fail after either another attempted store to load reserved address or any exception are inserted between the two instructions.
- ✓ only register-register instructions can be permitted;
- ✓ it is possible to create deadlock situations,

Synchronization: The Basics

Implementing Locks Using Coherence

- ✓ **Spin locks:** locks that a processor continuously tries to acquire, spinning around a loop until it succeeds.Spin locks are used to be held for a very short amount of time
- ✓ if there were no cache coherence, would be to keep the lock variables in memory.
- ✓ Processor continually try to acquire the lock using an atomic operation, say, atomic exchange, and test whether the exchange returned the lock as free.
- ✓ To release the lock, processor stores the value 0 to the lock.
- ✓ code sequence to lock a spin lock whose address is in x1. It uses EXCH as a macro for the atomic exchange sequence :

```
addi x2,R0,#1
lockit: EXCH x2,0(x1) ;atomic exchange
bnez x2,lockit ;already locked?
```

- ✓ If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently.

Synchronization: The Basics

- ✓ Caching locks has two advantages.
  - i. it allows an implementation where process of “spinning” could be done on a local cached copy rather requiring a global memory access on each attempt to acquire the lock.
  - ii. processor that used the lock last will use it again in the near future. lock value may reside in the cache of processor, greatly reducing the time to acquire the lock.
- ✓ Spin lock procedure spins by doing reads on a local copy of the lock until it successfully sees that the lock is available.
- ✓ it attempts to acquire the lock by doing a swap operation.
- ✓ A processor keeps reading and testing until the value of the read indicates that the lock is unlocked.
- ✓ All processes use a swap instruction that reads the old value and stores a 1 into the lock variable.
- ✓ single winner will see the 0, and the losers will see a 1 that was placed there by the winner.

Synchronization: The Basics

- ✓ winning processor executes the code after the lock and when finished, stores a 0 into the lock variable to release the lock.
- ✓ code to perform this spin lock (remember that 0 is unlocked and 1 is locked):

```
lockit: ld x2,0(x1) ;load of lock
bnez x2,lockit ;not available-spin
addi x2,R0,#1 ;load locked value
EXCH x2,0(x1) ;swap
bnez x2,lockit ;branch if lock wasn't 0
```

Synchronization: The Basics

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied.
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied.
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Figure 5.22 Cache coherence steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write invalidate coherence. P0 starts with the lock (step 1), and the value of the lock is 1 (i.e., locked); it is initially exclusive and owned by P0 before step 1 begins. P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3–5). P2 wins and enters the critical section (steps 6 and 7), while P1's attempt fails, so it starts spin waiting (steps 7 and 8). In a real system, these events will take many more than 8 clock ticks because acquiring the bus and replying to misses take much longer. Once step 8 is reached, the process can repeat with P2, eventually getting exclusive access and setting the lock to 0.

Synchronization: The Basics

Code sequence , optimized version using exchange (x1 has the address of the lock, the lr has replaced the LD, and the sc has replaced the EXCH):

lockit: lr x2,0(x1) ;load reserved
bnez x2,lockit ;not available-spin
addi x2,R0,#1 ;locked value
sc x2,0(x1) ;store
bnez x2,lockit ;branch if store fails

first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously

Models of Memory Consistency: An Introduction

How consistent memory must be seems simple with an example. Here are two code segments from processes P1 and P2, shown side by side:

```
P1:     A = 0;              P2:     B = 0;
        .....                       .....
        A = 1;                      B = 1;
L1:     if (B == 0)...      L2:     if (A == 0)...
```

- ✓ processes are running on different processors and that locations A and B are cached by both processors with the initial value of 0.
- ✓ If writes take immediate effect and are immediately seen by other processors, it will be impossible for both IF statements (labeled L1 and L2) to evaluate their conditions as true, since reaching the IF statement means that either A or B must have been assigned the value 1.
- ✓ write invalidate is delayed and processor is allowed to continue during this delay.
- ✓ P1 and P2 have not seen the invalidations for B and A before they attempt to read the values.

Models of Memory Consistency: An Introduction

- ✓ Sequential consistency requires result of any execution be the same as though the memory accesses executed by each processor were kept in order and the accesses were arbitrarily interleaved.
- ✓ To implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed.
- ✓ Effective to delay next memory access until the previous one is completed.
- ✓ Memory consistency involves operations among different variables: two accesses that must be ordered are actually to different memory locations

Models of Memory Consistency: An Introduction

Example Suppose we have a processor where a write miss takes 50 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 80 cycles for an invalidate to complete and be acknowledged once it is issued. Assuming that four other processors share a cache block, how long does a write miss stall the writing processor if the processor is sequentially consistent? Assume that the invalidates must be explicitly acknowledged before the coherence controller knows they are completed. Suppose we could continue executing after obtaining ownership for the write miss without waiting for the invalidates; how long would the write take?

Answer When we wait for invalidates, each write takes the sum of the ownership time plus the time to complete the invalidates. Because the invalidates can overlap, we need only worry about the last one, which starts $10+10+10+10=40$ cycles after ownership is established. Therefore the total time for the write is $50+40+80=170$ cycles. In comparison, the ownership time is only 50 cycles. With appropriate write buffer implementations, it is even possible to continue before ownership is established.

Models of Memory Consistency: An Introduction

Programmer's View:

- ✓ Program is synchronized if all accesses to shared data are ordered by synchronization operations.
- ✓ variables may be updated without ordering by synchronization are called data races because the execution outcome depends on the relative speed of the processors
- ✓ Programmers could attempt to guarantee ordering by constructing their own synchronization mechanisms.
- ✓ Programmers choose to use synchronization libraries that are correct and optimized for the multiprocessor and the type of synchronization.
- ✓ Use of standard synchronization primitives ensures that even if the architecture implements a more relaxed consistency model than sequential consistency.

Models of Memory Consistency: An Introduction

Relaxed Consistency Models: Basics and Release Consistency

Relaxed consistency models allow reads and writes to complete out of order, but use synchronization operations to enforce ordering so that a synchronized program behaves as processor were sequentially consistent.

Sequential consistency requires maintaining all four possible orderings: R--->W, R--->R,W---R, andW --->W. The relaxed models are defined relax:

1. Relaxing only theW--->R ordering yields a model known as total store ordering or processor consistency. Retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization.

2. Relaxing both the W-->R ordering and the W--->W ordering yields a model known as partial store order.

3. Relaxing all four orderings yields a variety of models including weak ordering, PowerPC consistency model and release consistency, RISC V consistency model.

By relaxing these orderings, processor may obtain significant performance advantages, which is the reason RISC V, ARMv8, C++ and C language standards chose release consistency model.

Models of Memory Consistency: An Introduction

- ✓ Release consistency distinguishes between synchronization operations:
- ✓ used to acquire access to a shared variable by SA)
- ✓ release an object to allow another processor to acquire access by SR.

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Partial store order	SPARC	$R \rightarrow R$, $R \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Weak ordering	PowerPC		$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Release consistency	MIPS, RISC V, Armv8, C, and C++ specifications		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_R$, $W \rightarrow S_R$, $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$

Figure 5.23 The orderings imposed by various consistency models are shown for both ordinary accesses and synchronization accesses. The models grow from most restrictive (sequential consistency) to least restrictive (release consistency), allowing increased flexibility in the implementation. The weaker models rely on fences created by synchronization operations, as opposed to an implicit fence at every memory operation. S_A and S_R stand for acquire and release operations, respectively, and are needed to define release consistency. If we were to use the notation S_A and S_R for each S consistently, each ordering with one S would become two orderings (e.g., $S \rightarrow W$ becomes $S_A \rightarrow W$, $S_R \rightarrow W$), and each $S \rightarrow S$ would become the four orderings shown in the last line of the bottom-right table entry.

Models of Memory Consistency: An Introduction

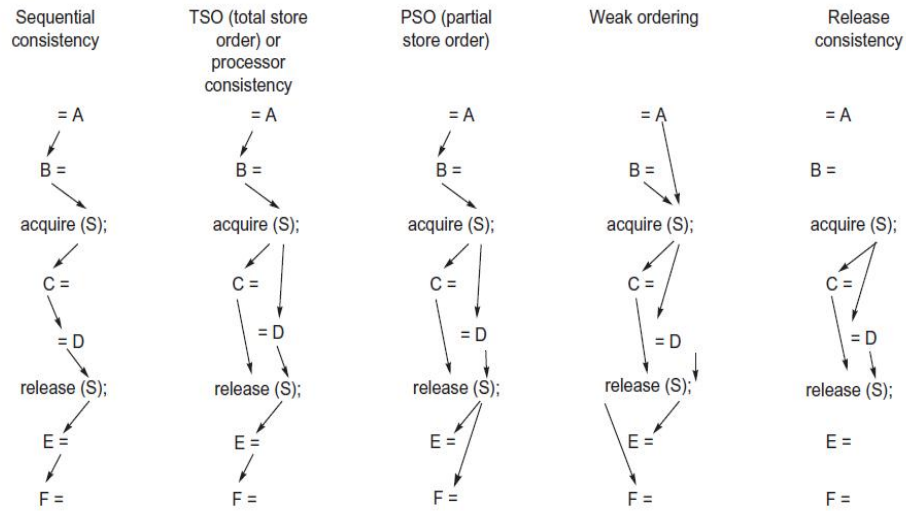


Figure 5.24 These examples of the five consistency models discussed in this section show the reduction in the number of orders imposed as the models become more relaxed. Only the minimum orders are shown with arrows. Orders implied by transitivity, such as the write of C before the release of S in the sequential consistency model or the acquire before the release in weak ordering or release consistency, are not shown.

Cross-Cutting Issues

Compiler Optimization and the Consistency Model

- ✓ Model for memory consistency is to specify range of legal compiler optimizations that can be performed on shared data.
- ✓ In parallel programs, compiler cannot interchange a read and a write of two different shared data items because such transformations affect the semantics of program.
- ✓ This restriction prevents even relatively simple optimizations, such as register allocation of shared data, because such a process usually interchanges reads and writes.
- ✓ In parallelized programs written in High Performance Fortran, programs must be synchronized and the synchronization points are known.
- ✓ Compilers can get advantage from more relaxed consistency models remains an open question.

Cross-Cutting Issues

Using Speculation to Hide Latency in Strict Consistency Models

- ✓ Speculation can also be used to hide latency arising from a strict consistency model, giving benefit of a relaxed memory model.
- ✓ Pocessor to use dynamic scheduling to reorder memory references which generate violations of sequential consistency. This possibility is avoided by using delayed commit feature of a speculative processor.
- ✓ If the processor receives an invalidation for a memory reference before the memory reference is committed, processor uses speculation recovery to back out of the computation and restart with memory reference whose address was invalidated.
- ✓ If the reordering of memory requests by the processor yields an execution order that could result in an outcome that differs from what would have been seen undersequential consistency, the processor will redo the execution.
- ✓ One open question is how successful compiler technology will be in optimizing memory references to shared variables. The state of optimization technology and the fact that shared data are often accessed via pointers or array indexing have limited the use of such optimizations. **j**

Cross-Cutting Issues

Inclusion and Its Implementation

- ✓ All multiprocessors use multilevel cache hierarchies to reduce demand on global interconnect and latency of cache misses.
- ✓ Many multiprocessors with multilevel caches enforce the inclusion property, although recent multiprocessors with smaller L1 caches and different block sizes have sometimes chosen not to enforce inclusion.
- ✓ This restriction is also called the subset property because each cache is a subset of the cache below it in the hierarchy.
- ✓ Consider a two-level example: Any miss in L1 either hits in L2 or generates a miss in L2, causing it to be brought into both L1 and L2.
- ✓ any invalidate that hits in L2 must be sent to L1, where it will cause the block to be invalidated if it exists.
- ✓ when the block sizes of L1 and L2 are different. Choosing different block sizes is quite reasonable, since L2 will be much larger and have a much longer latency component in its miss penalty, and will use a larger block size.

Cross-Cutting Issues

Performance Gains From Multiprocessing and Multithreading

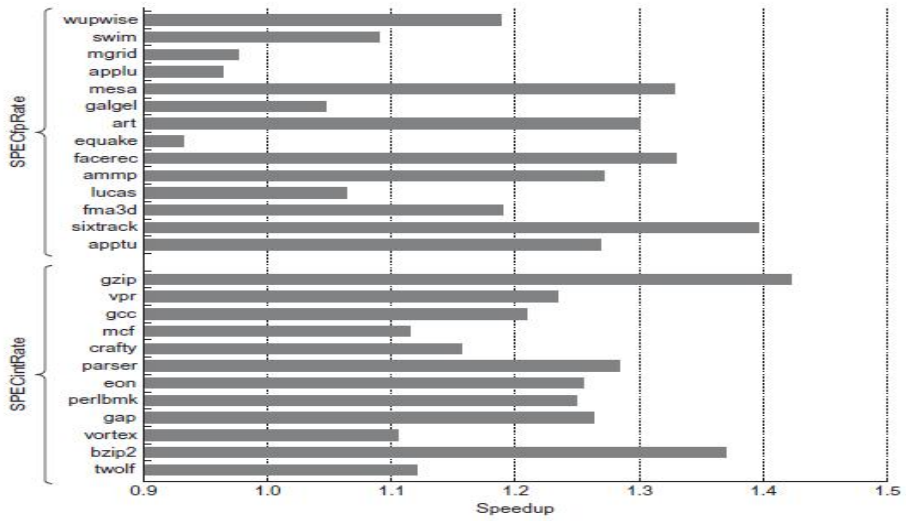


Figure 5.25 A comparison of SMT and single-thread (ST) performance on the 8-processor IBM eServer p5 575 using SPECfpRate (top half) and SPECintRate (bottom half) as benchmarks. Note that the *x*-axis starts at a speedup of 0.9, a performance loss. Only one processor in each Power5 core is active, which should slightly improve the results from SMT by decreasing destructive interference in the memory system. The SMT results are obtained by creating 16 user threads, whereas the ST results use only eight threads; with only one thread per processor, the Power5 is switched to single-threaded mode by the OS. These results were collected by John McCalpin at IBM. As we can see from the data, the standard deviation of the results for the SPECfpRate is higher than for SPECintRate (0.13 versus 0.07), indicating that the SMT improvement for FP programs is likely to vary widely.

Putting It All Together: Multicore Processors and Their Performance

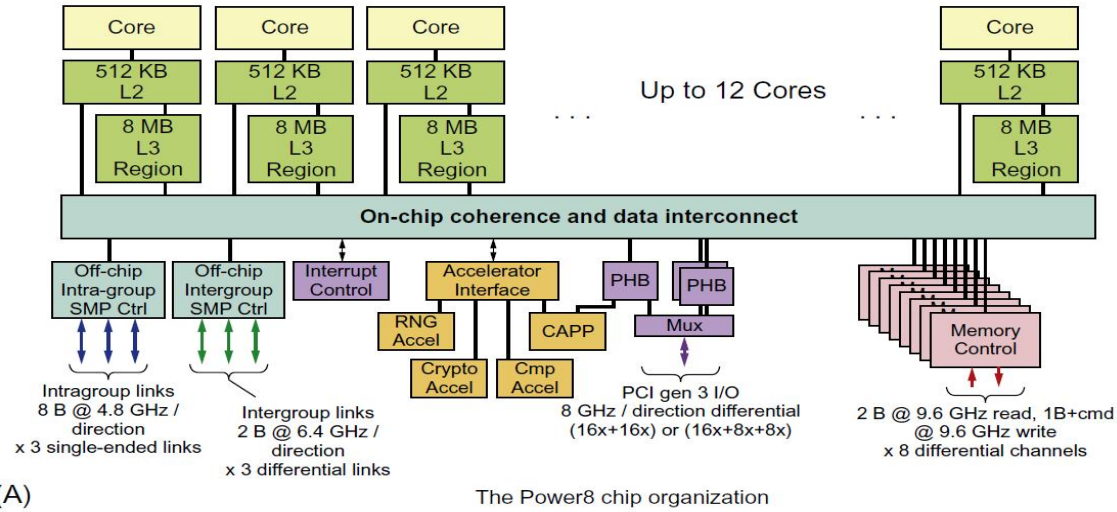
Performance of Multicore-Based Multiprocessors on a Multiprogrammed Workload:

Feature	IBM Power8	Intel Xeon E7	Fujitsu SPARC64 X+
Cores/chip	4, 6, 8, 10, 12	4, 8, 10, 12, 22, 24	16
Multithreading	SMT	SMT	SMT
Threads/core	8	2	2
Clock rate	3.1–3.8 GHz	2.1–3.2 GHz	3.5 GHz
L1 I cache	32 KB per core	32 KB per core	64 KB per core
L1 D cache	64 KB per core	32 KB per core	64 KB per core
L2 cache	512 KB per core	256 KB per core	24 MiB shared
L3 cache	L3: 32–96 MiB: 8 MiB per core (using eDRAM); shared with nonuniform access time	10–60 MiB @ 2.5 MiB per core; shared, with larger core counts	None
Inclusion	Yes, L3 superset	Yes, L3 superset	Yes
Multicore coherence protocol	Extended MESI with behavioral and locality hints (13-states)	MESIF: an extended form of MESI allowing direct transfers of clean blocks	MOESI
Multichip coherence implementation	Hybrid strategy with snooping and directory	Hybrid strategy with snooping and directory	Hybrid strategy with snooping and directory
Multiprocessor interconnect support	Can connect up to 16 processor chips with 1 or 2 hops to reach any processor	Up to 8 processor chips directly via Quickpath; larger system and directory support with additional logic	Crossbar interconnect chip, supports up to 64 processors; includes directory support
Processor chip range	1–16	2–32	1–64
Core count range	4–192	12–576	8–1024

Figure 5.26 Summary of the characteristics of three recent high-end multicore processors (2015–2017 releases) designed for servers. The table shows the range of processor counts, clock rates, and cache sizes within each pro-

Putting It All Together: Multicore Processors and Their Performance

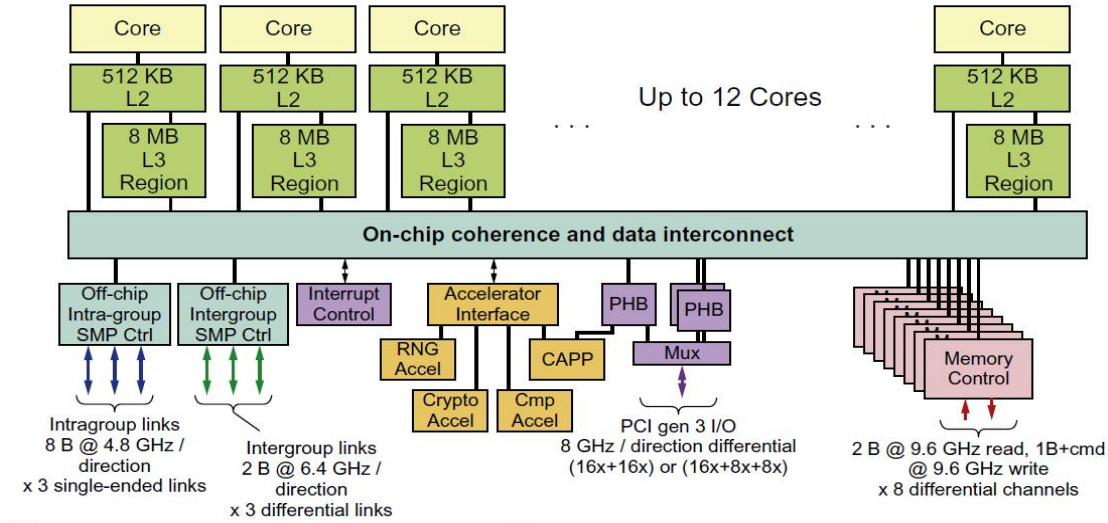
- ✓ Each core in the Power8 has an 8 MiB bank of L3 directly connected;
- ✓ other banks are accessed via the interconnection network, which has 8 separate buses.
- ✓ Power8 is NUCA (Nonuniform Cache Architecture), because access time to the attached bank of L3 will be much faster than accessing another L3.



(A) The Power8 chip organization

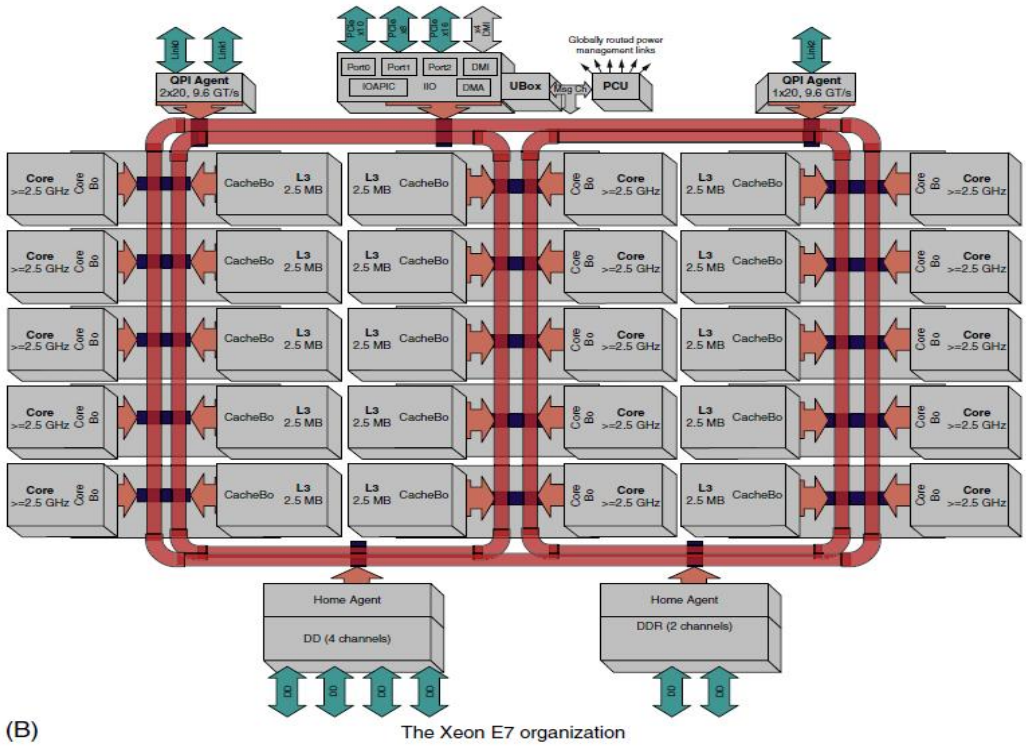
Putting It All Together: Multicore Processors and Their Performance

- ✓ Each Power8 chip has a set of links that can be used to build a large multiprocessor
- ✓ Memory links are connected to a special memory controller that includes an L4 and interfaces directly with DIMMs.



(A) The Power8 chip organization

Putting It All Together: Multicore Processors and Their Performance

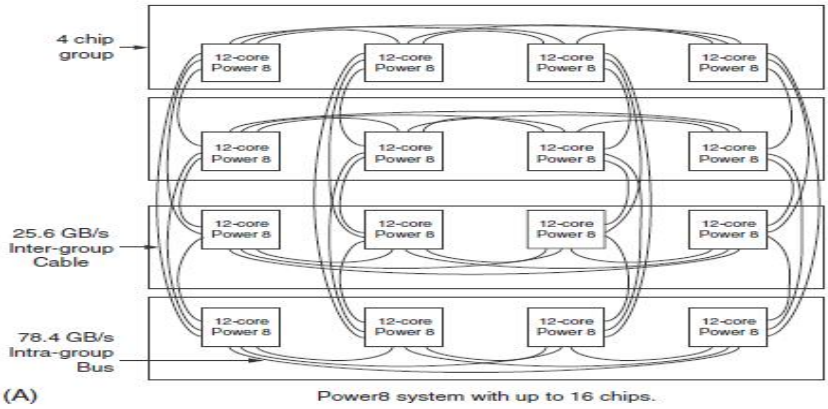


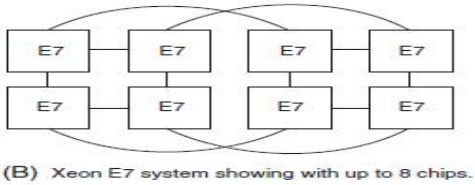
(B) The Xeon E7 organization

Putting It All Together: Multicore Processors and Their Performance

- ✓ Part B of Figure 5.27, shows Xeon E7 processor chip is organized when there are 18 or more cores.
- ✓ Three rings connect the cores and L3 cache banks and each core and each bank of L3 is connected to two rings.
- ✓ Any cache bank or any core is accessible from any other core by choosing the right ring.
- ✓ Within the chip, E7 has uniform access time.
- ✓ E7 is normally operated as a NUMA architecture by logically associating half the cores with each memory channel;
- ✓ Increases the probability that a desired memory page is open on a given access.
- ✓ E7 provides 3 QuickPath Interconnect (QPI) links for connecting multiple E7s.

Putting It All Together: Multicore Processors and Their Performance





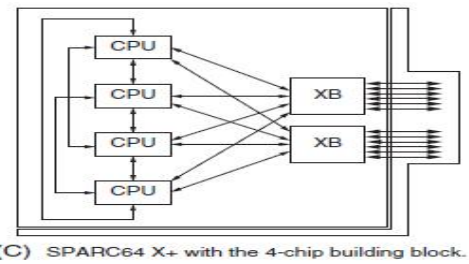


Figure 5.28 The system architecture for three multiprocessors built from multicore chips.

Putting It All Together: Multicore Processors and Their Performance

- ✓ Figure 5.28 shows 8 E7 processors can be connected; where every processor is one or two hops from every other processor.
- ✓ There are a number of Xeon-based multiprocessor servers that have more than 8 processor chips.
- ✓ To connect 4 processor chips with each processor connecting to two neighbors.
- ✓ Third QPI in each chip is connected to a crossbar switch.
- ✓ Memory accesses can occur at four locations with different timings: local to the processor, an immediate neighbor, neighbor in cluster that is two hops away and through the crossbar.
- ✓ Other organizations are possible and require less than a full crossbar in return for more hops to remote memory.
- ✓ SPARC64 X+ also uses a 4-processor module but each processor has three connections to its immediate neighbors plus two connections to a crossbar.
- ✓ 64 processor chips can be connected to two crossbar switches for a total of 1024 cores. Memory access is NUMA coherency is directory-based.

Putting It All Together: Multicore Processors and Their Performance

Performance of Multicore-Based Multiprocessors on a Multiprogrammed Workload:

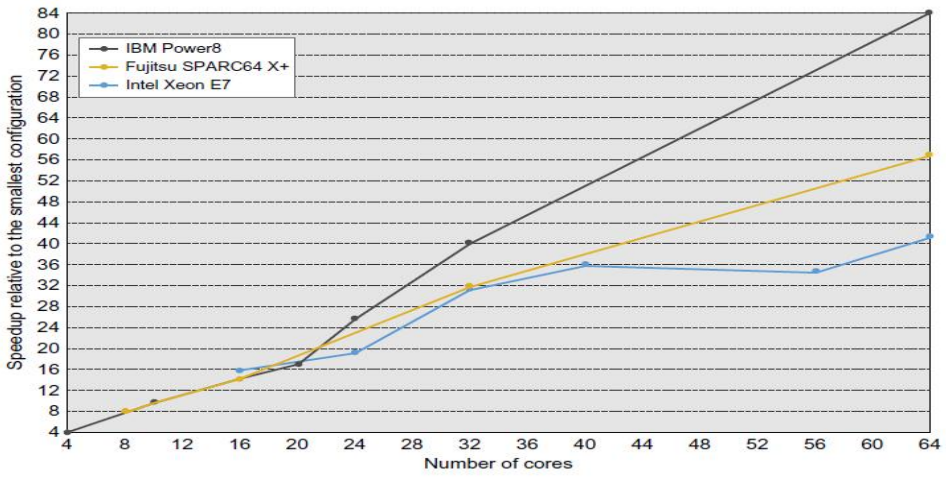


Figure 5.29 The performance scaling on the SPECintRate benchmarks for four multicore processors as the number of cores is increased to 64. Performance for each processor is plotted relative to the smallest configuration and assuming that configuration had perfect speedup. Although this chart shows how a given multiprocessor scales with additional cores, it does not supply any data about performance among processors. There are differences in the clock rates, even within a given processor family. These are generally swamped by the core scaling effects, except for the Power8 that shows a clock range spread of 1.5 × from the smallest configuration to the 64 core configuration.

Putting It All Together: Multicore Processors and Their Performance

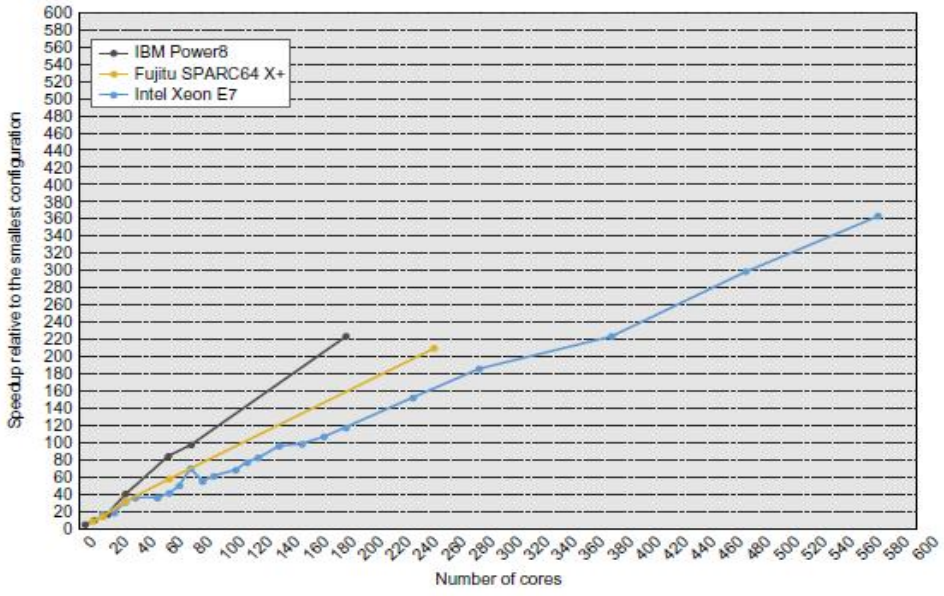


Figure 5.30 The scaling of relative performance for multiprocessor multicore. As before, performance is shown relative to the smallest available system. The Xeon result at 80 cores is the same L3 effect that showed up for smaller configurations. All systems larger than 80 cores have between 2.5 and 3.8 MiB of L3 per core, and the 80-core, or smaller, systems have 6 MiB per core.

Putting It All Together: Multicore Processors and Their Performance

Scalability in an Xeon MP With Different Workloads:

- **SPECjbb2015:** Models a supermarket IT system that handles a mix of point-of-sale requests, online purchases, and data-mining operations. Performance metric is throughput-oriented and use the maximum performance measurement on server side running multiple Java virtual machines.
- **SPECVirt2013:** Models a collection of virtual machines running independent mixes of other SPEC benchmarks, including CPU benchmarks, web servers, and mail servers. System must meet a quality of service guarantee for each virtual machine.
- **SPECOMP2012:** A collection of 14 scientific and engineering programs written with OpenMP standard for shared-memory parallel processing. Codes are written in Fortran, C, and C++ and range from fluid dynamics to molecular modeling to image manipulation

Putting It All Together: Multicore Processors and Their Performance

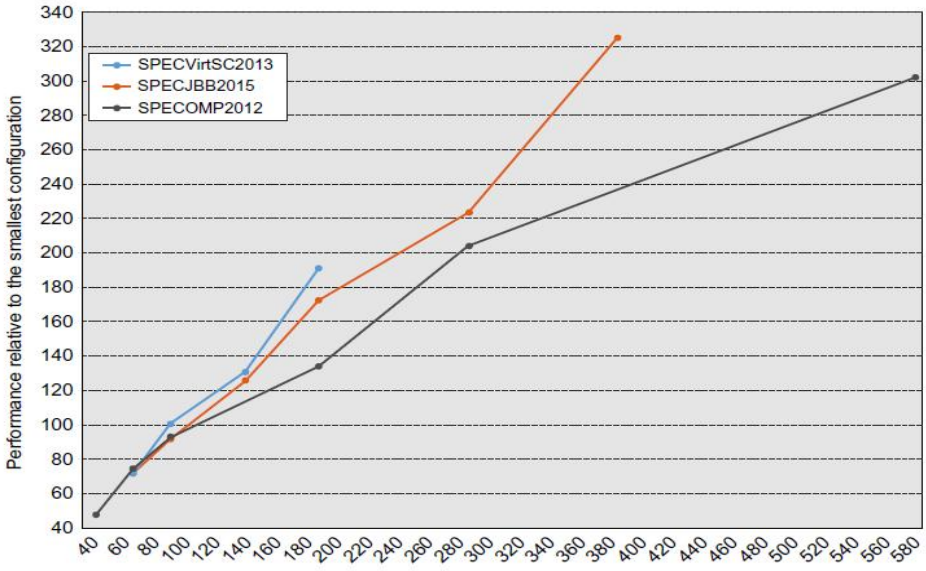


Figure 5.31 Scaling of performance on a range of Xeon E7 systems showing performance relative to the smallest benchmark configuration, and assuming that configuration gets perfect speedup (e.g., the smallest SPEWCOMP configuration is 30 cores and we assume a performance of 30 for that system). Only relative performance can be assessed from this data, and comparisons across the benchmarks have no relevance. Note the difference in the scale of the vertical and horizontal axes.

Putting It All Together: Multicore Processors and Their Performance

Performance and Energy Efficiency of the Intel i7 920 Multicore:

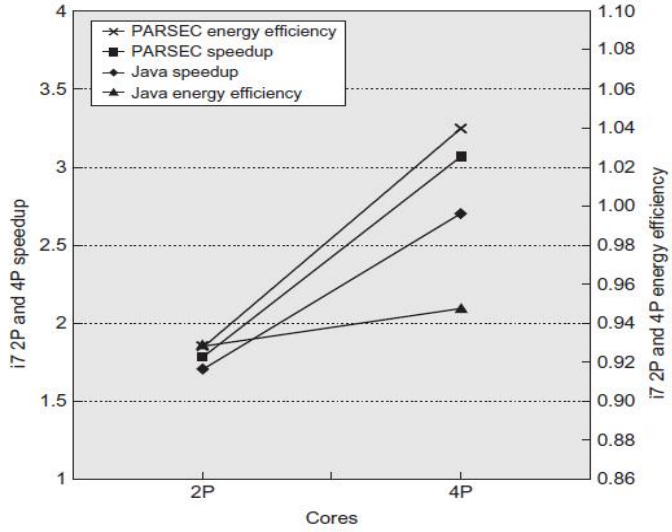


Figure 5.32 This chart shows the speedup and energy efficiency for two- and four-core executions of the parallel Java and PARSEC workloads without SMT. These data were collected by Esmaeilzadeh et al. (2011) using the same setup as described in Chapter 3. Turbo Boost is turned off. The speedup and energy efficiency are summarized using harmonic mean, implying a workload where the total time spent running each benchmark on 2 cores is equivalent.

Putting It All Together: Multicore Processors and Their Performance

Putting Multicore and SMT Together

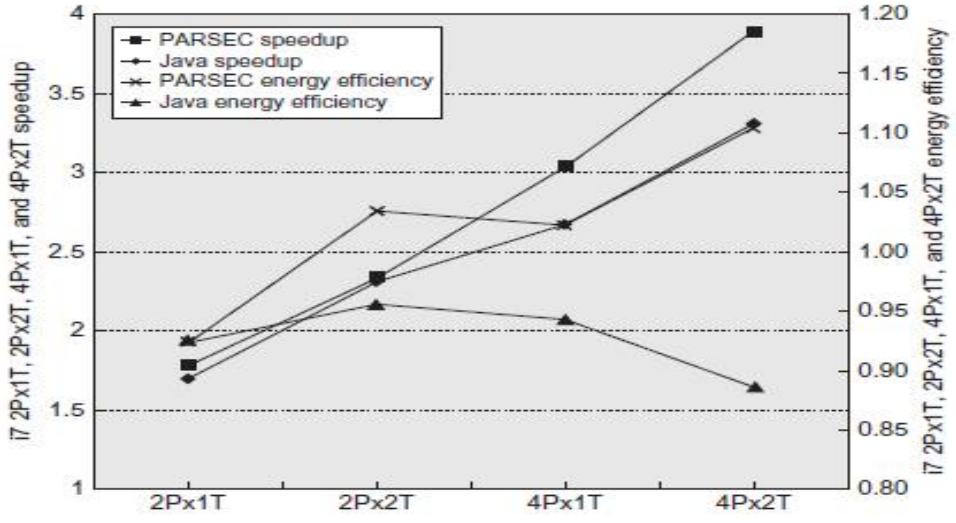


Figure 5.33 This chart shows the speedup for two- and four-core executions of the parallel Java and PARSEC workloads both with and without SMT. Remember that the preceding results vary in the number of threads from two to eight and reflect both architectural effects and application characteristics. Harmonic mean is used to summarize results, as discussed in the Figure 5.32 caption.