CS152: Computer Systems Architecture
Multiprocessing and Parallelism

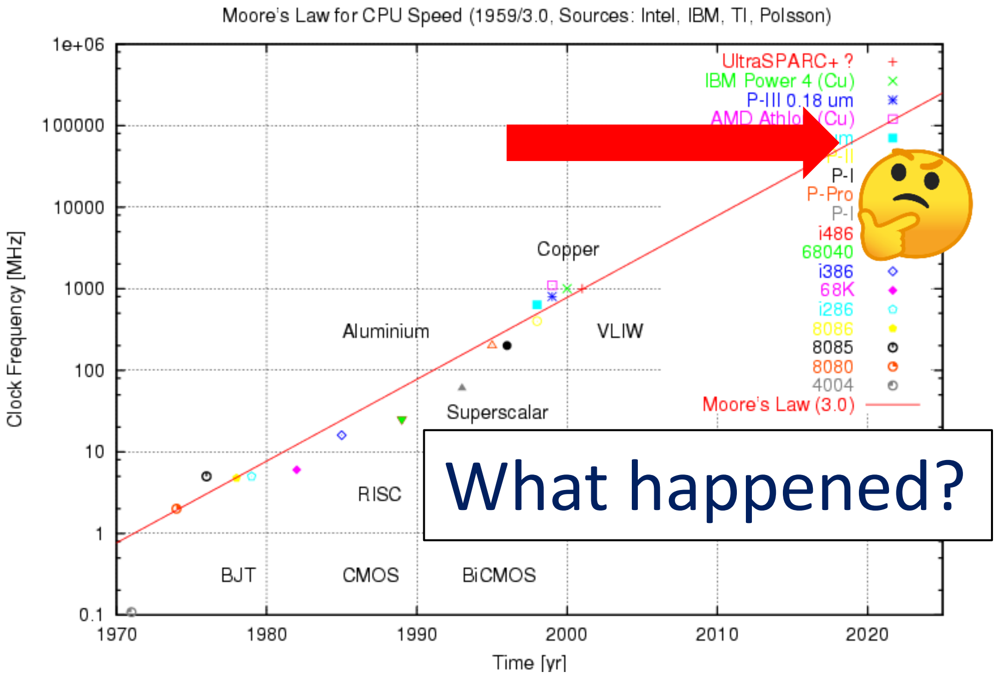
Sang-Woo Jun

Winter 2022

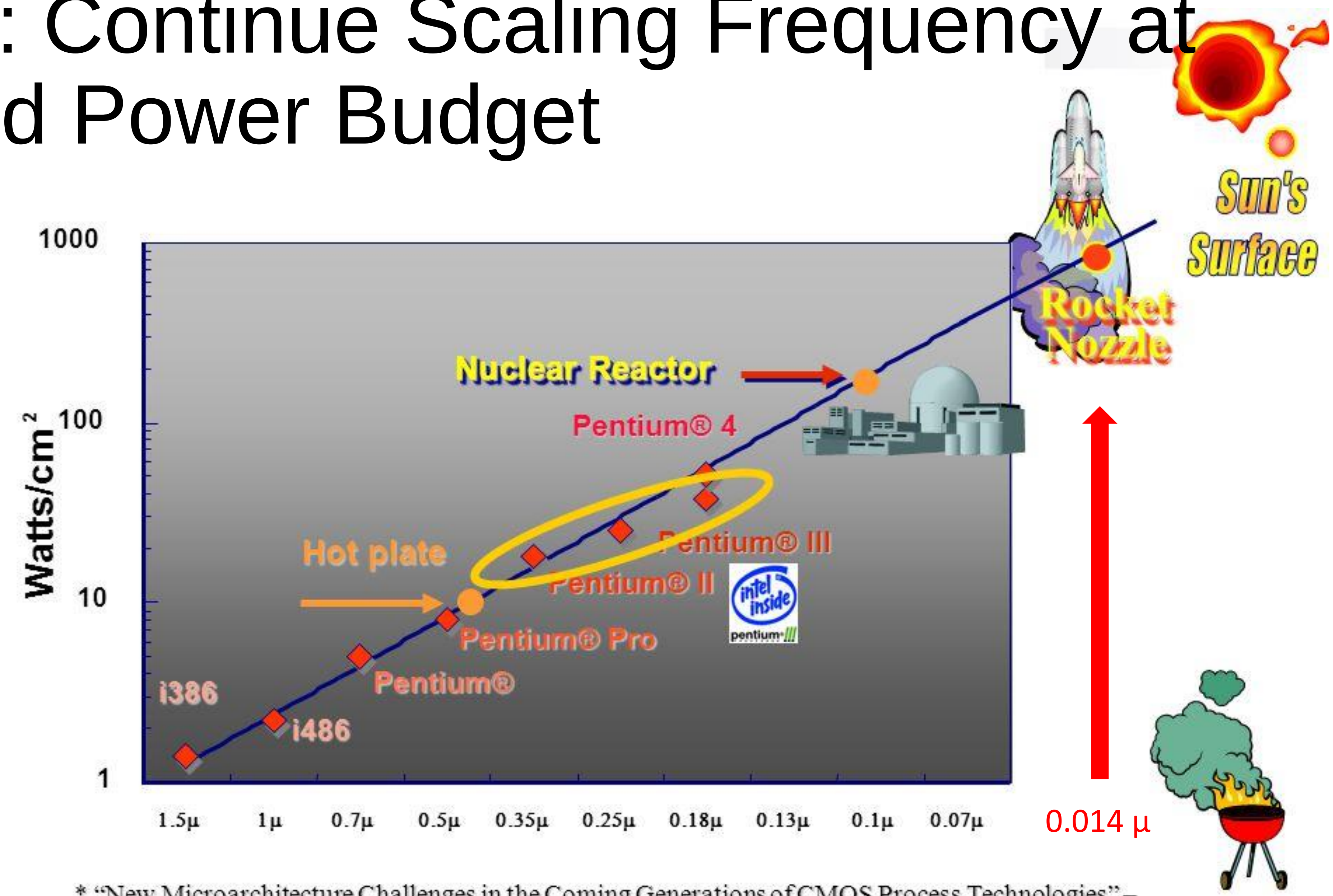
Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

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"

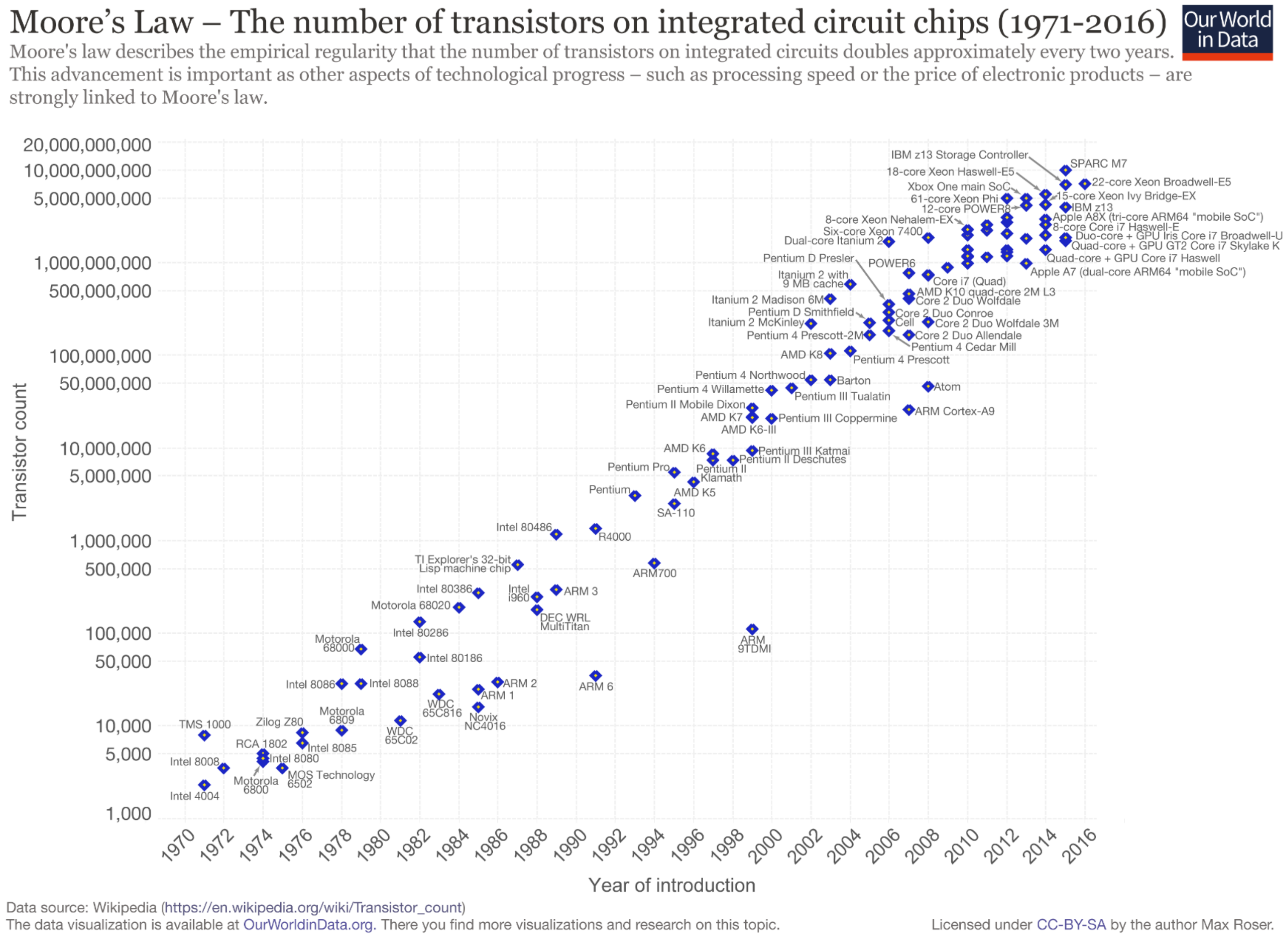


Option 1: Continue Scaling Frequency at Increased Power Budget



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

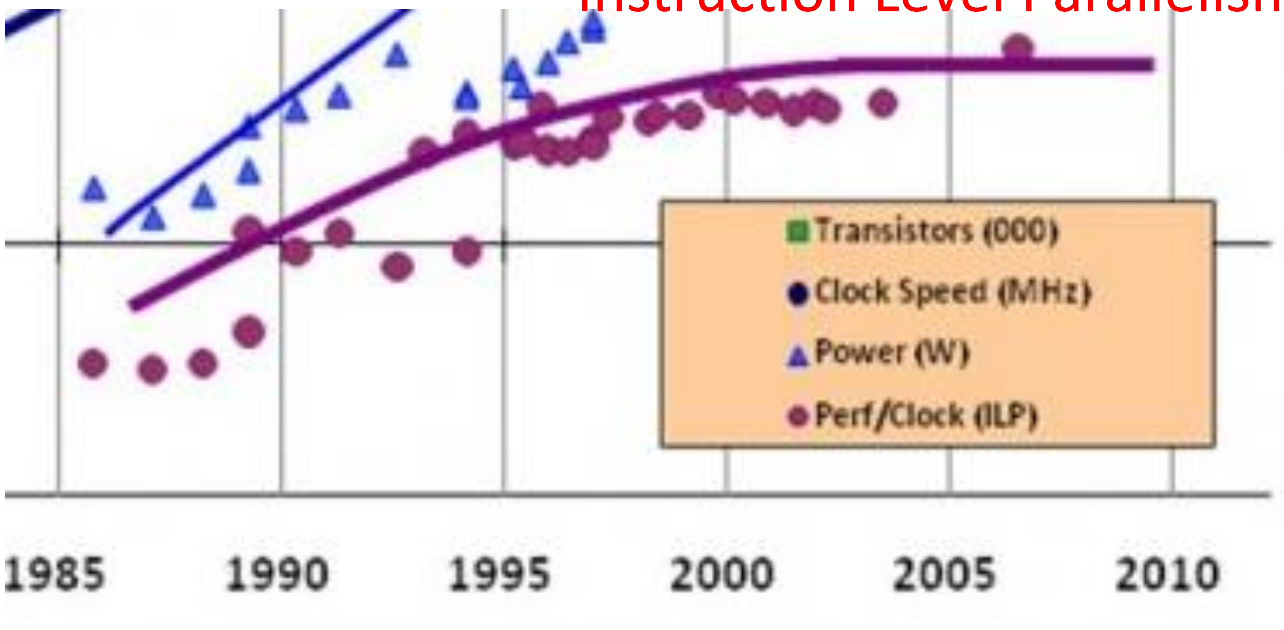
But Moore’s Law Continues Beyond 2006



State of Things at This Point (2006)

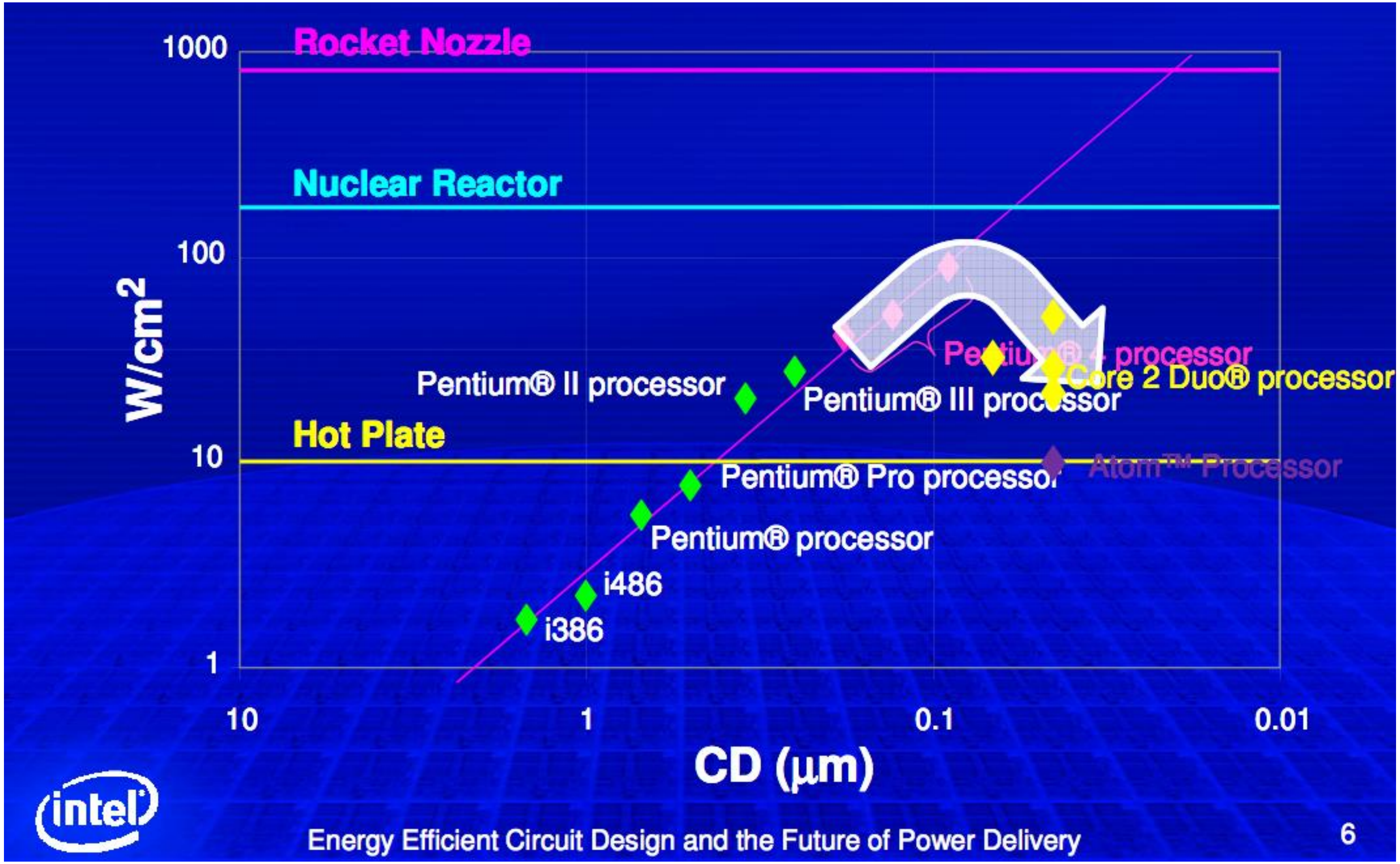
- ❑ Single-thread performance scaling ended
  - o Frequency scaling ended (Dennard Scaling)
  - o Instruction-level parallelism scaling stalled … also around 2005
- ❑ Moore’s law continues
  - o Double transistors every two years
  - o What do we do with them?

Instruction Level Parallelism

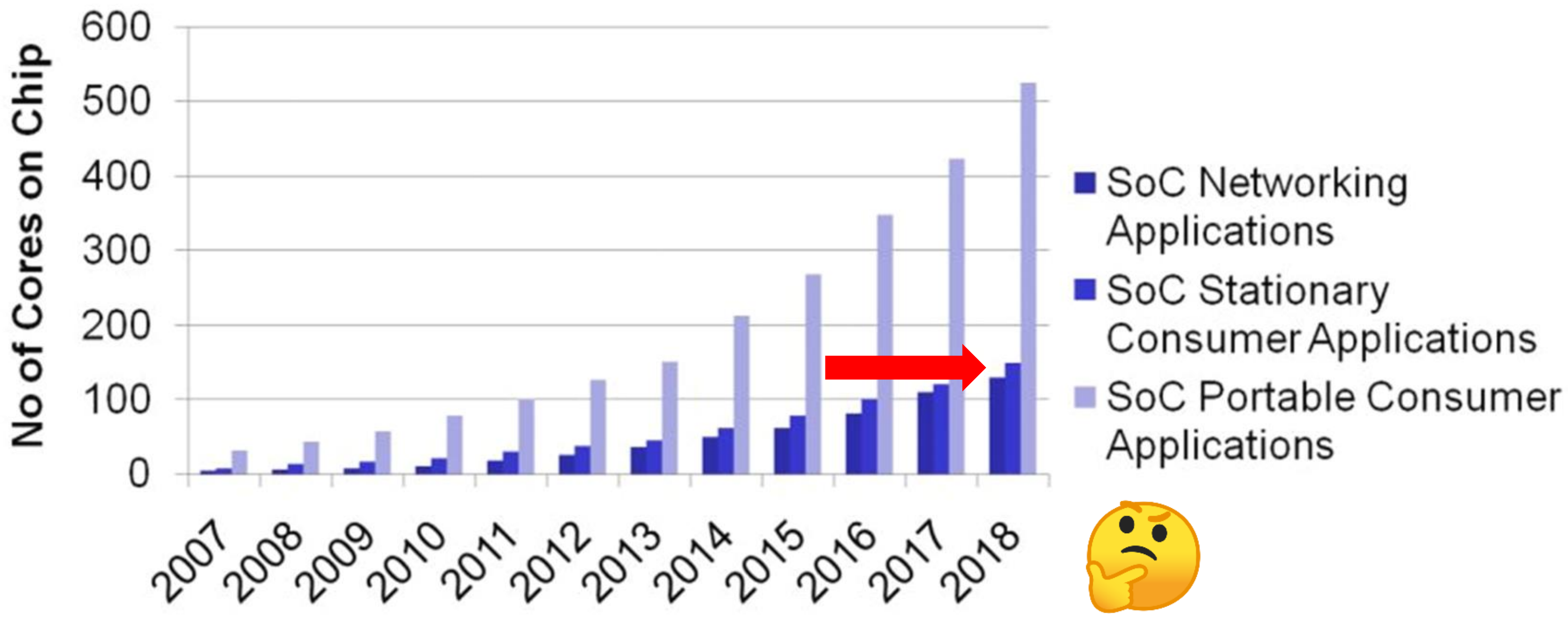


K. Olukotun, “Intel CPU Trends”

Crisis Averted With Manycores?



Crisis Averted With Manycores?



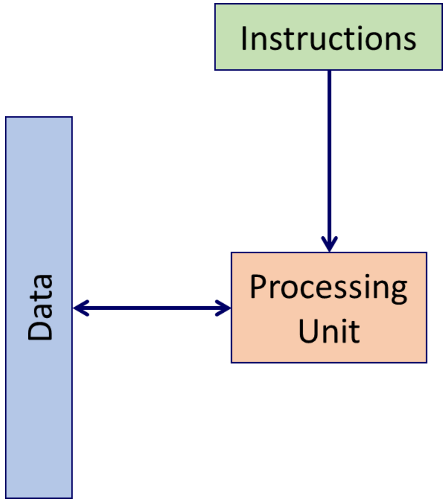
We'll get back to this point later. For now, multiprocessing!

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

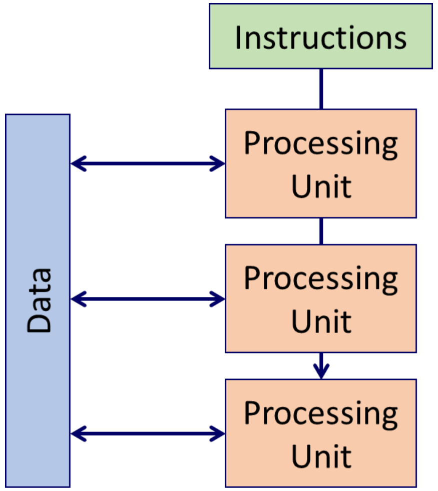
The hardware for parallelism: Flynn taxonomy (1966) recap

		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)

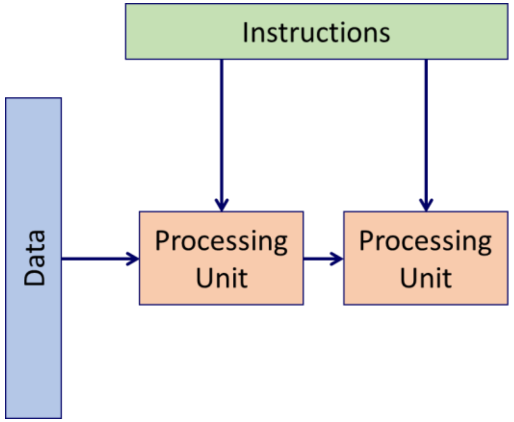
Flynn taxonomy



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

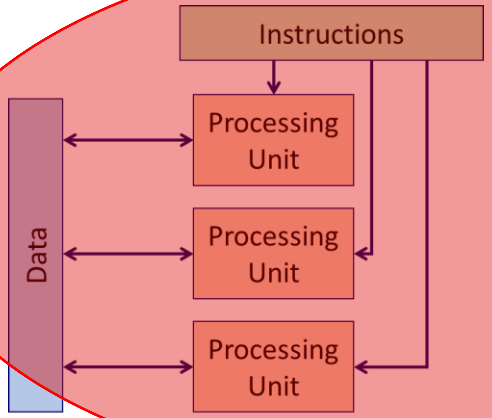


Single-Instruction
Multi-Data
(GPUs, Intel SIMD Extensions)



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

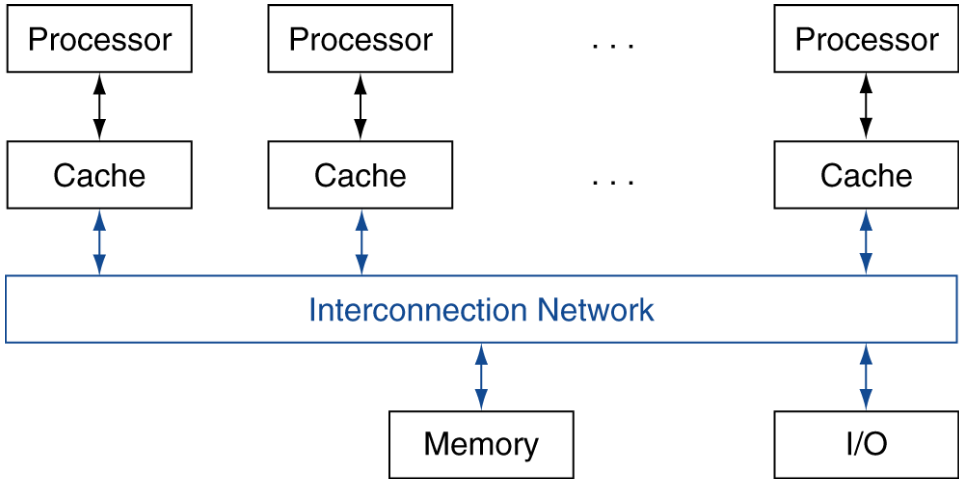
Today



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)

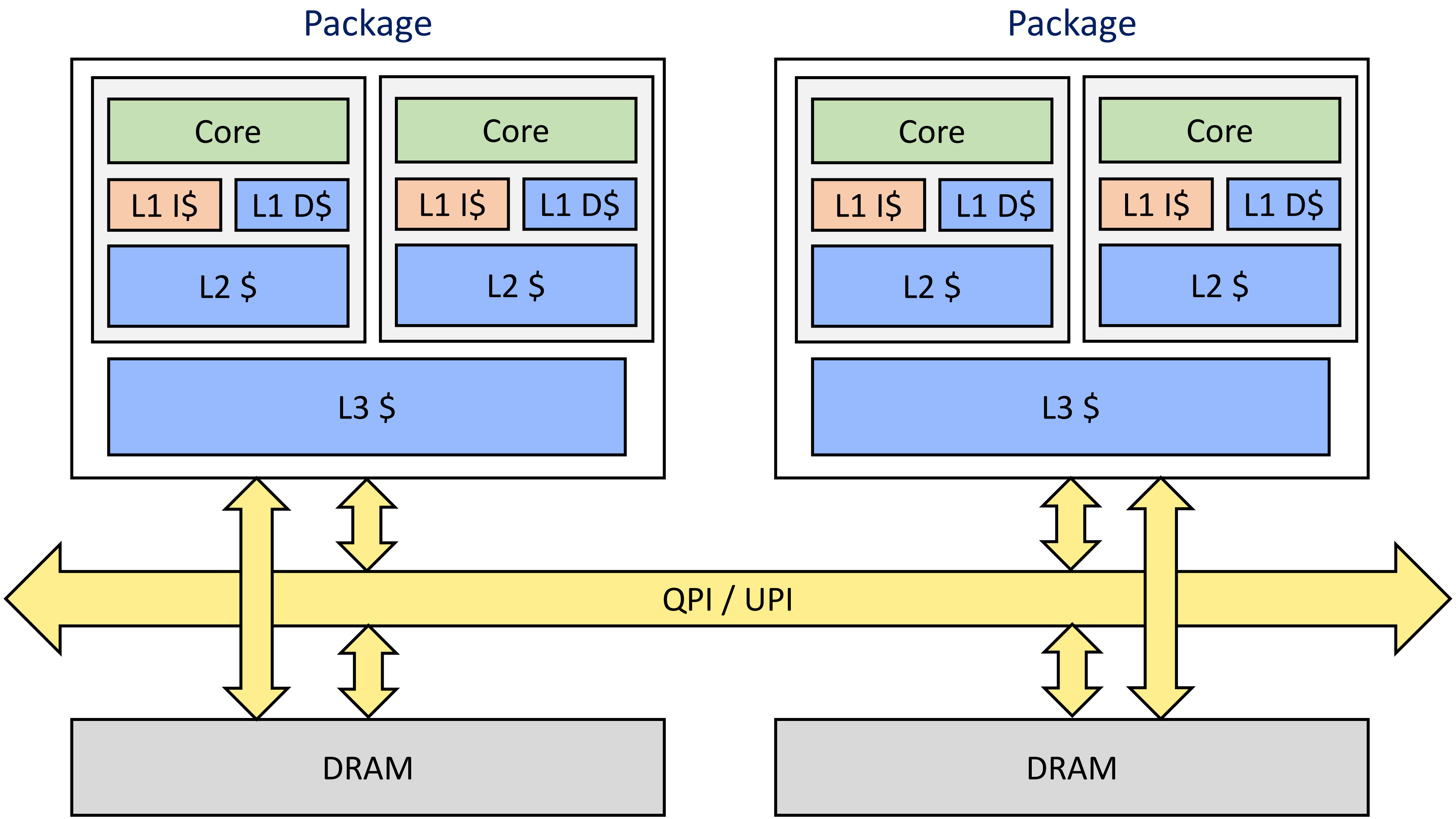


- SMP: Symmetric multiprocessor
  - The processors in the system are identical, and are treated equally

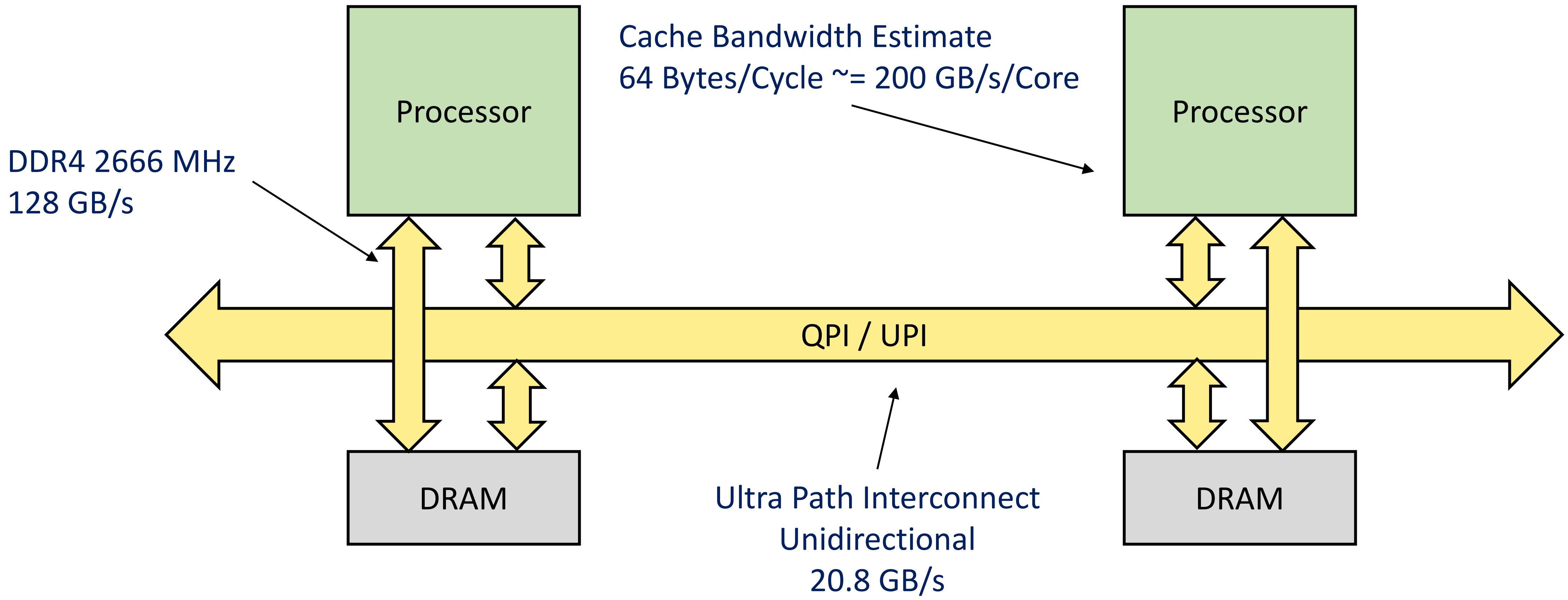
- Typical chip-multiprocessor (“multicore”) consumer computers

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!

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:	Processor 2:
1: A = 1;	3: B = 1;
2: print B	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

Grad level topic…

CS152: Computer Systems Architecture
Cache Coherency Introduction

Sang-Woo Jun

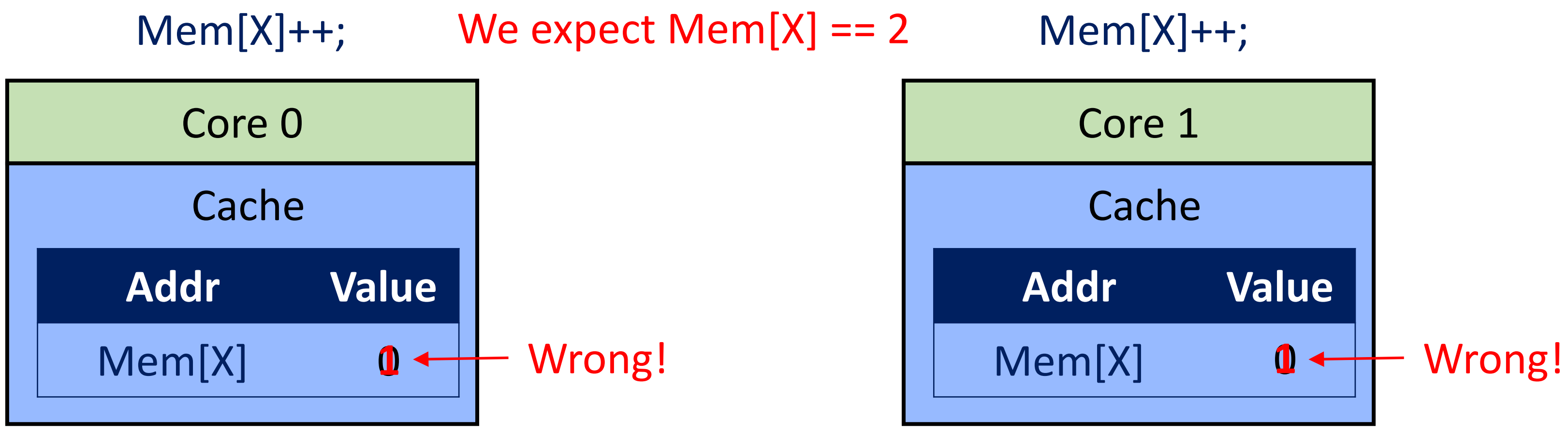
Winter 2022

UCI

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

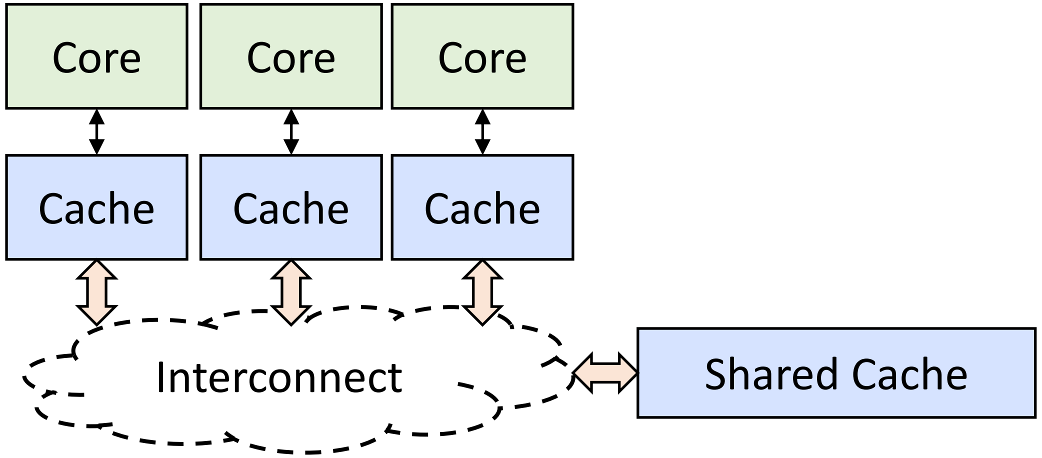
The cache coherency problem

- 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



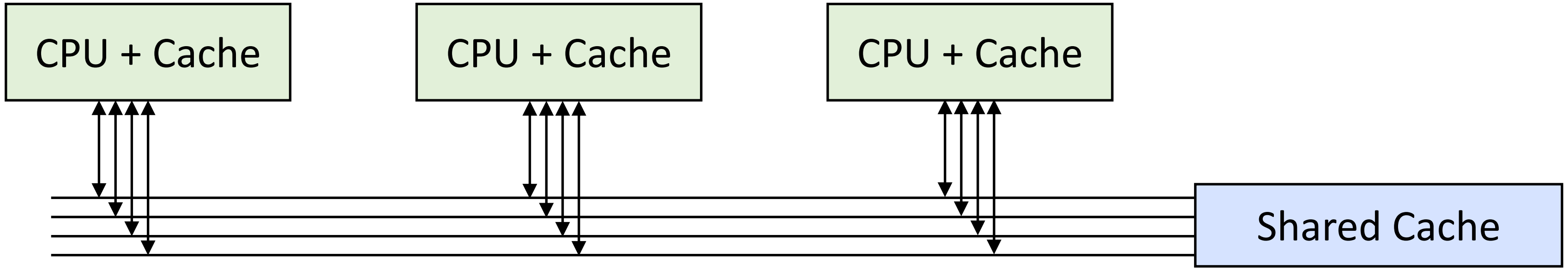
Background: On-chip interconnect

- 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



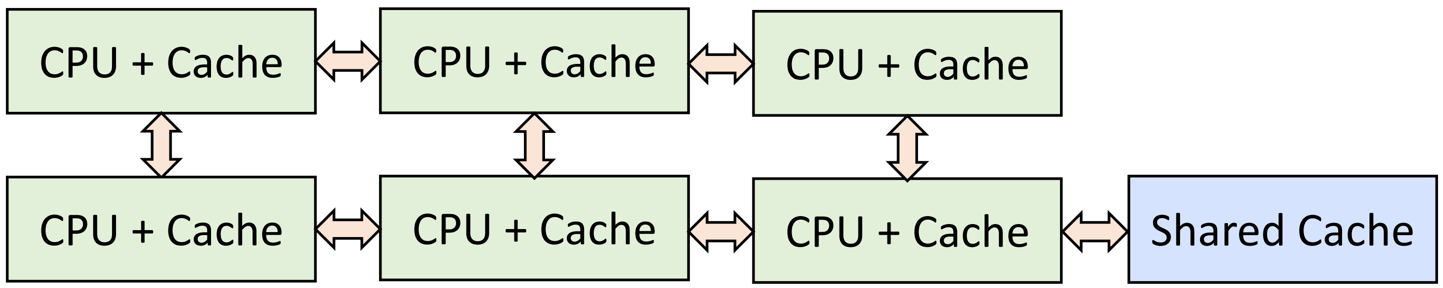
Background: Bus interconnect

- 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

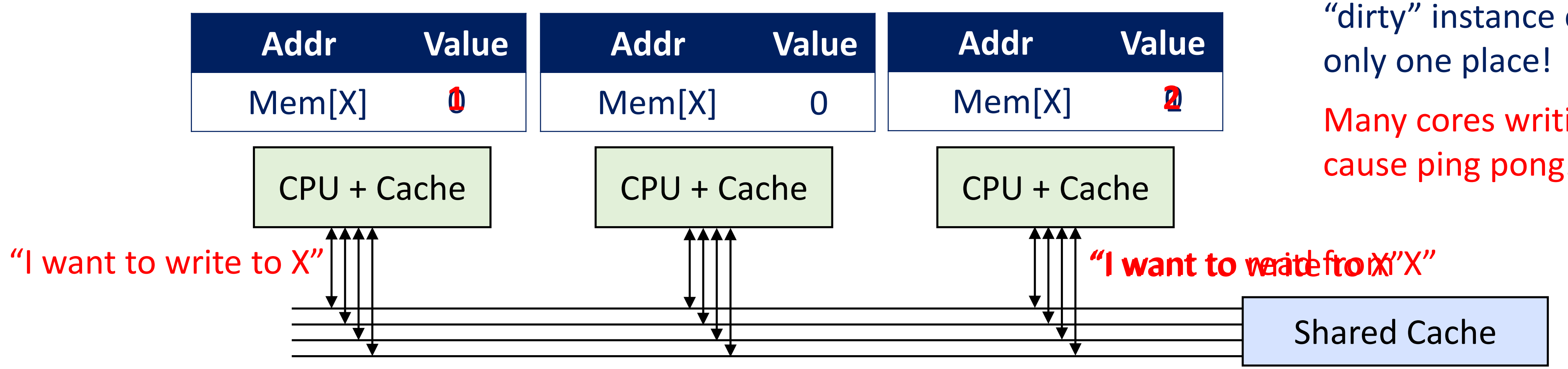
Keeping multiple caches coherent

- 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, …)



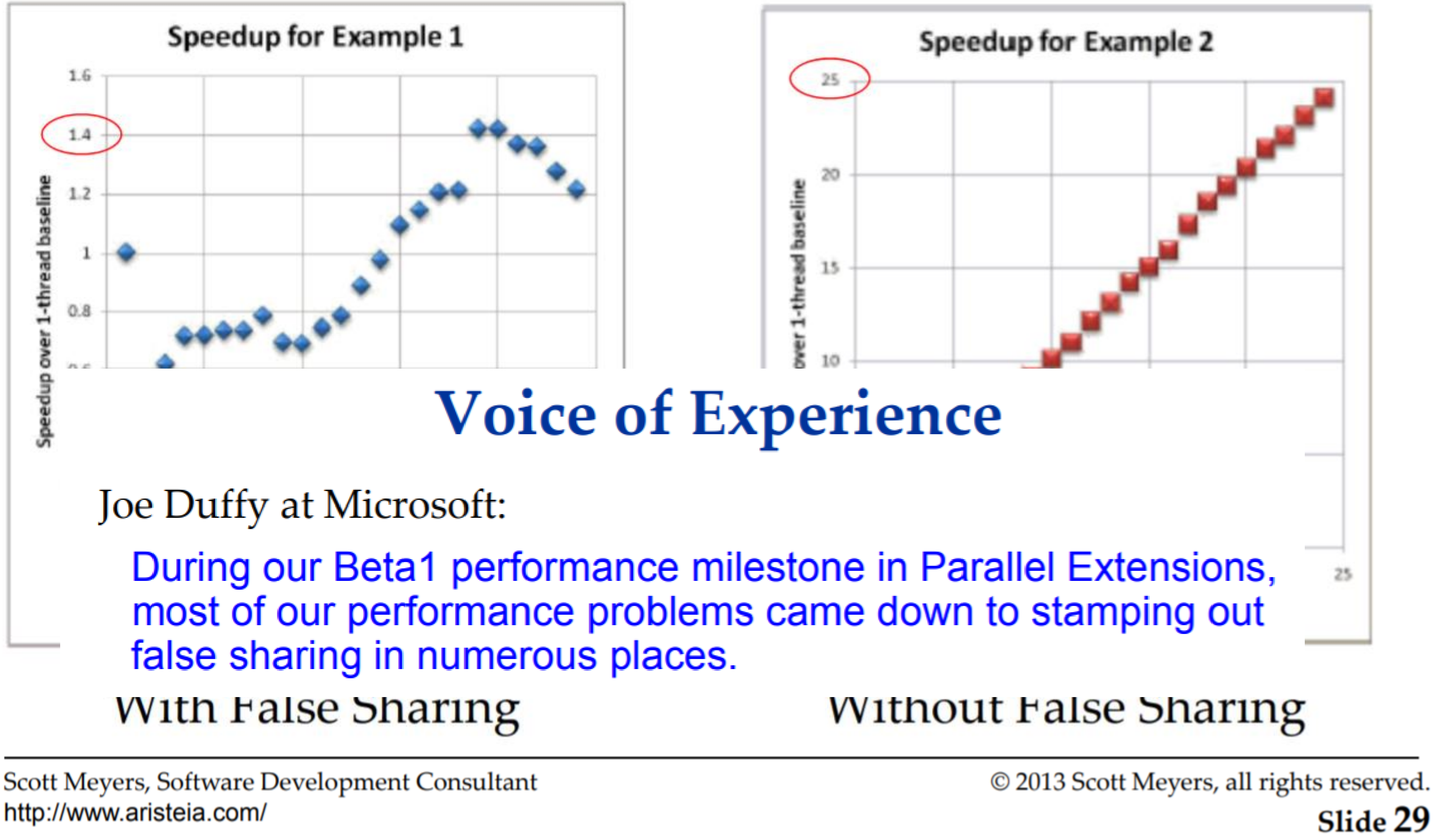
"dirty" instance can exist in only one place!

Many cores writing to X may cause ping pong…

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



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

Thread 1	Thread 2
while (lock==False);	while (lock==False);
lock = True;	lock = True;
// critical section	// critical section
lock = False;	lock = False;

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?