

CS250P: Computer Systems Architecture

Multiprocessing and Parallelism

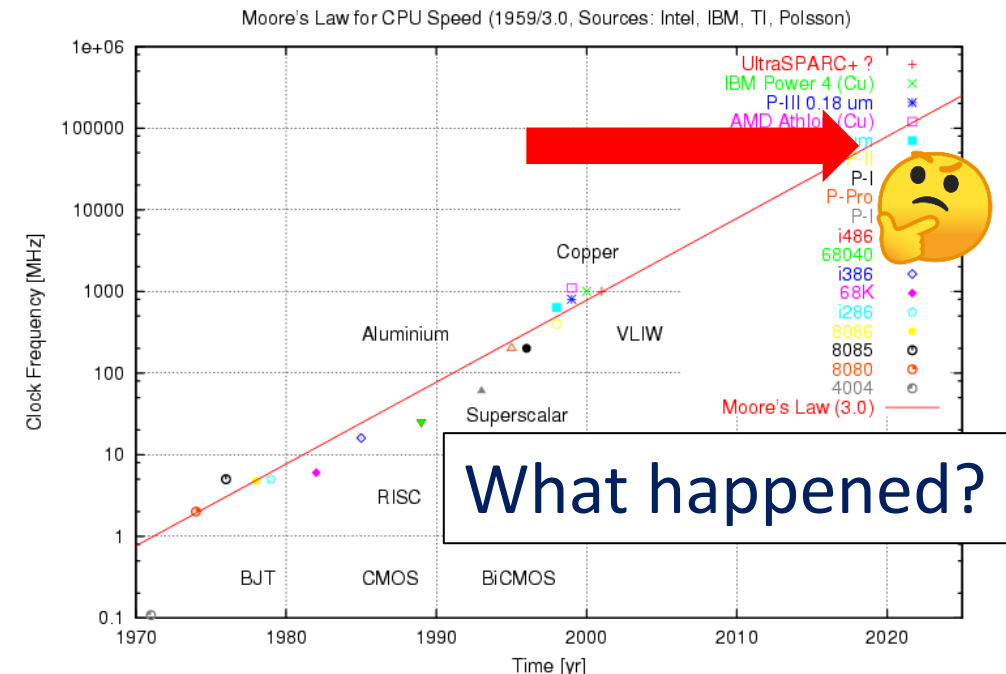


Sang-Woo Jun

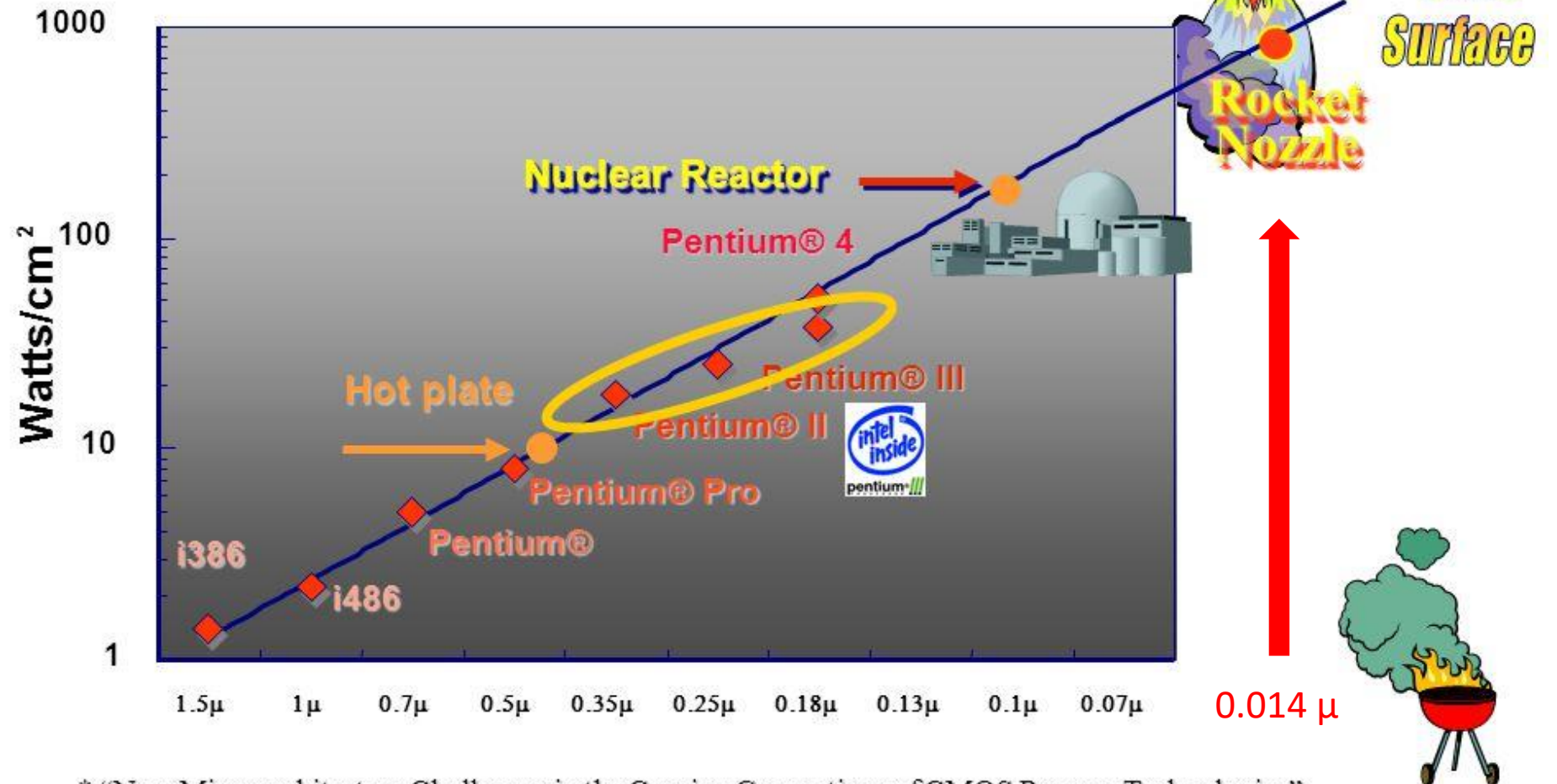
Fall 2022

Why focus on parallelism?

- ❑ Of course, large jobs require large machines with many processors
 - Exploiting parallelism to make the best use of supercomputers have always been an extremely important topic
- ❑ But now even desktops and phones are multicore!
 - Why? The end of “Dennard Scaling”



Option 1: Continue Scaling Frequency at Increased Power Budget



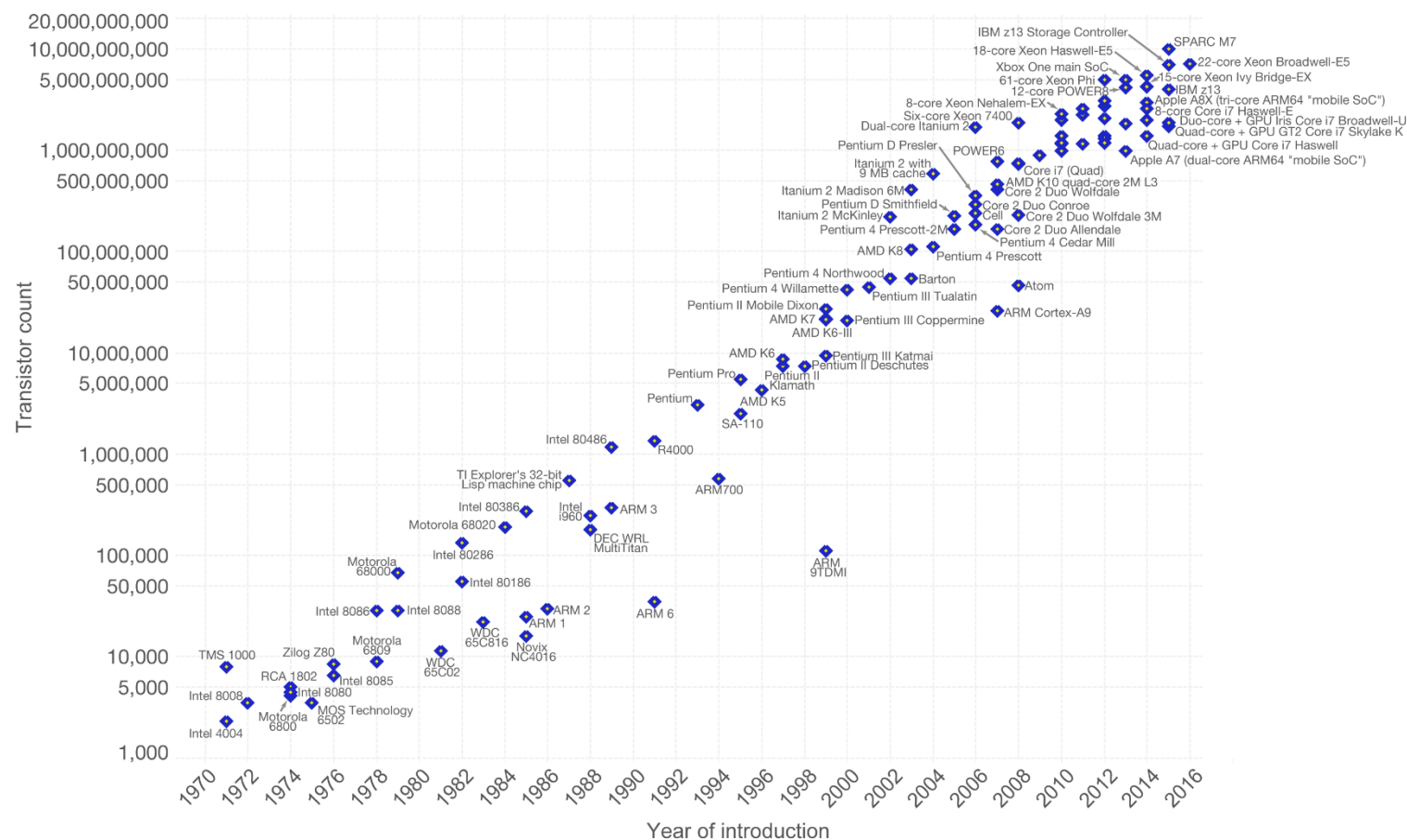
* "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" – Fred Pollack, Intel Corp. Micro32 conference key note - 1999.

But Moore's Law Continues Beyond 2006

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World
in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



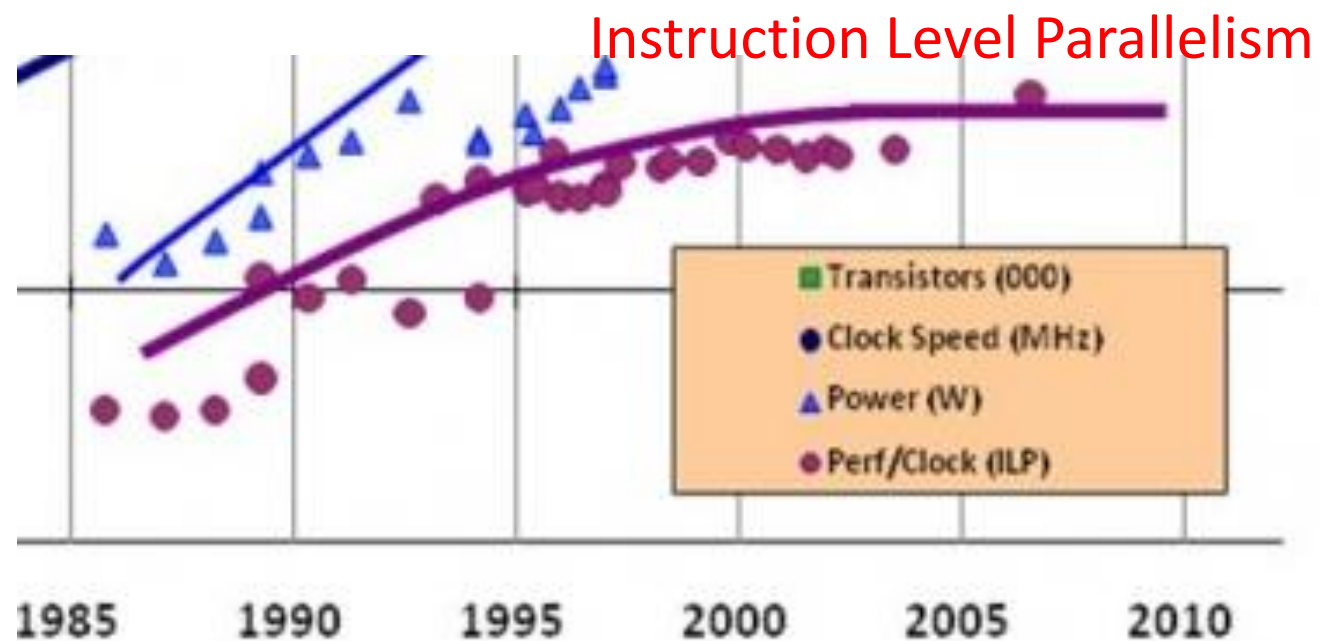
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at [OurWorldinData.org](https://ourworldindata.org). There you find more visualizations and research on this topic.

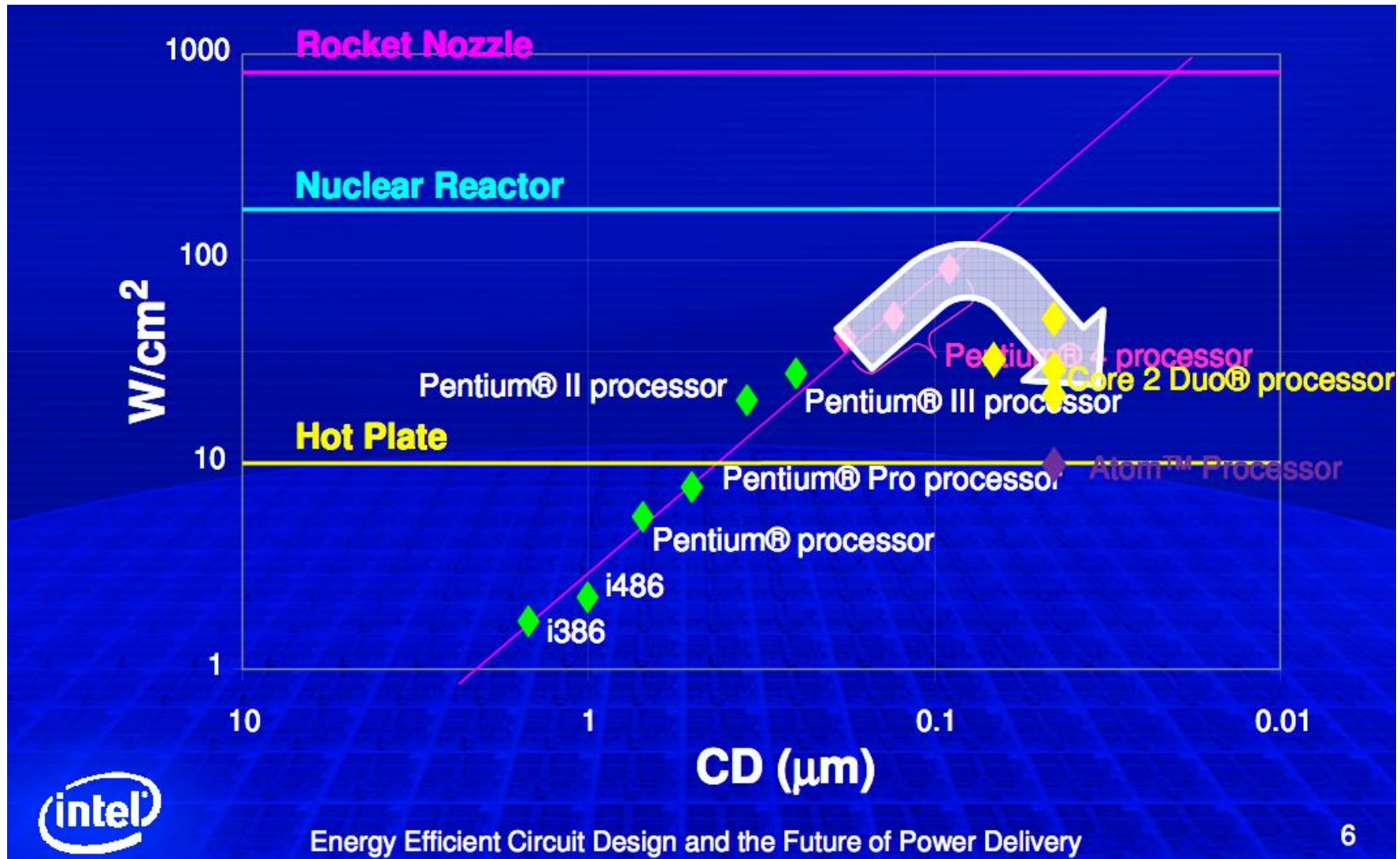
Licensed under [CC-BY-SA](#) by the author Max Roser.

State of Things at This Point (2006)

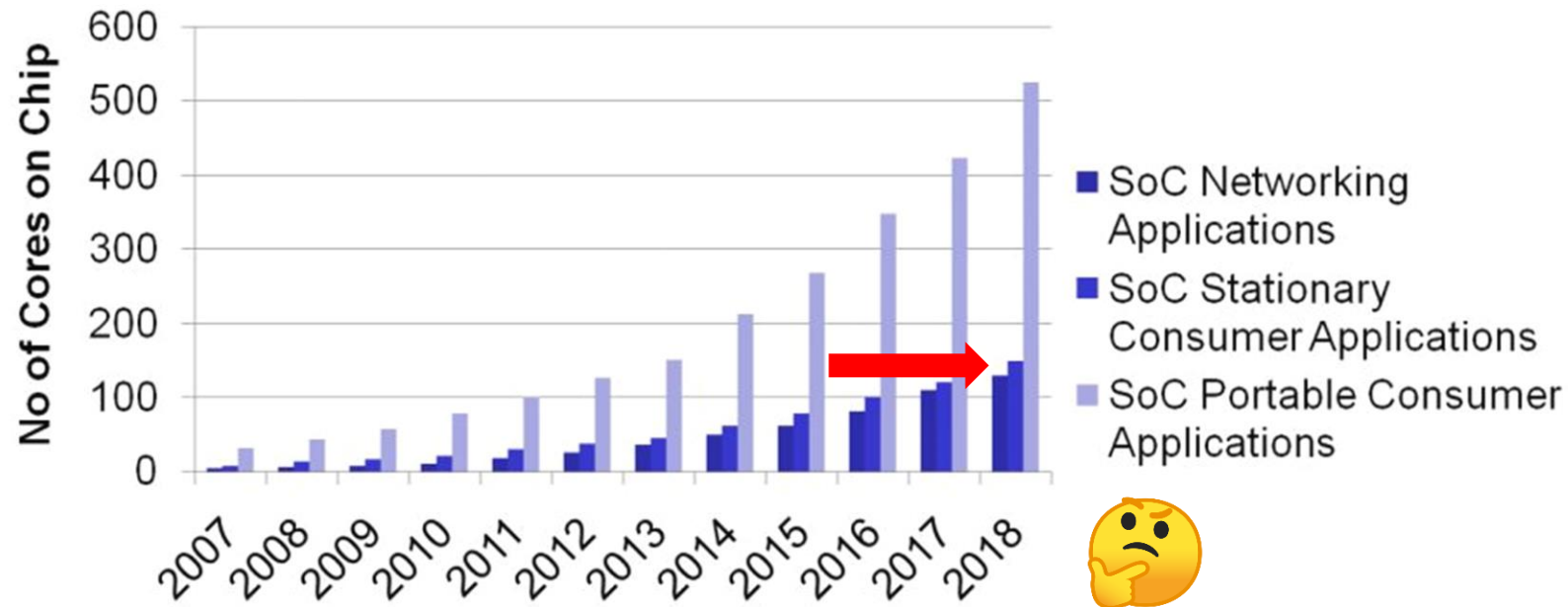
- ❑ Single-thread performance scaling ended
 - Frequency scaling ended (Dennard Scaling)
 - Instruction-level parallelism scaling stalled ... also around 2005
- ❑ Moore's law continues
 - Double transistors every two years
 - What do we do with them?



Crisis Averted With Manycores?



Crisis Averted With Manycores?



We'll get back to this point later. For now, multiprocessing!

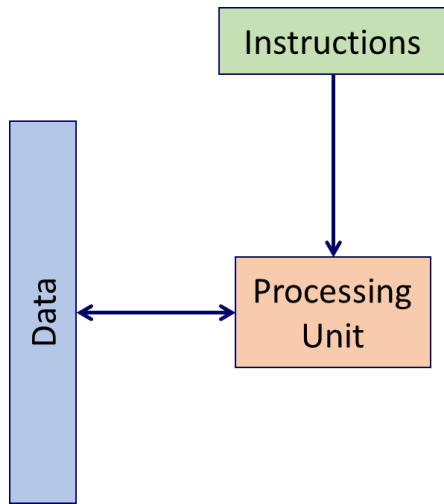
Source:

International Roadmap for Semiconductors 2007 edition (<http://www.itrs.net/>)

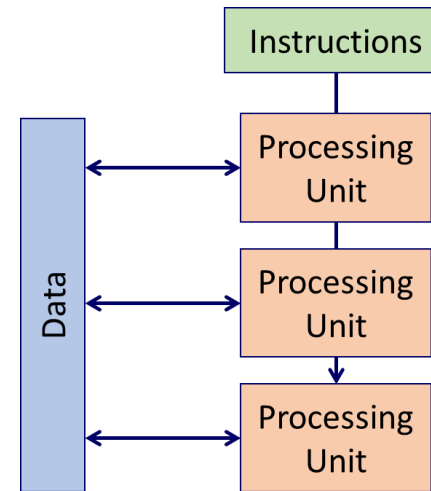
The hardware for parallelism: Flynn taxonomy (1966) recap

		Data Stream	
		Single	Multi
Instruction Stream	Single	SISD (Single-Core Processors)	SIMD (GPUs, Intel SSE/AVX extensions, ...)
	Multi	MISD (Systolic Arrays, ...)	MIMD (VLIW, Parallel Computers)

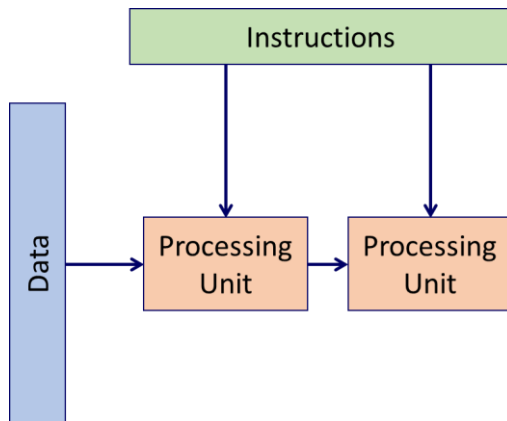
Flynn taxonomy



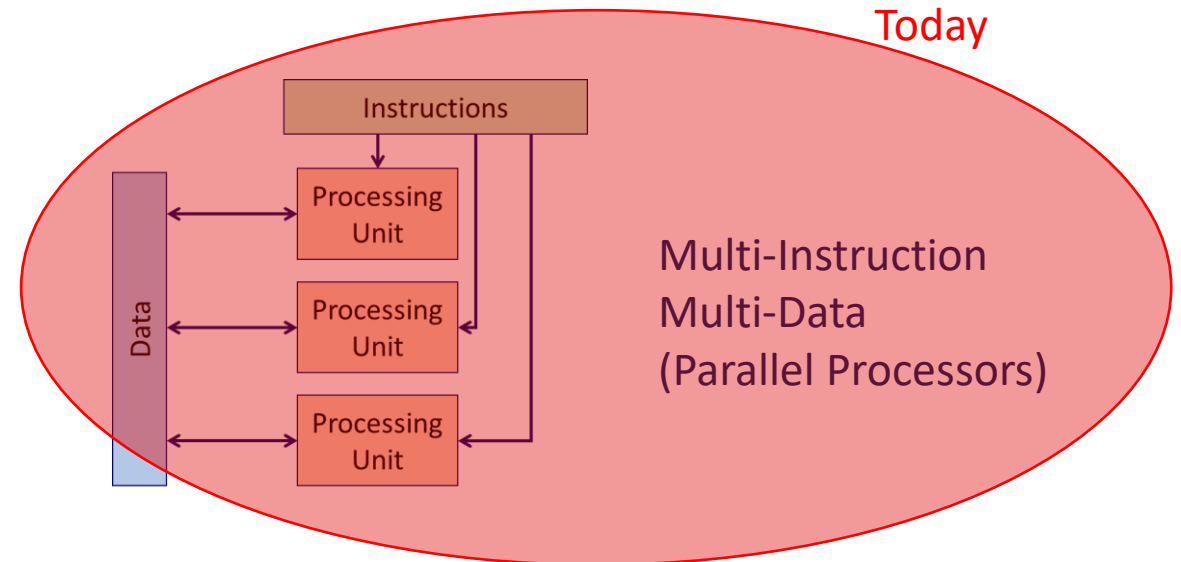
Single-Instruction
Single-Data
(Single-Core Processors)



Single-Instruction
Multi-Data
(GPUs, Intel SIMD Extensions)



Multi-Instruction
Single-Data
(Systolic Arrays,...)



Multi-Instruction
Multi-Data
(Parallel Processors)

Shared memory multiprocessor

❑ Shared memory multiprocessor

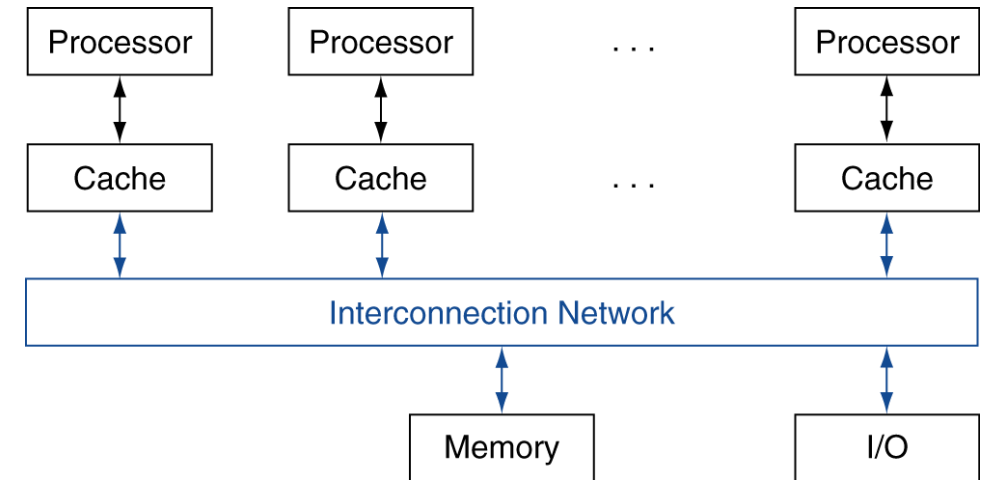
- Hardware provides single physical address space for all processors
- Synchronize shared variables using locks
- Memory access time
 - UMA (uniform) vs. NUMA (nonuniform)

❑ SMP: Symmetric multiprocessor

- The processors in the system are identical, and are treated equally

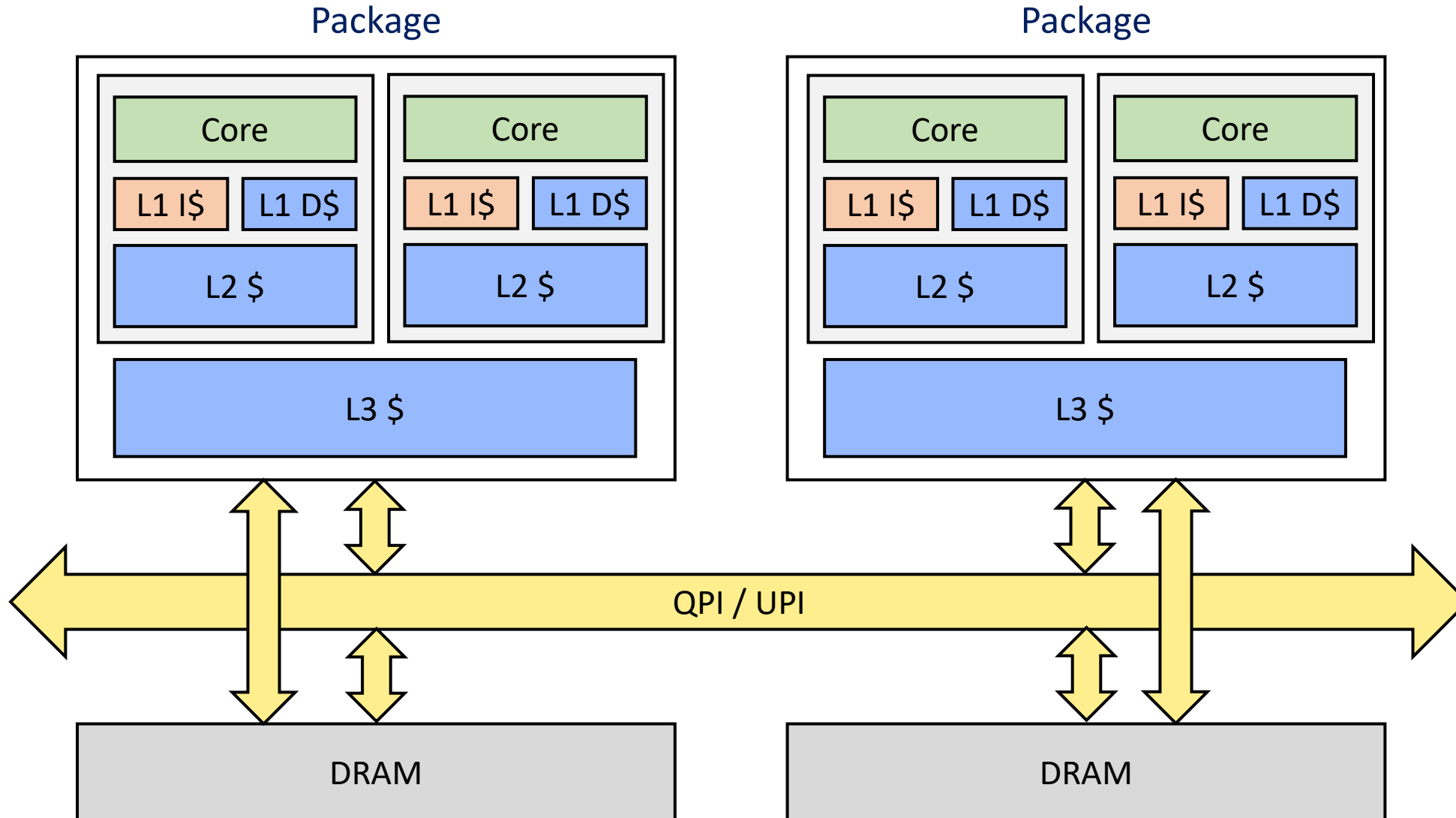
How identical? Very identical! They compete to become the boot core!

❑ Typical chip-multiprocessor (“multicore”) consumer computers

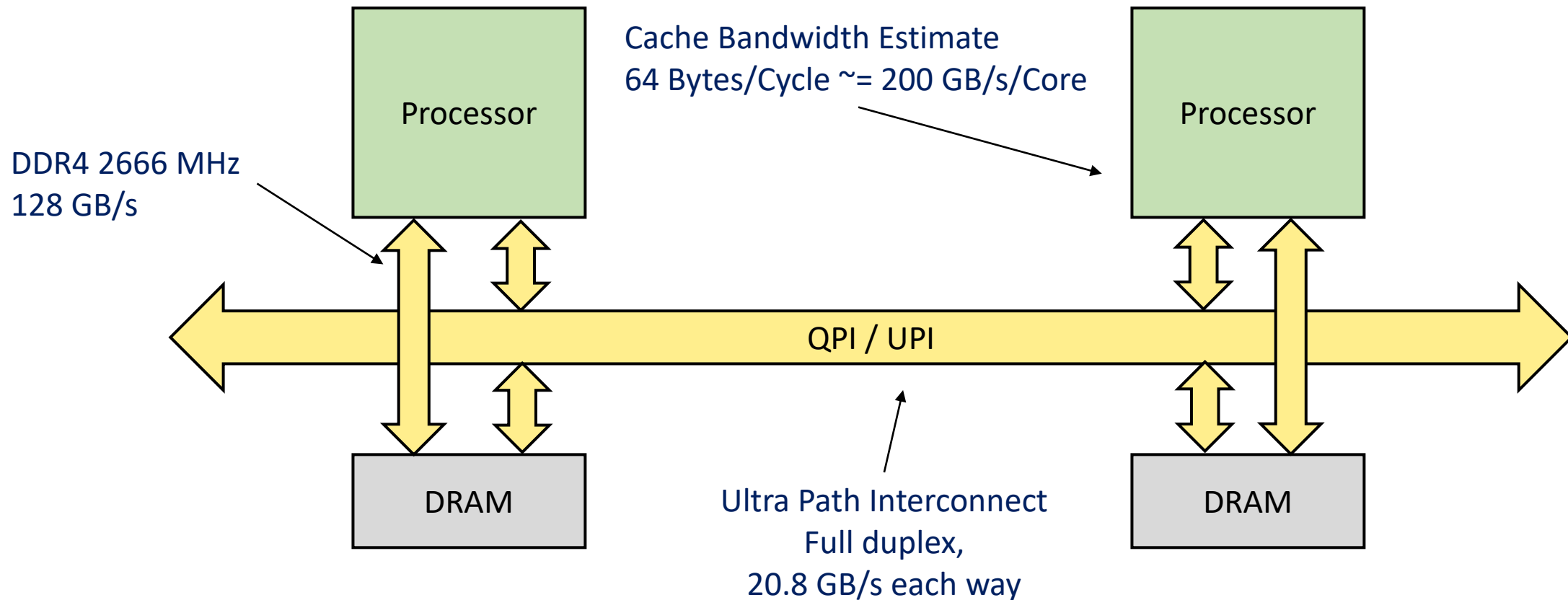


Memory System Architecture

UMA between cores sharing a package,
But NUMA across cores in different packages.
Overall, this is a NUMA system



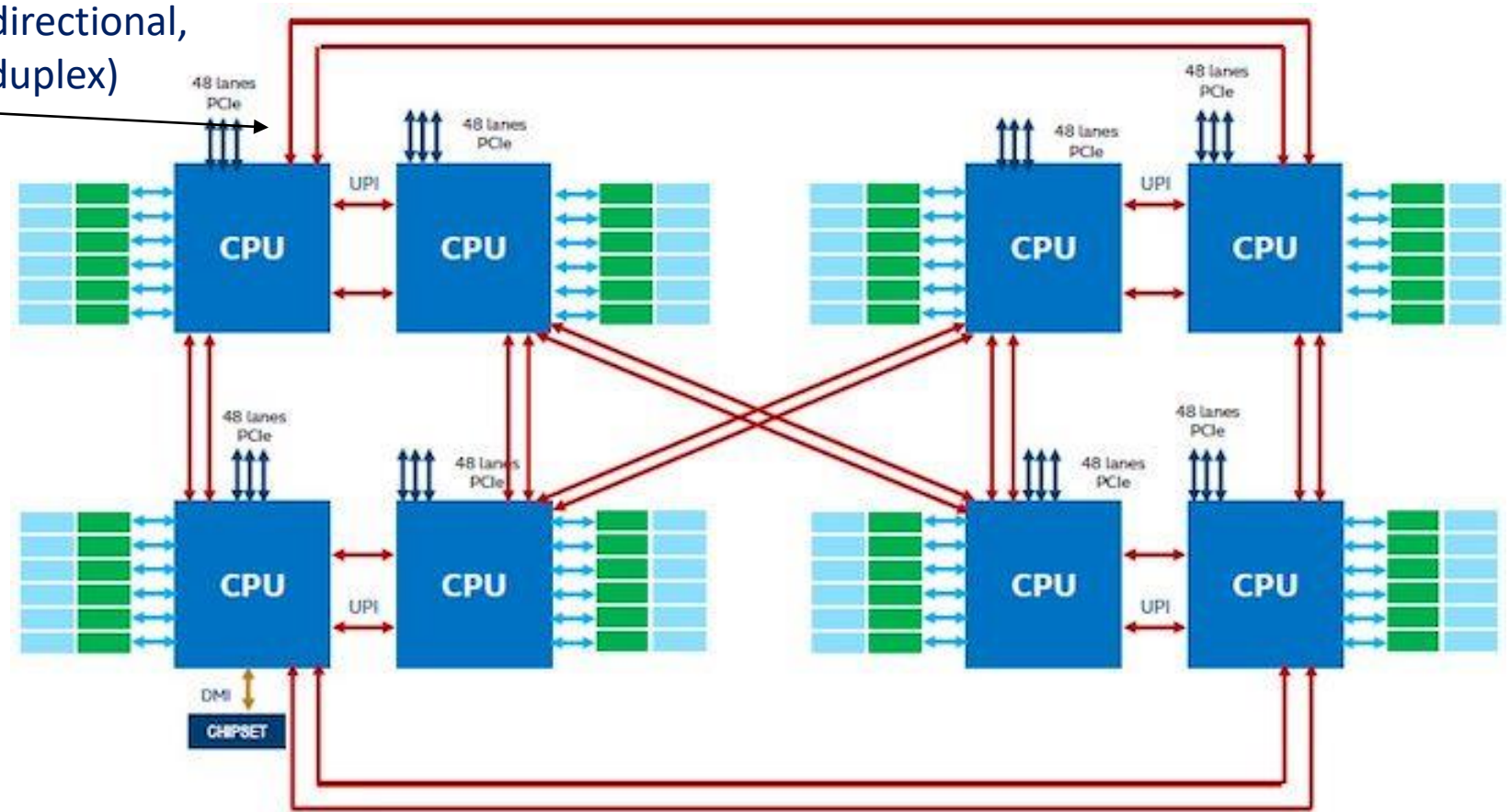
Memory System Bandwidth Snapshot (2021)



Memory/PCIe controller used to be on a separate “North bridge” chip, now integrated on-die
All sorts of things are now on-die! Even network controllers!

Example: Intel Xeon Sapphire Rapids (2021)

4x UPI
(4x 20.8 GB/s unidirectional,
8x 20.8 GB/s duplex)



“Twisted hypercube” topology
three nodes directly connected
four nodes two hops away

Memory system issues with multiprocessing (1)

- ❑ Suppose two CPU cores share a physical address space
 - Distributed caches (typically L1)
 - Write-through caches, but same problem for write-back as well

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Wrong data!

Memory system issues with multiprocessing (2)

❑ What are the possible outcomes from the two following codes?

- A and B are initially zero

Processor 1:

1: A = 1;
2: print B

Processor 2:

3: B = 1;
4: print A

- 1,2,3,4 or 3,4,1,2 etc : “01”
- 1,3,2,4 or 1,3,4,2 etc : “11”
- Can it print “10”, or “00”? Should it be able to?

“Memory model” defines what is possible and not
(Balance between performance and ease of use)

Memory problems with multiprocessing



Cache coherency (The two CPU example)

- Informally: Read to each address must return the most recent value
- Complex and difficult with many processors
- Typically: All writes must be visible at some point, and in proper order

Memory consistency (The two processes example)

- How updates to different addresses become visible (to other processors)
- Many models define various types of consistency
 - Sequential consistency, causal consistency, relaxed consistency, ...
- In our previous example, some models may allow “10” to happen, and we must program such a machine accordingly

CS250P: Computer Systems Architecture

Cache Coherency Introduction

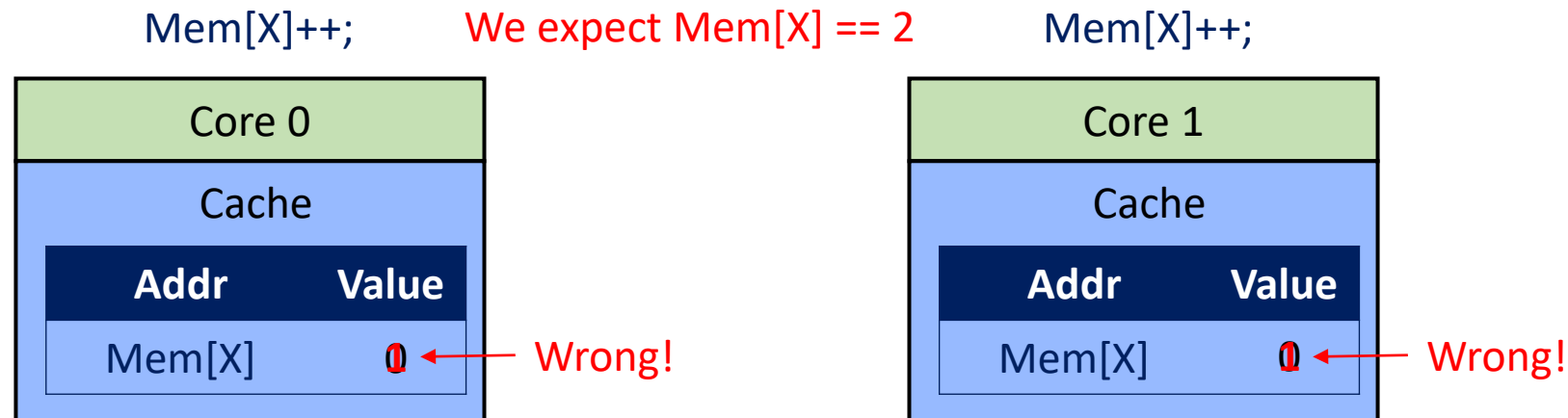


Sang-Woo Jun

Fall 2022

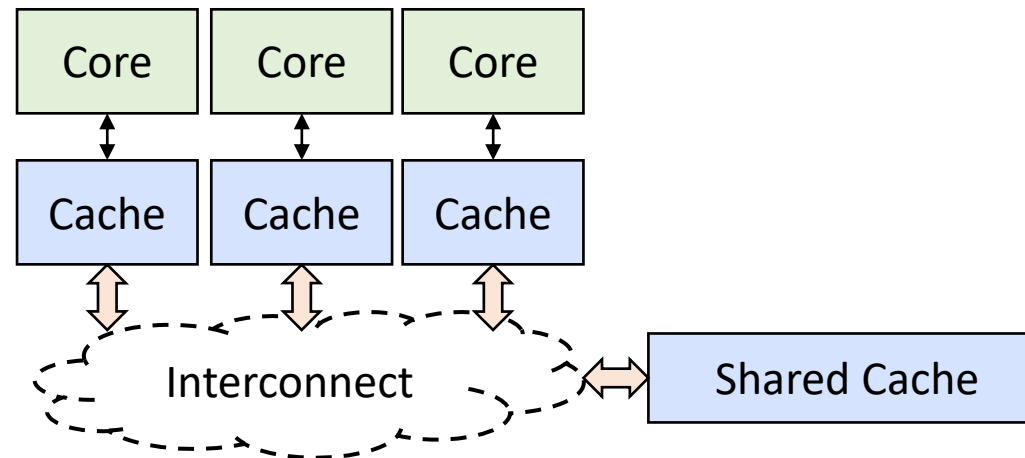
The cache coherency problem

- ❑ All cores may have their own cached copies for a memory location
- ❑ Copies become stale if one core writes only to its own cache
- ❑ Cache updates must be propagated to other cores
 - All cores broadcasting all writes to all cores undermines the purpose of caches
 - We want to privately cache writes without broadcasting, whenever possible



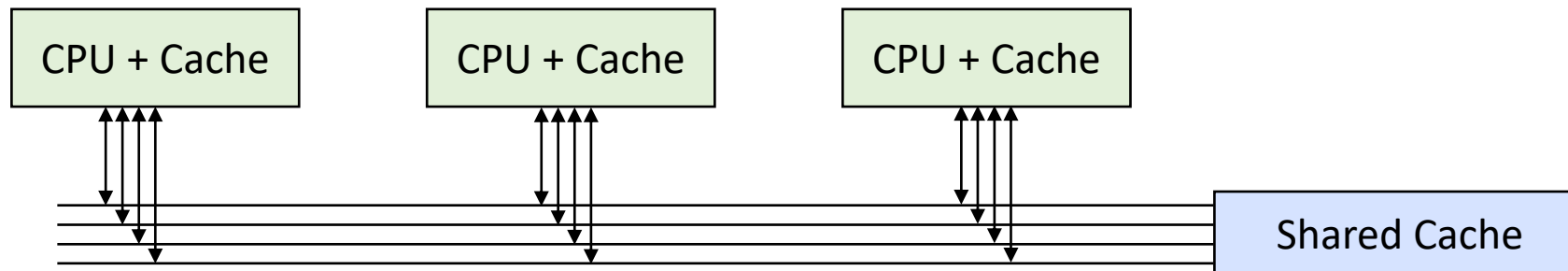
Background: On-chip interconnect

- ❑ An interconnect fabric connects cores and private caches to upper-level caches and main memory
 - Many different paradigms, architectures, and topologies
 - Packet-switched vs. Circuit-switched vs. ...
 - Ring topology vs. Tree topology vs. Torus topology vs. ...
- ❑ Data-driven decision of best performance/resource trade-off



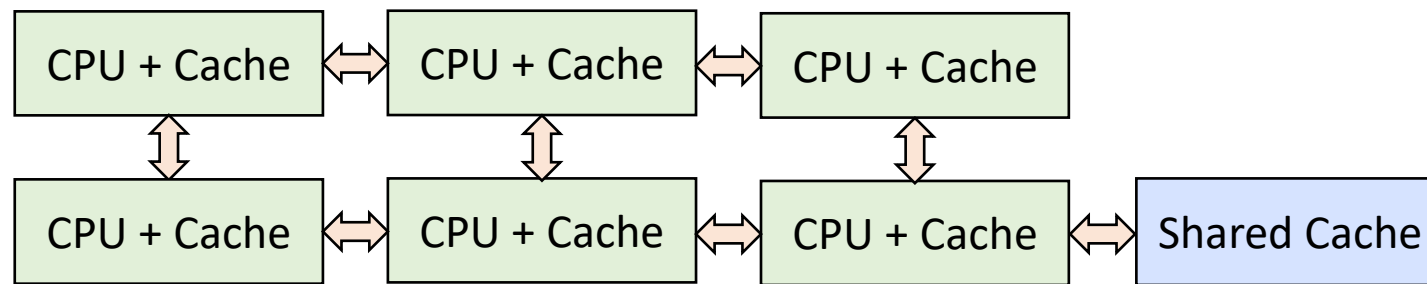
Background: Bus interconnect

- ❑ A bus is simply a shared bundle of wires
 - All data transfers are broadcast, and all entities on the bus can listen to all communication
 - All communication is immediate, single cycle
 - Only one entity may be transmitting at any given clock cycle
 - If multiple entities want to send data (a “multi-master” configuration) a separate entity called the “bus arbiter” must assign which master can write at a given cycle



Background: Mesh interconnect

- ❑ Each core acts as a network switch
 - Compared to bus, much higher aggregate bandwidth
 - Bus: 1 message/cycle, Mesh: Potentially as many messages as there are links
 - Much better scalability with more cores
 - Variable cycles of latency
 - A lot more transistors to implement, compared to bus



Desktop-class multicores migrating from busses to meshes (As of 2022)

Here we use busses for simplicity of description

Keeping multiple caches coherent

❑ Basic idea

- If a cache line is only read, many caches can have a copy
- If a cache line is written to, only one cache at a time may have a copy

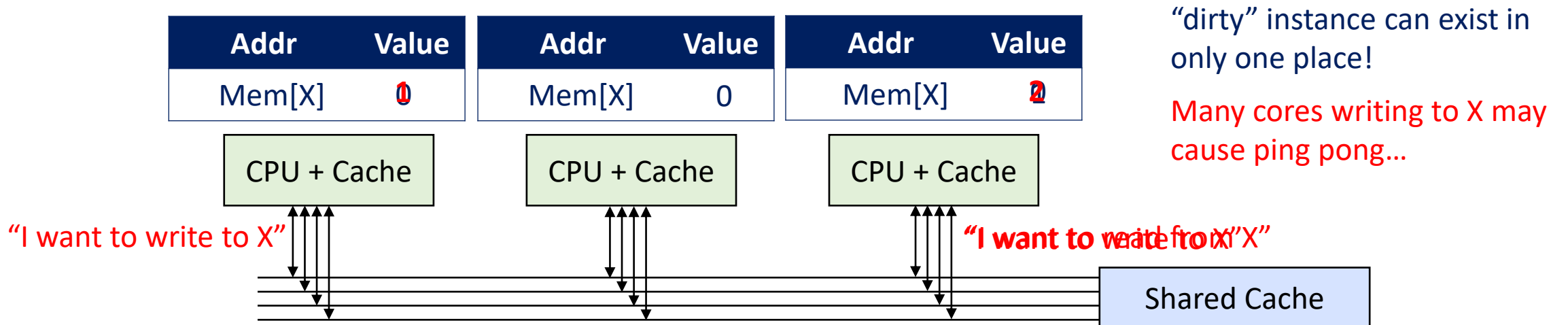
❑ Writes can still be cached (and not broadcast)!

❑ Typically two ways of implementing this

- “Snooping-based”: All cores listen to requests made by others on the memory bus
- “Directory-based”: All cores consult a separate entity called “directory” for each cache access

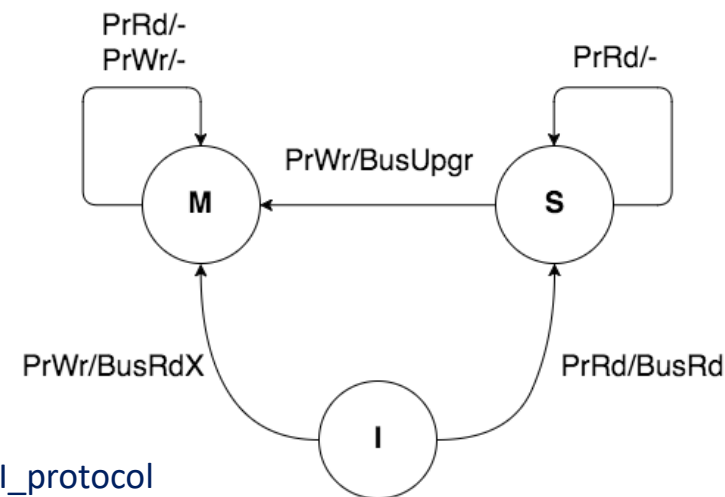
Snoopy cache coherence

- ❑ All caches listen (“snoop”) to the traffic on the memory bus
 - Some new information is added to read/write requests
- ❑ Before writing to a cache line, each core must broadcast its intention
 - All other caches must invalidate its own copies
 - Algorithm variants exist to make this work effectively (MSI, MSIE, ...)



Super high-level MSI introduction

- ❑ Each cache line can exist in one of three states
 - Modified 'M' : Dirty line. Only one copy of this line can exist across cores.
 - Shared 'S' : Unmodified read-only line. There can be multiple copies.
 - Invalid 'I' : Cache is no longer valid.
- ❑ Each cache line managed via a state machine
 - Cache reads fetched in 'S' state. Many cores can share.
 - One cache attempts to write
 - This attempt is broadcast onto bus, other 'S' copies are invalidated
 - Line upgrades itself to 'M'.
- ❑ Many variants exist: MESI, MOESI, ...

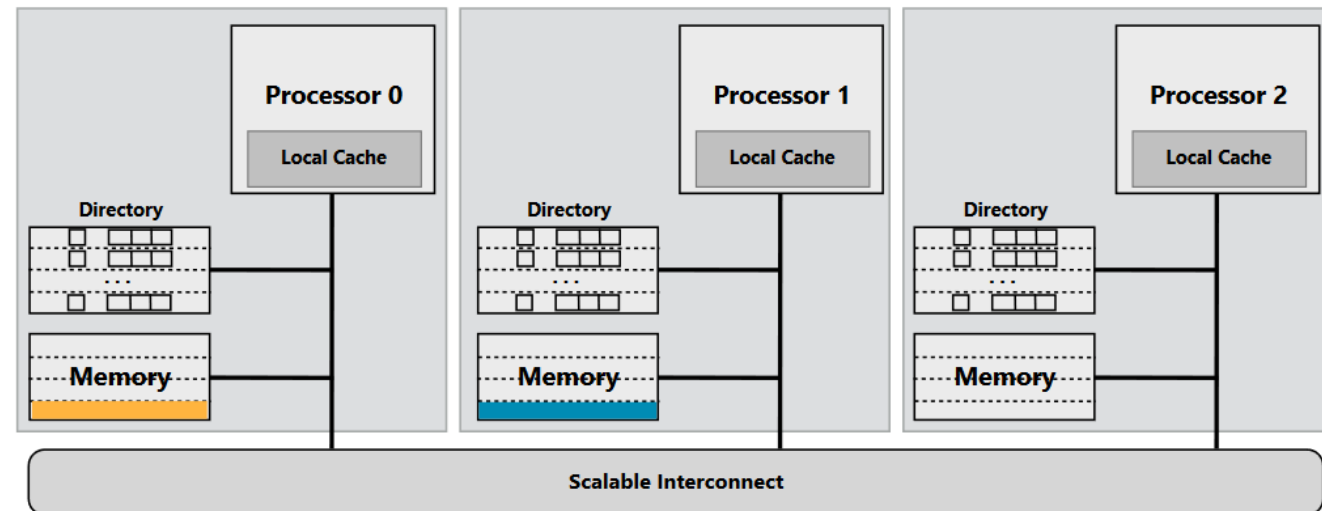


Directory-based coherency

- ❑ Issue with snoopy: All cores share a bus!
 - Only one processor can broadcast per cycle... Scalability issue!!
- ❑ Directory-based coherency assigns subsets of main memory to each core
 - Separate “Directory” per core manages caches mapped to its own subset
 - When accessing main memory, each core should query the responsible directory
 - Write request causes directory to send P2P invalidation messages
- ❑ Pros: Scalable! Multiple MSI-like protocol can run at each core
- ❑ Cons: Higher latency (multi-cycle request to directory)
- ❑ Cons: More memory requirement (directory data structure)
- ❑ More modern processors typically implement this

Directory-based coherency

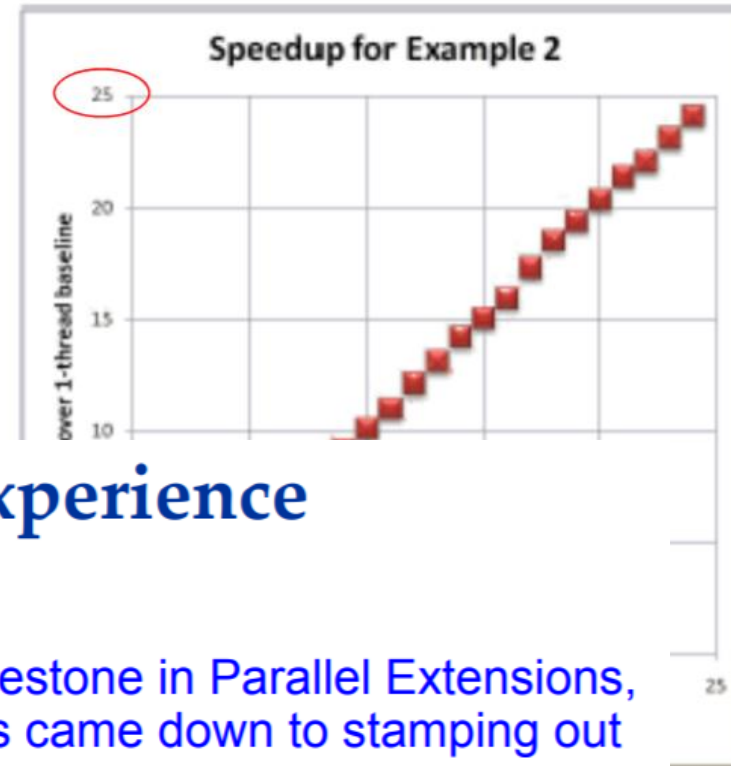
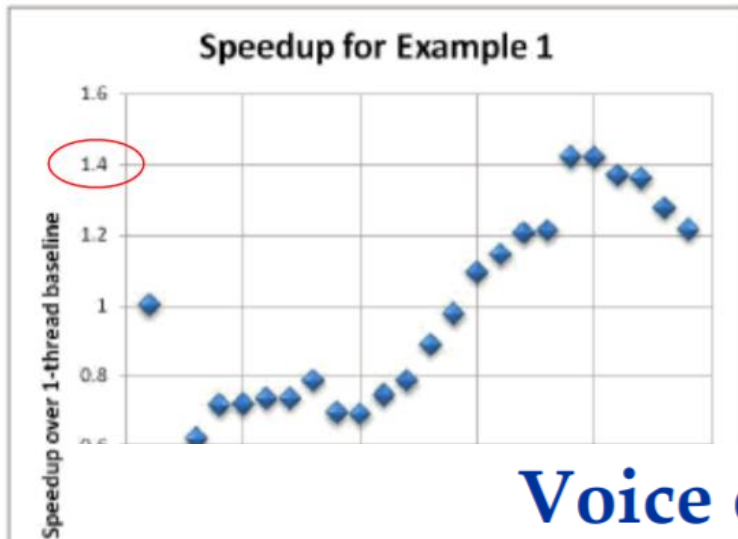
- ❑ Basic implementation: A bit vector per cache line
 - P bits per cache line for P cores (which core has a copy?)
 - 1 additional dirty bit
- ❑ Many possible optimizations!
 - e.g., bloom filter for smaller bit vector size
 - (false positive invalidation message is still safe)



Performance issue with cache coherence: False sharing

- ❑ Different memory locations, written to by different cores, mapped to same cache line
 - Core 1 performing “results[0]++;”
 - Core 2 performing “results[1]++;”
- ❑ Remember cache coherence
 - Every time a cache is written to, all other instances need to be invalidated!
 - “results” variable is ping-ponged across cache coherence every time
 - Bad when it happens on-chip, terrible over processor interconnect (QPI/UPI)
- ❑ Solution: Store often-written data in local variables

Some performance numbers with false sharing



Voice of Experience

Joe Duffy at Microsoft:

During our Beta1 performance milestone in Parallel Extensions, most of our performance problems came down to stamping out false sharing in numerous places.

With False Sharing

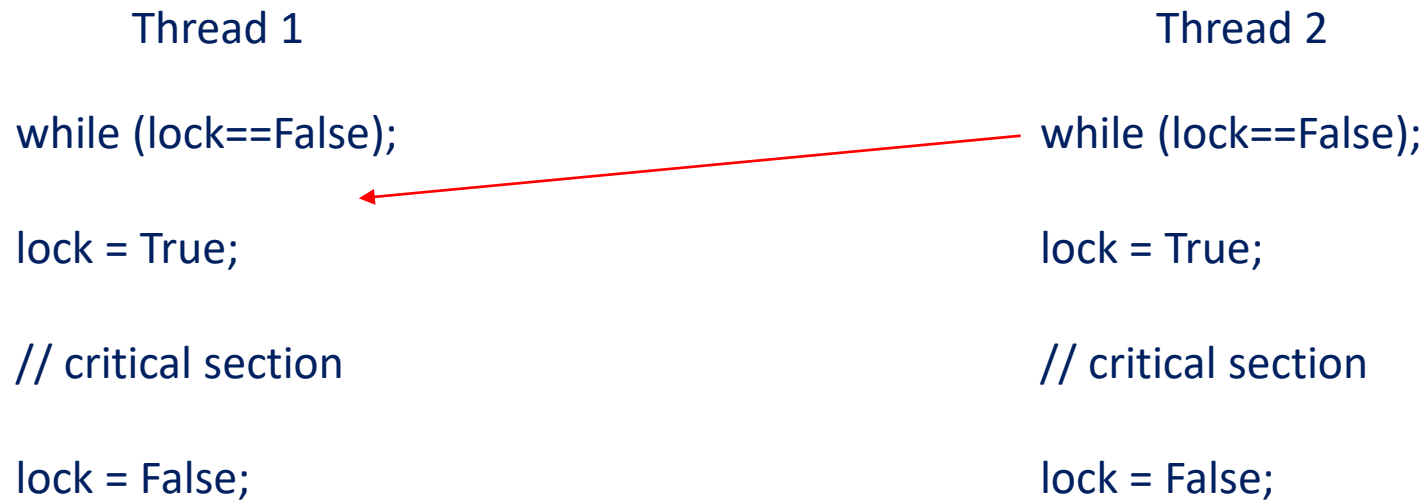
Without False Sharing

Hardware support for synchronization

- ❑ In parallel software, critical sections implemented via mutexes are critical for algorithmic correctness
- ❑ Can we implement a mutex with the instructions we've seen so far?
 - e.g.,
while (lock==False);
lock = True;
// critical section
lock = False;
 - Does this work with parallel threads?

Hardware support for synchronization

- ❑ By chance, both threads can think lock is not taken
 - e.g., Thread 2 thinks lock is not taken, before thread 1 takes it
 - Both threads think they have the lock



Algorithmic solutions exist! Dekker's algorithm, Lamport's bakery algorithm...

Hardware support for synchronization

- ❑ A high-performance solution is to add an “atomic instruction”
 - Memory read/write in a single instruction
 - No other instruction can read/write between the atomic read/write
 - e.g., “if (lock=False) lock=True”

Single instruction read/write is in the grey area of RISC paradigm...

RISC-V example

- ❑ Atomic instructions are provided as part of the “A” (Atomic) extension
- ❑ Two types of atomic instructions
 - Atomic memory operations (read, operation, write)
 - operation: swap, add, or, xor, ...
 - Pair of linked read/write instructions, where write returns fail if memory has been written to after the read
 - More like RISC!
 - With bad luck, may cause livelock, where writes always fail
- ❑ Aside: It is known all synchronization primitives can be implemented with only atomic compare-and-swap (CAS)
 - RISC-V doesn't define a CAS instruction though

Pipelined implementation of atomic operations

- ❑ In a pipelined implementation, even a single-instruction read-modify-write can be interleaved with other instructions
 - Multiple cycles through the pipeline

- ❑ Atomic memory operations
 - Modify cache coherence so that once an atomic operation starts, no other cache can access it
 - Other solutions?

CS250P: Computer Systems Architecture

Memory Consistency Introduction



Sang-Woo Jun

Fall 2022



Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Memory problems with multiprocessing

❑ Cache coherency (The two CPU example)

- Informally: Read to each address must return the most recent value
- Complex and difficult with many processors
- Typically: All writes must be visible at some point, and in proper order



❑ Memory consistency (The two processes example)

- How updates to different addresses become visible (to other processors)
- Many models define various types of consistency
 - Sequential consistency, causal consistency, relaxed consistency, ...
- In our previous example, some models may allow “10” to happen, and we must program such a machine accordingly

Memory consistency litmus test

❑ What are the possible outcomes from the two following codes?

- A and B are initially zero

Processor 1:

1: A = 1;
2: print B

Processor 2:

3: B = 1;
4: print A

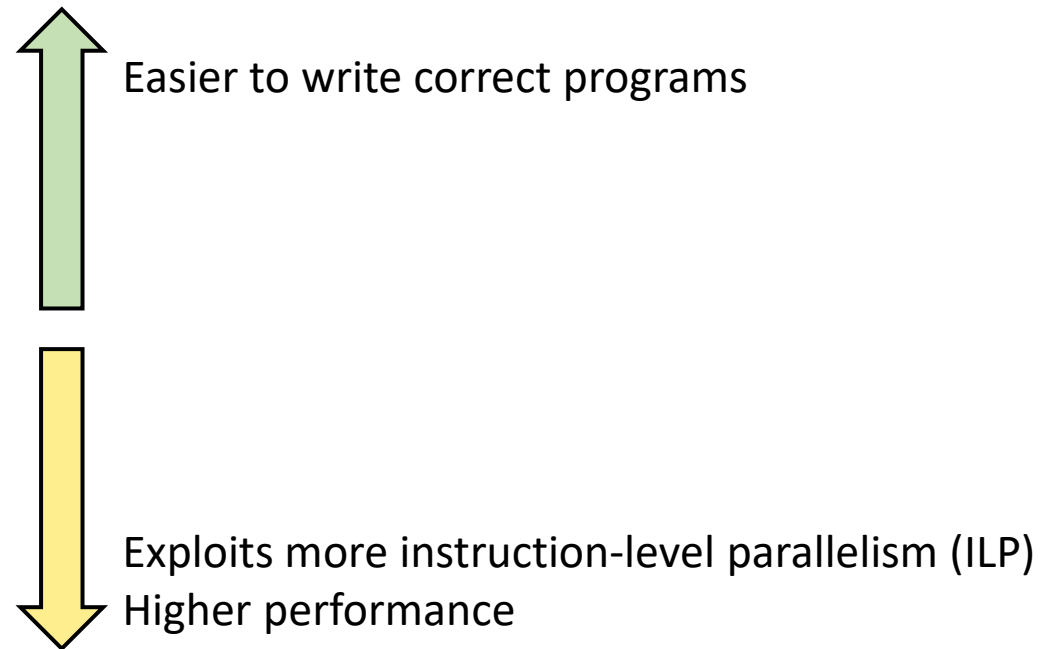
- 1,2,3,4 or 3,4,1,2 etc : “01”
- 1,3,2,4 or 1,3,4,2 etc : “11”
- Can it print “10”, or “00”? Should it be able to?

“Memory model” defines what is possible and not
(Balance between performance and ease of use)

Wide range of consistency models

☐ In rough order of strictness:

- ☒ Sequential consistency
- ☐ Total store order
- ☐ Causal consistency
- ☐ Processor consistency
- ☐ Release consistency
- ☐ Eventual consistency
- ☐ ...



Most strict: Sequential consistency (SC)

- ❑ The most “sane” model.
- ❑ Results are the same as /some/ sequential order between operations
 - Effects of each instruction is manifested, and visible to subsequently scheduled instruction, in every processor
 - e.g., 1: A=1 is visible to 2 or 4 if they are scheduled after

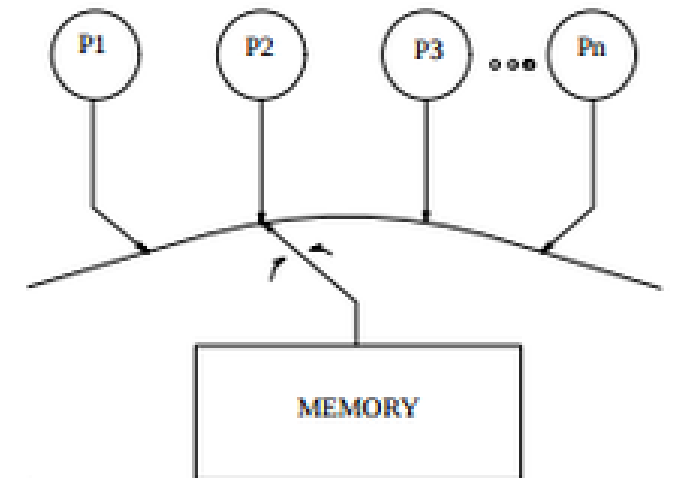
Processor 1:

1: A = 1;
2: print B

Processor 2:

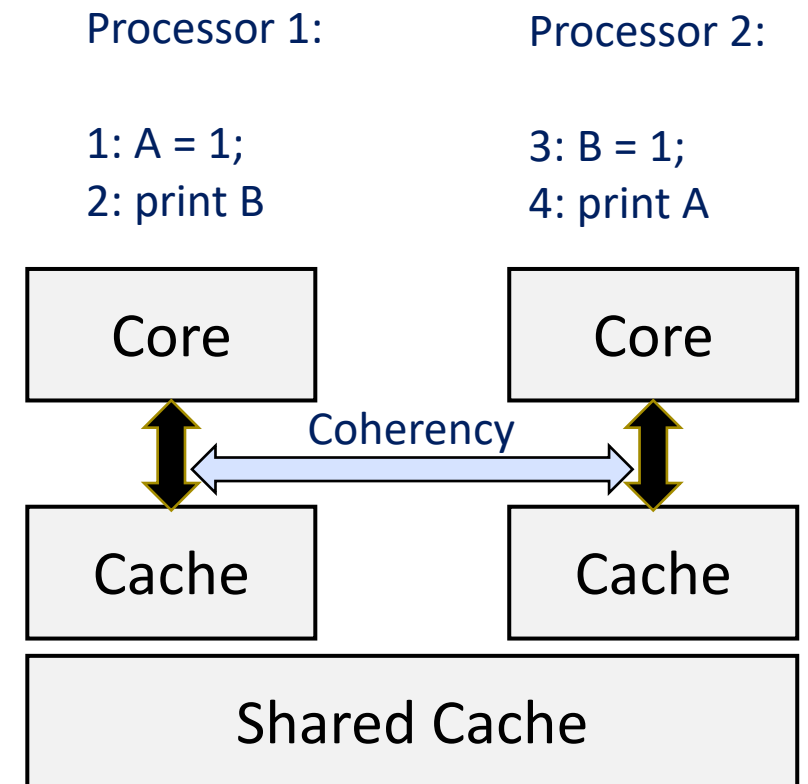
3: B = 1;
4: print A

Why do we need anything else?!



Why not SC? Distributed caches are slow...

- ❑ Writes to shared memory must be propagated to all other cores/caches!
 - Through coherency and/or shared caches
 - Through TLB! Through VM!
 - Even L1 cache! Where every cycle is precious
- ❑ If we sacrifice a little bit of consistency
 - Can we get a lot more performance?
 - Will the programming still be sane?



A slightly relaxed model: Total Store Order

❑ Let's say we add a small “store buffer” between core and cache

- Writes are cached in store buffer, applied later in-order
- All reads first scan store buffer before going to L1 cache
- Writes in store buffer is not coherent, not propagated to other cores
- Typically small! (32 slots?)

❑ Write instructions need not block the system!

- 2 and 4 can execute without waiting for
- 1 and 3 to propagate through the cache

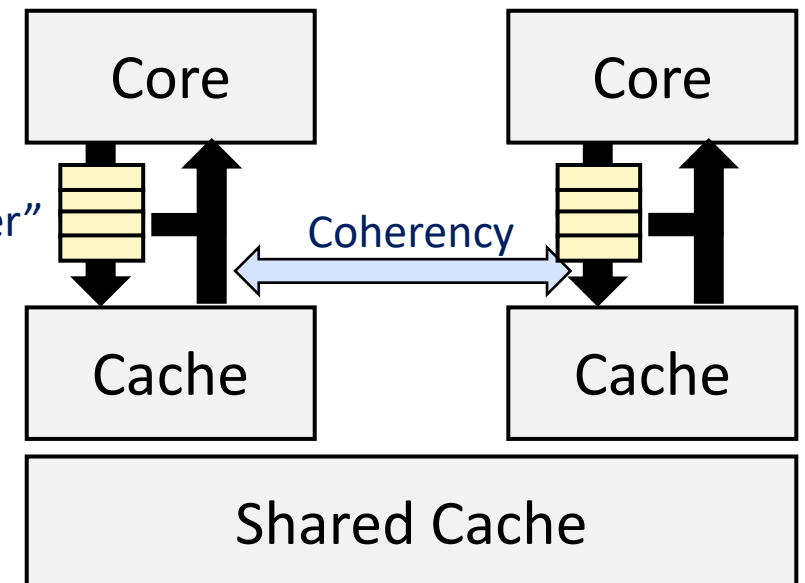
Processor 1:

1: A = 1;
2: print B

Processor 2:

3: B = 1;
4: print A

Multiple times faster performance in practice!



A slightly relaxed model: Total Store Order

❑ With a store buffer, can “00” happen?

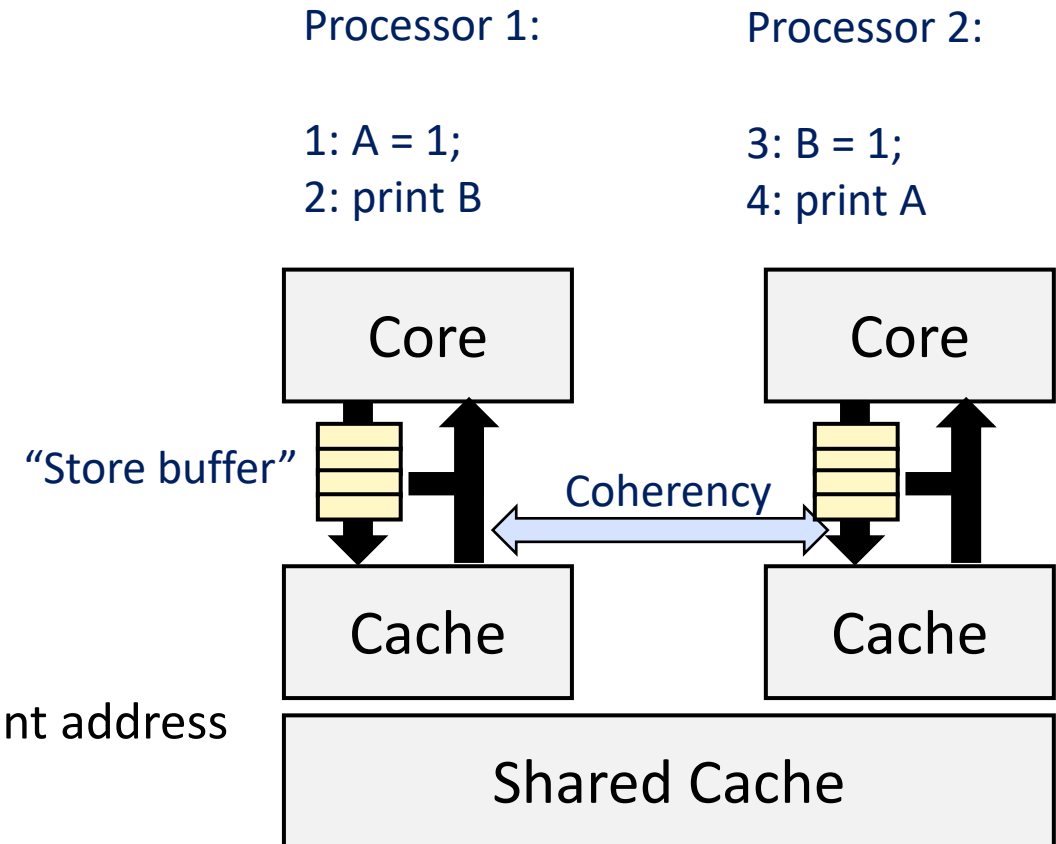
Yes! Neither 1 or 3 propagated

❑ Can “10” happen?

Yes! e.g., 3 propagated, but not 1

❑ Slightly more formally:

- TSO allows “reordering of load and stores”
 - Stores can be delayed beyond loads from different address
 - “00” Scenario: 2,4,1,3. “10”: 1,4,2,3
- Single thread behavior is maintained.
- But multi-thread is strange now!



TSO In the real world: x86

- ❑ Intel, ARM x86 is known to implement TSO-like model
 - But not “strictly” TSO.
 - Model is hardware design-driven, not driven by a mathematical model
- ❑ Programming manual provides series of “litmus tests”
 - Emphasize unexpected behaviors
- ❑ Many attempts to formalize the observed behavior into a clean model
 - e.g., Sewell et. al., “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”, CACM 2010

How can we write sane programs with such lack of restrictions?!

(Part of the) solution: Barrier instructions!

- ❑ Sometimes called “fence” instruction
- ❑ Enforces all state changes before it to propagate
 - In the TSO example, flushes the store buffer on a fence instruction
- ❑ Regardless of memory model, calling “fence” after EVERY instruction recreates sequential consistency
 - But would be slow
 - Only used when inter-thread communication/state is important
- ❑ Real world: x86 provides three fence instructions
 - LFENCE (orders loads), SFENCE (orders stores), MFENCE (orders both)
 - But TSO only reorders writes? x86 is not strict TSO!

Even weaker, less formal models

- ❑ RISC-V has memory model extensions
 - Default is RVWMO (RISC-V Weak Memory Order)
 - ZTSO extension for TSO, likely more
- ❑ SPARC allows selection between three
 - TSO, Partial Store Order (PSO), Weak
- ❑ ARM implements weak, very undocumented memory model
- ❑ “Weak”?
 - Allows load->store, store->load, load->load, store->store reordering
 - (Only to different addresses, of course)
 - How far can it reorder? Millions of instructions? Probably not...
 - Compilers need to know, to minimize fences and get high performance!

Some omitted complexities

- ❑ Reordering can be speculative!
 - TSO can reorder loads->loads and revert if remote write turn out to disallow it
- ❑ Out-of-Order design can also be influenced
 - OoO does not need to completely reconstruct program order!
- ❑ More complex terminology to describe memory models
 - Acquire/Release, Commit/Reconcile/Fences, ...
- ❑ Behavior of various weaker models
 - Partial Store Order, Processor Consistency, ...

Behaviors of existing memory models

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	AMD64	IA-64	zSeries
Loads reordered after loads	✓	✓	✓	✓	✓					✓	
Loads reordered after stores	✓	✓	✓	✓	✓					✓	
Stores reordered after stores	✓	✓	✓	✓	✓	✓				✓	
Stores reordered after loads	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic reordered with loads	✓	✓		✓	✓					✓	
Atomic reordered with stores	✓	✓		✓	✓	✓				✓	
Dependent loads reordered	✓										
Incoherent instruction cache pipeline	✓	✓		✓	✓	✓	✓	✓		✓	✓

Hot take: TSO regarded cleanest model with best performance and least silicon...?

Reminder: Amdahl's Law

$$\square \text{ Speedup} = \frac{1}{(1-p) + \frac{p}{s}}$$

- p = proportion of execution time that can be parallelized
- s = speedup of the parallelized portion

Reminder: Amdahl's Law

□ $Speedup = \frac{1}{(1-p) + \frac{p}{s}}$

- p = proportion of execution time that can be parallelized
- s = speedup of the parallelized portion

□ Anecdote

- If parallel portion is “only” 50%
Even infinite processors can only achieve 2x performance!

