
CSE 431

Computer Architecture

Fall 2017

Introduction to

Multiprocessor/Multicore Systems

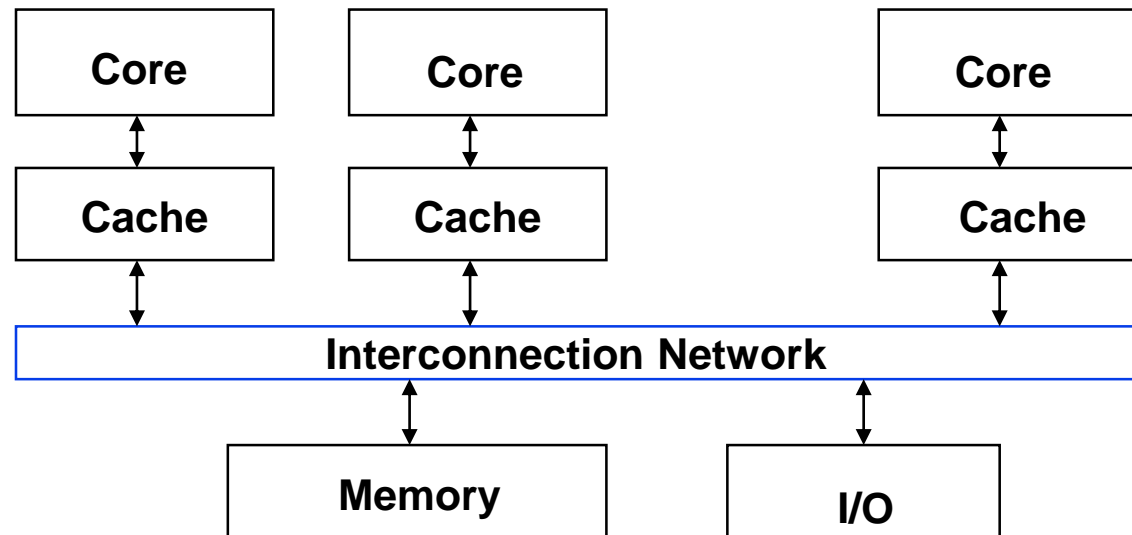
Mahmut Taylan Kandemir (www.cse.psu.edu/~kandemir)

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, MK

With additional thanks/credits to Amir Roth, Milo Martin, CIS/UPenn]

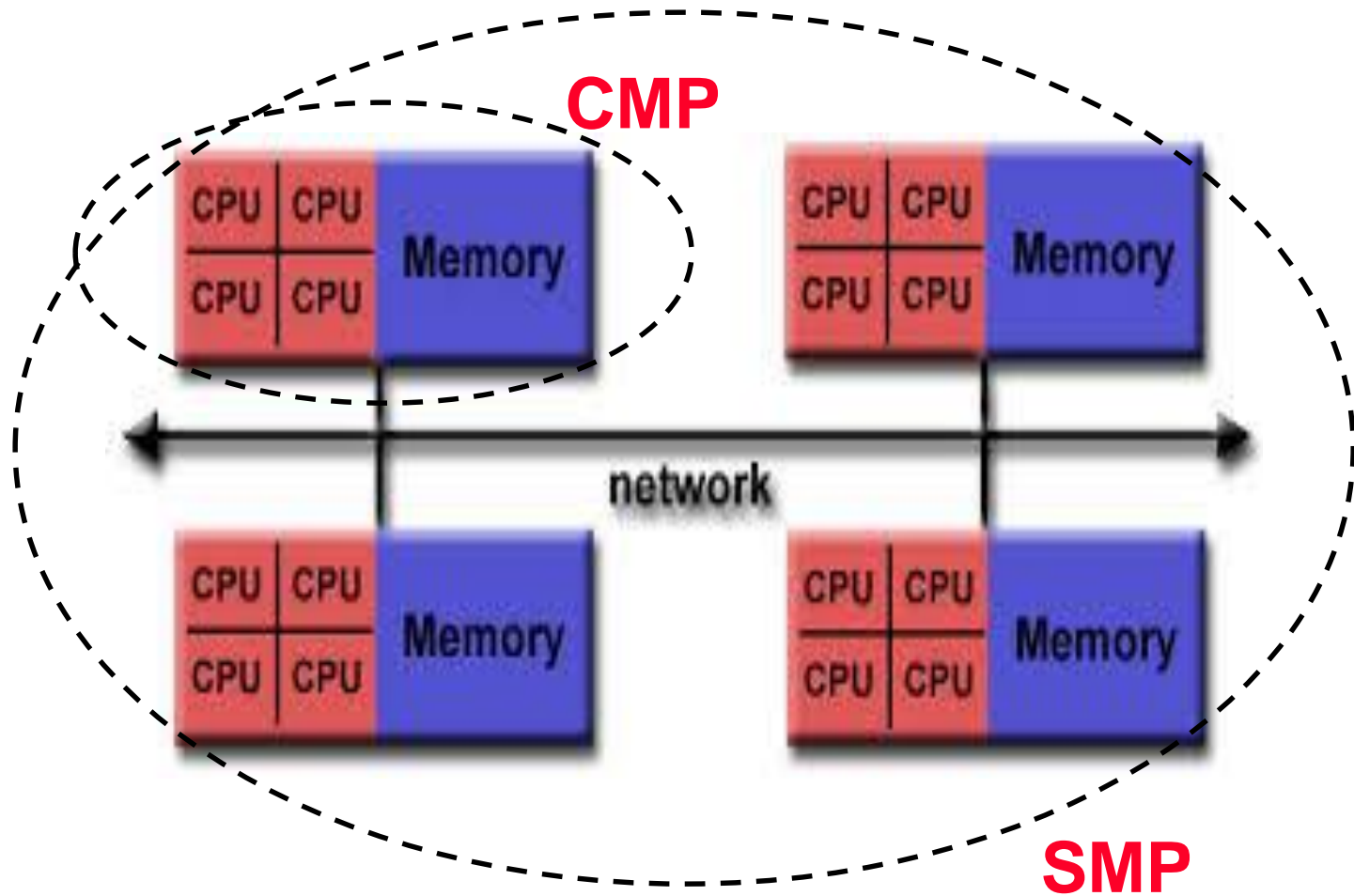
The Big Picture: Where are We Now?

- ❑ **Multiprocessor** – a computer system with at least 2 cores
- ❑ **Multicore** – a chip (processor) with at least 2 cores

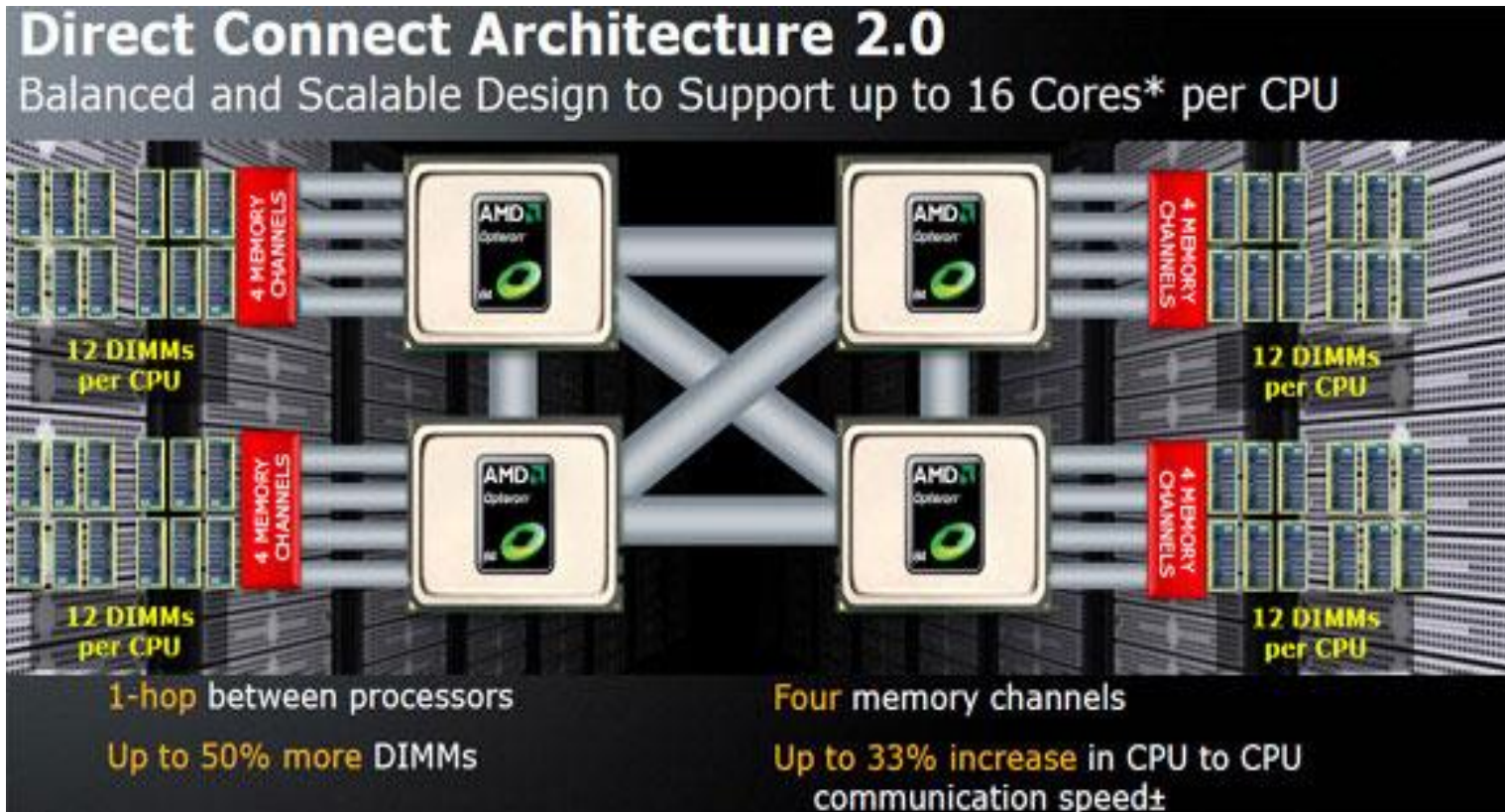


- Can deliver high throughput for multiple independent jobs – **multiprogramming** – via **task-level** or **process-level parallelism**
- Can improve the run time of a single program that has been specially crafted to run on a multiprocessor – a **multithreaded (parallel) program**

SMP vs CMP (Multicore)



AMD SMP



Multicores (aka CMPs) Now the Norm

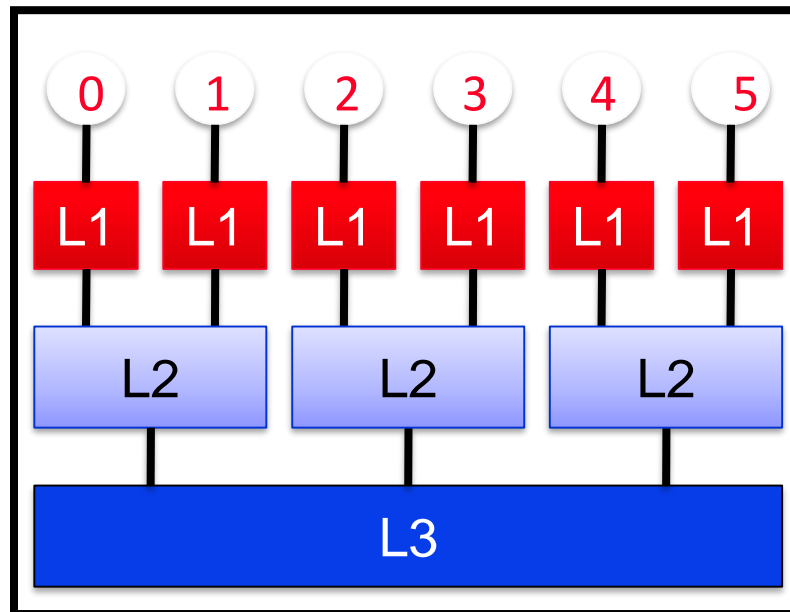
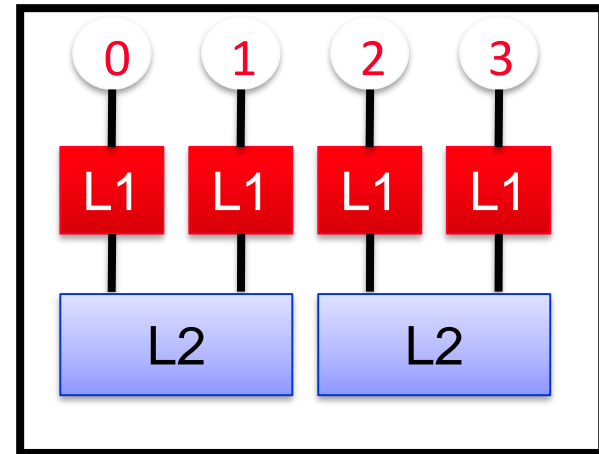
- ❑ Power constraints have stalled clock rate improvements
 - If you want better performance, have to get it by effectively using multiple cores on the same chip !
- ❑ Today's processors contain more than one core – **Chip Multicore microProcessors (CMPs)** – in a single IC (chip (chip = processor by Intel))

	AMD Opteron 6174	Intel Xeon 7500	IBM POWER7	Sun Niagara 2
Cores/chip	6	8	8	8
Clock rate	2.2 GHz	2.3GHz	4.14 GHz	1.4 GHz
Chip power	115 W	130W	~100 W?	94 W

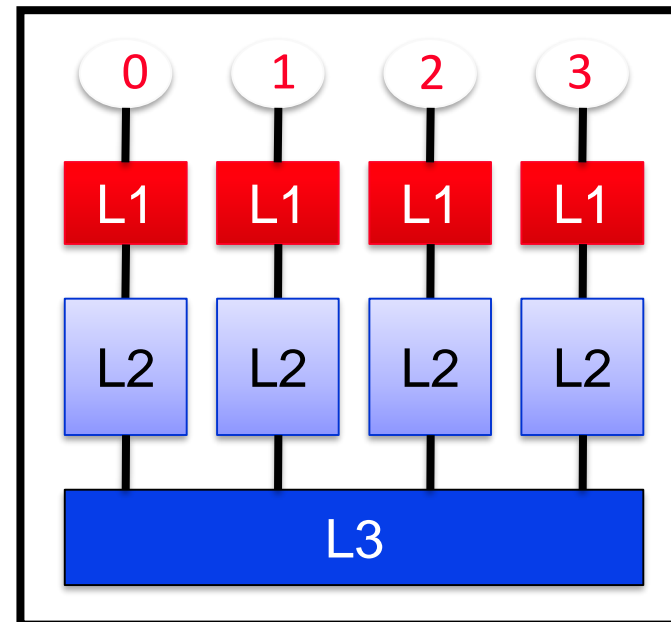
Multicore = Platform Variety

Harpertown

- ❑ Number/type of cores
- ❑ Levels and structure of the cache hierarchy
- ❑ Number of memory controllers (MCs) on-chip
- ❑ Type of on-chip interconnect

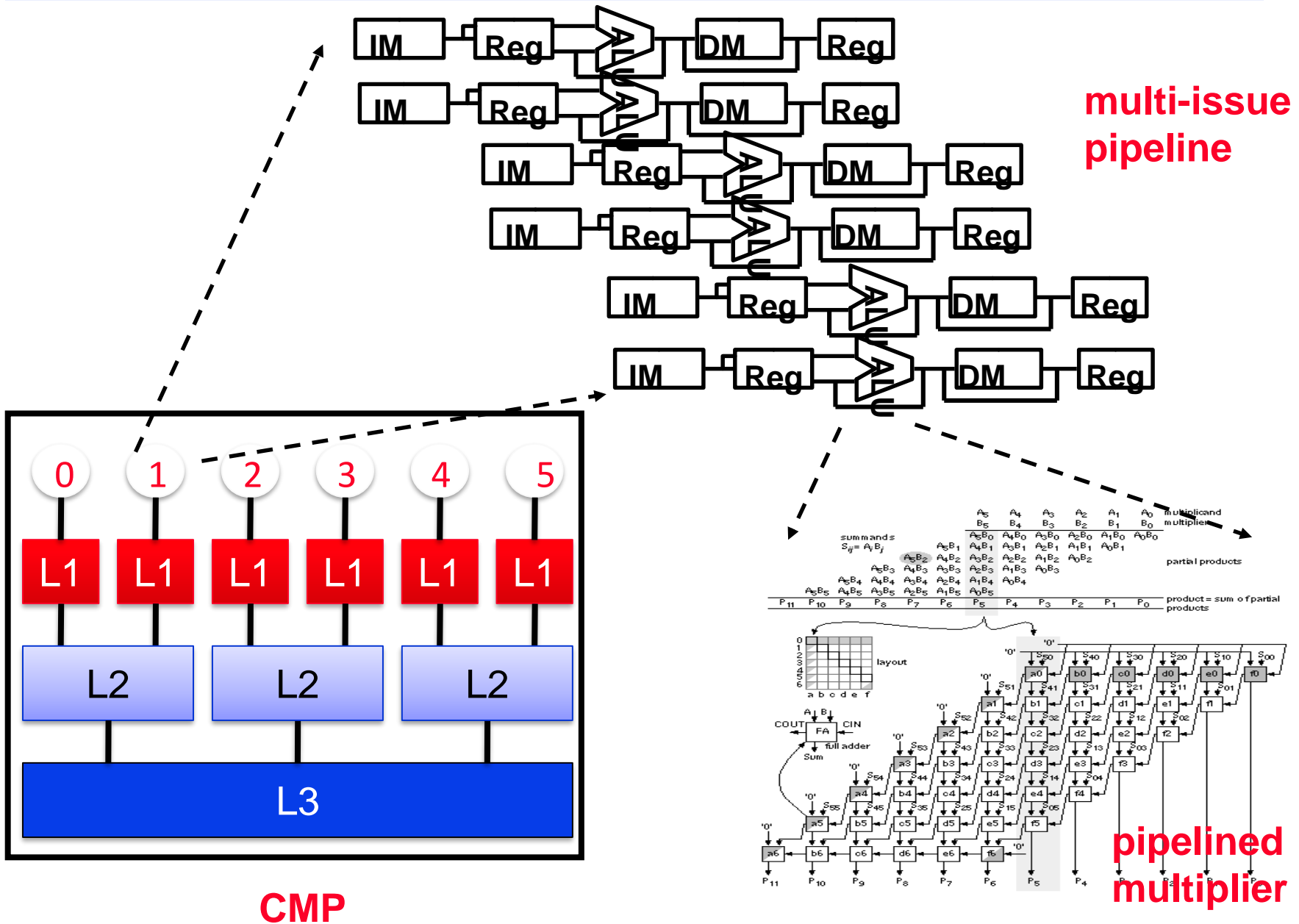


Dunnington



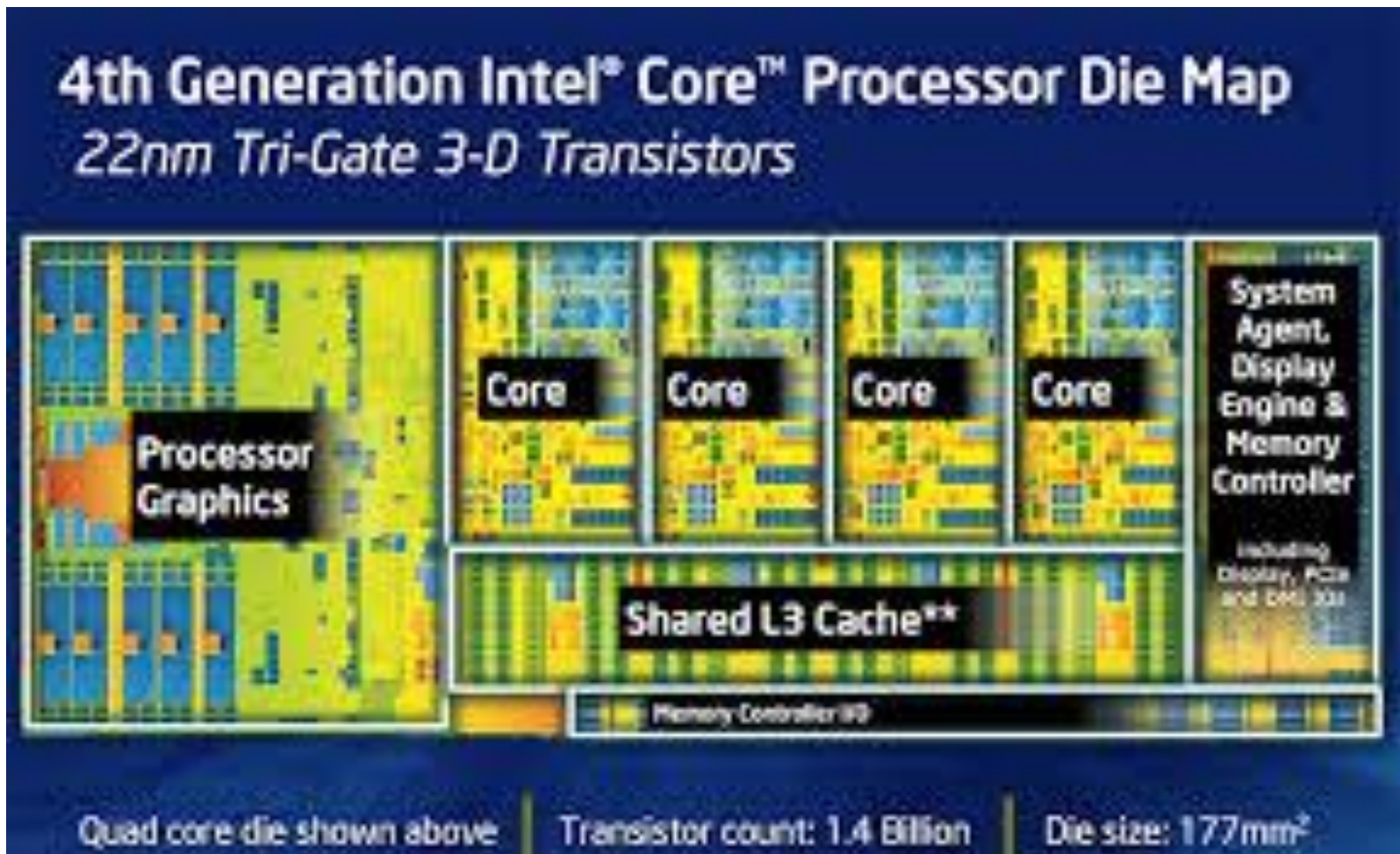
Nehalem

Hierarchical Parallelism



Intel's Haswell Multicore

- ❑ 4 cores + graphics co-processor, shared LLC, 2 MCs



Key Multiprocessor Design Questions

- ❑ Q1 – How do they share data?
- ❑ Q2 – How do they coordinate?
- ❑ Q3 – How scalable is the architecture? How many cores can be supported?

Shared Memory multiProcessor (SMP)

- ❑ Q1 – Single physical address space shared by all cores
- ❑ Q2 – Threads on cores coordinate/communicate through shared variables in memory (via loads and stores)
 1. Use of shared data must be coordinated via **synchronization** primitives (locks) that allow access to data to only one core at a time (used to implement critical sections)
 2. Caches must be kept **coherent** (read of a data item returns the most recently written value of that data item)
- ❑ SMPs come in two styles
 1. Uniform memory access (**UMA**) multiprocessors
 2. Nonuniform memory access (**NUMA**) multiprocessors
- ❑ Programming NUMAs is the harder of the two
- ❑ But NUMAs can scale to larger sizes and have lower latency to **local** memory

SMP Multithreaded Programming Model

- ❑ Creating multithreaded programs for multicores ?

- ❑ Programmer explicitly creates multiple software threads
 - Java has thread support built in, C/C++ supports P-threads library
 - Other tools such as OpenMP, Thread Blocks, Cilk Plus
 - Speedup via Thread-Level Parallelism (TLP)

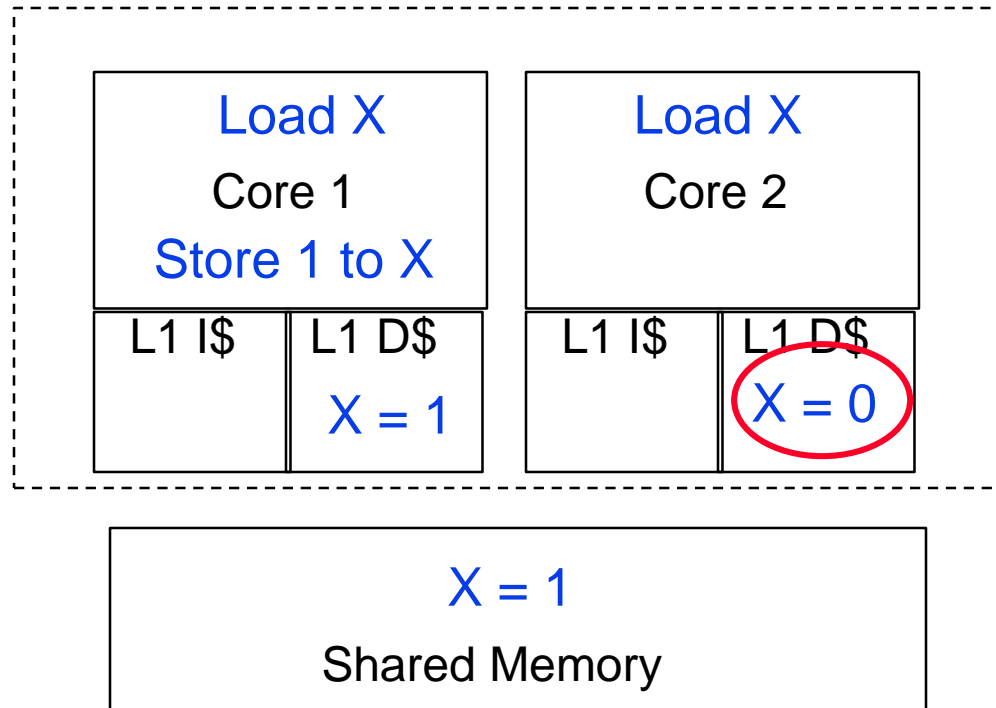
- ❑ All loads & stores are to a single **shared memory** space
 - Each thread has a private stack frame for its local variables

- ❑ A “thread switch” can occur at any time
 - SMT, FGMT
 - Pre-emptive multithreading by OS
 - Hardware timer interrupt occasionally triggers OS
 - Quickly swapping threads gives illusion of concurrent execution

Cache Coherence in Multicores

- ❑ Illustration of the **cache coherence** problem in a two core processor with private L1I\$ and L1D\$ caches and a (common) shared memory.

1. Core 1 does Load X
2. Core 2 does Load X
3. Core 1 does Store 1 to X
- ...
4. Core 1 does WB of X



A Coherent Memory System

- ❑ Any read of a data item should return the most recently written value of the data item
 - Coherence – defines **what values** can be returned by a read
 - Stores to the same location are **serialized** (two stores to the same location must be seen in the same order by all cores)
 - Consistency – determines **when** a written value will be returned by a read

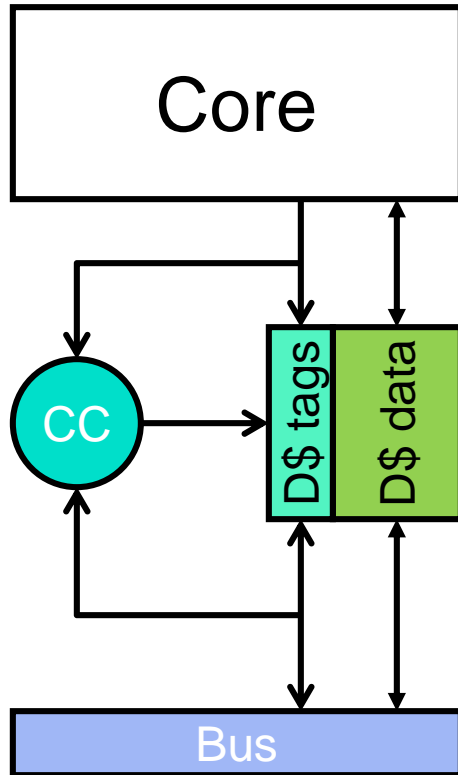
Critical for performance

- ❑ To enforce coherence, caches must provide
 - **Replication** of shared data items in multiple cores' caches
 - Replication reduces both latency and contention for a read shared data item
 - **Migration** of shared data items to a core's local cache
 - Migration reduces the latency of the access the data and the bandwidth demand on the shared memory

Cache Coherence Protocols

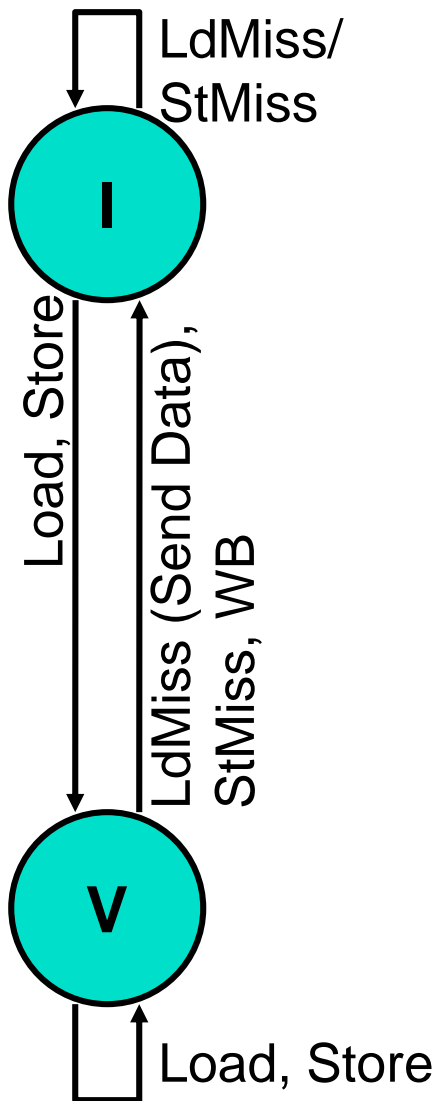
- ❑ Need a hardware mechanism to ensure cache coherence, the most popular of which is **snooping**
 - The cache controllers monitor (snoop) on the broadcast medium (e.g., bus) with duplicate address tag hardware (so they don't interfere with core's access to the cache) to determine if their cache has a copy of a block that is requested
- ❑ **Write invalidate protocol** – **writes** require exclusive access and **invalidate** *all* other existing copies
 - Exclusive access ensure that no other readable or writable copies of an item exists
- ❑ If two processors attempt to write the same data at the same time, one of them wins the race, causing the other core's copy to be invalidated. For the other core to complete, it must obtain the new data from the first core's cache – thus enforcing **write serialization**

Hardware Cache Coherence



- ❑ Write-back caches (rather than write-through) for performance reasons
 - ❑ Coherence Controller (**CC**)
 - Monitors (“snoops”) bus traffic (addresses and data)
 - Executes the **coherence protocol**
 - What to do the with local copy when you see different things happening on bus
- ❑ Three core-initiated events
 - **Ld**: load **St**: store **WB**: write-back
 - ❑ Two remote-initiated (bus) events
 - **LdMiss**: read miss from ***another*** core
 - **StMiss**: write miss from ***another*** core

VI (MI) coherence protocol



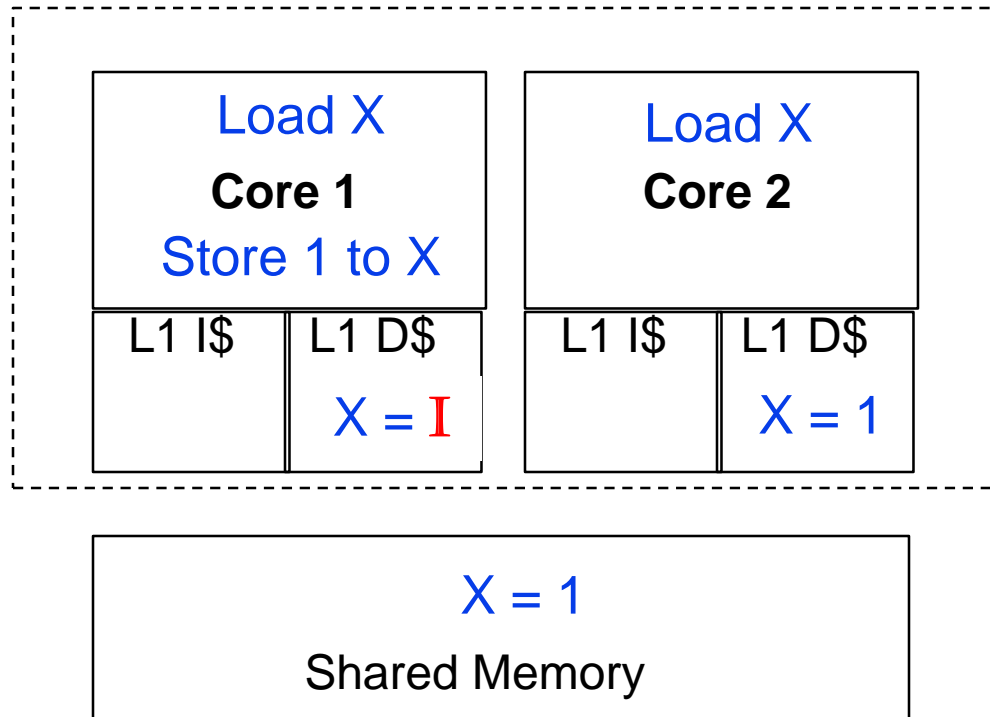
- ❑ **VI (valid-invalid) protocol**: aka MI
 - Two states (per block in cache)
 - **V (valid)**: have block
 - **I (invalid)**: don't have block
 - + Can implement with one bit (the valid bit)
- ❑ Protocol diagram (left)
 - If anyone **else** wants to read/write block
 - Give it up: transition to **I** state
 - WriteBack (WB) if your own copy is dirty
- ❑ This is an **invalidate protocol**
- ❑ **Update protocol**: copy data (write-through), don't invalidate
 - Sounds good, but wastes a lot of bus bandwidth

VI Protocol State Transition Table

State	<i>From This Core</i>		<i>From Other Cores</i>	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → V	Miss → V	---	---
Valid (V)	Hit	Hit	Send Data → I	→ I

- ❑ Rows are “states” (**I** vs **V**) of data blocks in the caches
- ❑ Columns are “events” in the cores
 - WriteBack events not shown
 - **V** → **I** on a LdMiss also updates memory (Send Data) **if** block is Dirty
- ❑ Memory sends data when no core responds

Example of VI Snooping Protocol



LdMiss seen by Core 1, so Core 1 invalidates its copy

StMiss seen by Core 2, so Core 2 invalidates its copy

LdMiss seen by Core 1, so Core 1 does a Send Data and then invalidates its copy

- ❑ When the second miss by Core 2 occurs, Core 1 responds with the data value for Core 2 (canceling the response from the shared memory), updating the value of X in the shared memory, and invalidating its own copy

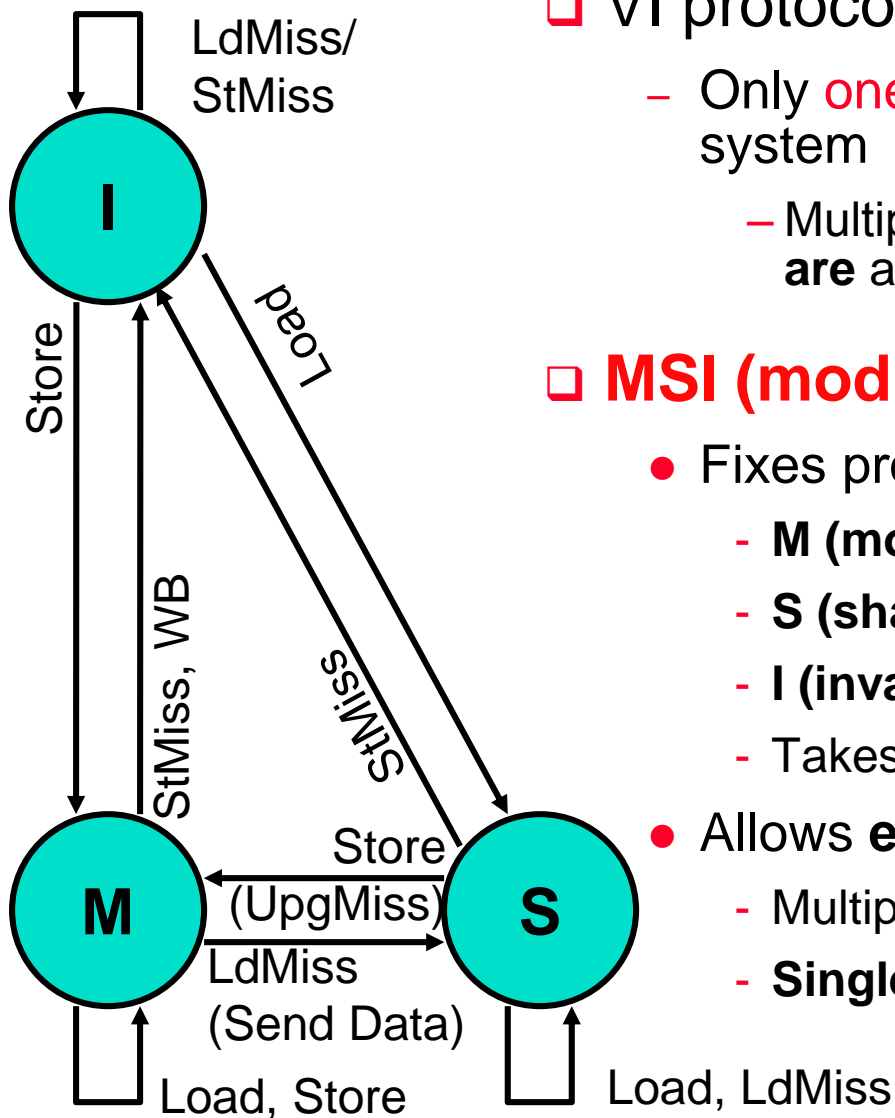
VI → MSI

❑ VI protocol is inefficient

- Only **one** cached copy allowed in entire system
- Multiple caches copies can't exist even if they **are** all **read-only**

❑ MSI (modified-shared-invalid)

- Fixes problem: splits “V” state into two states
 - **M (modified)**: local dirty copy
 - **S (shared)**: local clean copy
 - **I (invalid)**
 - Takes 2 bits to implement
- Allows **either**
 - Multiple read-only copies (**S**-state) **--OR--**
 - **Single** read/write copy (**M**-state)

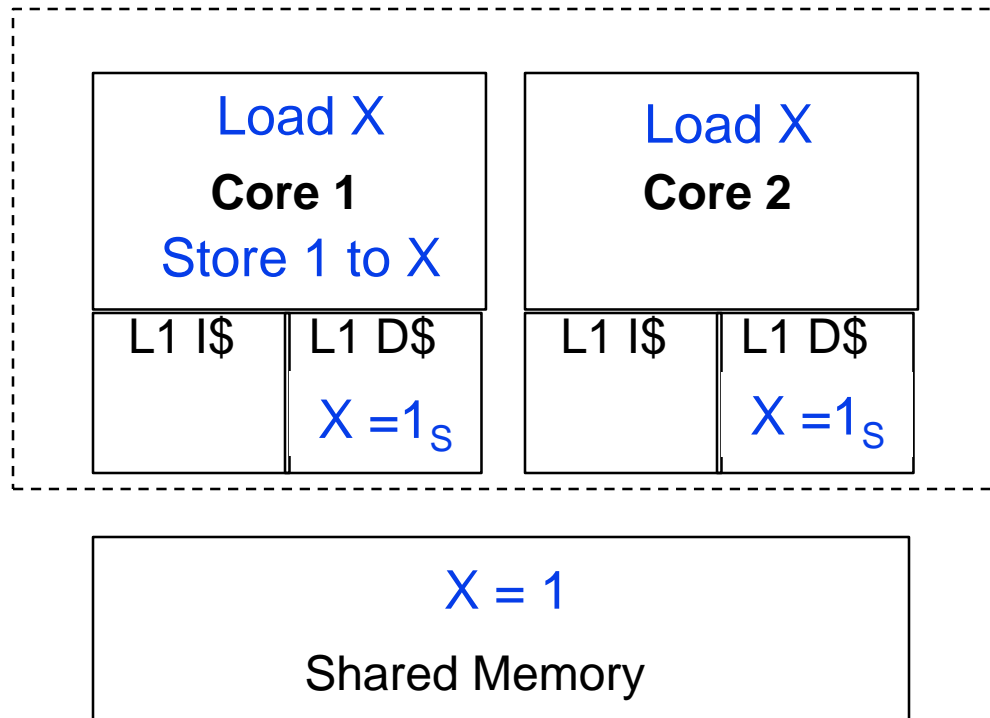


MSI protocol state transition table

State	<i>From This Core</i>		<i>From Other Cores</i>	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → S	Miss → M	---	---
Shared (S)	Hit	Upg Miss → M	---	→ I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

- ❑ M → S transition on a LdMiss also updates memory (Sends Data) since the block is Dirty
 - After which memory will respond (as all cores will be in S)

Example of MSI Snooping Protocol



LdMiss seen by Core 1 (no action required)

Core 1 marks X as modified and sends out an UpgMiss

StMiss (UpgMiss) seen by Core 2 so Core 2 invalidates its copy

LdMiss seen by Core 1, so Core 1 does a Send Data of X and marks its copy as shared

- ❑ When the second miss by Core 2 occurs, Core 1 responds with the value canceling the response from the shared memory (and also updating the value of X in the shared memory and marking its copy as shared)

Other Coherence Protocols

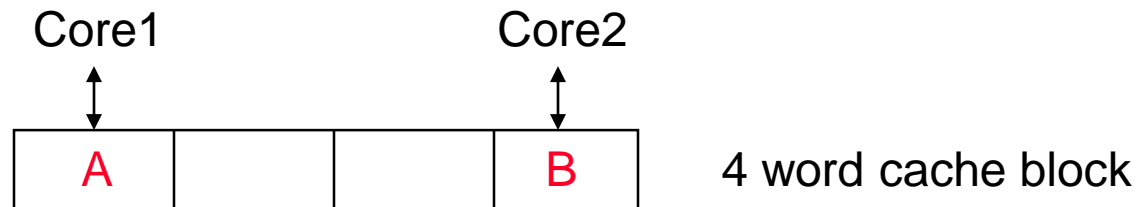
- ❑ Another write-invalidate protocol used in the Pentium 4 (and many other processors) is **MESI** with four states:
 - **M**odified – same
 - **E**xclusive – only one copy of the shared data is allowed to be cached
 - Since there is only one copy of the data, write hits don't need to send UpgMiss broadcast
 - **S**hared – same (multiple copies of the shared data may be cached (i.e., data permitted to be cached with more than one core))
 - **I**nvalid – load miss is E if no other core is caching the data, else its S
- ❑ Other variations include MOSI, MOESI, MERSI, MESIF, Berkeley, Firefly and Dragon
- ❑ Snooping protocols (and buses) don't scale well to many-cores
 - They use **directory based**, non-broadcast coherence protocols where a directory in memory keeps track of data ownership

Directory-Based Coherence

- ❑ No bus and no broadcast !
- ❑ Each cache/memory contains a portion of the directory
- ❑ Directory entry tells
 - Whether the block is in the cache
 - If so, where (which cache) and its status
- ❑ One possible algorithm:
 - Use physical address to go to the directory
 - Check whether the block is in the cache
 - If yes:
 - Go to the cache that contains the block
 - If no:
 - Issue a memory request [we have a cache miss]
 - The block is brought to the requesting core/cache
 - The directory is updated
- ❑ There are many ways to build a directory (e.g., distr vs shared)

Block Size Effects

- ❑ Writes to one word in a multi-word block mean that the full block is invalidated
- ❑ Multi-word blocks can also result in **false sharing**: when two cores are writing to two different variables that happen to fall in the same cache block
 - With VI false sharing increases cache miss rates



- ❑ Compilers can help reduce false sharing by allocating highly correlated data to the same cache block


Coherence Misses

1. **True-sharing misses** arise from communication of data through cache-coherence mechanism
 - Invalidates due to write to shared word
 - Reads by another core of modified block in different cache
 - Miss would still occur if block size were 1 word
2. **False-sharing misses** when a block is invalidated because some word in the block, other than the one being read, is written into
 - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
 - Block is shared, but no word in block is actually shared
 - ⇒ Miss would not occur if block size were 1 word

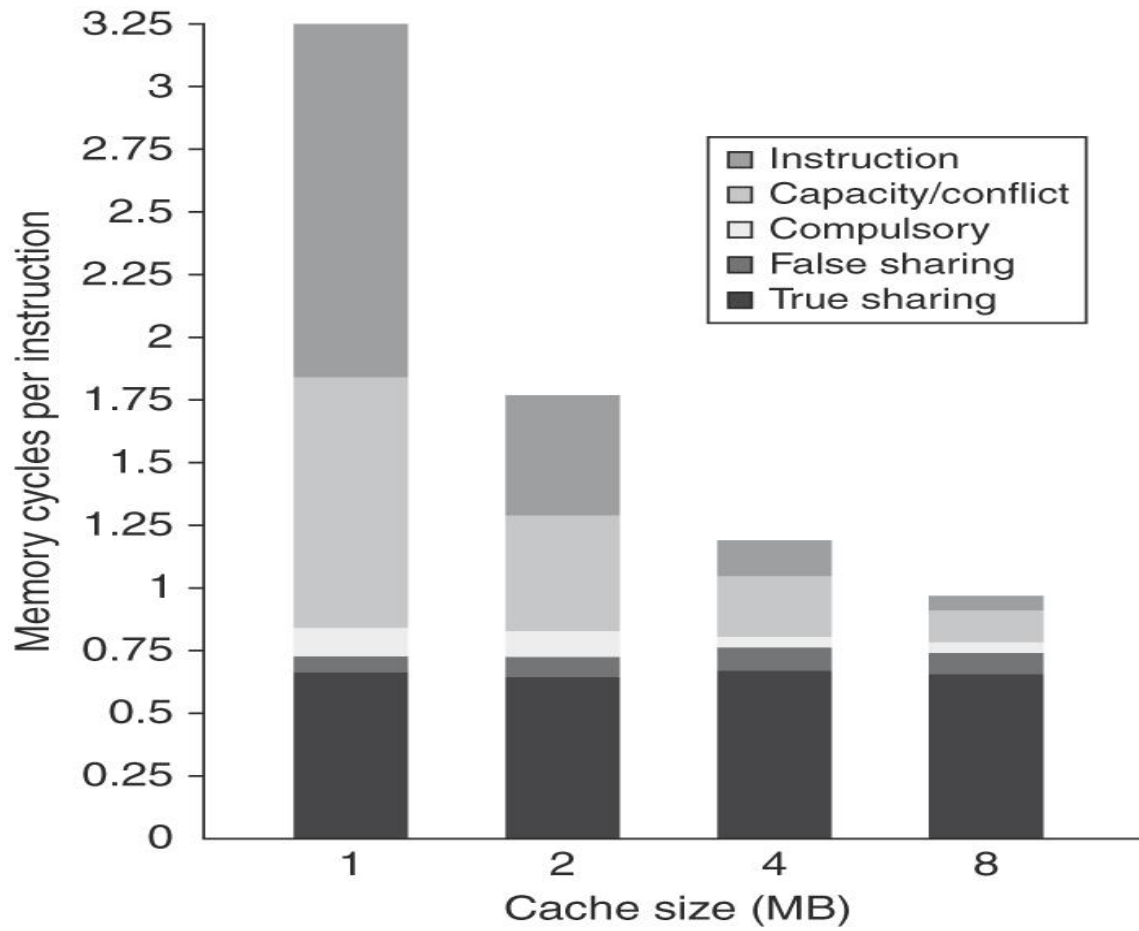
Example: True Sharing vs False Sharing

Assume d1 and d2 in same cache block.
P1 and P2 both read d1 and d2 before.

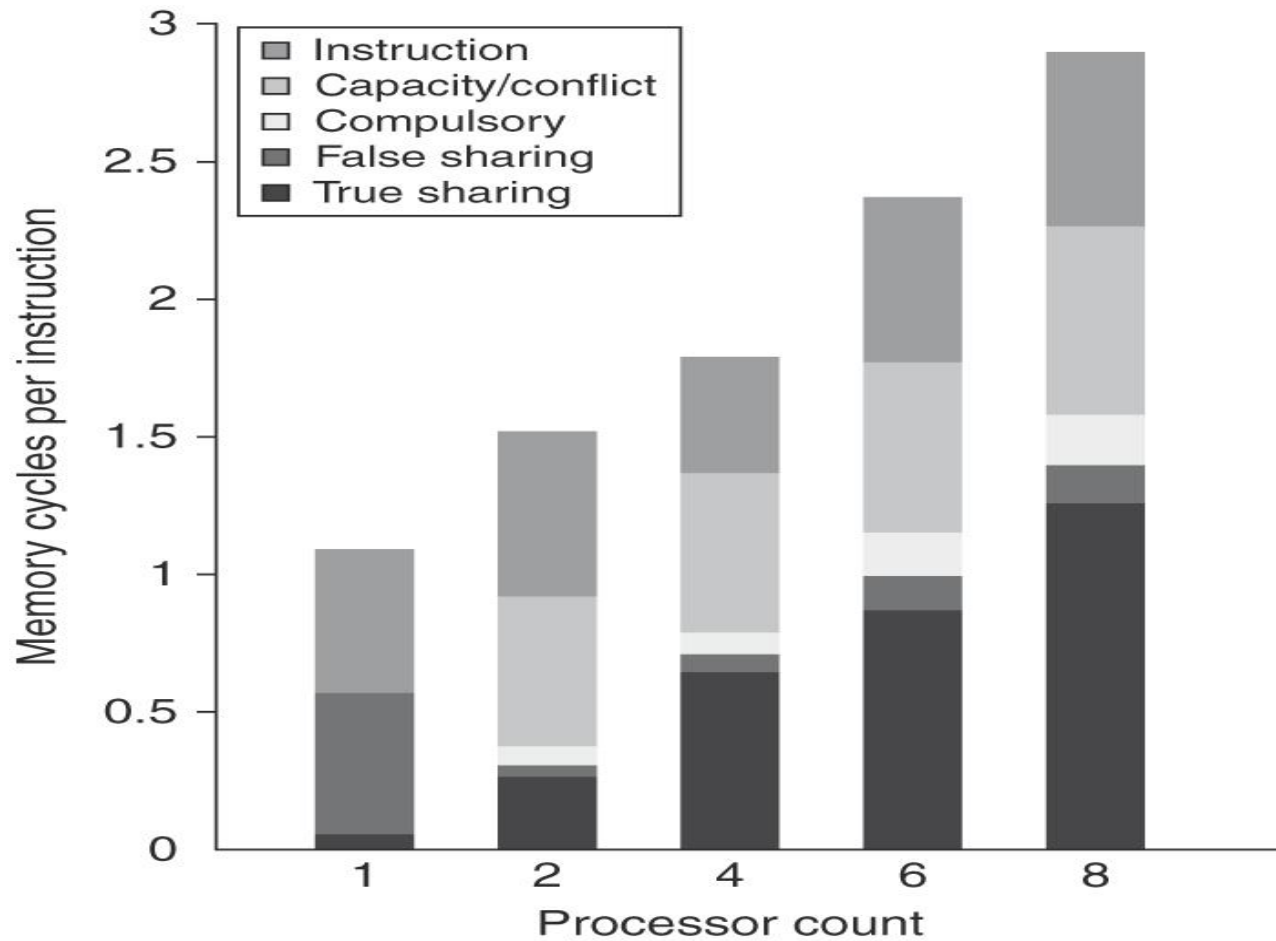
Step	P1	P2	True, False, Hit? Why?
1	Write d1		Invalidate the cache block in P2
2		Read d2	False miss; d1 irrelevant to P2
3	Write d1		Invalidate the cache block in P2
4		Write d2	Invalidate the cache block in P1
5	Read d2		True miss



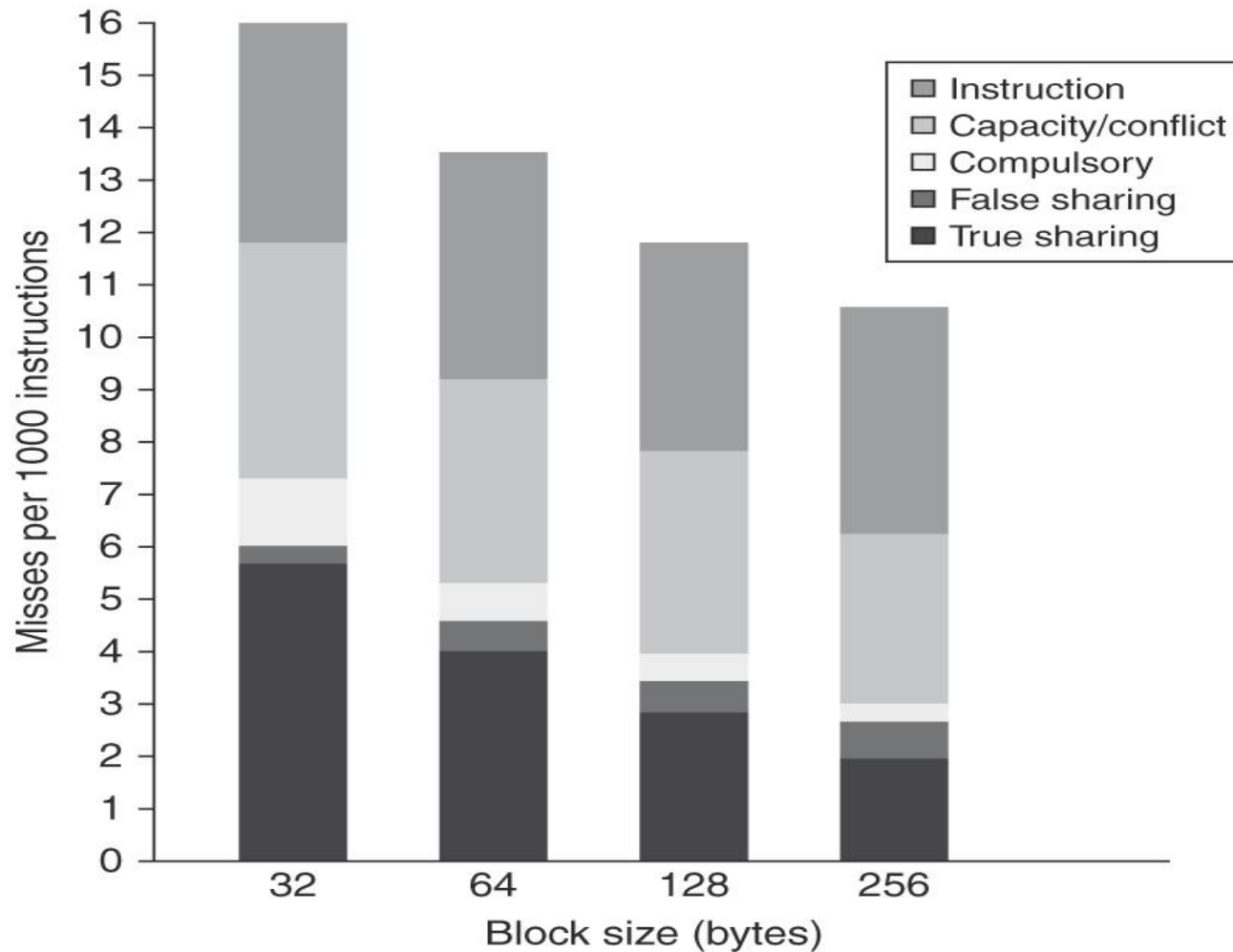
Memory Cycle Distribution



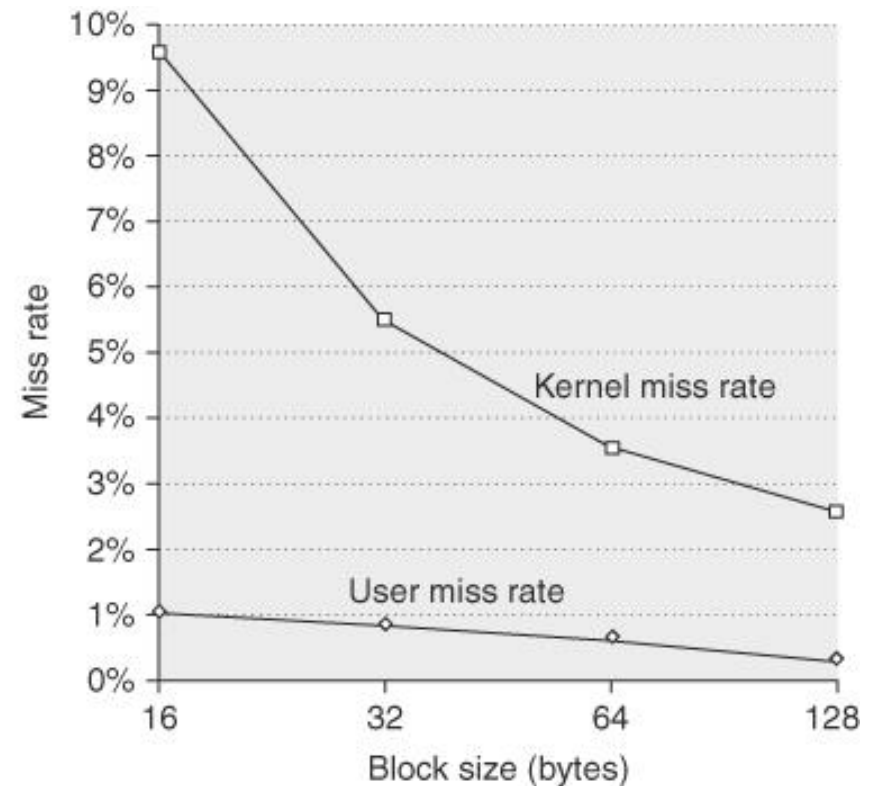
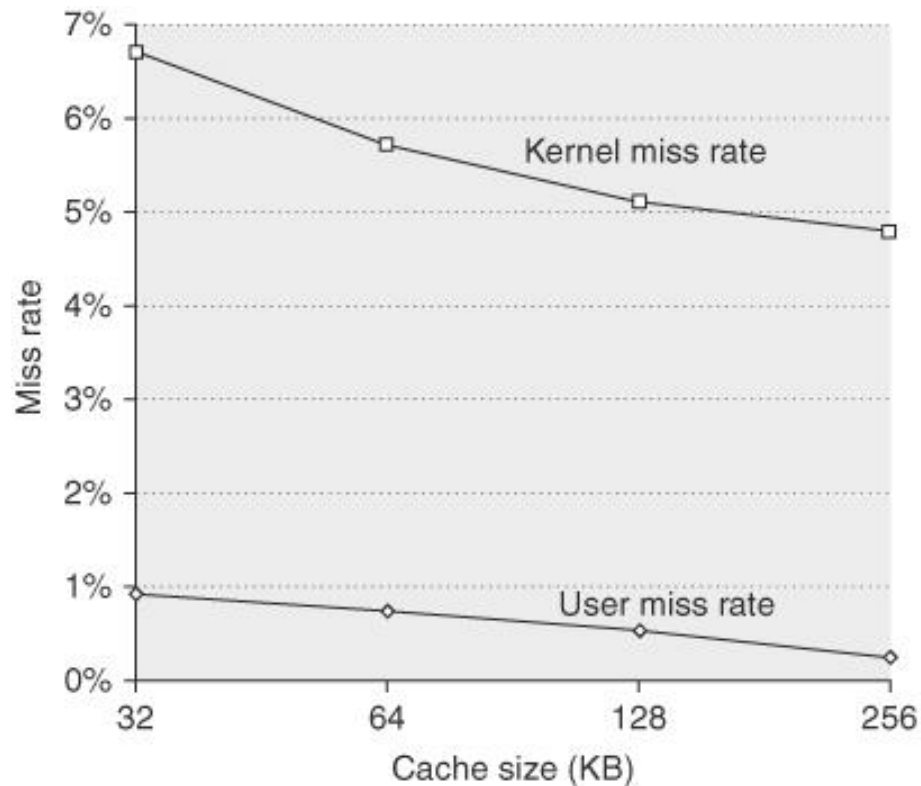
Memory Cycle Distribution



Impact of L3 Block Size



User Mode vs Kernel Mode



Memory Consistency

❑ When are stores seen by other cores ?

- “Seen” means a load returns the written value
- Can’t be instantaneously !

❑ Assumptions: **Write Serialization**

- A store completes only when all cores have seen it
- A core does not reorder its own stores with other memory accesses (true on in-order commit datapaths)

❑ Consequences

- C1 stores to location X then C2 stores to location X (and in MSI, C1 invalidates its copy of X)
- Serializing the stores ensures that **every** core will first see the store by C1 and then, eventually, the store by C2
- Cores can reorder loads, but not stores

Importance of Consistency Model

- ❑ Impact of the memory consistency model on a shared memory machine is profound
- ❑ **Programmability**
 - The model affects programmability because programmers must use it to reason about the correctness of their programs.
- ❑ **Performance**
 - The model affects the performance of the system because it determines the types of optimizations that may be exploited by the hardware and the system software.
- ❑ **Portability**
 - Due to a lack of consensus on a single model, portability can be affected when moving software across systems supporting different models.

Consistency Models: Example Programs

Initially, $A = B = 0$

P1

$A = 1$

if ($B == 0$)

critical section

P2

$B = 1$

if ($A == 0$)

critical section

P1

$Data = 2000$

$Head = 1$

P2

while ($Head == 0$)

{ }

... = $Data$

Initially, $A = B = 0$

P1

$A = 1$

P2

if ($A == 1$)

$B = 1$

P3

if ($B == 1$)

register = A

Coherence vs. Consistency

- Recall that coherence guarantees (i) that a write will eventually be seen by other processors, and (ii) write serialization (all processors see writes to the same memory location in the same order)
- The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions

Sequential Consistency (SC) (MIPS, PA-RISC)

- ❑ The most commonly assumed memory consistency model
- ❑ Formally defined by Lamport as follows:
 - A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors/cores were executed in some sequential order, and the operations of each individual processor/core appear in this sequence in the order specified by its program.
- ❑ Three important aspects
 - Maintaining program order among operations from individual processors
 - Each instruction executes atomically
 - Maintaining a single sequential order among operations from all processors. This makes it appear as if a memory operation executes atomically or instantaneously with respect to other memory operations

Sequential Consistency

P1	P2
Instr-a	Instr-A
Instr-b	Instr-B
Instr-c	Instr-C
Instr-d	Instr-D
...	...

We assume:

- Within a program, program order is preserved
- Each instruction executes atomically
- Instructions from different threads can be interleaved arbitrarily

Valid executions:

abAcBCDdeE... or ABCDEFabGc... or abcAdBe... or
aAbBcCdDeE... or

Sequential Consistency

- Programmers assume SC; makes it much easier to reason about program behavior
- Hardware innovations can disrupt the SC model
- For example, if we assume write buffers, or out-of-order execution, or if we drop ACKS in the coherence protocol, the previous programs yield unexpected outputs

Consistency Example - I

- An OOO core will see no dependence between instructions dealing with A and instructions dealing with B; those operations can therefore be re-ordered; this is fine for a single thread, but not for multiple threads

Initially A = B = 0	
P1	P2
A \leftarrow 1	B \leftarrow 1
...	...
if (B == 0)	if (A == 0)
Crit.Section	Crit.Section

The consistency model lets the programmer know what assumptions they can make about the hardware's reordering capabilities

Consistency Example - 2

Initially, $A = B = 0$

P1
 $A = 1$

P2
if ($A == 1$)
 $B = 1$

P3
if ($B == 1$)
 register = A

If a coherence invalidation didn't require ACKs, we can't confirm that everyone has seen the value of A.

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion
- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow
- **Alternatives:**
 - Add optimizations to the hardware
 - Offer a relaxed memory consistency model and fences

Relaxed Consistency Models

- We want an intuitive programming model (such as sequential consistency) and we want high performance
- We care about data races and re-ordering constraints for some parts of the program and not for others – hence, we will relax some of the constraints for sequential consistency for most of the program, but enforce them for specific portions of the code
- Fence instructions are special instructions that require all previous memory accesses to complete before proceeding (sequential consistency)

Fences

P1

```
{  
  Region of code  
  with no races  
}
```

Fence
Acquire_lock
Fence

```
{  
  Racy code  
}
```

Fence
Release_lock
Fence

P2

```
{  
  Region of code  
  with no races  
}
```

Fence
Acquire_lock
Fence

```
{  
  Racy code  
}
```

Fence
Release_lock
Fence

Relaxing Constraints

- Sequential consistency constraints can be relaxed in the following ways (allowing higher performance):
 - within a processor, a read can complete before an earlier write to a different memory location completes (this was made possible in the write buffer example and is of course, not a sequentially consistent model)
 - within a processor, a write can complete before an earlier write to a different memory location completes
 - within a processor, a read or write can complete before an earlier read to a different memory location completes
 - a processor can read the value written by another processor before all processors have seen the invalidate
 - a processor can read its own write before the write is visible to other processors

Relaxed Memory Consistency Models

❑ **Processor consistency (PC)** (x86, SPARC)

- Allows an in-order Store Buffer
 - Stores can be deferred, but must be put into the cache **in order**

❑ **Release consistency (RC)** (ARM, Itanium, PowerPC)

- Allows an un-ordered Store Buffer
 - Stores can be put into cache **in any order**

-
- ❑ At the present time, many processors being built support some sort of relaxed consistency model
 - ❑ The expectation is that most programmers will use standard synchronization libraries, written by system programmers

Shared Memory multiProcessor (SMP)

- ❑ Q1 – Single memory address space shared by all cores
- ❑ Q2 – Cores coordinate/communicate through shared variables in memory (via loads and stores)
 1. Use of shared data must be coordinated via **synchronization** primitives (locks) that allow access to data to only one core at a time (used to implement critical sections)
 2. Caches must be kept coherent (cores have the same shared data)
- ❑ SMPs come in two styles
 1. Uniform memory access (**UMA**) multiprocessors
 2. Nonuniform memory access (**NUMA**) multiprocessors
- ❑ Programming NUMAs is the harder of the two
- ❑ But NUMAs can scale to larger sizes and have lower latency to **local** memory

Synchronization

- ❑ Need to be able to coordinate processes working on a common task; sharing data must be coordinated carefully
 - Lock variables (**semaphores**) are used to coordinate or synchronize processes
 - **mutual exclusion** – restrict data access to one core at a time
 - **sequential ordering** – must complete the first operation before the second operation is allowed to begin
- 1. Need an architecture-supported arbitration mechanism to decide which core gets access to the lock variable
 - Single bus provides an arbitration mechanism, since the bus is the only path to memory – the core that gets the bus wins
- 2. Need an architecture-supported operation that locks the variable
 - Locking can be done via an **atomic swap operation** which allows a core to both read a location and set it to the locked state – **test-and-set** – in the same bus operation

Synchronization

- ❑ Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions
- ❑ Synchronization can be a performance bottleneck
- ❑ One typical hardware support is **atomic exchange**
 - Interchanges a value in a register for a value in memory
 - Assume we want to build a simple lock where **0** is used to indicate the lock is free and **1** is used to indicate the lock is unavailable
 - A processor performs an exchange of 1, which is in a register, with the memory address corresponding to the lock
 - The value read (after the exchange)
 - 1: the lock is unavailable (some other processor is currently using it)
 - 0: we got the lock
- ❑ **Key property:** the operation is atomic!

Synchronization

- ❑ An alternative (as in MIPS) 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
 - If all other operations executed by any processor occurred before or after the pair (too strict?)
- ❑ **Load Linked** and **Store Conditional**
 - Used in sequence
 - Load Linked returns the initial value
 - Store Conditional returns 1 only if it succeeds

Atomic Exchange Support

❑ **Atomic exchange** (atomic swap) – interchanges a value in a register for a value in memory atomically, i.e., as one operation (instruction)

- Implementing an atomic exchange would require **both** a memory read and a memory write in a single, uninterruptable instruction

❑ MIPS provides **ll/sc**: load-linked / store-conditional

- Atomic load/store pair

```
ll r2,0(&lock)      #load linked
// potentially other load/store's by other cores
sc r1,0(&lock)      #store conditional
```

- On **ll**, the SRAM cache controller (or DRAM memory controller) remembers the core id and the load address (in a **link register**) ...
 - And watches for **stores** by other cores (or for any exceptions)
- If a store by another core to the same address is detected or a context switch occurs, the **sc** fails (i.e., the store is not performed)
 - the **sc** returns a 1 to the core if the store was successful, 0 on fail

Atomic Exchange with ll and sc

- ❑ If the contents of the memory location specified by the `ll` are changed (by some other core) before the `sc` to the same address executes, the `sc` fails (returns a zero)

```
lock: add $t0, $zero, $s4      # $t0 = $s4 (exchange value)
      ll  $t1, 0($s1)          # load memory value to $t1
      // potentially other load/store instr's executed by other cores
      sc  $t0, 0($s1)          # try to store the exchange
                                # value to memory, if fail
                                # $t0 will be 0, if succeed
                                # $t0 will be 1
      beq $t0, $zero, lock      # try again on failure
      add $s4, $zero, $t1      # put exchange value in $s4
```

If the `sc` fails (does not succeed in storing the exchange value into memory) and returns a 0 in `$t0`, the code sequence tries again

On success, the contents of `$s4` and the memory location specified by `$s1` have been atomically exchanged

Atomic Exchange with ll and sc

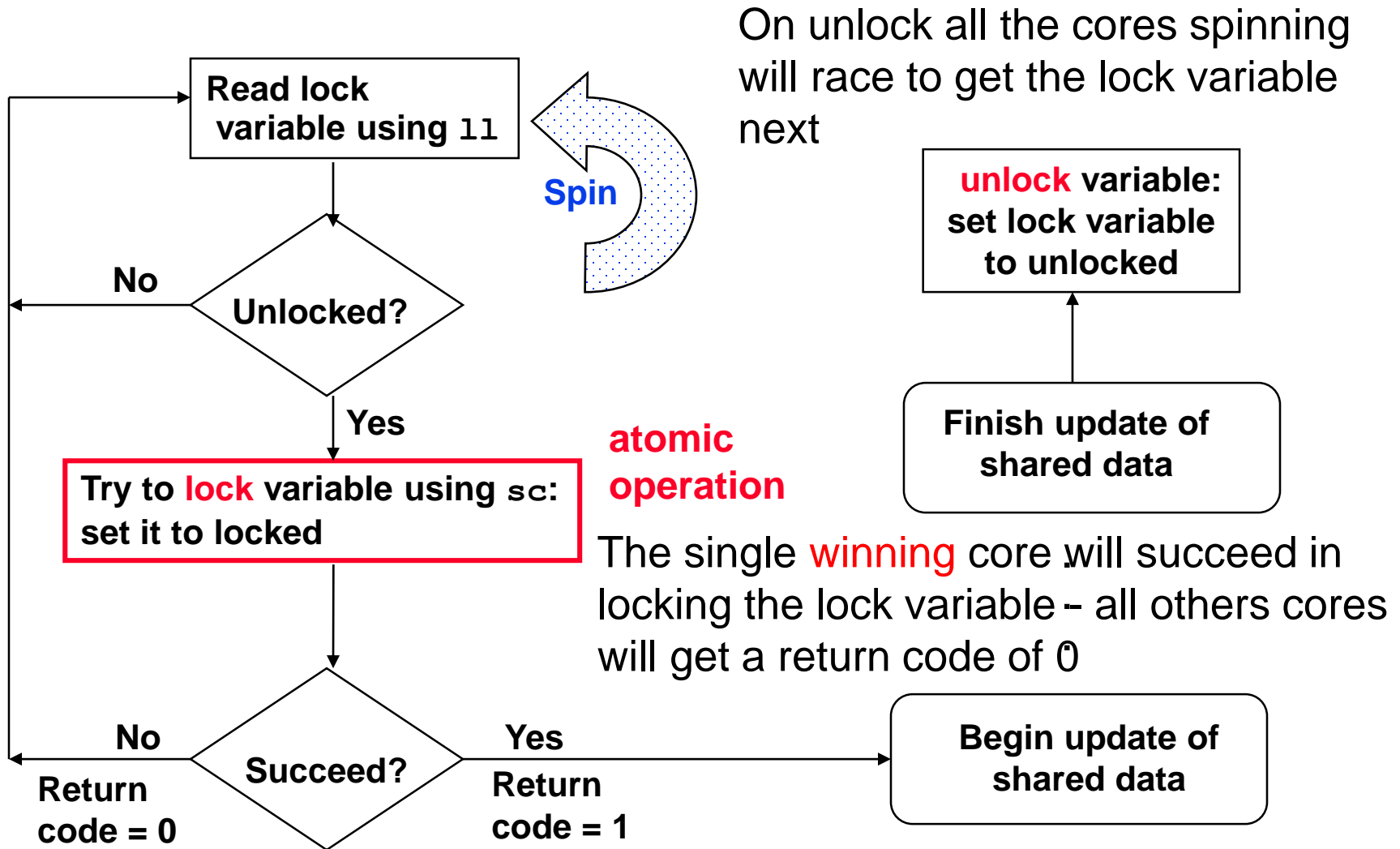
- ❑ On ll, the memory controller remembers the load address (in a link register) and watches for stores by (other) cores to the same address (or for any exceptions).
- ❑ If the sc succeeds, then a 1 is returned.
- ❑ Use care when purposely inserting instructions between the ll and sc. Only register-register instructions can be safely permitted; otherwise, it is possible to create deadlock where the processor can never complete the sc because of repeated page faults.

Atomic Exchange Example

- ❑ Assume all loads and stores (including `ll` and `sc`) hit in the L1 cache

Core0	Core1	Cycle	Core0		Mem (\$s1)	Core1	
			\$t0	\$t1		\$t0	\$t1
		0	30	40	55	10	11
	<code>ll \$t1, 0(\$s1)</code>	1					55
<code>ll \$t1, 0(\$s1)</code>		2		55			
	<code>sc \$t0, 0(\$s1)</code>	3			10	1	
<code>sc \$t0, 0(\$s1)</code>		4	0		10		

Spin Lock Synchronization



Summing 100,000 Numbers on 100 Core SMP

- ❑ Cores start by running a thread loop that sums their subset of vector A numbers (vectors A and sum are **shared** variables, Cn is the core's number, i is a **private** variable)

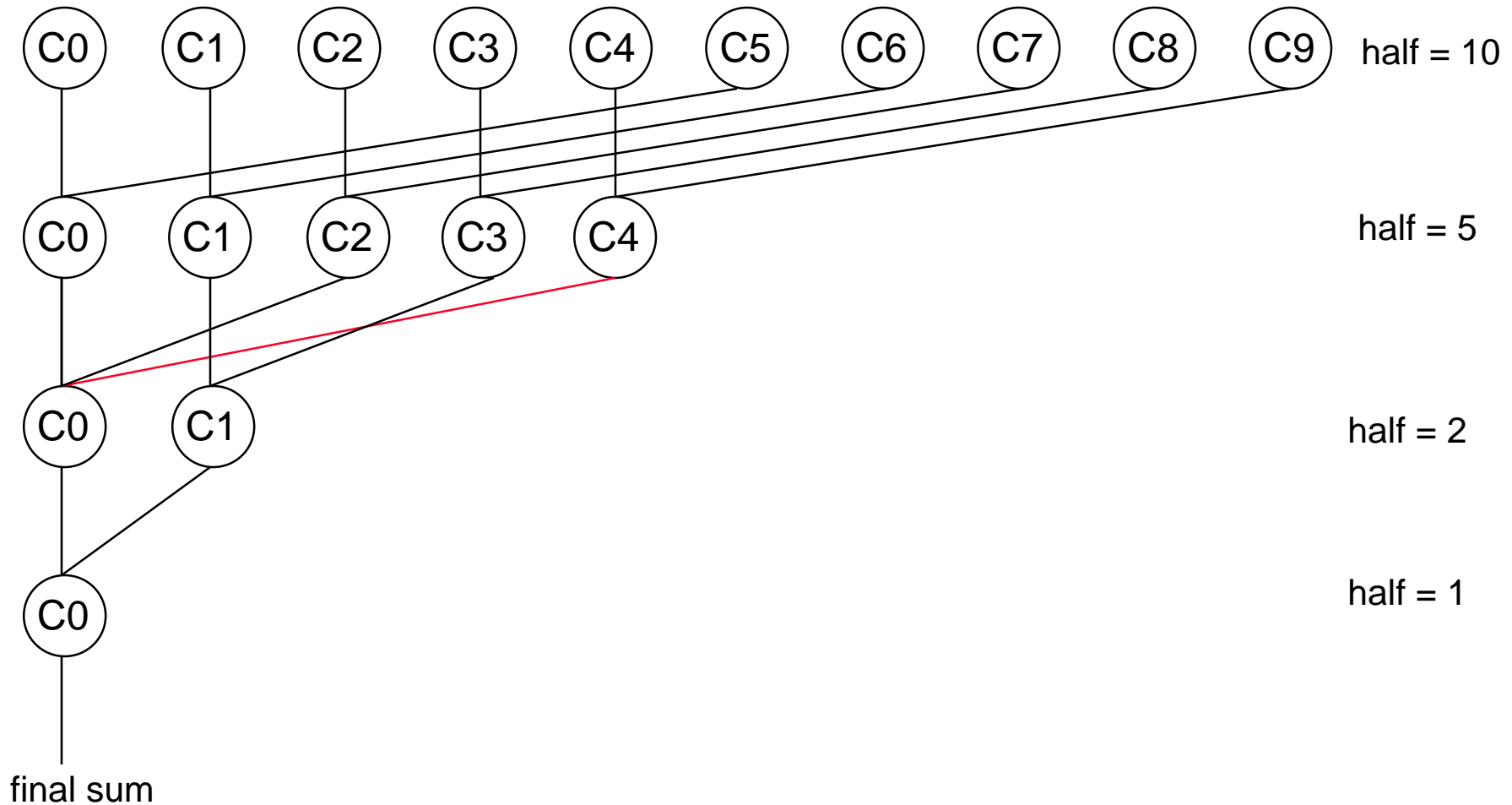
```
sum[Cn] = 0;
for (i = 1000*Cn; i < 1000*(Cn+1); i = i + 1)
    sum[Cn] = sum[Cn] + A[i];
```

- ❑ The cores' threads then coordinate in adding together the partial sums (half is a **private** variable initialized to 100 (the number of cores)) – **reduction**

```
repeat
    synch();                /*synchronize first
    if (half%2 != 0 && Cn == 0)
        sum[0] = sum[0] + sum[half-1];
    half = half/2
    if (Cn < half) sum[Cn] = sum[Cn] + sum[Cn+half]
until (half == 1);          /*final sum in sum[0]
```

An Example with 10 Cores (10 Threads)

sum[C0]sum[C1]sum[C2] sum[C3]sum[C4]sum[C5]sum[C6] sum[C7]sum[C8] sum[C9]



Review: Summing Numbers on a SMP

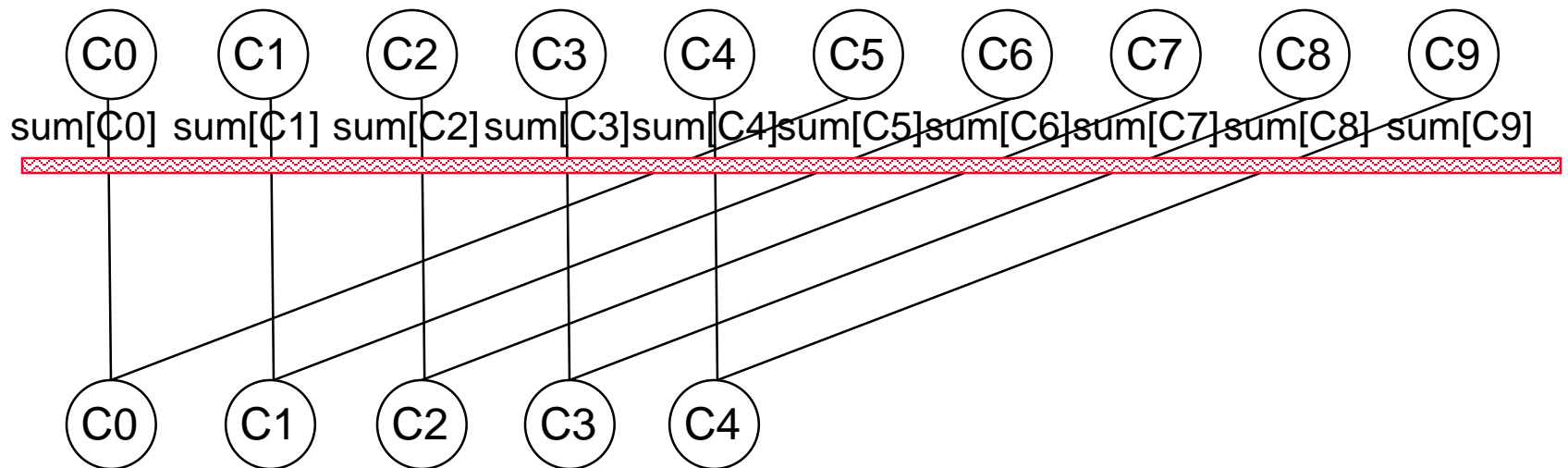
- C_n is the core's number, vectors A and sum are **shared** variables, i is a **private** variable, $half$ is a **private** variable initialized to the number of cores

```
sum[Cn] = 0;
for (i = 1000*Cn; i < 1000*(Cn+1); i = i + 1)
    sum[Cn] = sum[Cn] + A[i];
                                /* each core sums its
                                /* subset of vector A

repeat                          /* adding together the
                                /* partial sums
    synch () ;                  /*synchronize first
    if (half%2 != 0 && Cn == 0)
        sum[0] = sum[0] + sum[half-1];
    half = half/2
    if (Cn < half) sum[Cn] = sum[Cn] + sum[Cn+half];
until (half == 1);             /*final sum in sum[0]
```

An Example with 10 Cores (10 Threads)

- ❑ `synch()`: Cores (threads) must synchronize before the “consumer” core (thread) tries to read the results from the memory location written by the “producer” core (thread)
 - **Barrier synchronization** – cores wait at the barrier, not proceeding until every core has reached it



Key SMP Multiprocessor Design Questions

❑ Q1 – How do they share data?

A single physical address space shared by all cores

❑ Q2 – How do they coordinate?

Program threads on cores coordinate/
communicate through atomic operations on shared
variables in memory (via loads and stores)

❑ Q3 – Scalability - How cores can be supported?

			# of Cores
Communication model	SMP	NUMA	8 to 256 +
		UMA	2 to 32
Physical interconnect	Network		8 to 256 +
	Bus		2 to 8