An Introduction to Parallel Architectures

Marko Bertogna

marko.bertogna@unimore.it

[Courtesy: Andrea Marongiu]

Impact of Parallel Architectures

- From cell phones to supercomputers
- In regular CPUs as well as GPUs

1.4GHz Quad core Exynos (ARM)

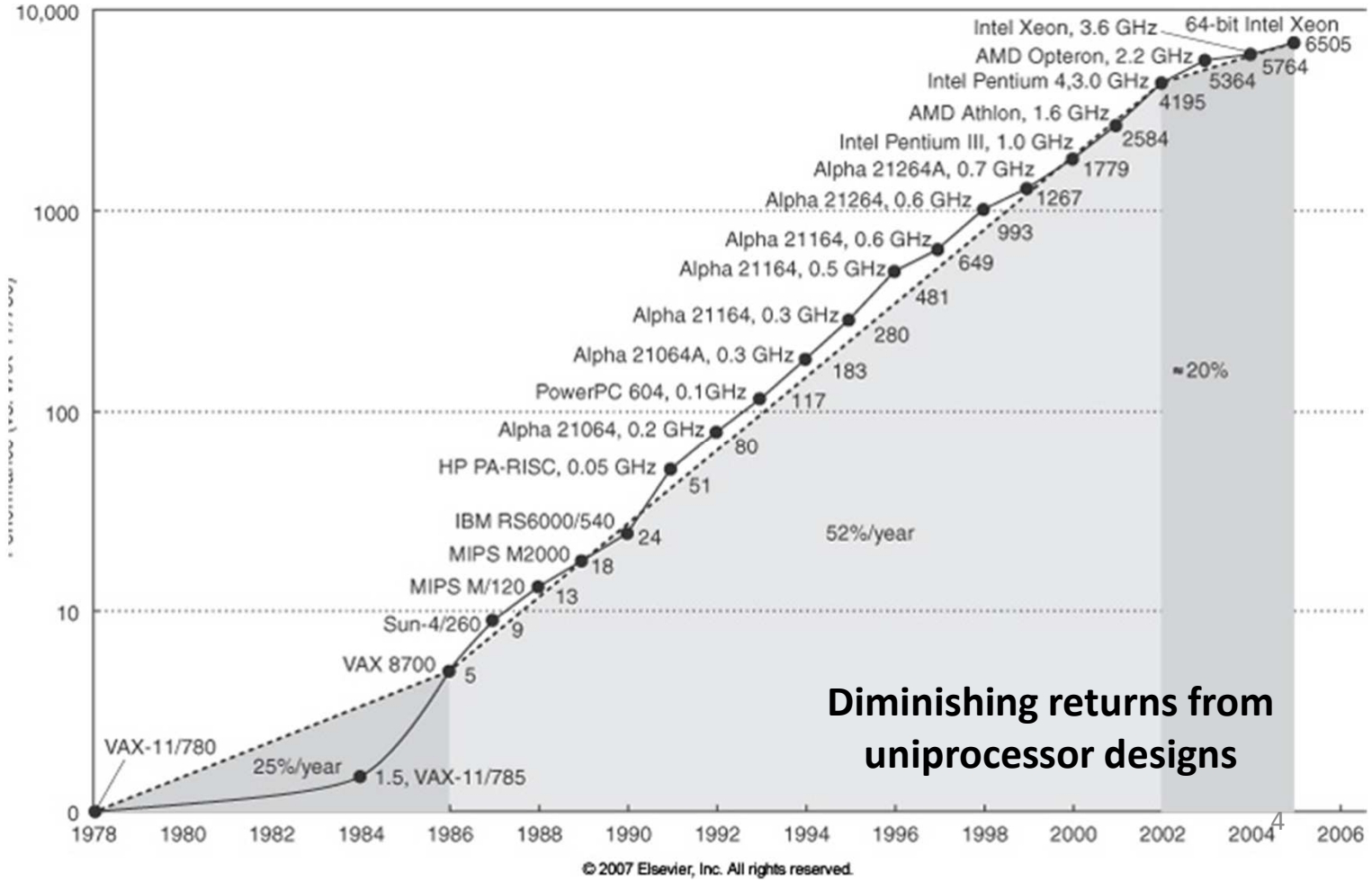
Parallel HW Processing

- Why?
- Which types?
- Which novel issues?

Why Multicores?

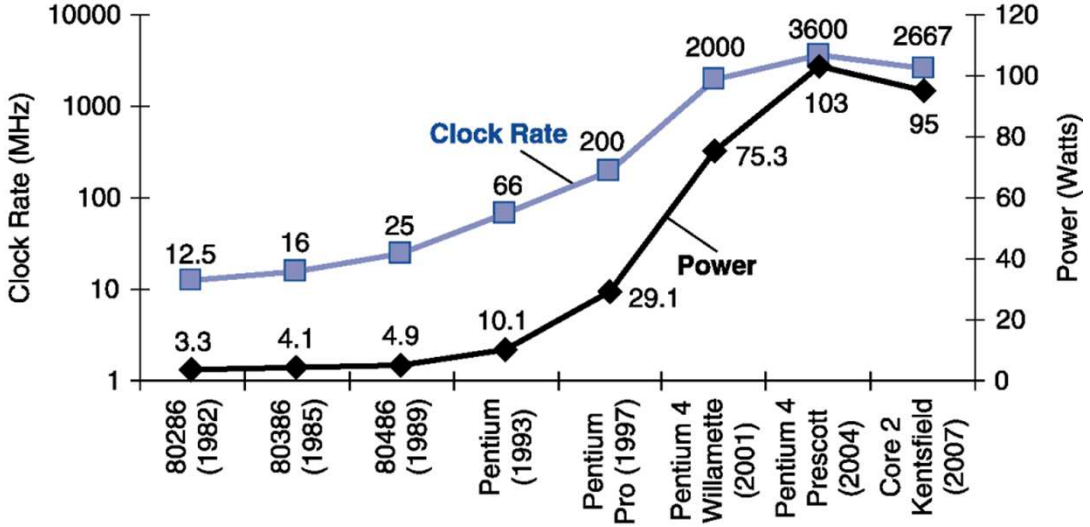
The SPECint performance of the hottest chip grew by 52% per year from 1986 to 2002, and then grew only 20% in the next three years (about 6% per year).

[from Patterson & Hennessy]



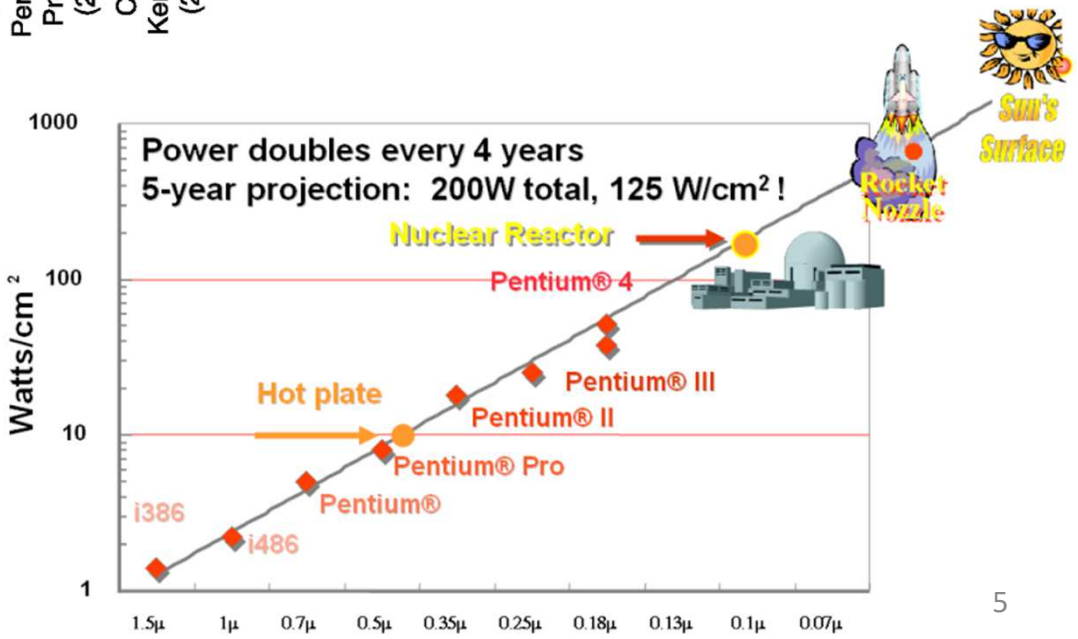
Power Wall

[from Patterson & Hennessy]



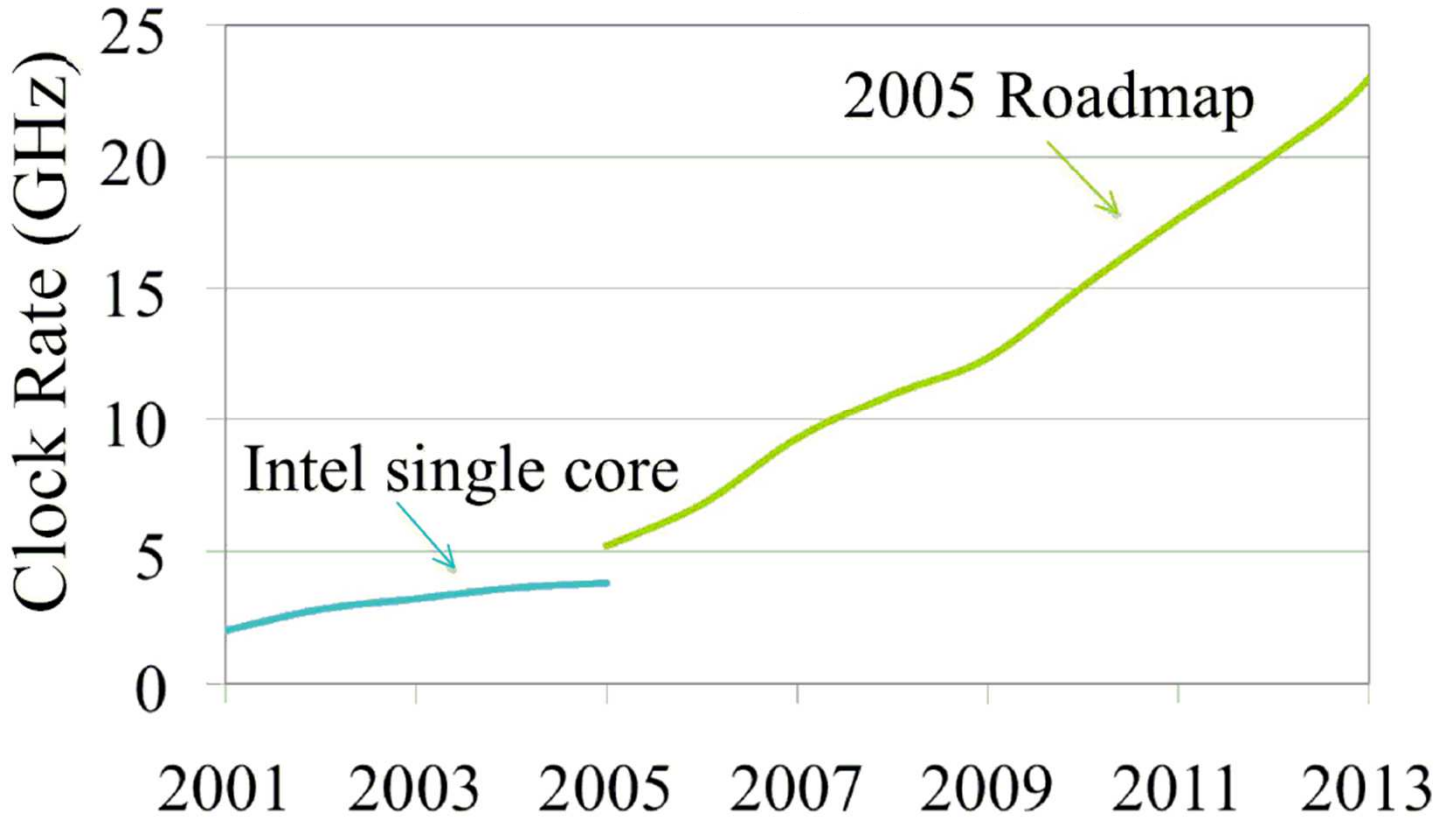
- The design goal for the late 1990's and early 2000's was to drive the clock rate up.
  - by adding more transistors to a smaller chip.

- Unfortunately, this increased the power dissipation of the CPU chip beyond the capacity of inexpensive cooling techniques



Roadmap for CPU Clock Speed: Circa 2005

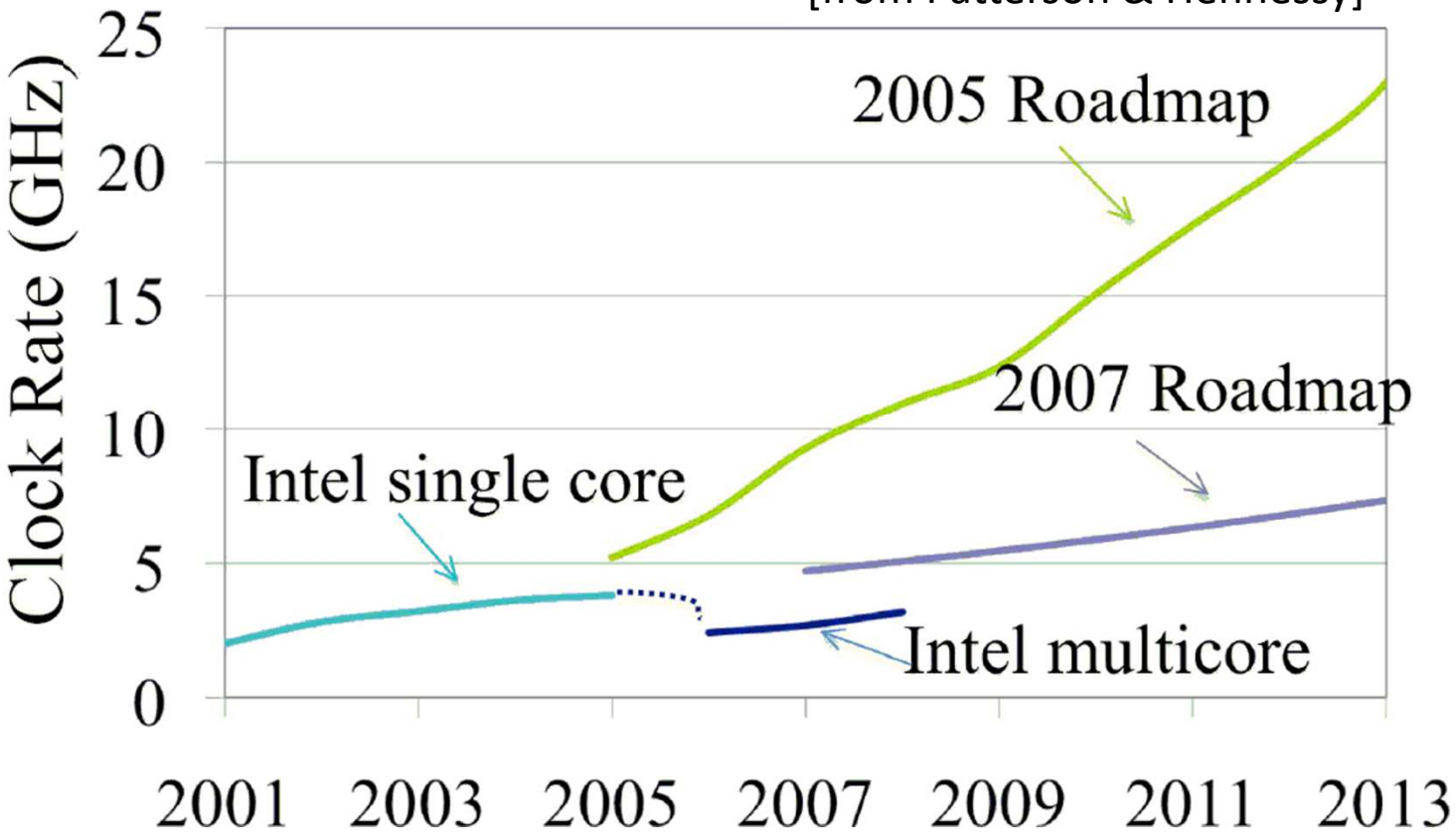
[from Patterson & Hennessy]



Here is the result of the best thought in 2005. By 2015, the clock speed of the top “hot chip” would be in the 20 – 25 GHz range.

The CPU Clock Speed Roadmap (A Few Revisions Later)

[from Patterson & Hennessy]

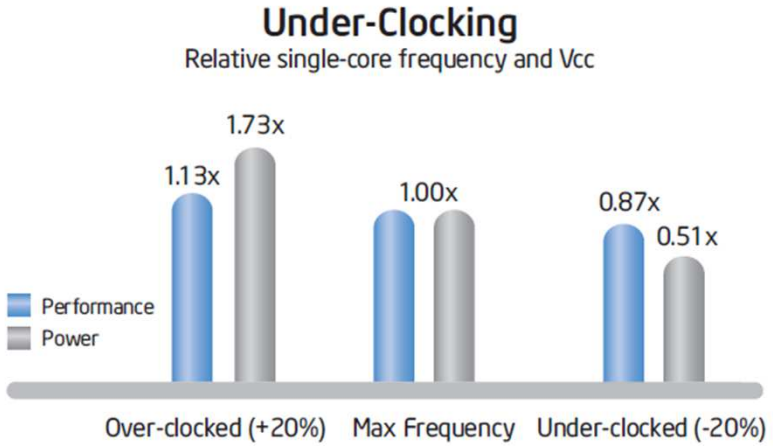


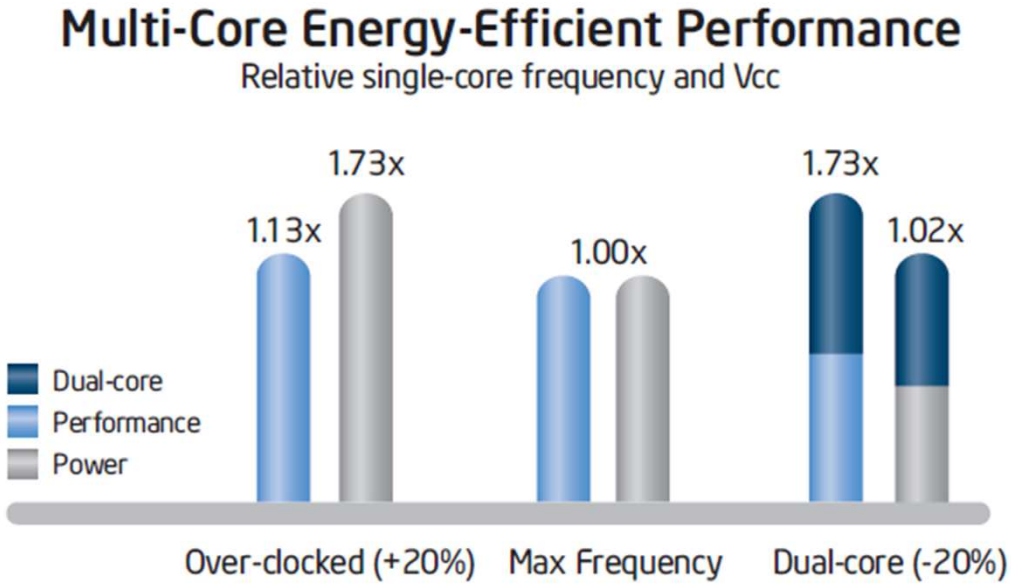
This reflects the practical experience gained with dense chips that were literally "hot"; they radiated considerable thermal power and were difficult to cool.
Law of Physics: All electrical power consumed is eventually radiated as heat.

The Multicore Approach

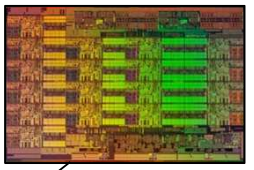
Multiple cores on the same chip

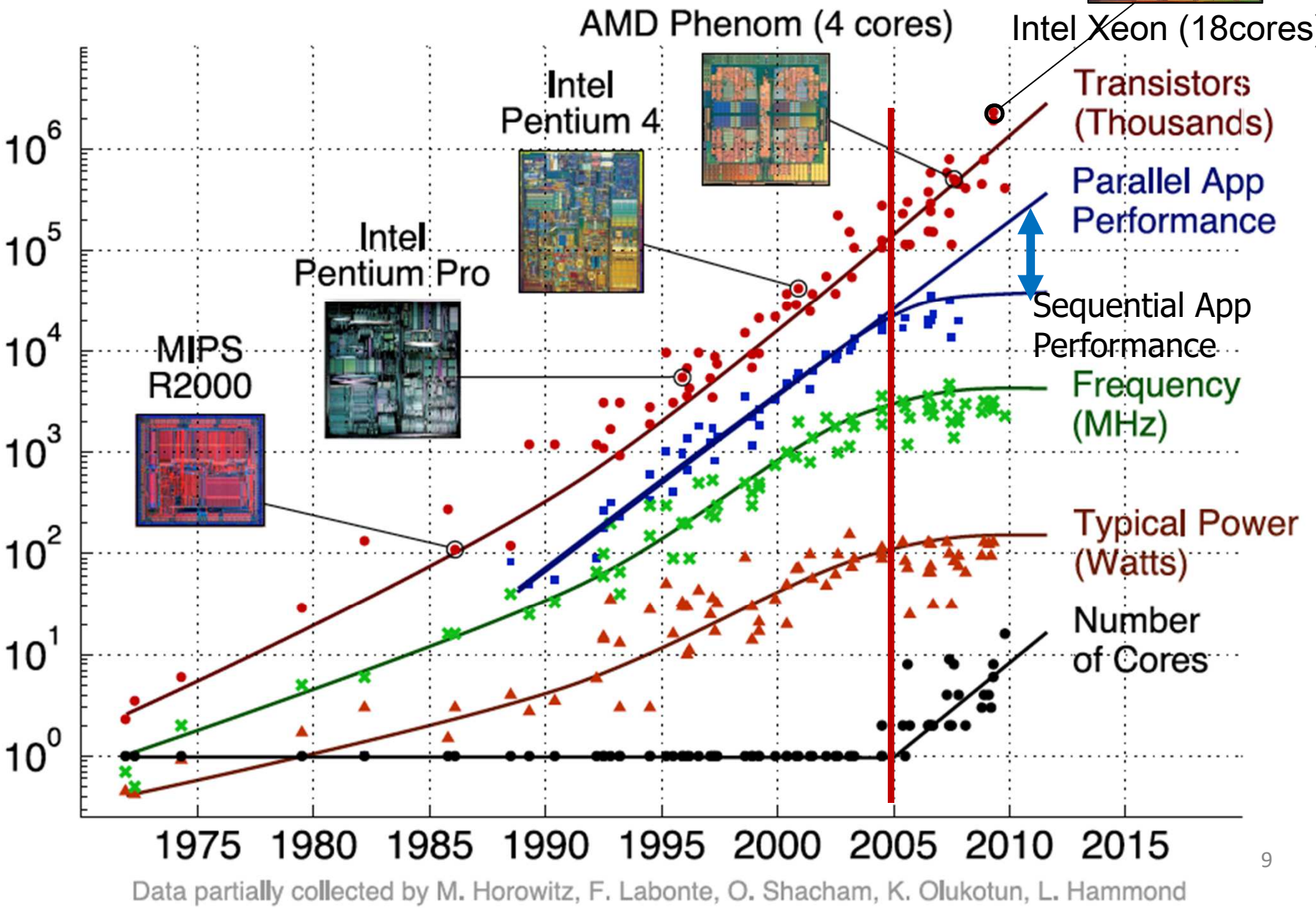
- Simpler
- Slower
- Less power demanding





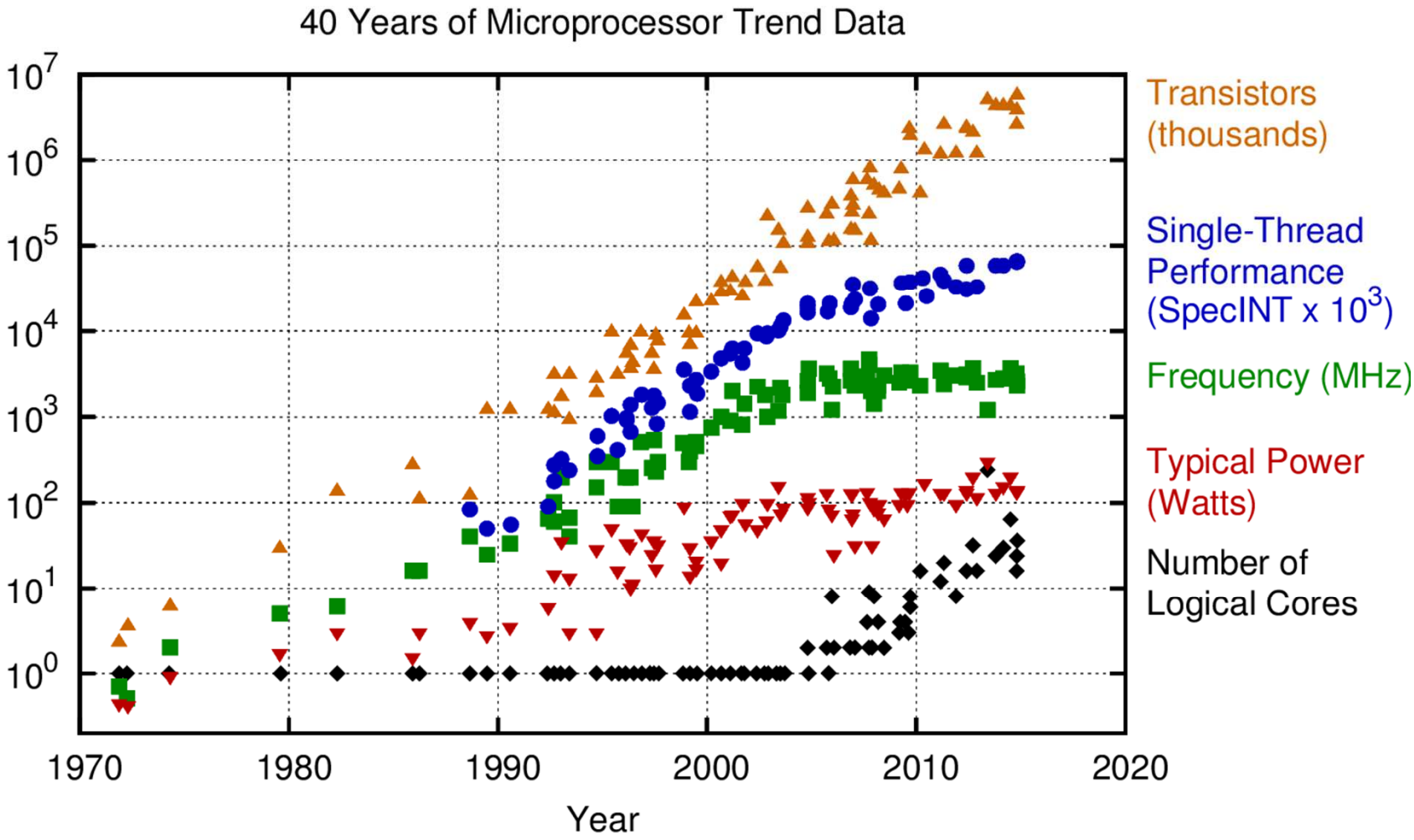
Transition to Multicore





Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

The trend goes on…

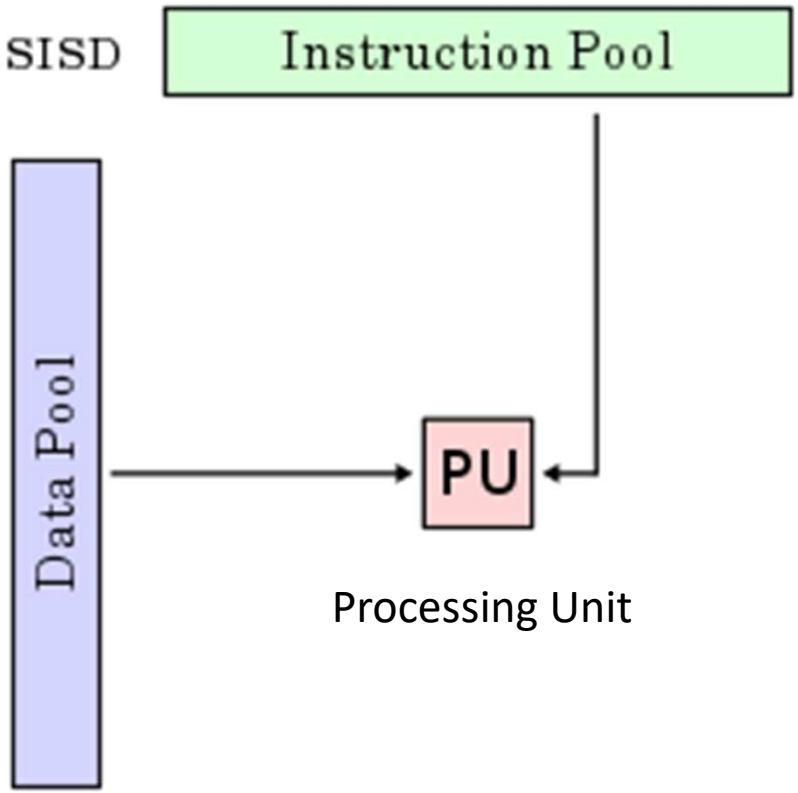


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Flynn Taxonomy of parallel computers

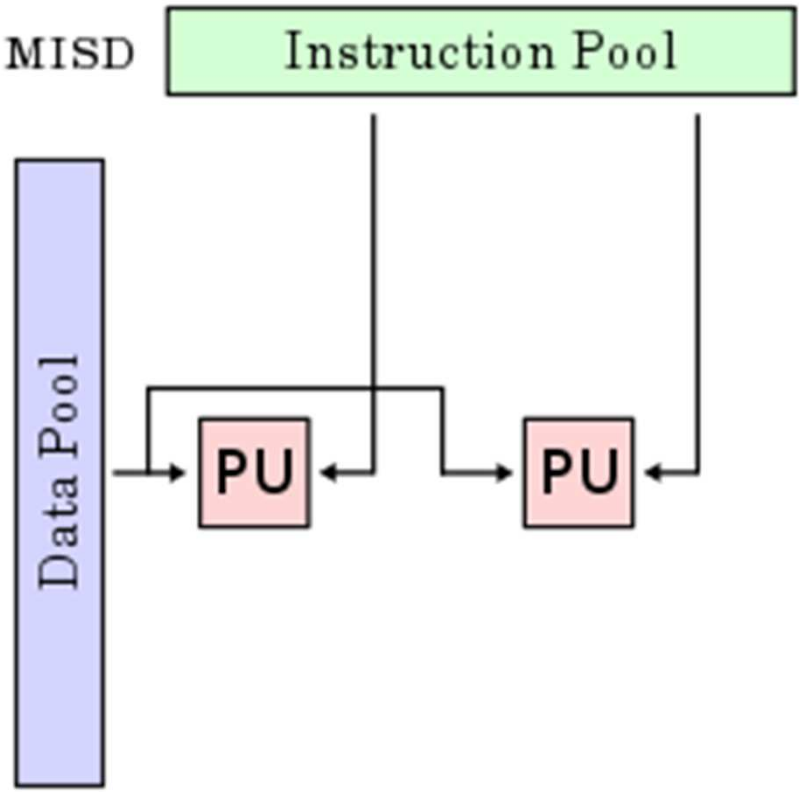
		Data streams	
		Single	Parallel
Instruction Streams	Single	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

Alternative Kinds of Parallelism: Single Instruction/Single Data Stream



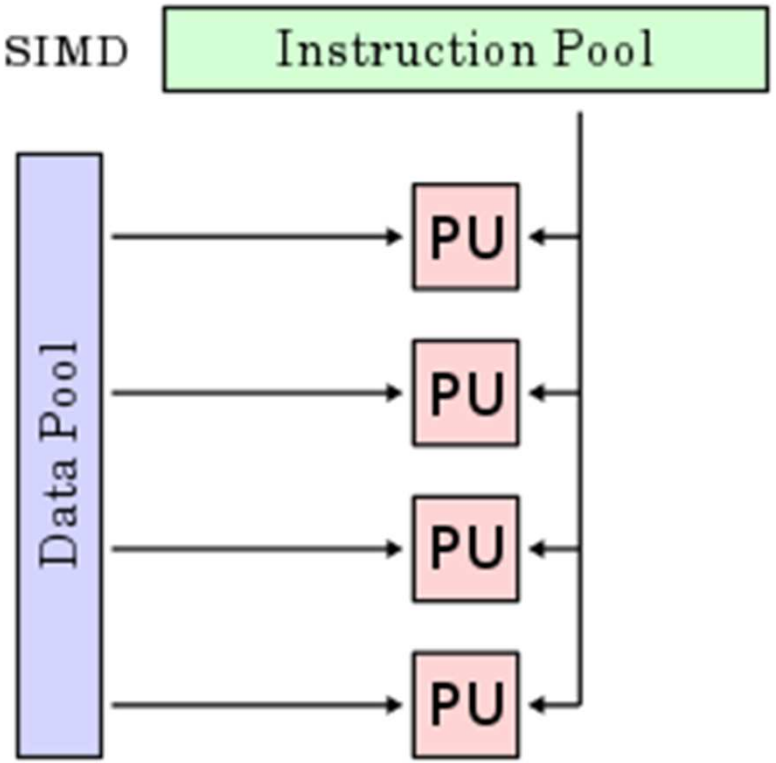
- Single Instruction, Single Data stream (SISD)
  - Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

Alternative Kinds of Parallelism: Multiple Instruction/Single Data Stream



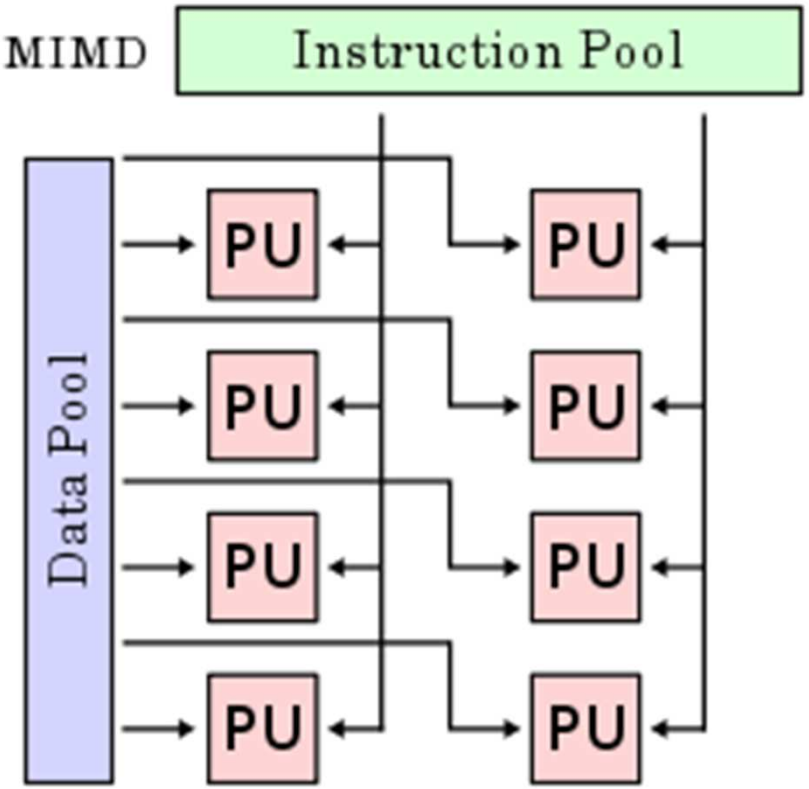
- Multiple Instruction, Single Data streams (MISD)
  - Computer that exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized. For example, certain kinds of array processors.
  - No longer commonly encountered, mainly of historical interest only

Alternative Kinds of Parallelism: Single Instruction/Multiple Data Stream



- Single Instruction, Multiple Data streams (SIMD)
  - Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., SIMD instruction extensions or Graphics Processing Unit (GPU)

Alternative Kinds of Parallelism: Multiple Instruction/Multiple Data Streams



- Multiple Instruction, Multiple Data streams (MIMD)
  - Multiple autonomous processors simultaneously executing different instructions on different data.
  - MIMD architectures include multicore and Warehouse Scale Computers (datacenters)

Flynn Taxonomy of parallel computers

		Data streams	
		Single	Parallel
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE x86, ARM neon, GPGPUs, …
	Multiple	**MISD**: No examples today	**MIMD**: SMP (Intel, ARM, …)

- **From 2011, SIMD and MIMD most common parallel computers**
- **Most common parallel processing programming style: Single Program Multiple Data ("SPMD")**
  - Single program that runs on all processors of a MIMD
  - Cross-processor execution coordination through conditional expressions (thread parallelism)
- **SIMD (aka hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays**
  - Scientific computing, signal processing, multimedia (audio/video processing)

SIMD Architectures

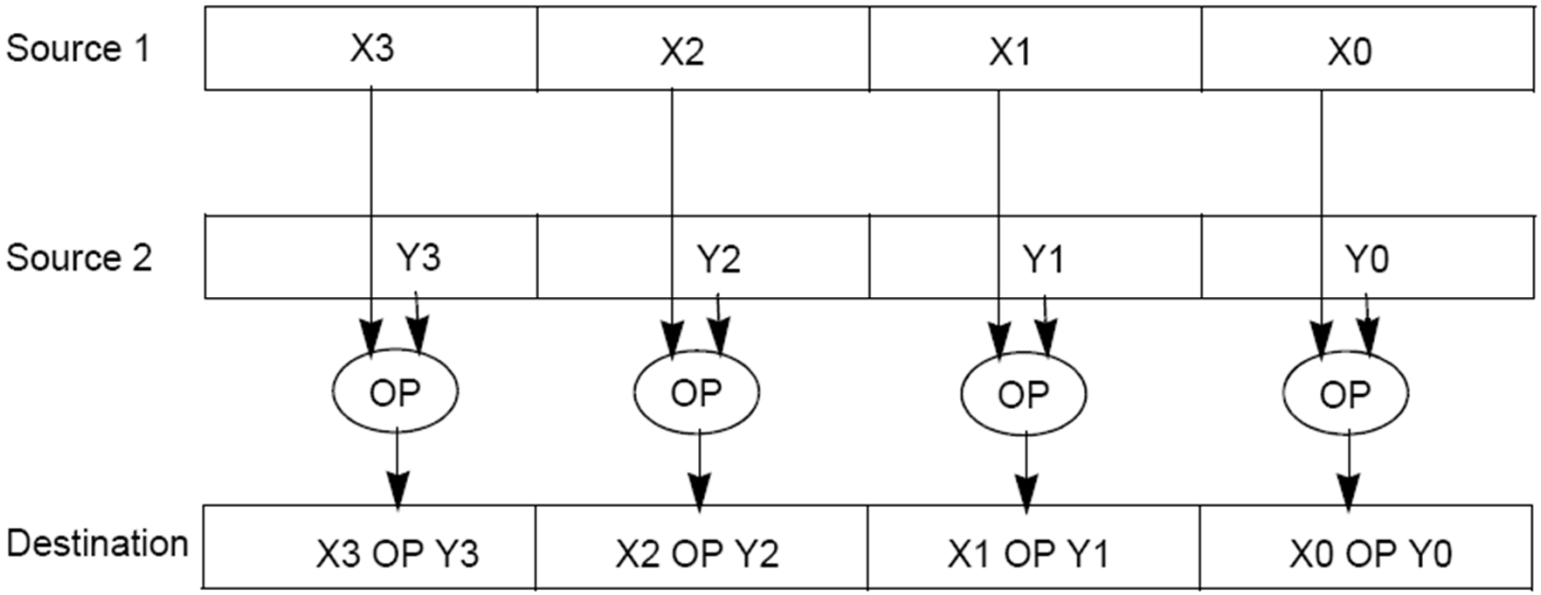
- *Data parallelism*: executing **one operation on multiple data streams**
  - Single control unit
  - Multiple datapaths (processing elements – PEs) running in parallel
    - *PEs are interconnected and exchange/share data as directed by the control unit*
    - *Each PE performs the same operation on its own local data*

- Example to provide context:
  - Multiplying a coefficient vector by a data vector (e.g., in filtering)

$$y[i] := c[i] \times x[i],\ 0 \leq i < n$$

“Advanced Digital Media Boost”

- To improve performance, SIMD instructions
  - Fetch one instruction, do the work of multiple instructions



Example: SIMD Array Processing

```
for each f in array
    f = sqrt(f)
```

SISD

```
for each f in array
{
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}
```

SIMD

```
for each 4 members in array
{
    load 4 members to the SIMD register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
```

Data Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

```
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- How to obtain more data level parallelism than available in a single iteration of a loop?
- *Unroll loop* and adjust iteration rate

Loop Unrolling

Loop Unrolling can be implemented from C code

```
for(i=1000; i>0; i=i-1)
 x[i] = x[i] + s;
```

into

```
for(i=1000; i>0; i=i-4)
{
  x[ i ] = x[ i ] + s;
  x[i-1] = x[i-1] + s;
  x[i-2] = x[i-2] + s;
  x[i-3] = x[i-3] + s;
}
```

Loop Unrolling (MIPS)

Assumptions:

- R1 is initially the address of the element in the array with the highest address
- F2 contains the scalar value s
- 8(R2) is the address of the last element to operate on.

```
for(i=1000; i>0; i=i-1)
  x[i] = x[i] + s;
```

```
Loop:
1. l.d      F0, 0(R1)     ; F0=array element
2. add.d    F4,F0,F2      ; add s to F0
3. s.d      F4,0(R1)      ; store result
4. addui    R1,R1,#-8     ; decrement pointer 8 byte
5. bne      R1,R2,Loop    ; repeat loop if R1 != R2
```

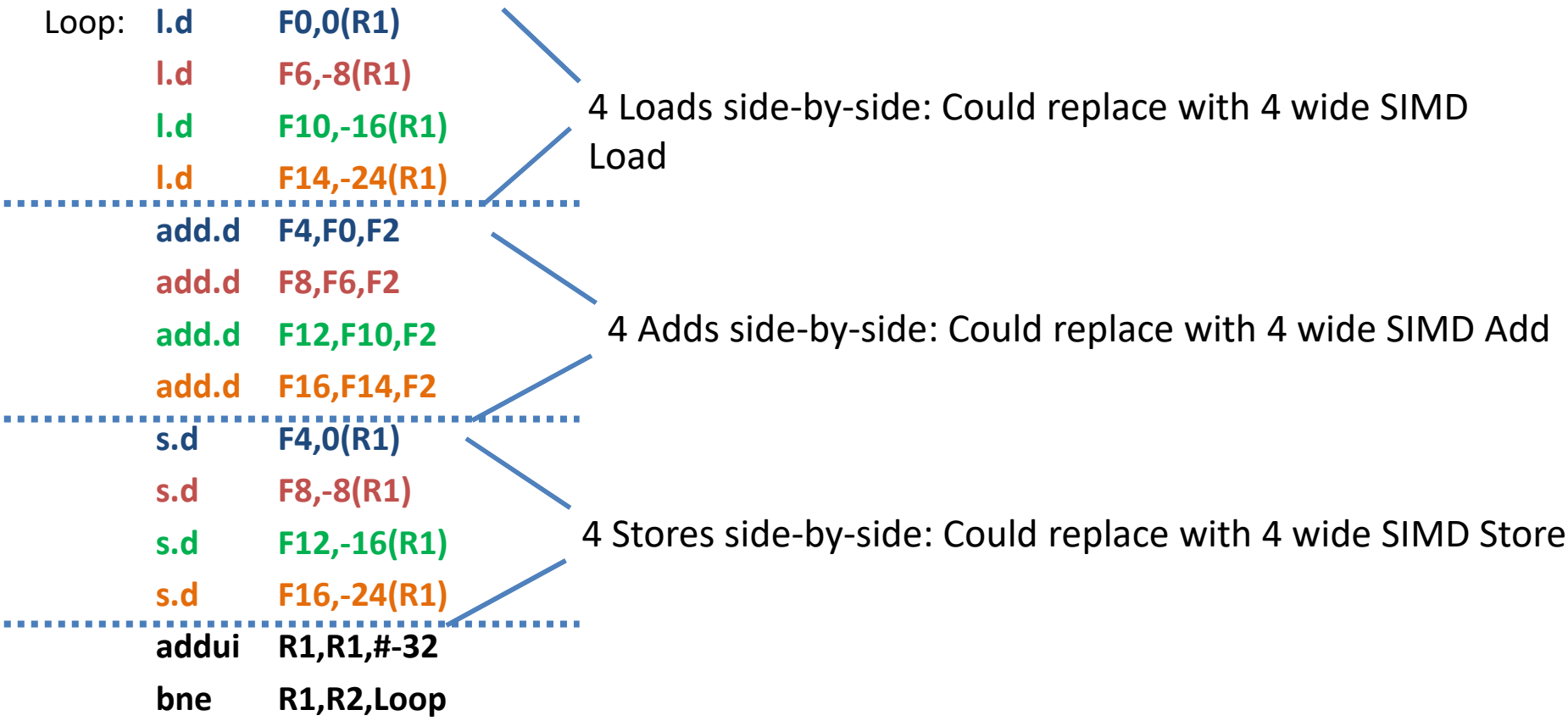
Loop Unrolled

```
Loop:  l.d     F0,0(R1)
       add.d   F4,F0,F2
       s.d     F4,0(R1)
       l.d     F6,-8(R1)
       add.d   F8,F6,F2
       s.d     F8,-8(R1)
       l.d     F10,-16(R1)
       add.d   F12,F10,F2
       s.d     F12,-16(R1)
       l.d     F14,-24(R1)
       add.d   F16,F14,F2
       s.d     F16,-24(R1)
       addui   R1,R1,#-32
       bne     R1,R2,Loop
```

NOTE:

1. Different Registers eliminate stalls
2. Only 1 Loop Overhead every 4 iterations
3. This unrolling works if loop_limit(mod 4) = 0

Loop Unrolled Scheduled



Generalizing Loop Unrolling

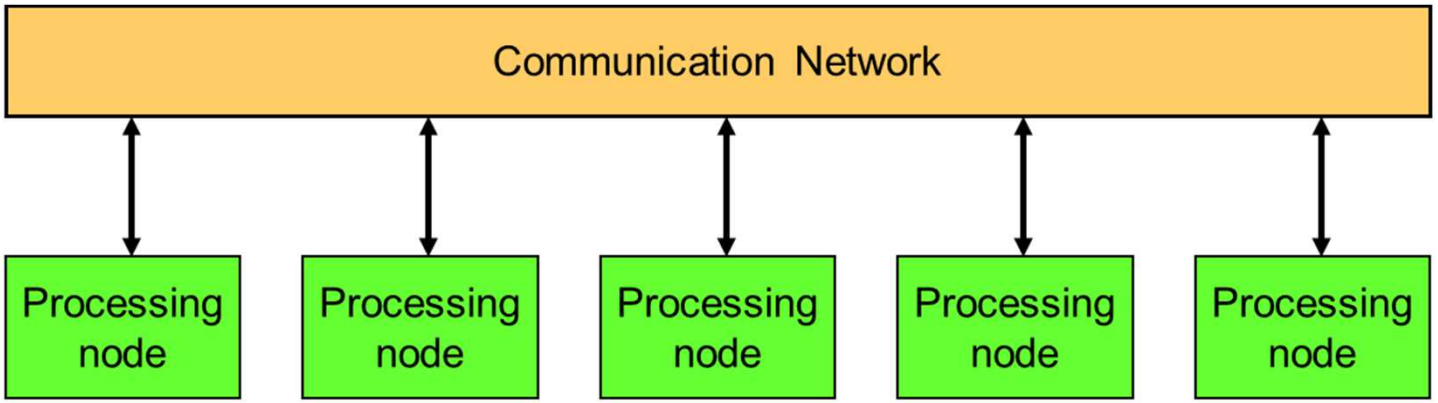
- A loop of **n iterations**
- **k copies** of the body of the loop

Then we will run the loop with 1 copy of the body **n(mod k)** times and

with k copies of the body **floor(n/k)** times

MIMD Architectures

- **Multicore architectures**
- At least 2 processors interconnected via a communication network
  - abstractions (HW/SW interface)
  - organizational structure to realize abstraction efficiently



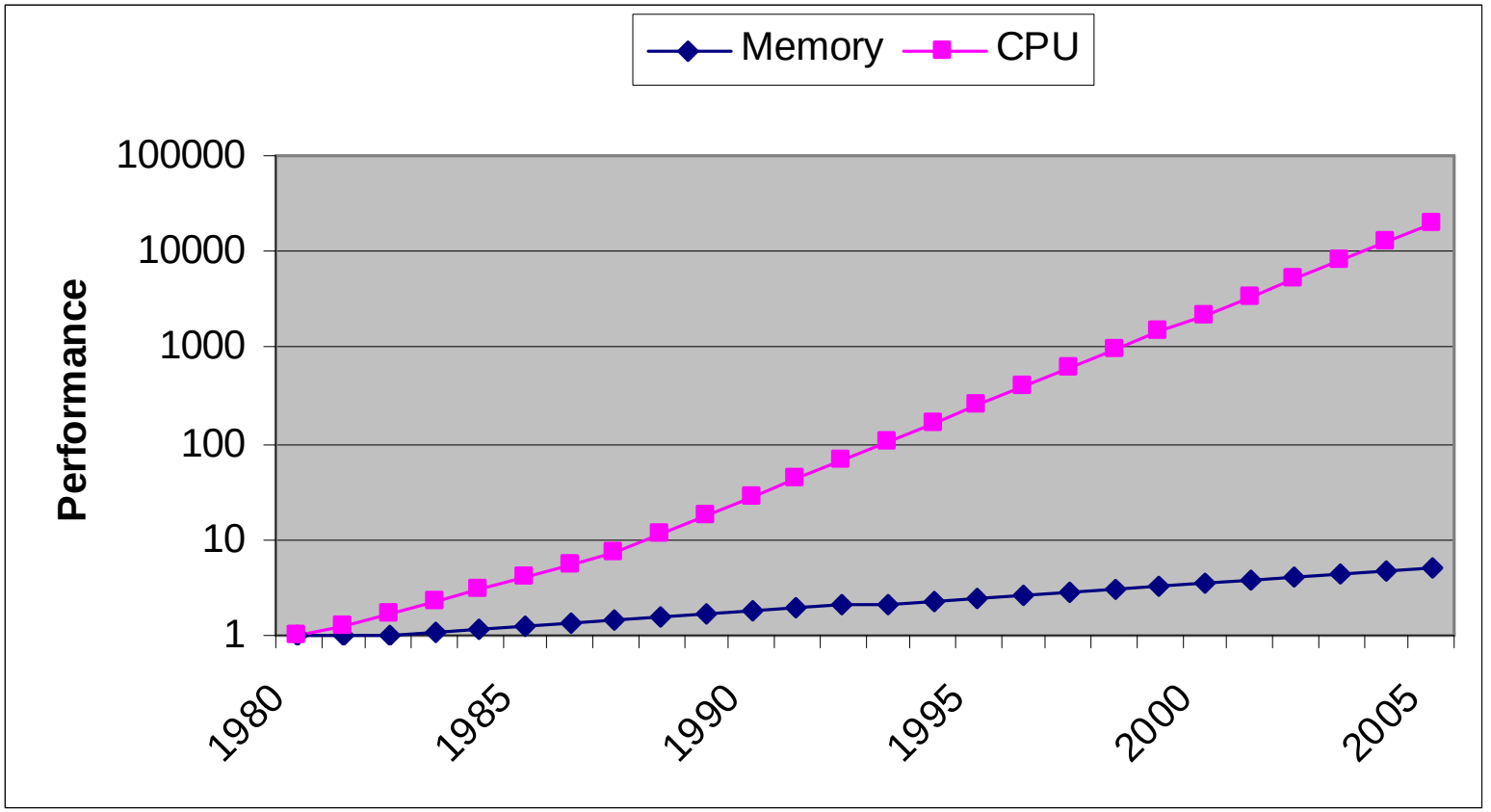
MIMD Architectures

- Thread-Level parallelism
  - Have multiple program counters
  - Targeted for tightly-coupled shared-memory multiprocessors
- For *n* processors, need *n* threads
- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

MIMD Architectures

- And what about memory?
- How is data accessed by multiple cores?
- How to design accordingly the memory system?
- How do traditional solutions from uniprocessor systems affect multicores?

The Memory Gap



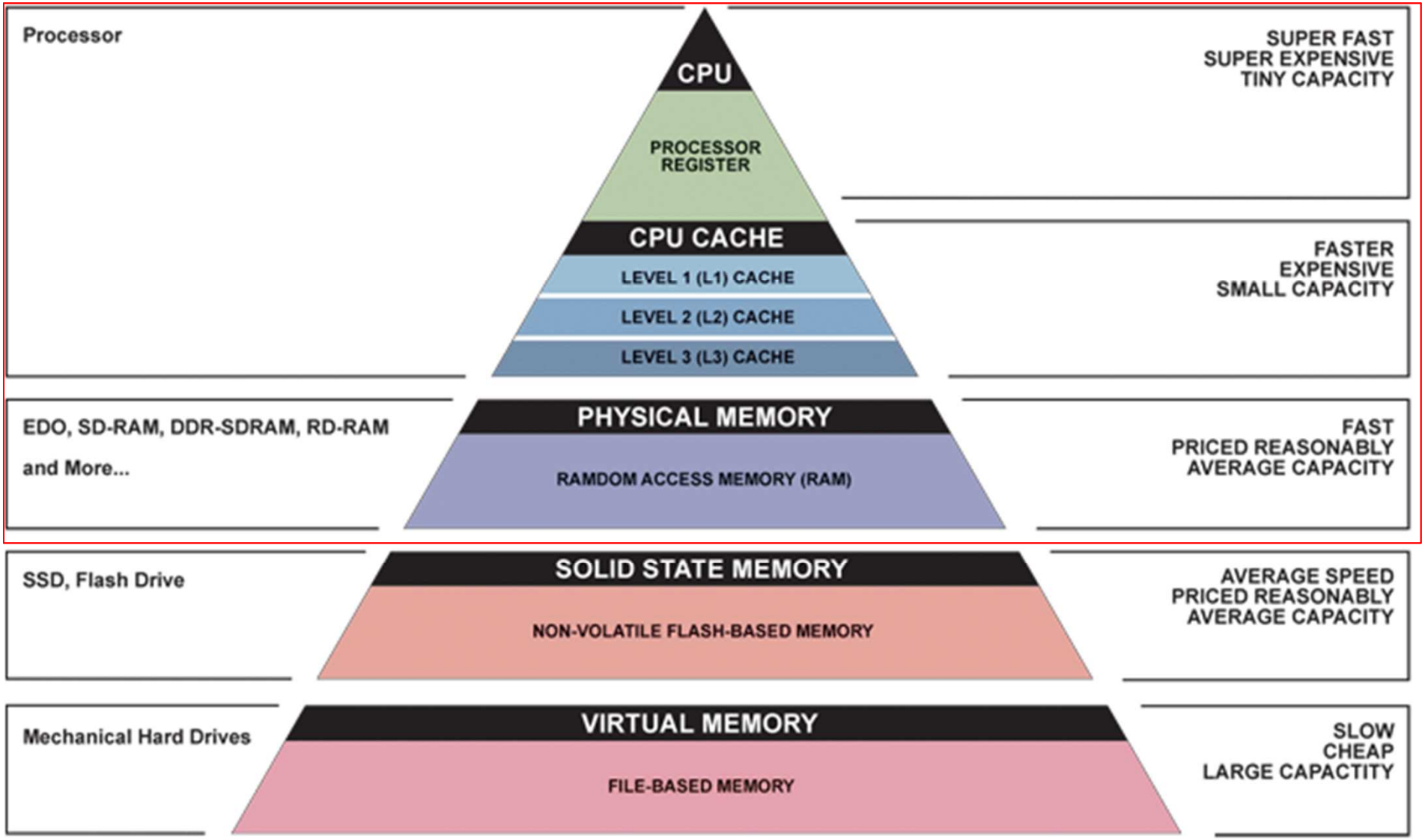
H&P
Fig. 5.2

- **Bottom-line: memory access is increasingly expensive and CA must devise new ways of hiding this cost**

The memory wall

- How do multicores attack the memory wall?
- Same issue as uniprocessor systems
  - By building on-chip cache hierarchies

Memory Hierarchy

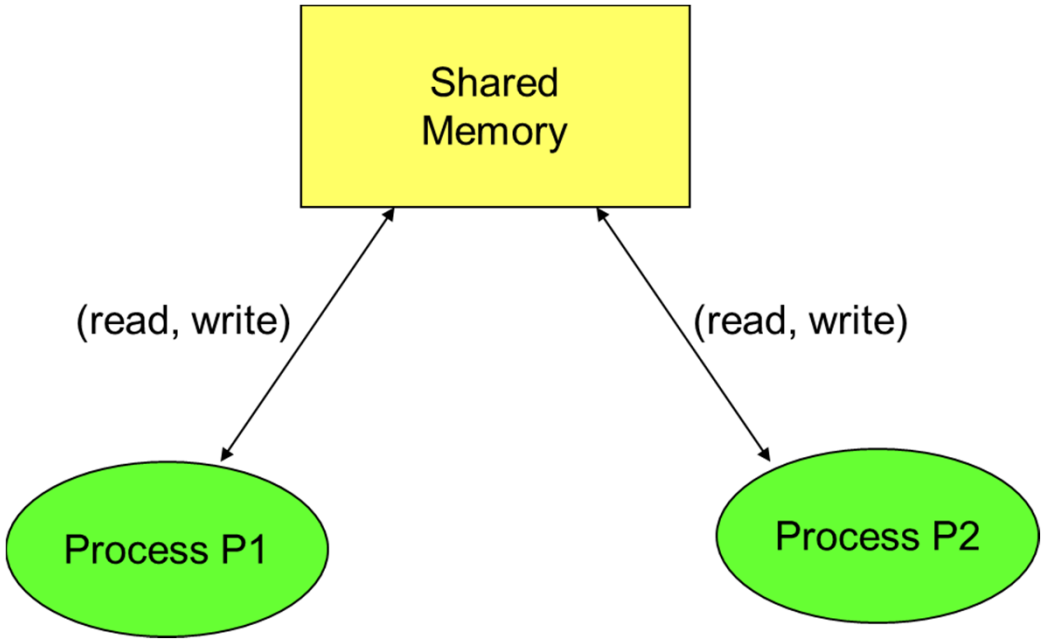


The memory wall

- How do multicores attack the memory wall?
- Same issue as uniprocessor systems
  - By building on-chip cache hierarchies

- Which novel issues arise?
  - Coherence
  - Consistency
  - Scalability

Communication models: Shared Memory



- Coherence problem
- Memory consistency issue
- Synchronization problem

Communication models: **Shared memory**

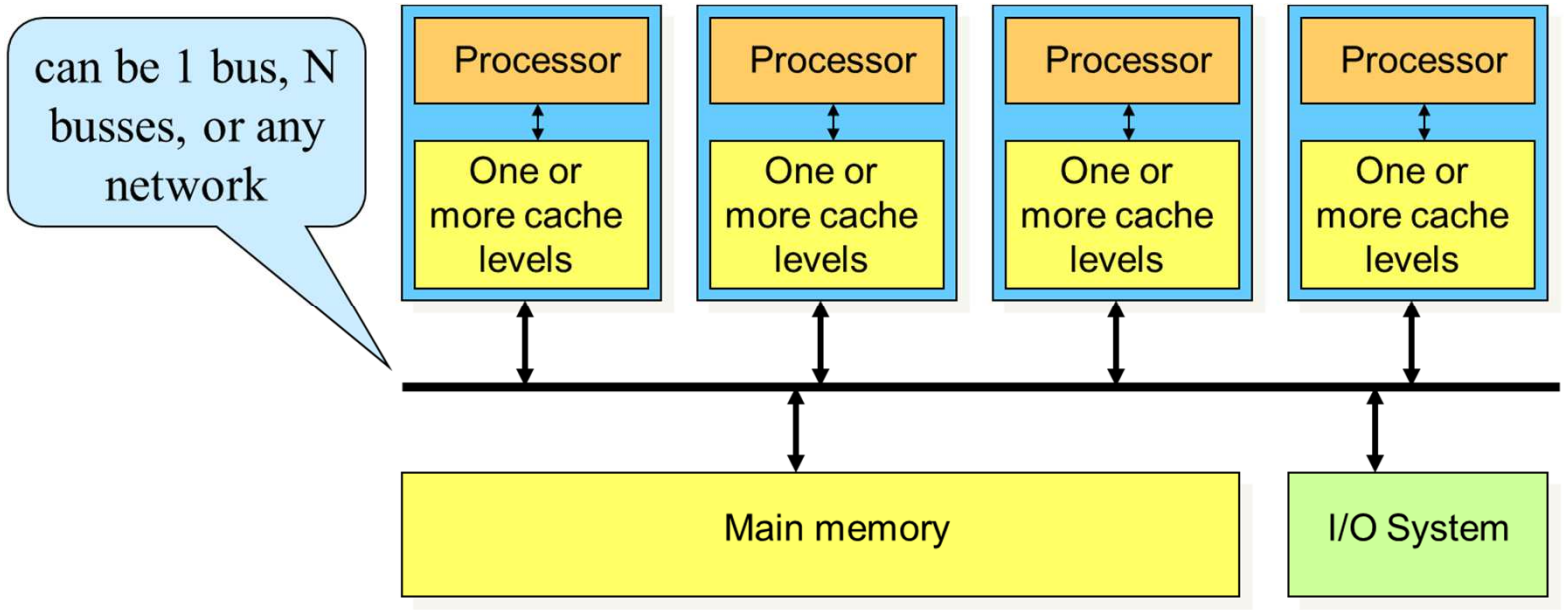
- Shared address space
- Communication primitives:
  - load, store, atomic swap

Two varieties:

- Physically shared => Symmetric Multi-Processors (**SMP**)
  - usually combined with local caching
- Physically distributed => Distributed Shared Memory (**DSM**)

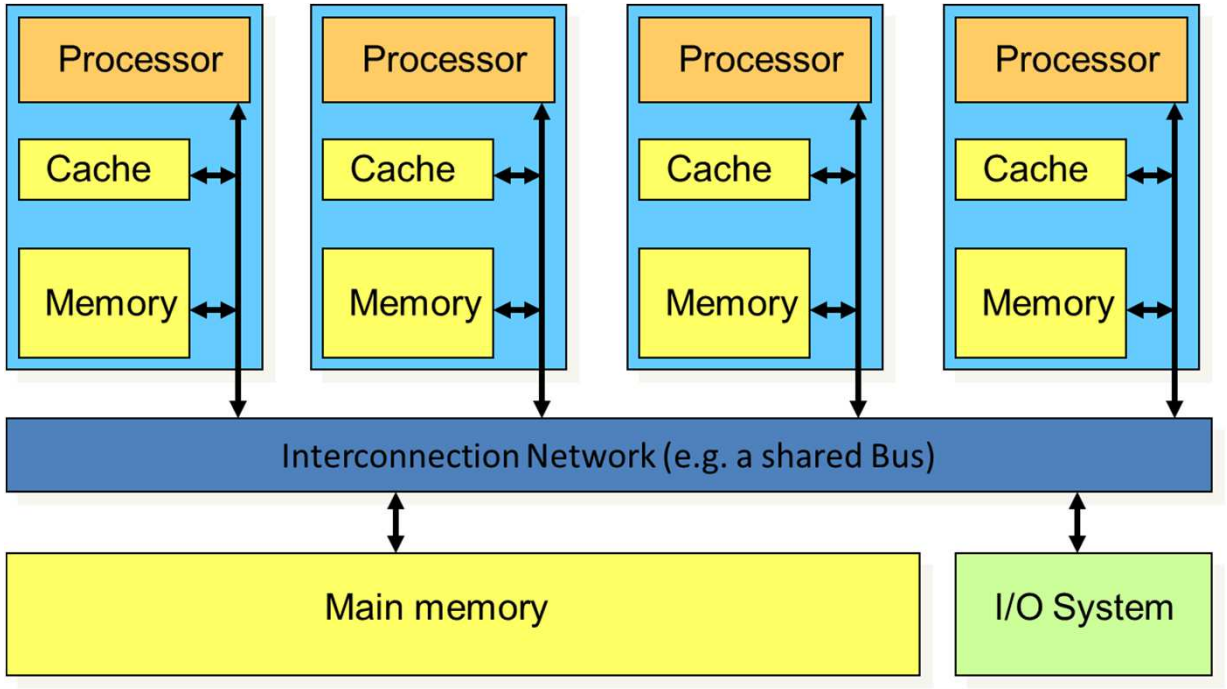
SMP: Symmetric Multi-Processor

- Memory: centralized with uniform access time (***UMA***) and bus interconnect, I/O
- Examples: Sun Enterprise 6000, SGI Challenge, Intel



DSM: Distributed Shared Memory

- Nonuniform access time (***NUMA***) and scalable interconnect (distributed memory)

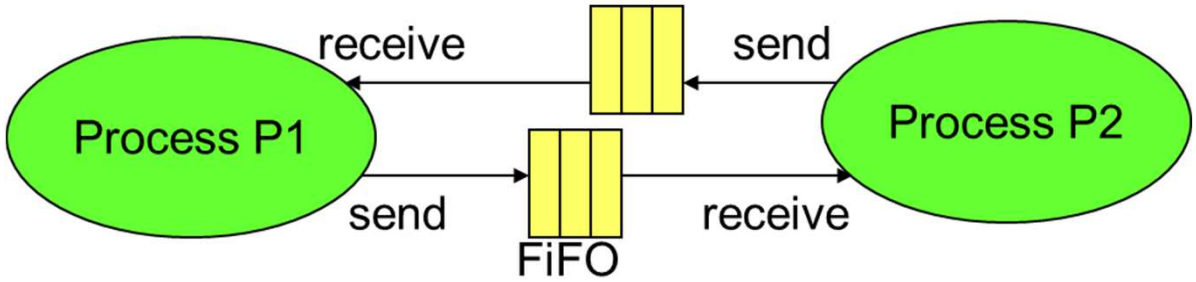


Shared Address Model Summary

- Each processor can address every physical location in the machine
- Each process can address all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Memory hierarchy model applies:
  - communication moves data to local proc. cache

Communication models: **Message Passing**

- Communication primitives
  - e.g., send, receive library calls
  - standard MPI: Message Passing Interface
    - www.mpi-forum.org
- *Note that MP can be built on top of SM and viceversa!*

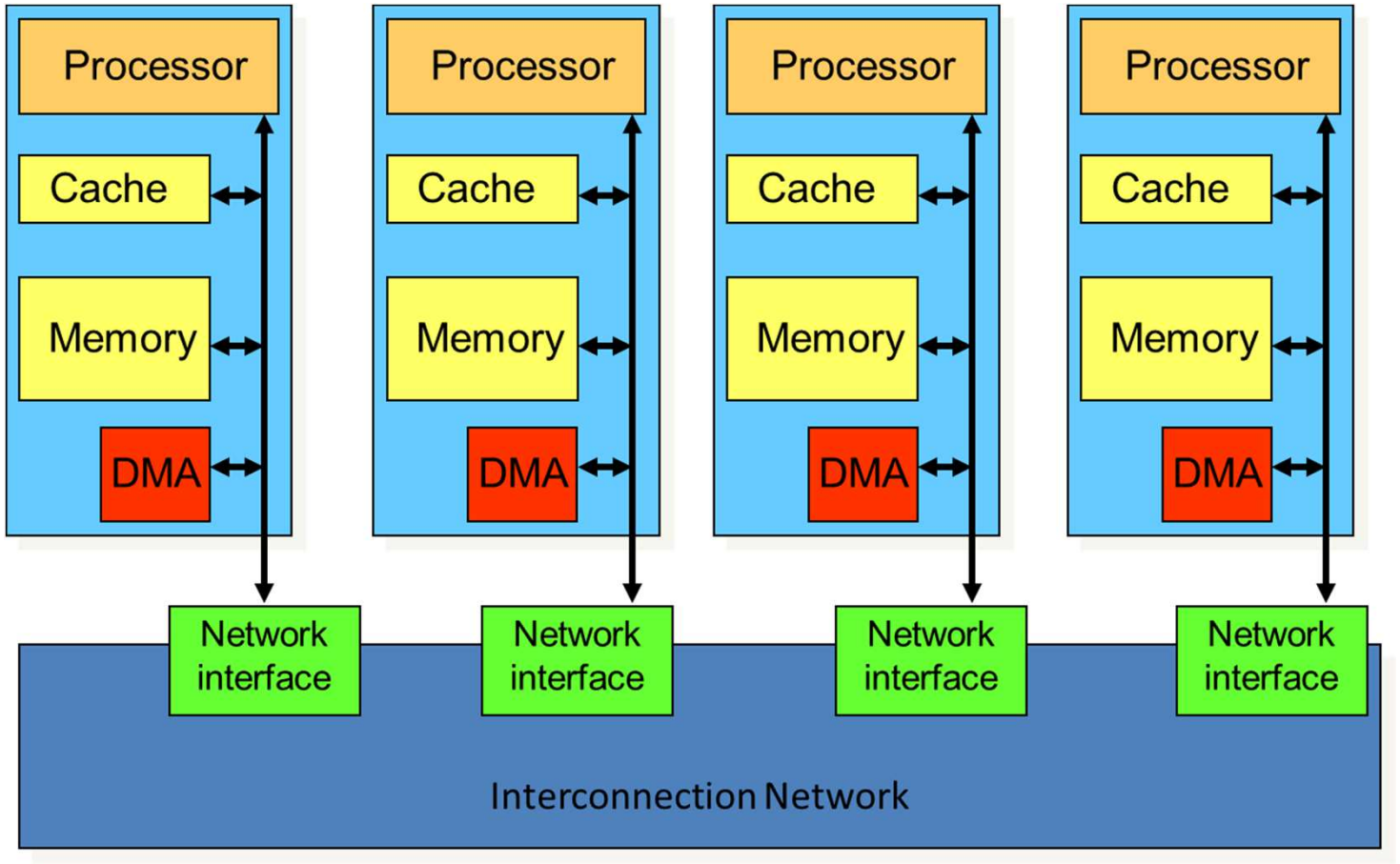


Message Passing Model

- Explicit message send and receive operations
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
- Typically blocking communication, but may use DMA

Message structure

Message passing communication



Communication Models: Comparison

- Shared-Memory (used by e.g. **OpenMP**)
  - Compatibility with well-understood (language) mechanisms
  - Ease of programming for complex or dynamic communications patterns
  - Shared-memory applications; sharing of large data structures
  - Efficient for small items
  - Supports hardware caching
- Messaging Passing (used by e.g. **MPI**)
  - Simpler hardware
  - Explicit communication
  - Implicit synchronization (with any communication)

Three fundamental issues for shared memory multiprocessors

- **Coherence**,
  about: *Do I see the most recent data?*

- **Consistency**,
  about: ***When** do I see a written value?*
  - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

- **Synchronization**
  How to synchronize processes?
  - how to protect access to shared data?

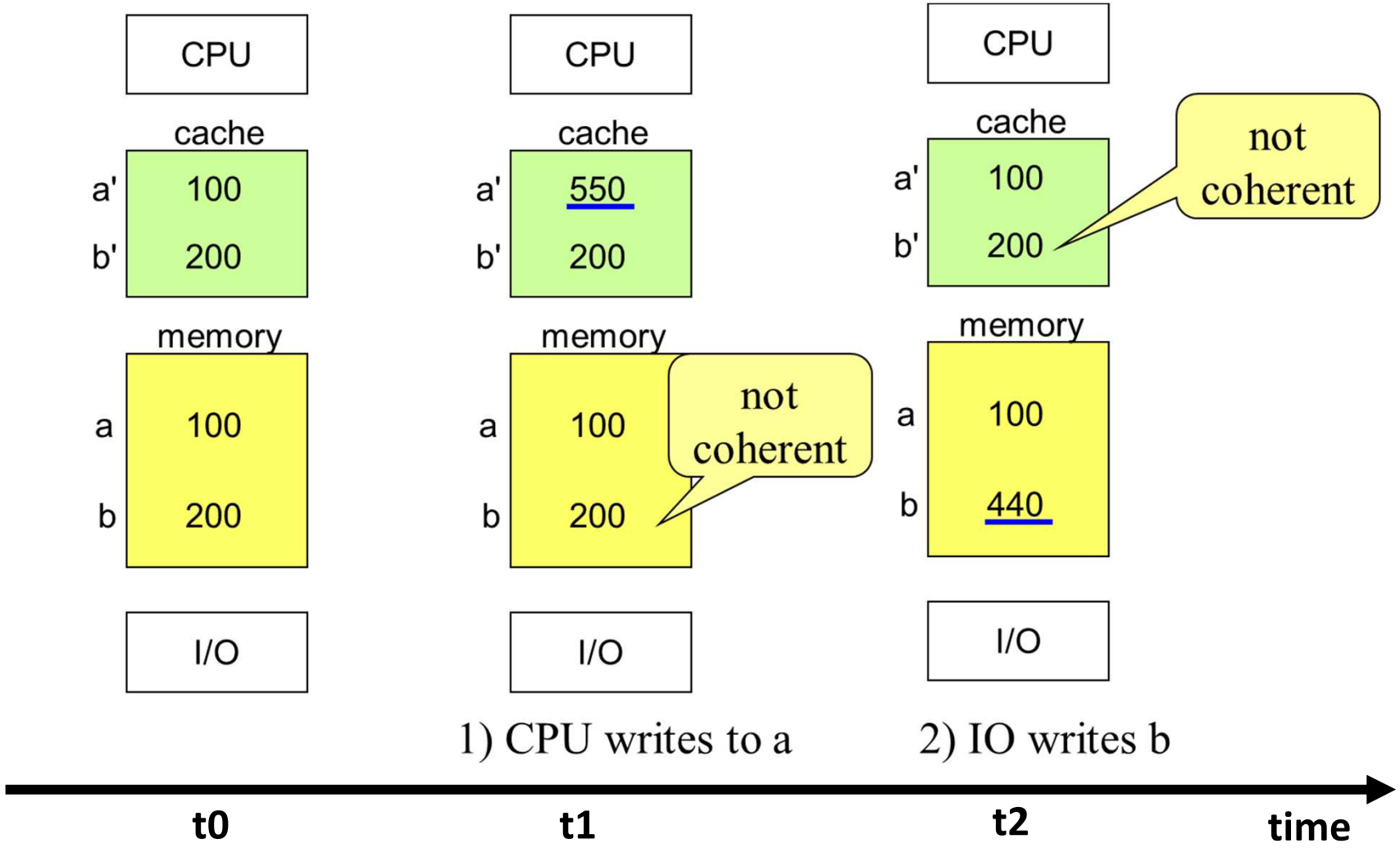
Three fundamental issues for shared memory multiprocessors

- **Coherence**,
  about: *Do I see the most recent data?*

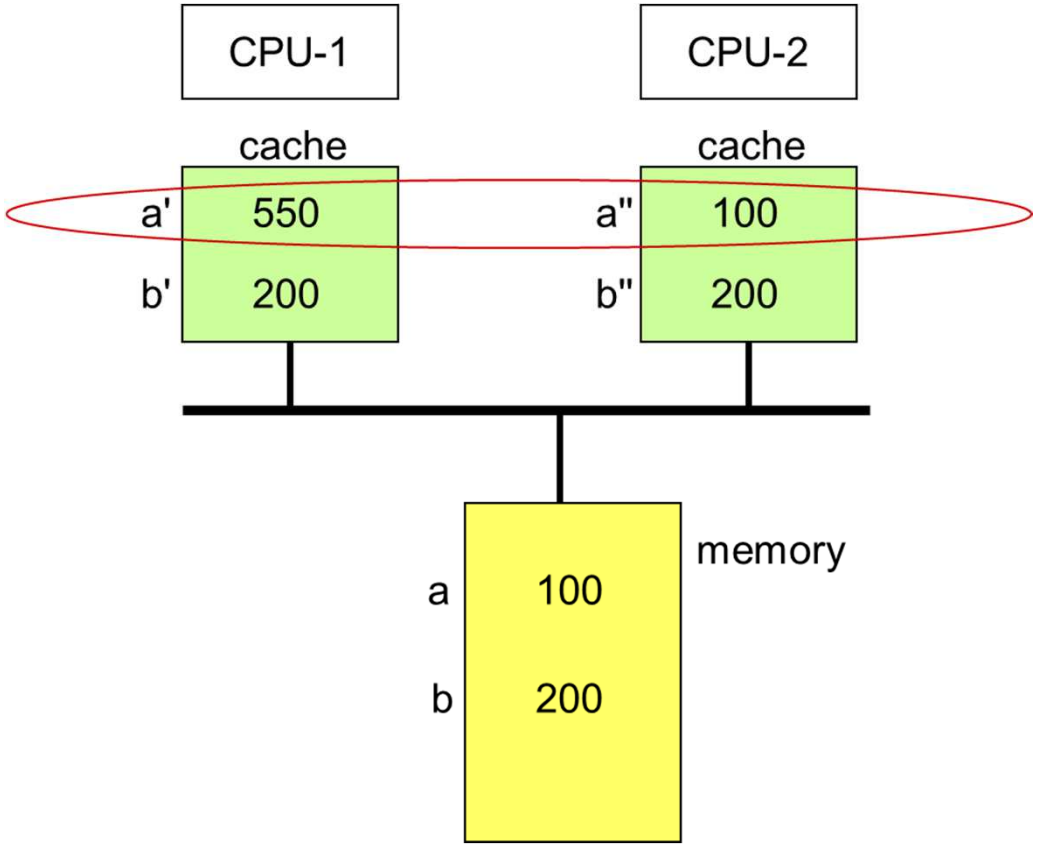
- **Consistency**,
  about: ***When** do I see a written value?*
  - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

- **Synchronization**
  How to synchronize processes?
  - how to protect access to shared data?

Coherence problem, in single CPU system



Coherence problem, in Multi-Proc system



What Does Coherency Mean?

- Discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion
- Different levels are possible:
  - Every write operation appears to occur instantaneously
    - Too expensive and inefficient
  - All processors see exactly the same sequence of changes of values for each separate operand
  - Different processors may see an operand and assume different sequences of values
    - non-coherent behavior

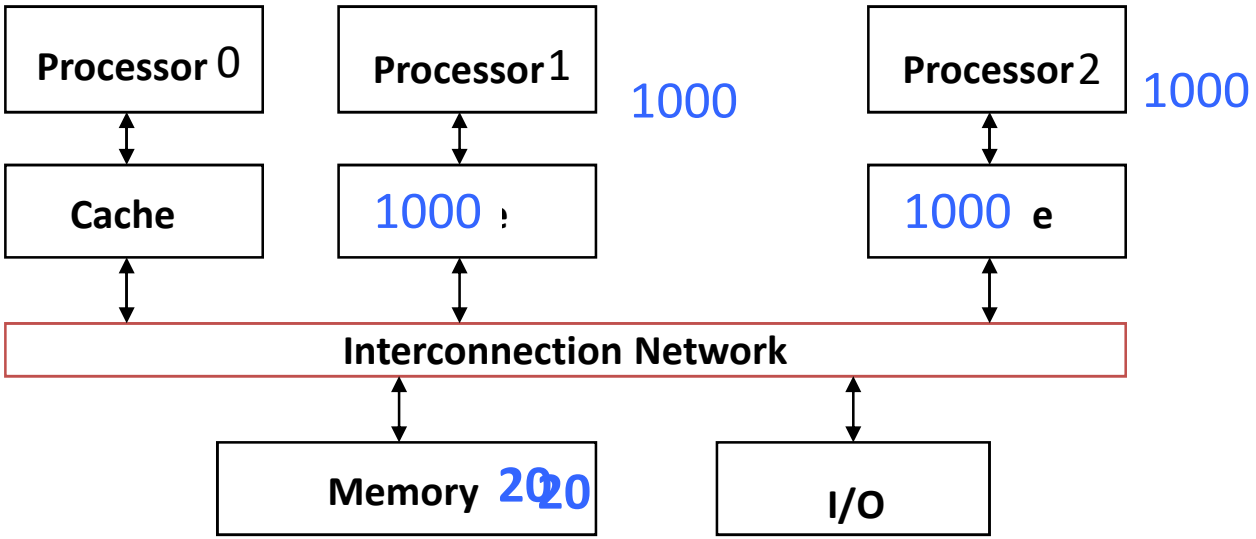
Two rules to ensure coherency

- “If P1 writes x and P2 reads it, P1’s write will be seen by P2 if the read and write are sufficiently far apart”

- Writes to a single location are serialized: seen in one order
  - ‘Latest’ write will be seen
  - Otherwise could see writes in illogical order (could see older value after a newer value)

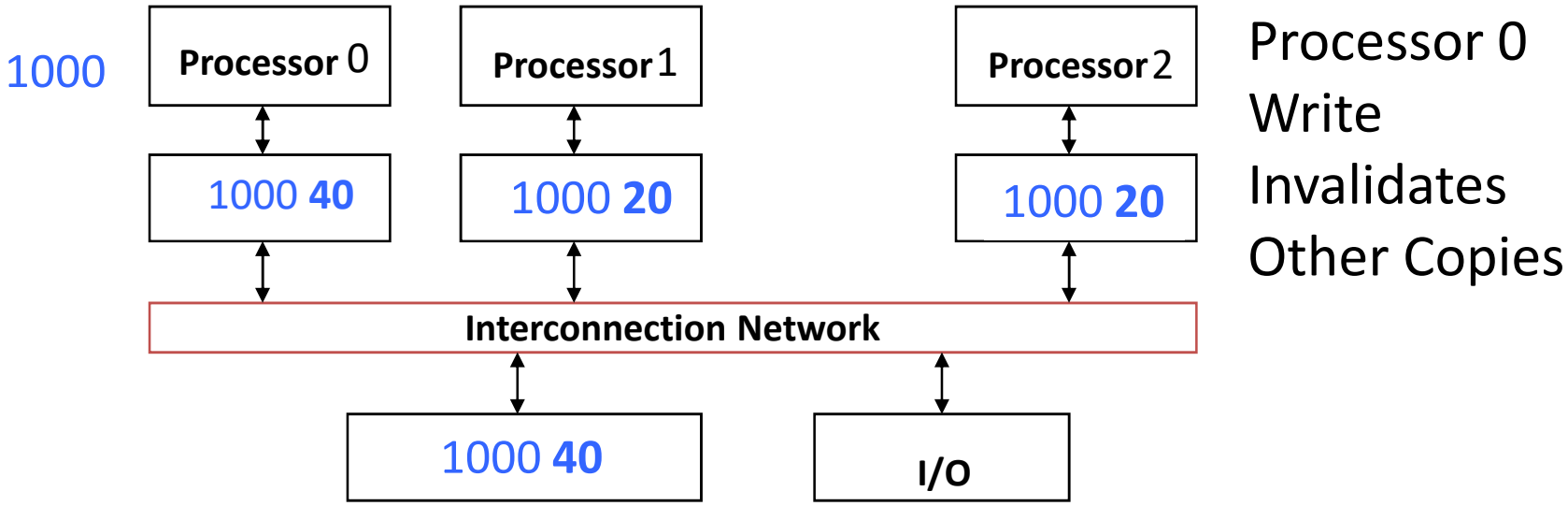
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



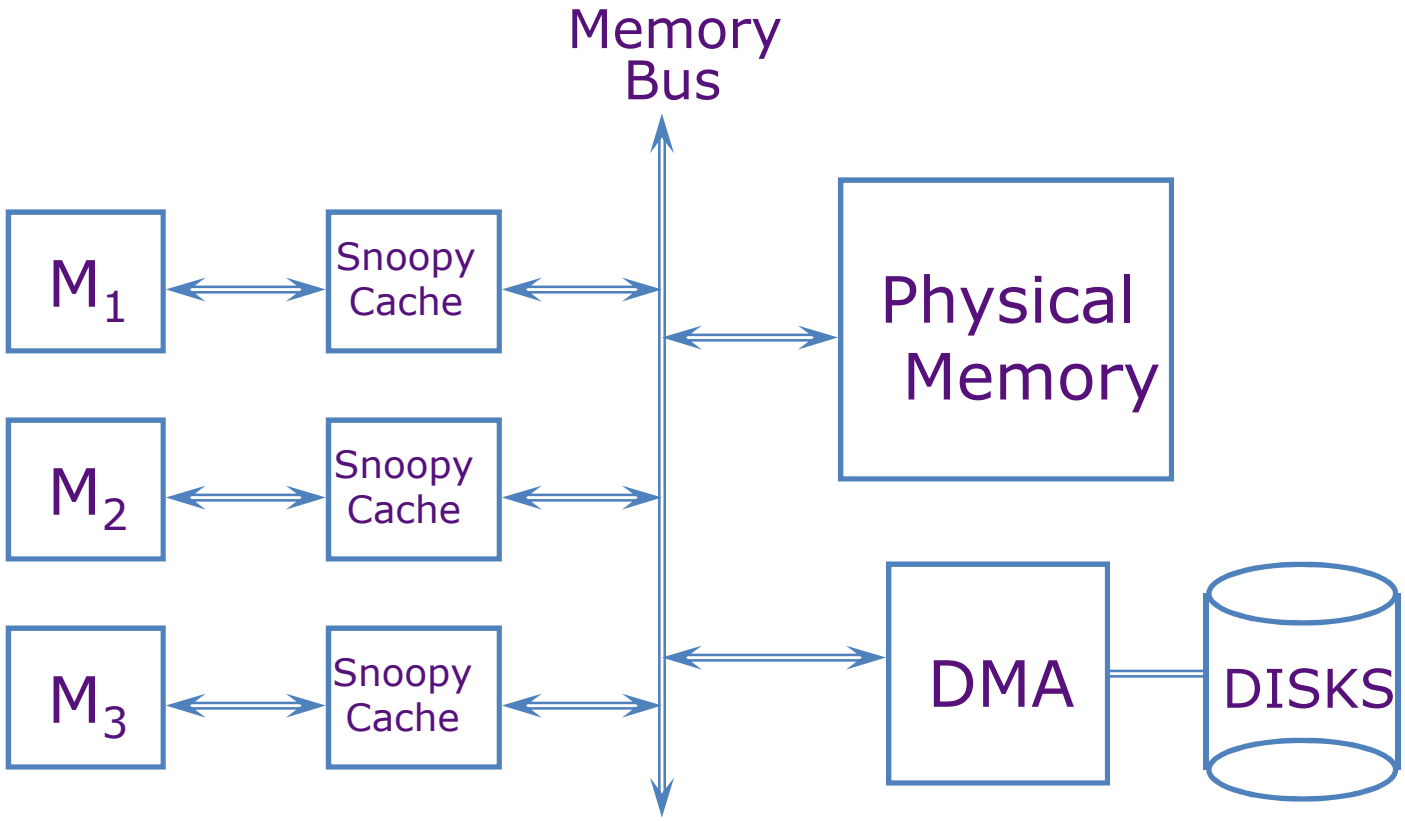
Keeping Multiple Caches Coherent

- **Architect's job**: shared memory => keep cache values coherent

- **Idea**: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate all other copies

- Shared written result can "ping-pong" between caches

Shared Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

Snoopy Cache Coherence Protocols

write miss:

the address is *invalidated* in all other caches *before* the write is performed

read miss:

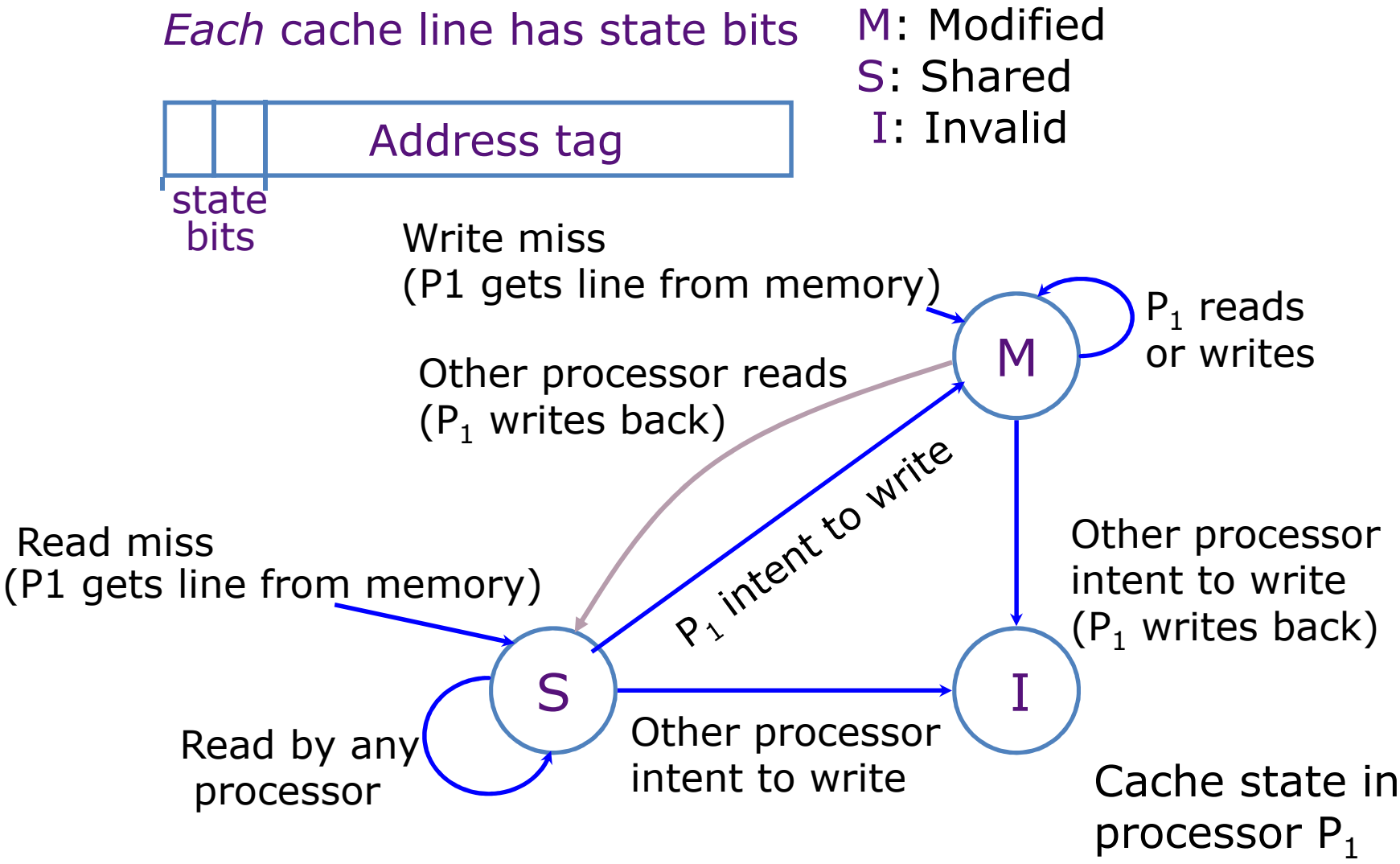
if a dirty copy is found in some cache, a write-back is performed before the memory is read

MSI Protocol States

- **M**odified: The cache modified the block
  - The data in the cache is inconsistent with the backing store (e.g., main memory)
  - Cache must write-back the block when evicted
- **S**hared: The block is unmodified
  - The cache can evict the data without writing it to the backing store.
- **I**nvalid: The block is not present in the cache
  - It must be fetched from memory or another cache

Cache State Transition Diagram

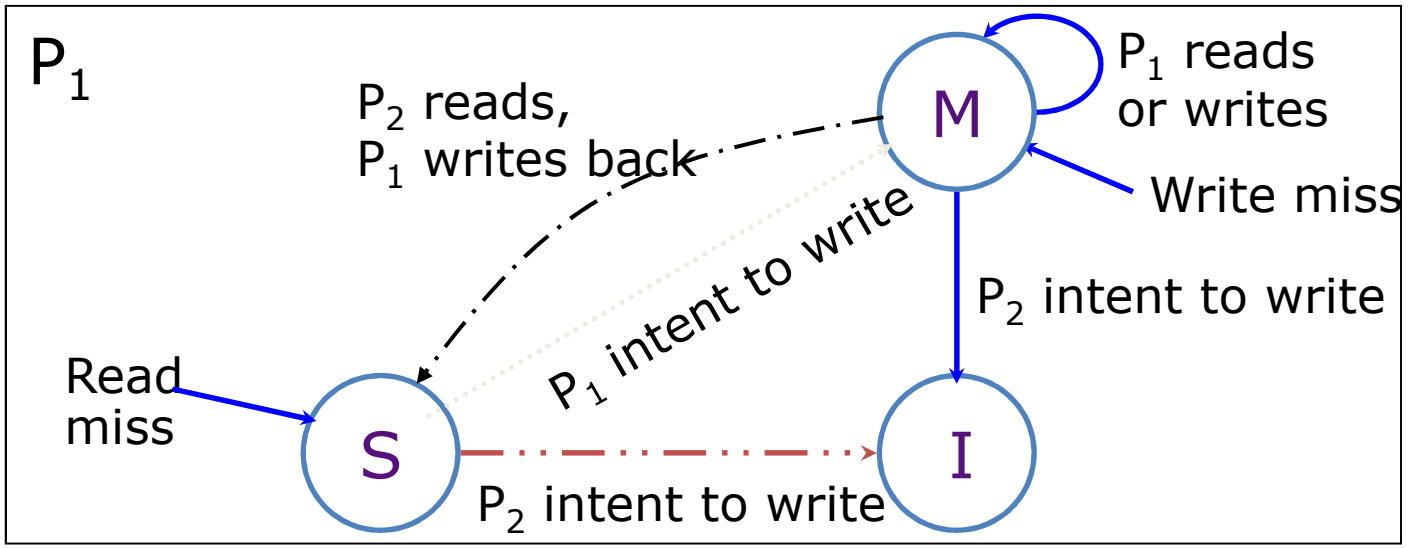
The MSI protocol

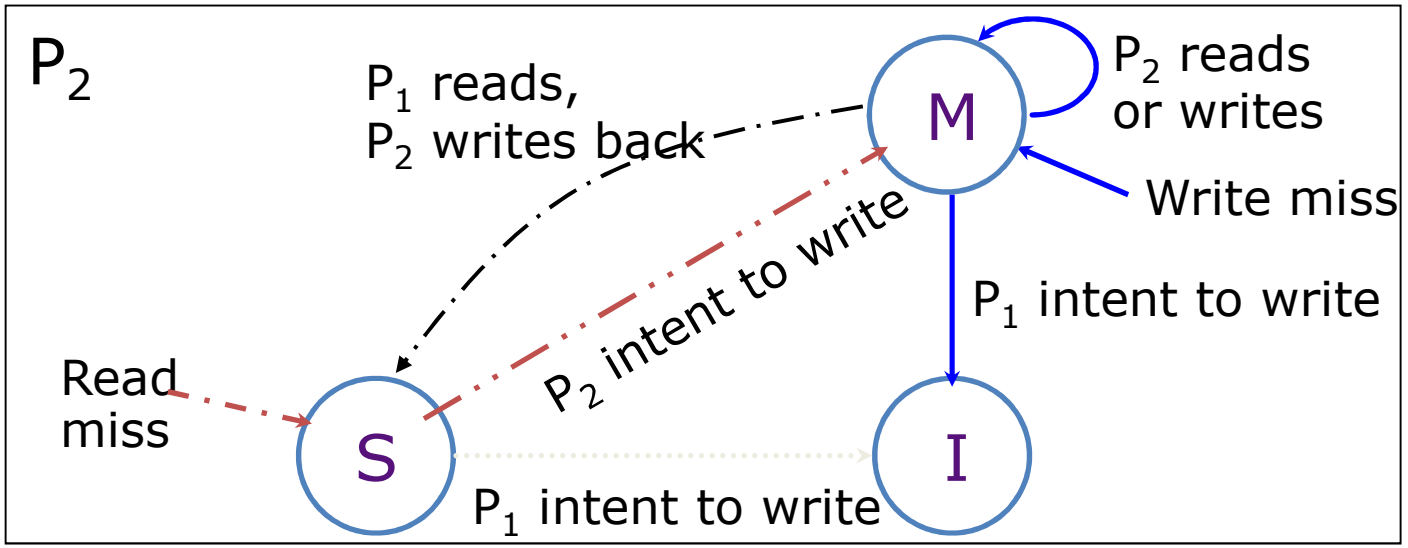


Two Processor Example

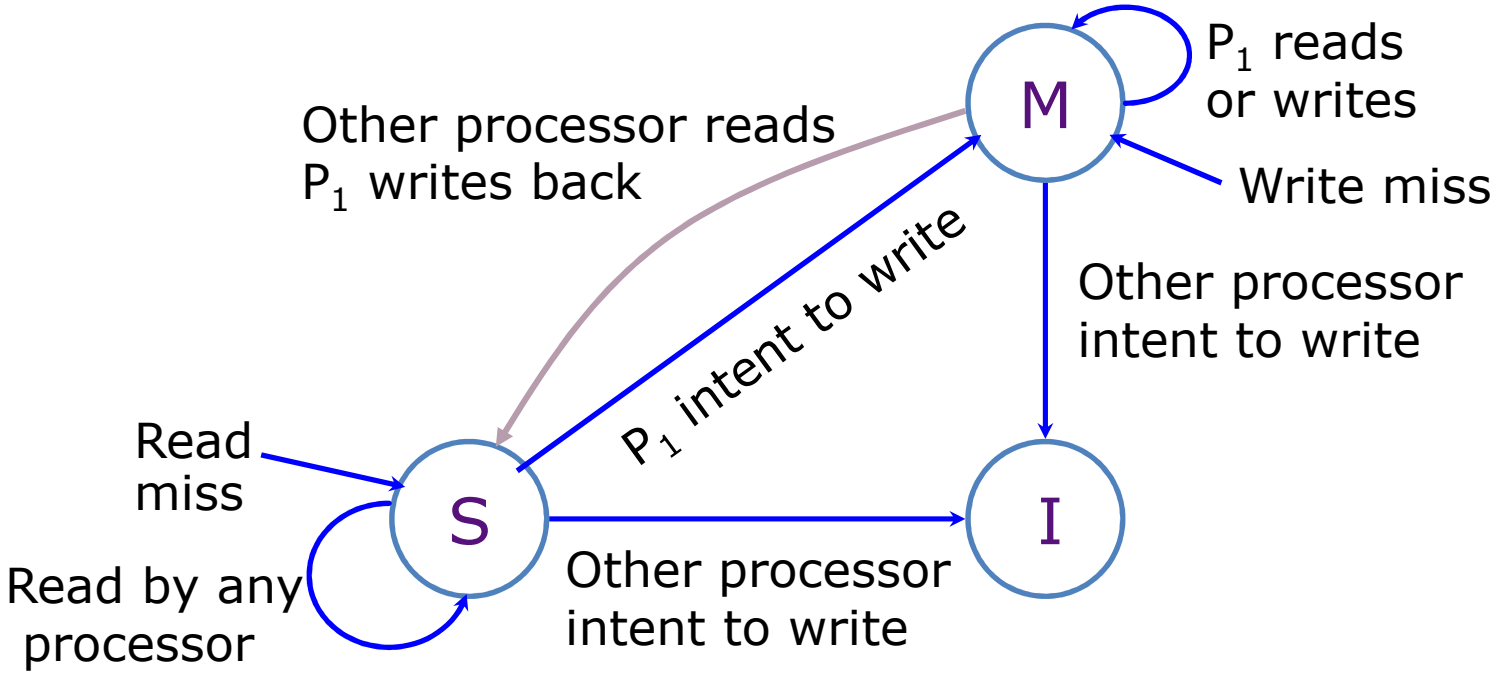
(Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes





Observation



- If a line is in the M state then no other cache can have a copy of the line!
  - Memory stays coherent, multiple differing copies cannot exist

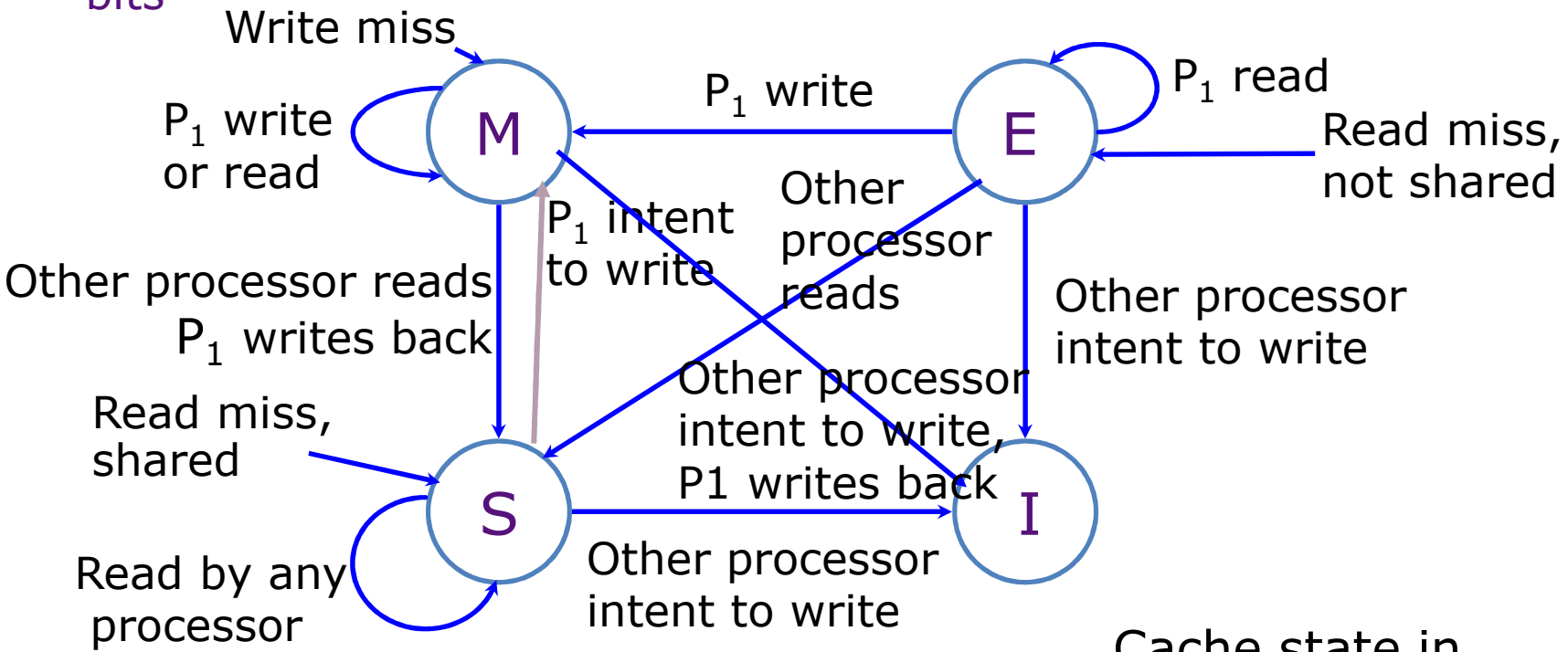
MESI: An Enhanced MSI protocol

- Adds an **E** state to MSI
  - **E**xclusive but unmodified
  - Data is only in this cache and is clean
    - It matches main memory
  - A read request from another cache will change it to **S**hared
  - Writing it will change it to **M**odified
    - No invalidation needed since it was exclusive!
- Increased performance for private data

MESI: An Enhanced MSI protocol

Each cache line has a tag

M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



Cache state in processor P_1

Considerations on MESI

- A read can be satisfied in any state except **I**nvalid
  - An **I**nvalid line must be fetched to **S** or **E** state
- A write can be satisfied in **M** or **E**
  - A **S**hared line must first invalidate other copies
    - Broadcast operation: Request For Ownership (RFO)
- **S,E,I** lines can be always discarded
  - A **M**odified line must be written back first
- A **M**odified line must snoop other read attempts
  - Back-off the reader, write-back, and change to **S**hared
- A **S**hared line must listen for invalidate/RFO requests
  - Change to **I**nvalid
- An **E**xclusive line must listen for read requests
  - Change to **S**hared

Performance of Symmetric Shared-Memory Multiprocessors

Cache performance is combination of:

1. Uniprocessor cache miss traffic
2. Traffic caused by communication
   - Results in invalidations and subsequent cache misses

- Adds 4th C: *coherence miss*
  - Joins Compulsory, Capacity, Conflict
  - (Sometimes called a *Communication* miss)

Coherency Misses

1. True sharing misses arise from the communication of data through the cache coherence mechanism
   - Invalidates due to 1st write to shared block
   - Reads by another CPU 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
     $\Rightarrow$ miss would not occur if block size were 1 word

False Sharing

A cache block contains more than one word

Cache-coherence is done at the block-level and not word-level

Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same block address.

What can happen?

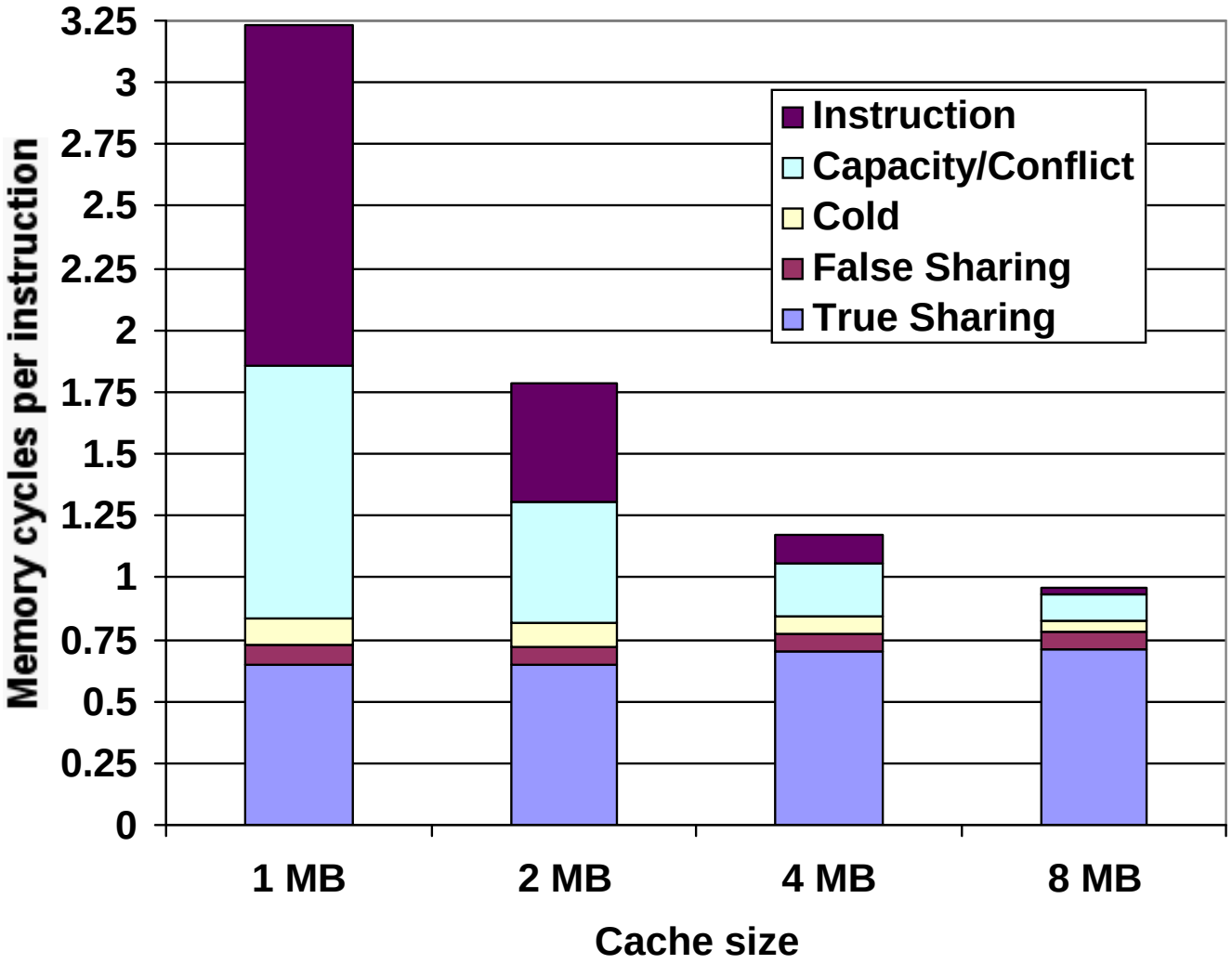
Example: True v. False Sharing v. Hit?

- **Assume x1 and x2 in same cache block.**
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss

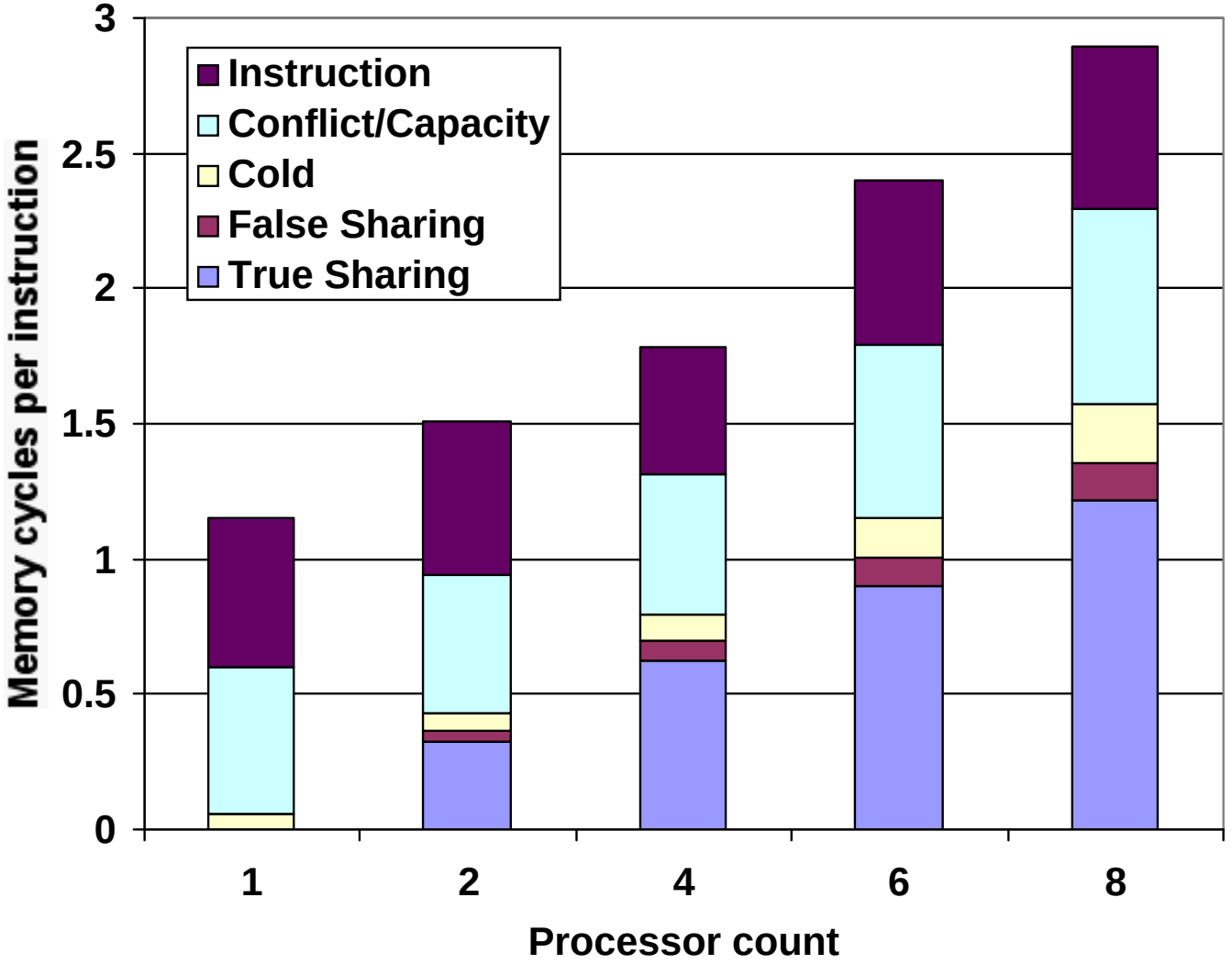
MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)
- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)



MP Performance 2MB Cache
Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs



A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (0) Determine when to invoke coherence protocol
  - (a) Find info about state of address in other caches to determine action
    - whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (invalidate/update)
- (0) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an "access fault" occurs on the line
- Different approaches distinguished by (a) to (c)

Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a “search”
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn’t scale with number of processors, P
  - on bus, bus bandwidth doesn’t scale
  - on scalable network, every fault leads to at least P network transactions
- Scalable coherence:
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol

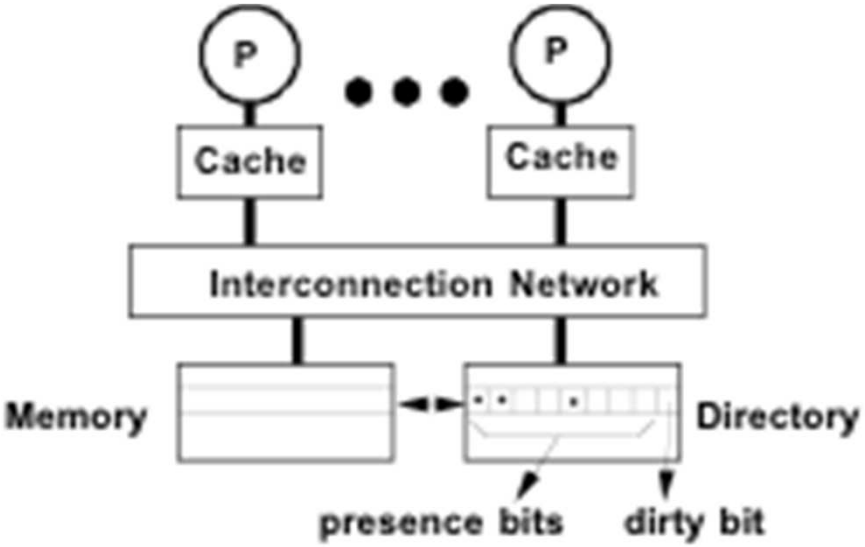
Need for a more scalable protocol

- Snoopy schemes do not scale because they rely on broadcast
- Hierarchical snoopy schemes have the root as a bottleneck
- **Directory** based schemes allow scaling
  - They avoid broadcasts by keeping track of all CPUs caching a memory block, and then using point-to-point messages to maintain coherence
  - They allow the flexibility to use any scalable point-to-point network

Scalable Approach: Directories

- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Basic Operation of Directory



- k processors.
- With each cache-block in memory:
  k presence-bits, 1 dirty-bit
- With each cache-block in cache:
  1 valid bit, and 1 dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit ON then { recall line from dirty proc (downgrade cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}
- Write to main memory by processor i:
  - If dirty-bit OFF then {send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }