Parallelism, Multicore, and Synchronization

Anne Bracy

CS 3410

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer. Also some slides from Amir Roth & Milo Martin in here.

P & H Chapter 4.10, 1.7, 1.8, 5.10, 6

Performance Improvement 101

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

2 Classic Goals of Architects:

⬇ Clock period (⬆ Clock frequency)

⬇ Cycles per Instruction (⬆ IPC)

Clock frequencies have stalled

Darling of performance improvement for decades

Why is this no longer the strategy?

Hitting Limits:

- Pipeline depth
- Clock frequency
- Moore’s Law & Technology Scaling
- Power

Improving IPC via ILP

Exploiting Intra-instruction parallelism:

Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP):

Multiple issue pipeline (2-wide, 4-wide, *etc.*)

- Statically detected by compiler (VLIW)
- Dynamically detected by HW

Dynamically Scheduled (OoO)

Static Multiple Issue

a.k.a. Very Long Instruction Word (VLIW)

Compiler groups instructions to be issued together

- Packages them into “issue slots”

How does HW detect and resolve hazards?

It doesn’t. ☺ Compiler must avoid hazards

Example: Static Dual-Issue 32-bit MIPS

- Instructions come in pairs (64-bit aligned)
  - One ALU/branch instruction (or nop)
  - One load/store instruction (or nop)

MIPS with Static Dual Issue

Two-issue packets

- One ALU/branch instruction
- One load/store instruction
- 64-bit aligned
  - ALU/branch, then load/store
  - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Scheduling Example

Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)        # $t0=array element
      addu $t0, $t0, $s2      # add scalar in $s2
      sw   $t0, 0($s1)        # store result
      addi $s1, $s1,–4        # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,–4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

Clicker Question: What is the IPC of this machine?

(A) 0.8 (B) 1.0 (C) 1.25 (D) 1.5 (E) 2.0

Techniques and Limits of Static Scheduling

Goal: larger instruction windows (to play with)

- Predication
- Loop unrolling
- Function in-lining
- Basic block modifications (superblocks, *etc.*)

Roadblocks

- Memory dependences (aliasing)
- Control dependences

Improving IPC via ILP

Exploiting Intra-instruction parallelism:

Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP):

Multiple issue pipeline (2-wide, 4-wide, *etc.*)

- Statically detected by compiler (VLIW)
- Dynamically detected by HW

Dynamically Scheduled (OoO)

Dynamic Multiple Issue

aka SuperScalar Processor (c.f. Intel)

- CPU chooses multiple instructions to issue each cycle
- Compiler can help, by reordering instructions….
- … but CPU resolves hazards

Even better: Speculation/Out-of-order Execution

- Execute instructions as early as possible
- Aggressive register renaming (indirection to the rescue!)
- Guess results of branches, loads, etc.
- Roll back if guesses were wrong
- Don’t commit results until all previous insns committed

Effectiveness of OoO Superscalar

It was awesome, but then it stopped improving

Limiting factors?

- Programs dependencies
- Memory dependence detection → be conservative
  - e.g. Pointer Aliasing: A[0] += 1; B[0] *= 2;
- Hard to expose parallelism
  - Still limited by the fetch stream of the static program
- Structural limits
  - Memory delays and limited bandwidth
- Hard to keep pipelines full, especially with branches

Improving IPC via ~~ILP~~ TLP

Exploiting Thread-Level parallelism

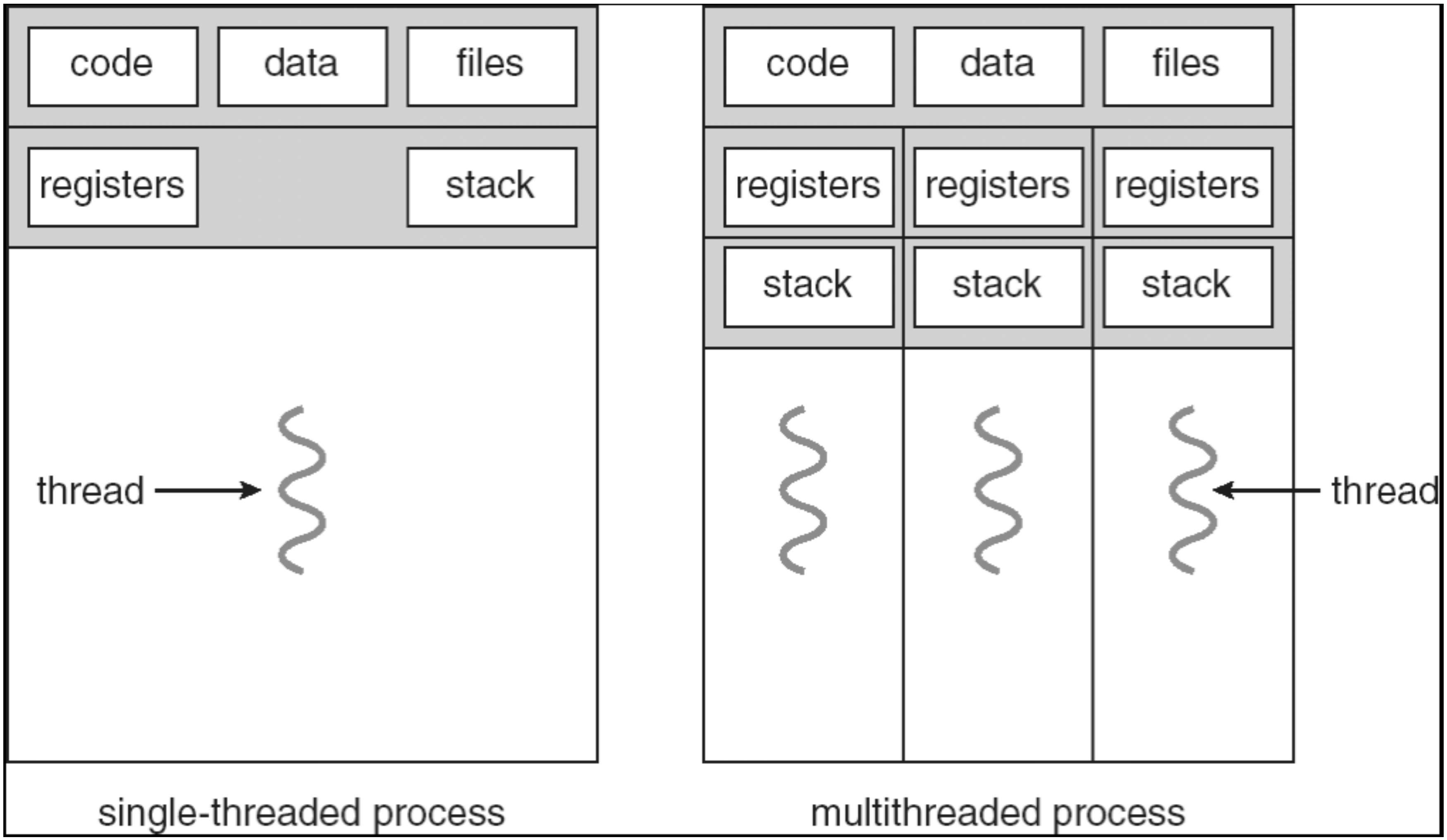
Hardware multithreading to improve utilization:

- Multiplexing multiple threads on single CPU
- Sacrifices latency for throughput
- Single thread cannot fully utilize CPU? *Try more!*
- Three types:
  - Course-grain (has preferred thread)
  - Fine-grain (round robin between threads)
  - Simultaneous (hyperthreading)

What is a thread?

Process: multiple threads, code, data and OS state

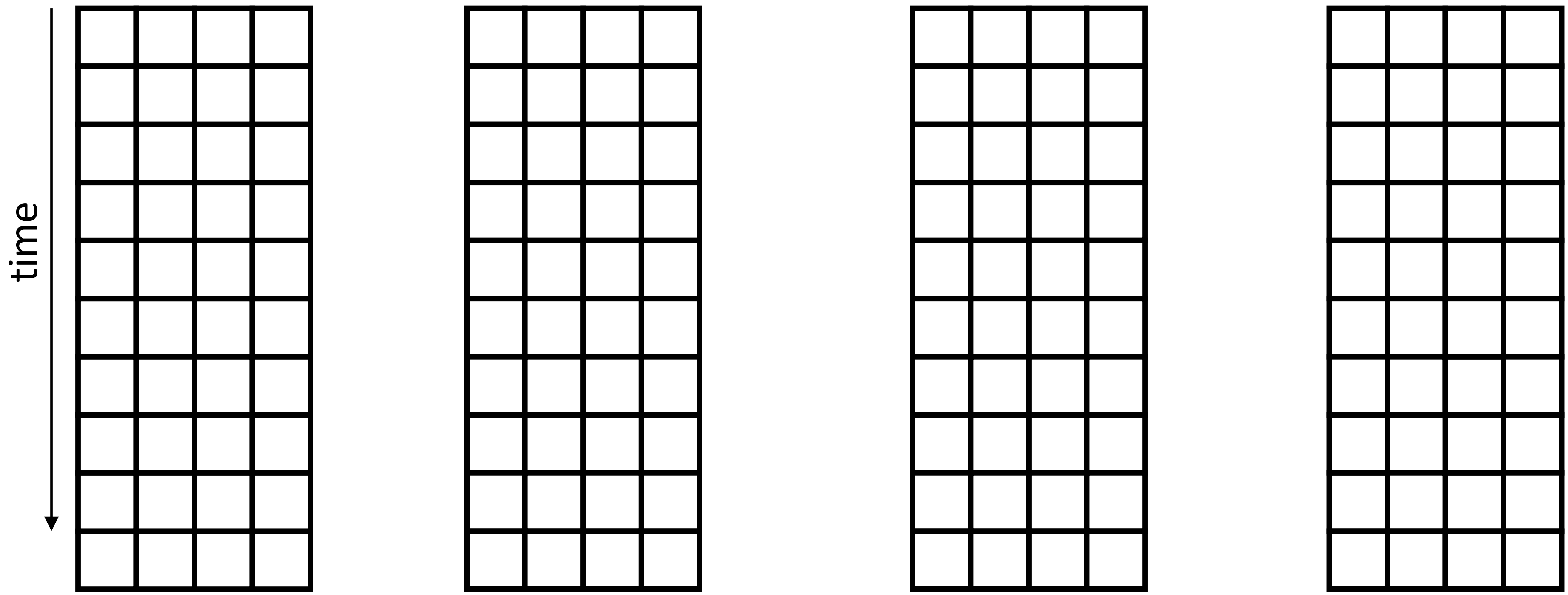
Threads: share code, data, files, **not** regs or stack



Standard Multithreading Picture

Time evolution of issue slots

- Color = thread, white = no instruction



4-wide Superscalar

CGMT

Switch to thread B on thread A L2 miss

FGMT

Switch threads every cycle

SMT

Insns from multiple threads coexist

Power Efficiency

CPU	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
Core i5 Nehal	2010	3300MHz	14	4	Yes	1	87W
Core i5 Ivy Br	2012	3400MHz	14	4	Yes	8	77W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

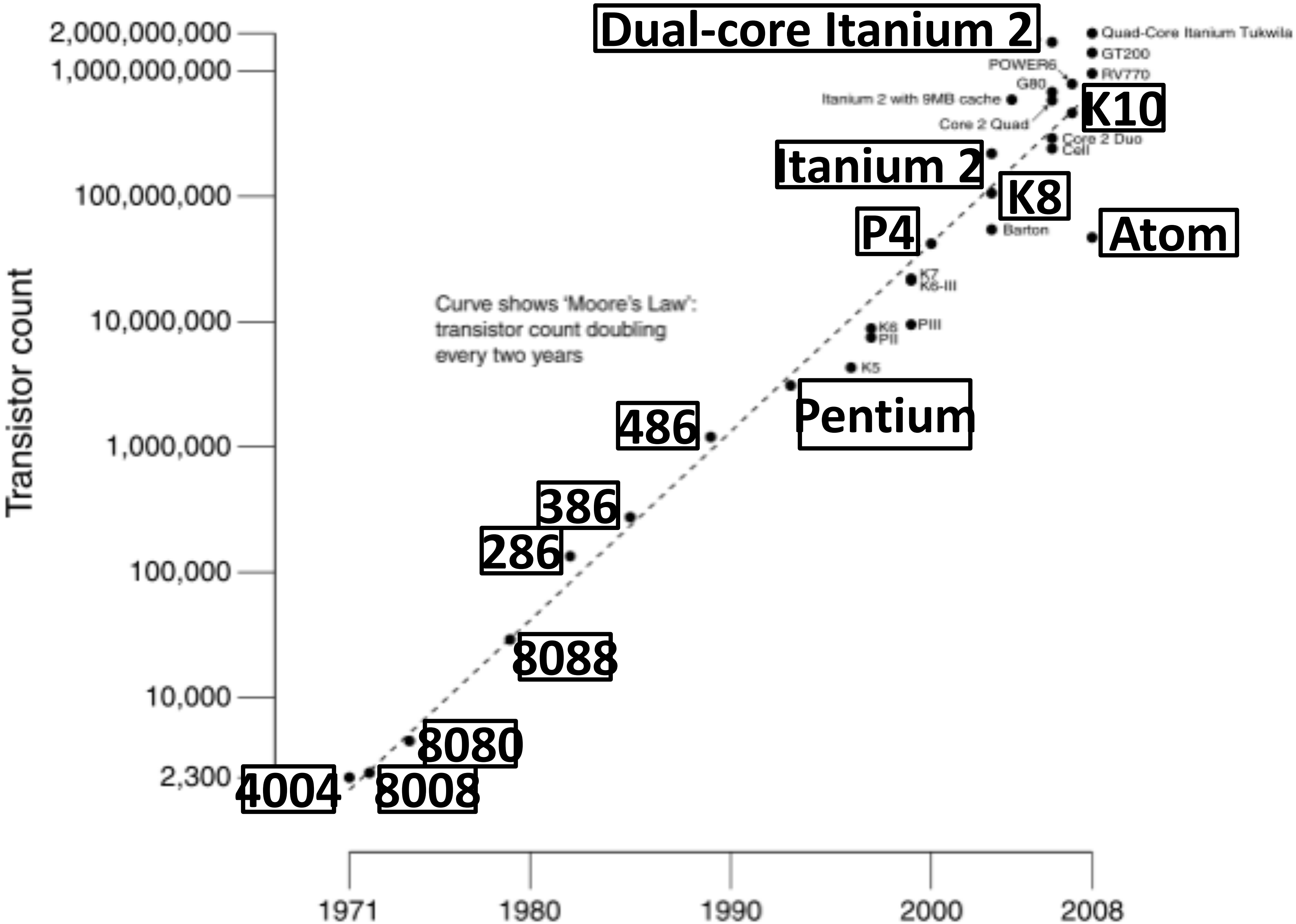
Those simpler cores did something very right.

Why Multicore?

Moore's law

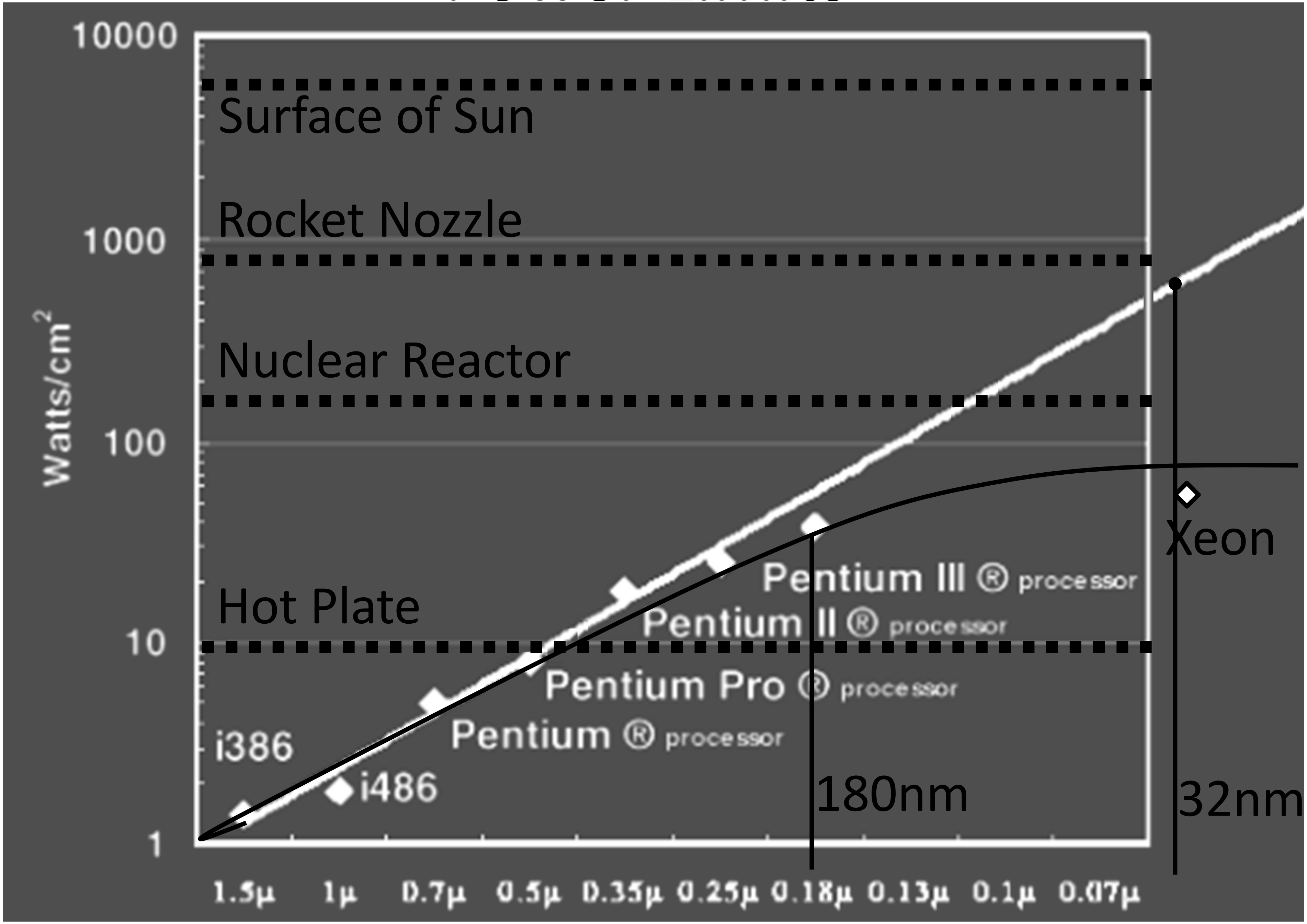
- A law about transistors
- Smaller means more transistors per die
- And smaller means faster too

But: Power consumption growing too…



Transistor count
2,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2008
Curve shows 'Moore's Law': transistor count doubling every two years
Dual-core Itanium 2
Quad-Core Itanium Tukwila
GT200
POWER6
RV770
G80
Itanium 2 with 9MB cache
K10
Core 2 Quad
Core 2 Duo
Cell
Itanium 2
K8
P4
Barton
Atom
K7
K6-III
K6
PII
PIII
K5
Pentium
486
386
286
8088
8080
4004
8008

Power Limits



Power Wall

Power = capacitance * voltage^2 * $\underbrace{\text{frequency}}_{\text{Lower Frequency}}$

In practice: Power ~ voltage^3

Reducing voltage helps (a lot)

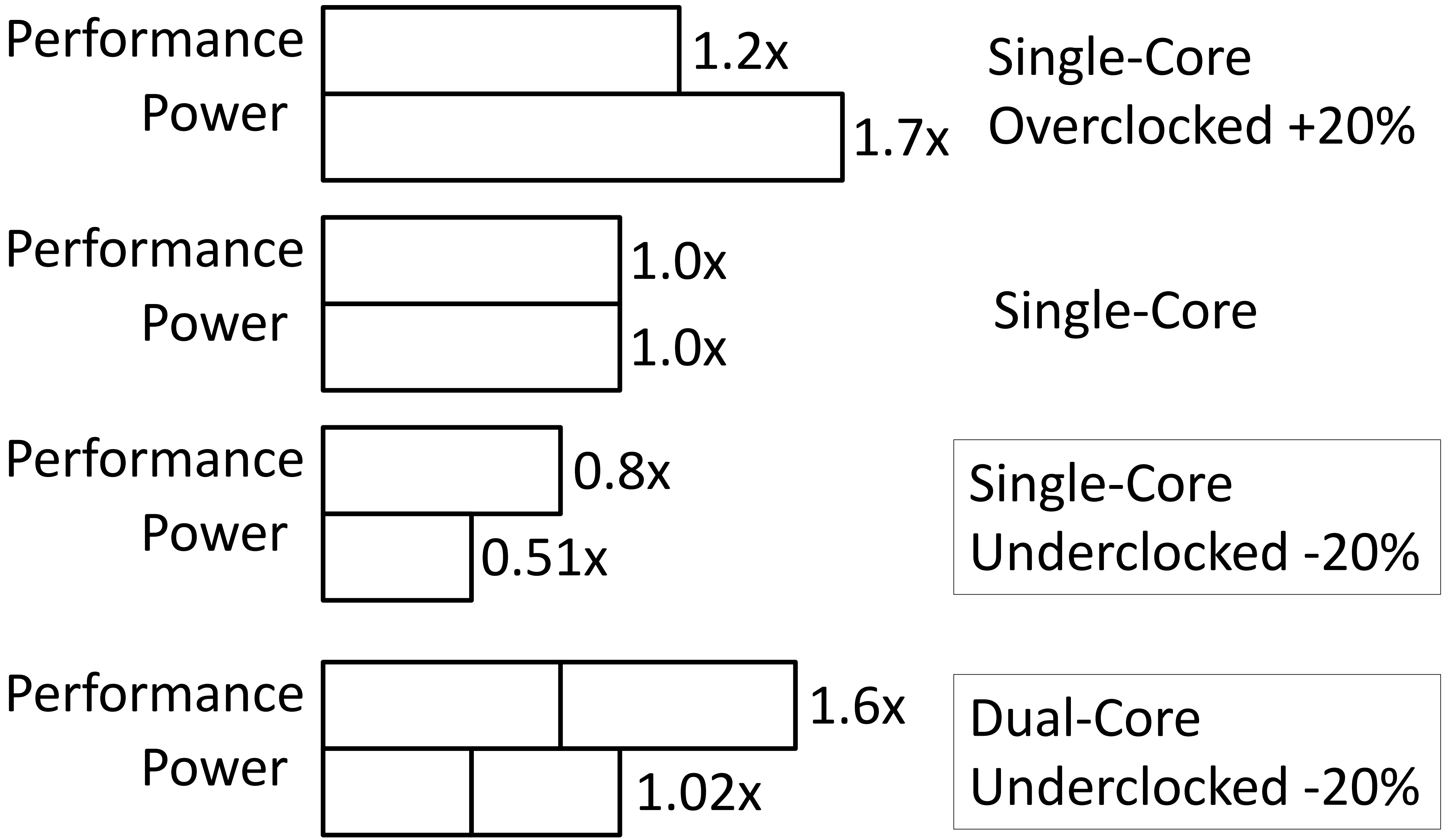
... so does reducing clock speed

Better cooling helps

The power wall

- We can’t reduce voltage further
- We can’t remove more heat

Why Multicore?



Parallel Programming

Q: So lets just all use multicore from now on!

A: Software must be written as parallel program

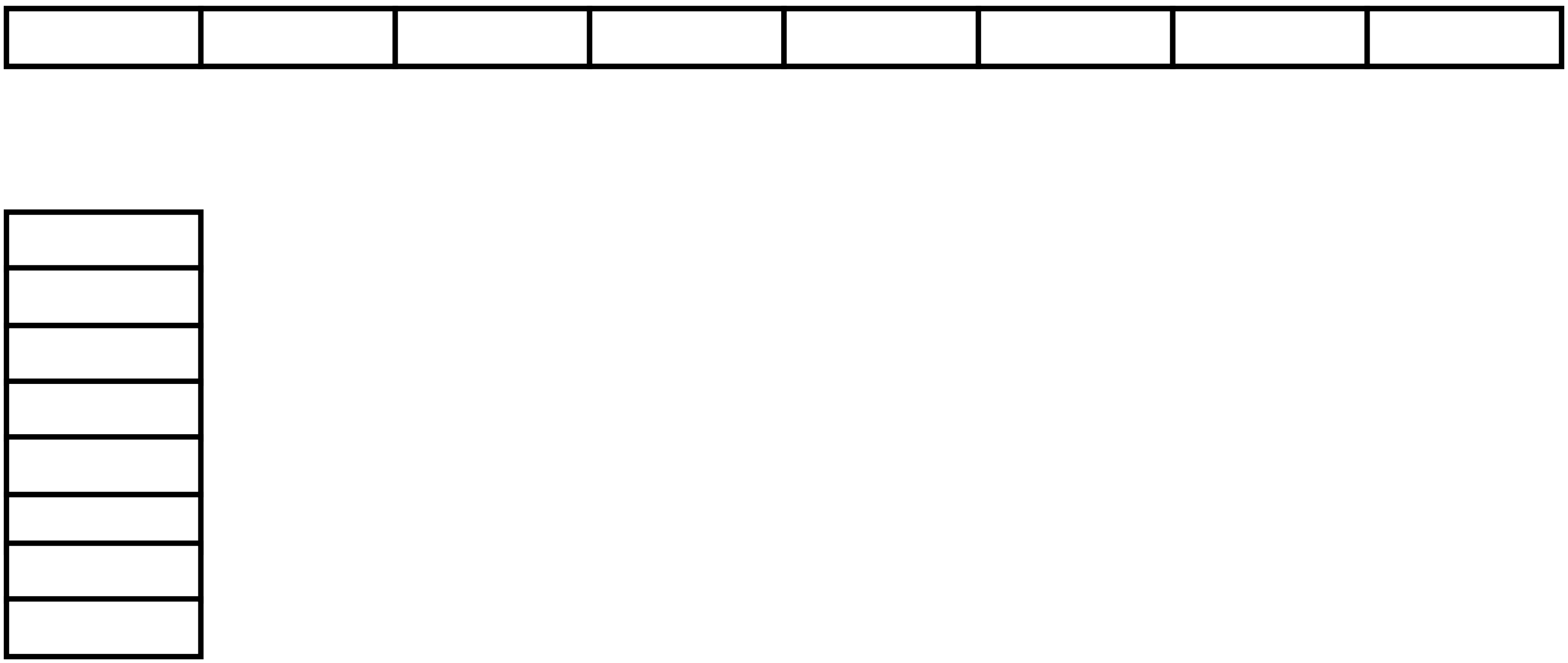
Multicore difficulties

- Partitioning work
- Coordination & synchronization
- Communications overhead
- How do you write parallel programs?

  ... without knowing exact underlying architecture?

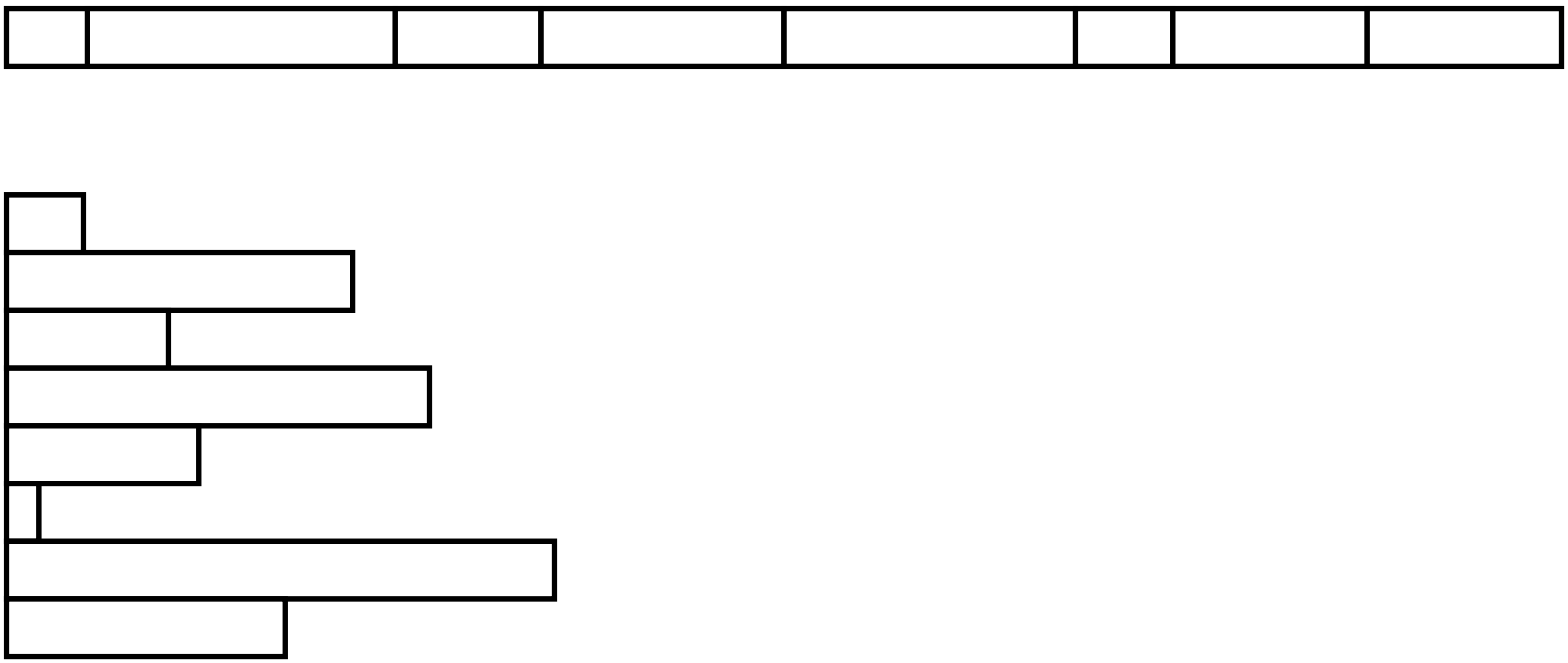
Work Partitioning

Partition work so all cores have something to do



Load Balancing

Need to partition so all cores are actually working



Amdahl's Law

If tasks have a serial part and a parallel part…

Example:

step 1: divide input data into *n* pieces

step 2: do work on each piece

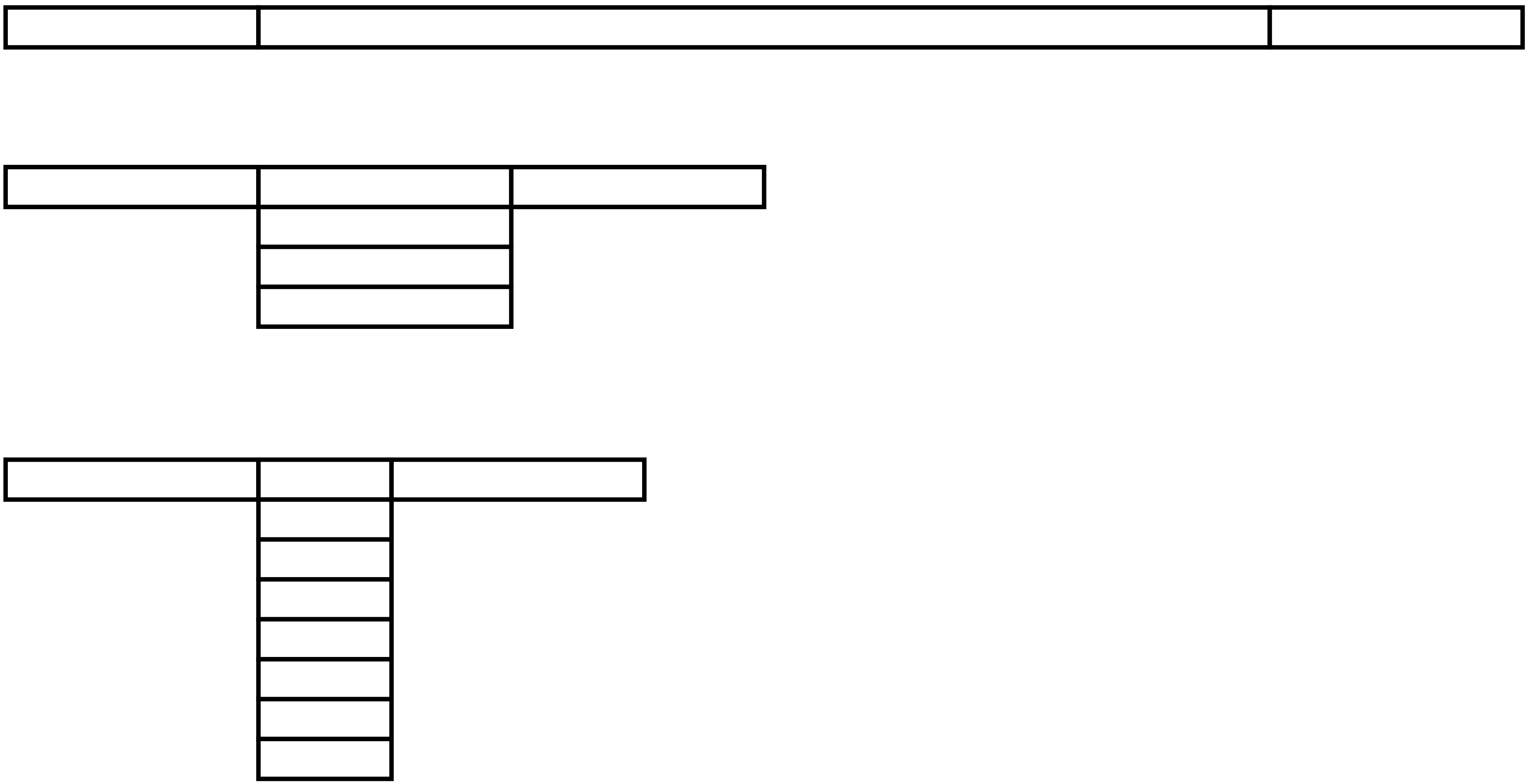
step 3: combine all results

Recall: Amdahl's Law

As number of cores increases …

- time to execute parallel part? goes to zero
- time to execute serial part? Remains the same
- *Serial part eventually dominates*

Amdahl's Law



Parallelism is a necessity

Necessity, not luxury

- Power wall

Not easy to get performance out of

Many solutions

- Pipelining
- Multi-issue
- Multithreading
- Multicore

Parallel Programming

Q: So lets just all use multicore from now on!

A: Software must be written as parallel program

Multicore difficulties

- Partitioning work
- Coordination & synchronization
- Communications overhead
- How do you write parallel programs?
  ... without knowing exact underlying architecture?

Parallelism & Synchronization

Cache Coherency

- Processors cache *shared* data → they see different (incoherent) values for the *same* memory location

Synchronizing parallel programs

- Atomic Instructions
- HW support for synchronization

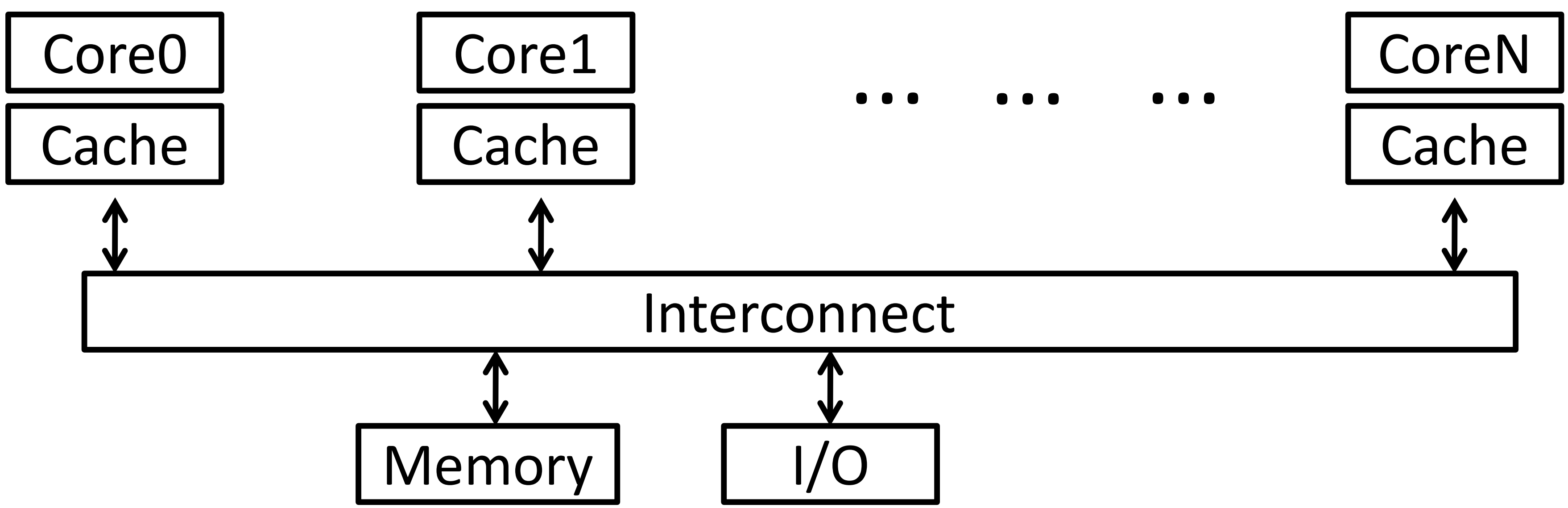
How to write parallel programs

- Threads and processes
- Critical sections, race conditions, and mutexes

Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- Hardware provides *single physical address* space for all processors



Cache Coherency Problem

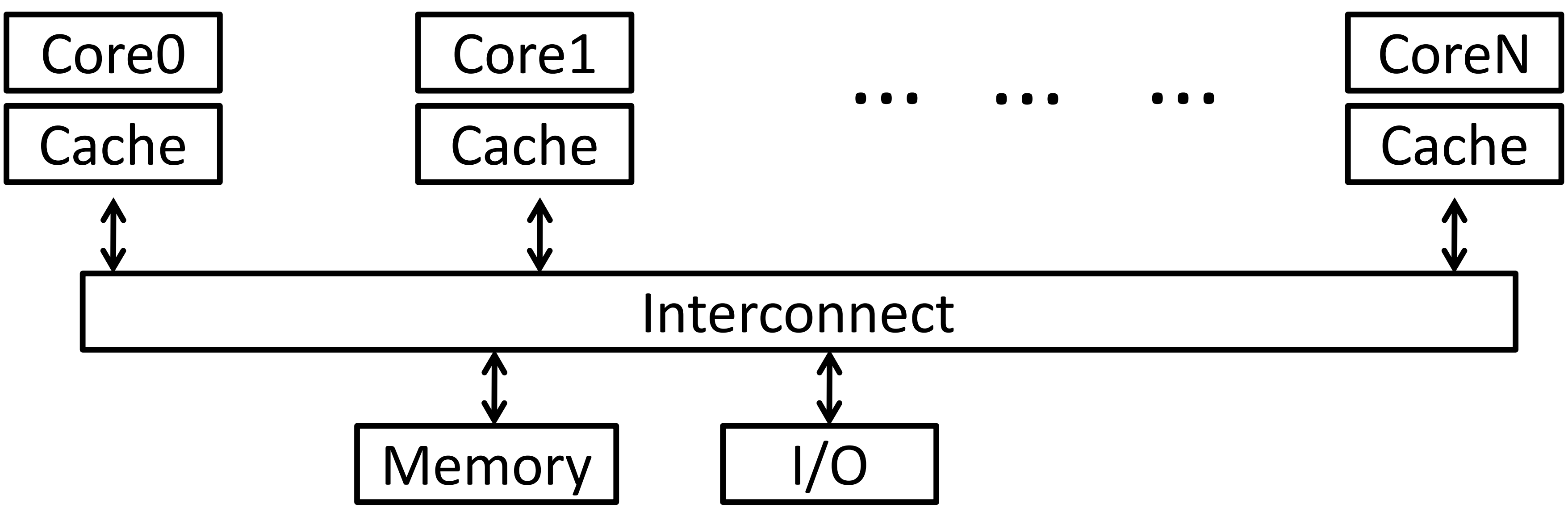
Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
        x = x + 1;
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        x = x + 1;
}
```

What will the value of x be after both loops finish?



Cache Coherency Problem

Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
        x = x + 1;
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        x = x + 1;
}
```

What will the value of x be after both loops finish?

a) 6
b) 8
c) 10
d) Could be any of the above
e) Couldn't be any of the above

Cache Coherency Problem, WB $

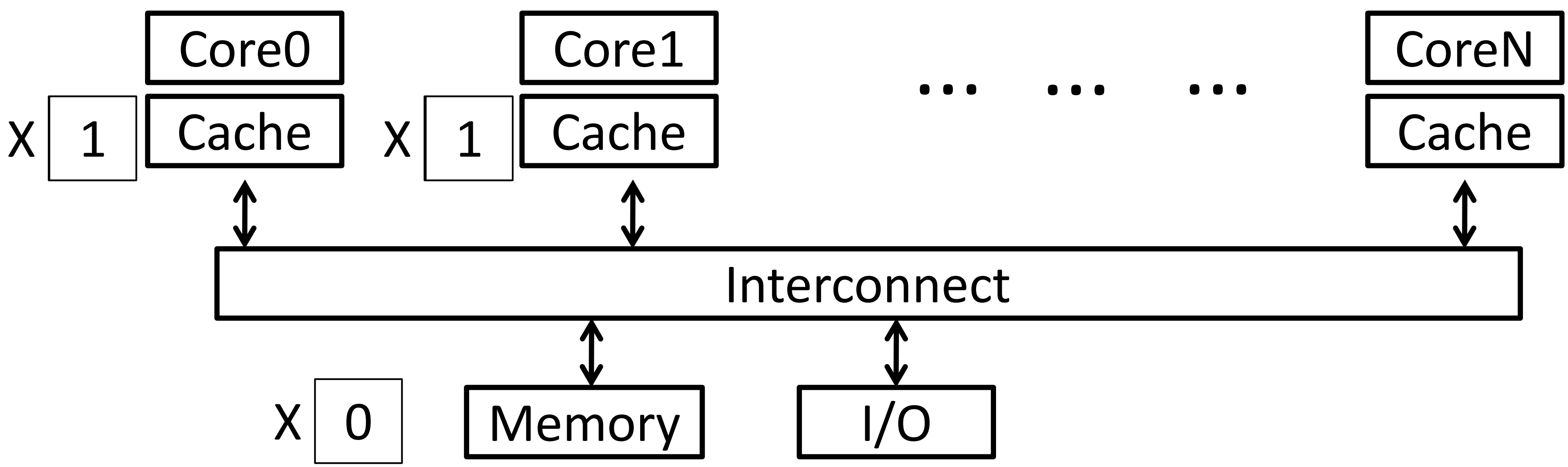
Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
$t0=0    LW $t0, addr(x)
$t0=1    ADDIU $t0, $t0, 1
x=1      SW $t0, addr(x)
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
$t0=0    LW $t0, addr(x)
$t0=1    ADDIU $t0, $t0, 1
x=1      SW $t0, addr(x)
}
```

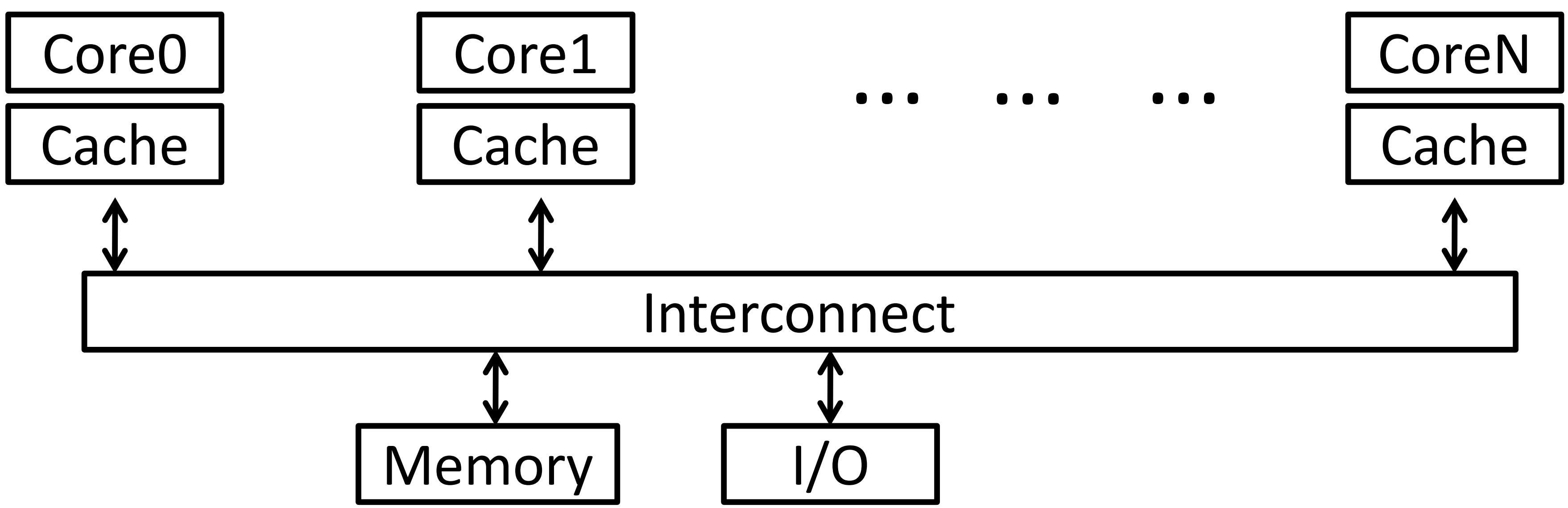
Problem!



Not just a problem for Write-Back Caches

Executing on a write-thru cache:

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**



Two issues

Coherence

- What values can be returned by a read
- Need a globally uniform (consistent) view of a single memory location

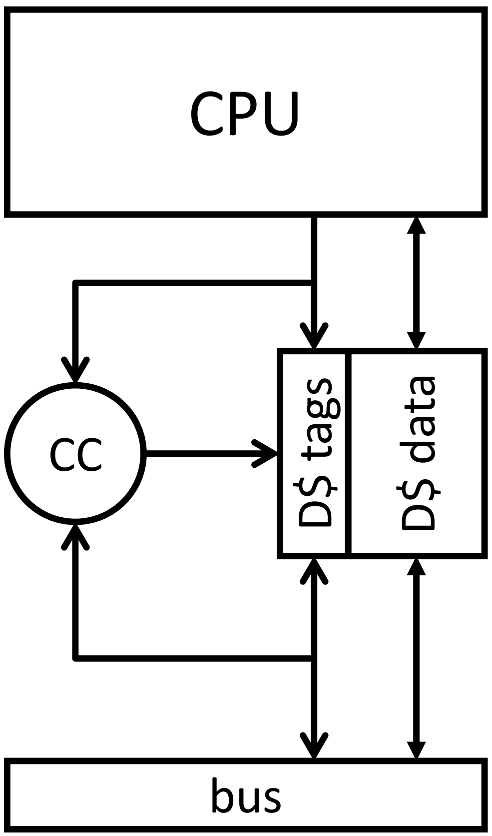
Solution: Cache Coherence Protocols

Consistency

- When a written value will be returned by a read
- Need a globally uniform (consistent) view of *all memory locations relative to each other*

Solution: Memory Consistency Models

Hardware Cache Coherence



Coherence

- all copies have same data at all times

Coherence controller:

- Examines bus traffic (addresses and data)
- Executes **coherence protocol**
  - What to do with local copy when you see different things happening on bus

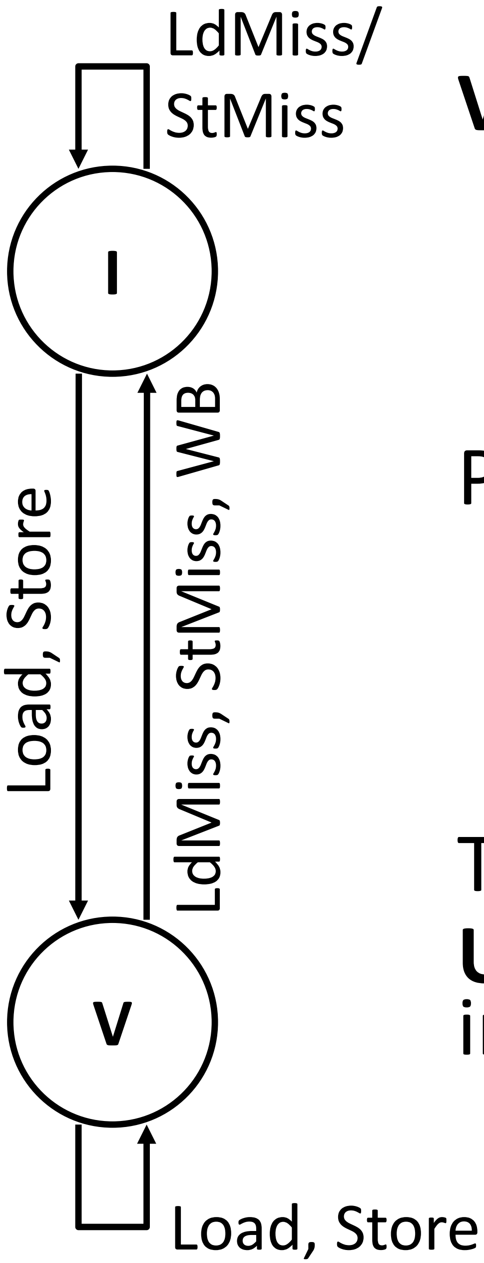
Three processor-initiated events

- **Ld**: load
- **St**: store
- **WB**: write-back

Two remote-initiated events

- **LdMiss**: read miss from ***another*** processor
- **StMiss**: write miss from ***another*** processor

VI Coherence Protocol



VI (valid-invalid) protocol:

- Two states (per block in cache)
  - **V (valid)**: have block
  - **I (invalid)**: don't have block
  - \+ Can implement with valid bit

Protocol diagram (left)

- If *you* load/store a block: transition to **V**
- If anyone *else* wants to read/write block:
  - Give it up: transition to **I** state
  - Write-back if your own copy is dirty

This is an **invalidate protocol**

Update protocol: copy data, don't invalidate

- Sounds good, but wastes a lot of bandwidth

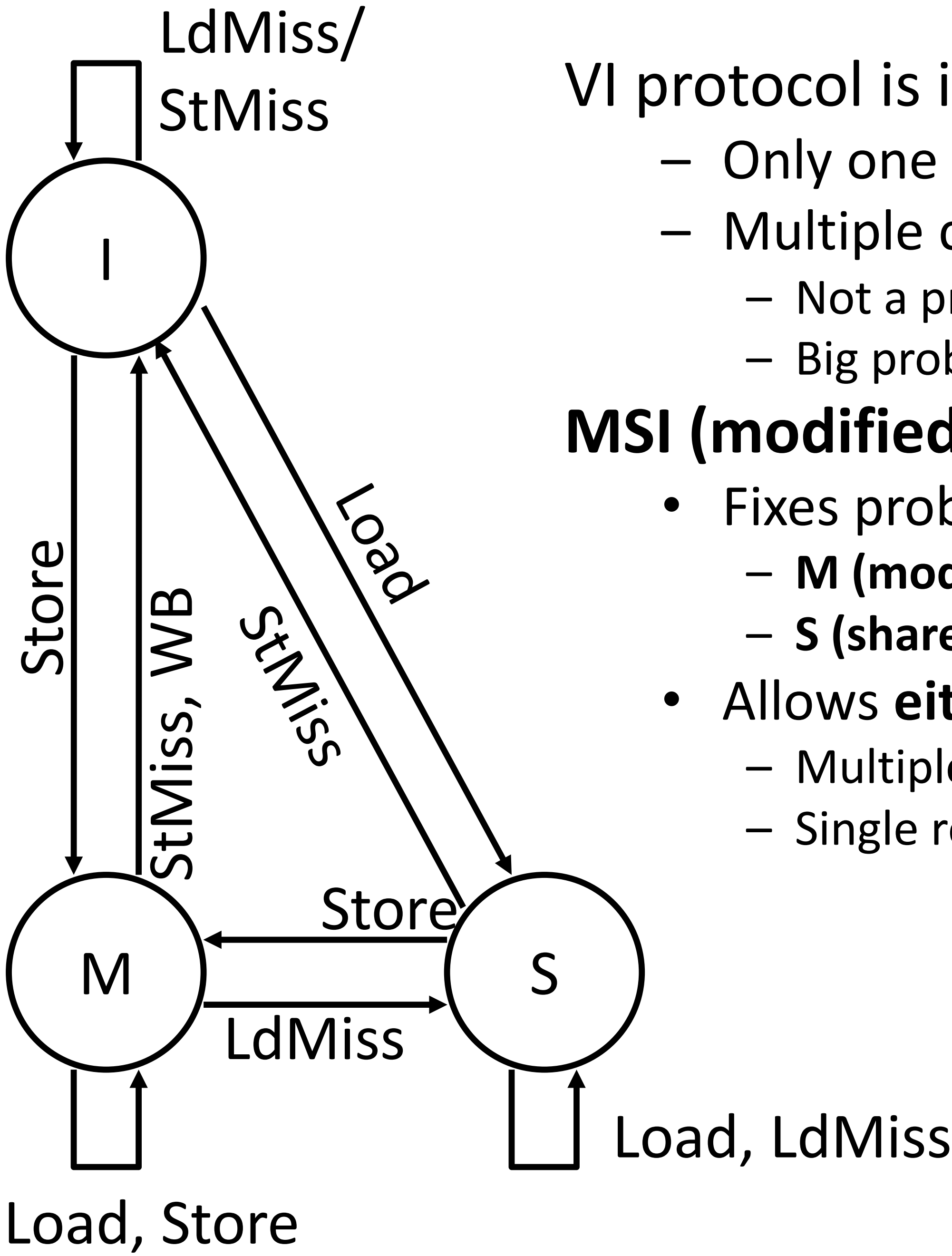
VI Protocol (Write-Back Cache)

Thread A	Thread B	CPU0	CPU1	Mem
				0
`lw t0, 0(r3),`		V:0		0
`ADDIU $t0,$t0,1`				
`sw t0,0(r3)`		V:1		0
	`lw t0, 0(r3)`	I:	V:1	1
	`ADDIU $t0,$t0,1`			
	`sw t0,0(r3)`		V:2	1

`lw` by Thread B generates an "other load miss" event (LdMiss)

- Thread A responds by sending its dirty copy, transitioning to **I**

VI → MSI



VI protocol is inefficient

- Only one cached copy allowed in entire system
- Multiple copies can't exist even if read-only
  - Not a problem in example
  - Big problem in reality

MSI (modified-shared-invalid)

- Fixes problem: splits "V" state into two states
  - **M (modified)**: local dirty copy
  - **S (shared)**: local clean copy
- Allows **either**
  - Multiple read-only copies (S-state) **--OR--**
  - Single read/write copy (M-state)

MSI Protocol (Write-Back Cache)

Thread A	Thread B	CPU0	CPU1	Mem
				0
`lw t0, 0(r3),`		S:0		0
`ADDIU $t0,$t0,1`				
`sw t0,0(r3)`		M:1		0
	`lw t0, 0(r3),`	S:1	S:1	1
	`ADDIU $t0,$t0,1`			
	`sw t0,0(r3)`	I:	M:2	1

`lw` by Thread B generates a "other load miss" event (LdMiss)

- Thread A responds by sending its dirty copy, transitioning to **S**

`sw` by Thread B generates a "other store miss" event (StMiss)

- Thread A responds by transitioning to **I**

Cache Coherence and Cache Misses

Coherence introduces two new kinds of cache misses

- **Upgrade miss**
  - On stores to read-only blocks
  - Delay to acquire write permission to read-only block
- **Coherence miss**
  - Miss to a block evicted by another processor's requests

Making the cache larger…

- Doesn't reduce these type of misses
- As cache grows large, these sorts of misses dominate

False sharing

- Two or more processors sharing parts of the same block
- But *not* the same bytes within that block (no actual sharing)
- Creates pathological "ping-pong" behavior
- Careful data placement may help, but is difficult

More Cache Coherence

In reality: many coherence protocols

- Snooping: VI, MSI, MESI, MOESI, …
  - But Snooping doesn't scale
- Directory-based protocols
  - Caches & memory record blocks' sharing status in directory
  - Nothing is free → directory protocols are slower!

Cache Coherency:

- requires that reads return most recently written value
- Is a hard problem!

Are We Done Yet?

Thread A	Thread B	CPU0	CPU1	Mem
				0
`lw t0, 0(r3)`		S:0		0
	`lw t0, 0(r3)`	S:0	S:0	0
	`ADDIU $t0,$t0,1`			
	`sw t0,0(x)`	I:	M:1	0
`ADDIU $t0,$t0,1`				
`sw t0,0(x)`		M:1	I:	1

What just happened???

Is MSI Cache Coherency Protocol Broken??

Programming with threads

Within a thread: execution is sequential

Between threads?

- No ordering or timing guarantees
- Might even run on different cores at the same time

Problem: hard to program, hard to reason about

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Cache coherency is necessary but **not** sufficient…

Need explicit synchronization to make guarantees about concurrent threads!

Race conditions

Timing-dependent error involving access to shared state

Race conditions depend on how threads are scheduled

- i.e. who wins “races” to update state

Challenges of Race Conditions

- Races are intermittent, may occur rarely
- Timing dependent = small changes can hide bug

Program is correct *only* if *all possible* schedules are safe

- Number of possible schedules is huge
- Imagine adversary who switches contexts at worst possible time

Hardware Support for Synchronization

Atomic read & write memory operation

- Between read & write: *no writes to that address*

Many atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and exchange (x86, ARM deprecated)
- linked load / store conditional (pair of insns) (MIPS, ARM, PowerPC, DEC Alpha, …)

Synchronization in MIPS

Load linked: `LL rt, offset(rs)`

"I want the value at address X. Also, start monitoring any writes to this address."

Store conditional: `SC rt, offset(rs)`

"If no one has changed the value at address X since the LL, perform this store and tell me it worked."

- Data at location has not changed since the `LL`?
  - SUCCESS:
    - Performs the store
    - Returns 1 in rt
- Data at location has changed since the LL?
  - FAILURE:
    - Does not perform the store
    - Returns 0 in rt

Using LL/SC to create Atomic Increment

Load linked: `LL rt, offset(rs)`

Store conditional: `SC rt, offset(rs)`

i++

↓

```
LW $t0, 0($s0)
ADDIU $t0, $t0, 1
SW $t0, 0($s0)
```

⟶

atomic(i++)

↓

```
try:  LL $t0, 0($s0)
      ADDIU $t0, $t0, 1
      SC $t0, 0($s0)
      BEQZ $t0, try
```

Value in memory changed between LL and SC ?

→ SC returns 0 in $t0 → retry

Atomic Increment in Action

Load linked: `LL $t0, offset($s0)`

Store conditional: `SC $t0, offset($s0)`

Time	Thread A	Thread B	Thread A $t0	Thread B $t0	Mem [$s0]
0					0
1	try: LL $t0, 0($s0)		0		0
2		try: LL $t0, 0($s0)		0	0
3	ADDIU $t0, $t0, 1		1	0	0
4		ADDIU $t0, $t0, 1	1	1	0
5	SC $t0, 0($s0)		**1**	1	1
6	BEQZ $t0, try		1	1	1
7		SC $t0, 0 ($s0)	1	**0**	1
8		BEQZ $t0, try	1	0	1

Success!

Failure!

Critical Sections

Create atomic version of every instruction? NO

Does not scale *or solve the problem*

To eliminate races: identify *Critical Sections*

- only one thread can be in
- Contending threads must wait to enter

time

T1

```
CSEnter();
  Critical
  section
CSExit();
```

T1

T2

```
CSEnter();
  # wait
  # wait
  Critical
  section
CSExit();
```

T2

Mutual Exclusion Lock (Mutex)

Implementation of CSEnter and CSExit

- Only one thread can hold the lock at a time

  “I have the lock”

Mutex from LL and SC

This is called a
Spin lock
aka spin waiting

```
m = 0;
mutex_lock(int *m) {
  test_and_set: LI $t0, 1
                LL $t1, 0($a0)
                BNEZ $t1, test_and_set
                SC $t0, 0($a0)
                BEQZ $t0, test_and_set
}

mutex_unlock(int *m) {
      SW $zero, 0($a0)
}
```

2 threads attempt to grab the lock

`mutex_lock(int *m)`

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			$t0	$t1	$t0	$t1	M[$a0]
0							0
1	try: LI $t0, 1	try: LI $t0, 1	1		1		0
2	LL $t1, 0($a0)	LL $t1, 0($a0)	1	0	1	0	0
3	BNEZ $t1, try	BNEZ $t1, try	1	0	1	0	0
4		SC $t0, 0 ($a0)			**1**	0	1
5	SC $t0, 0($a0)		0	0	1	0	1
6	BEQZ $t0, try	BEQZ $t0, try	0	0	1	0	1
7	try: LI $t0, 1	Critical section					

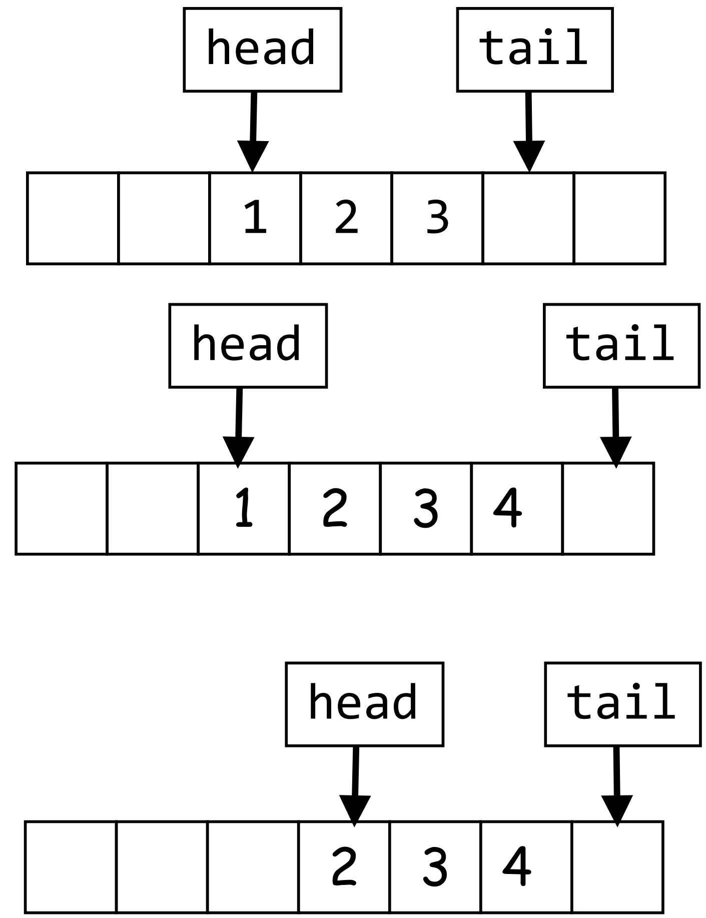
Failure! *Success!*

Producer/Consumer Example (1)

```
// invariant:
// data in A[h … t-1]
char A[100];
int h = 0, t = 0;

// producer: add to tail if room
void put(char c) {
  A[t] = c;
  t = (t+1)%n;
}
// consumer: take from head
char get() {
  while (t == h) { };
  char c = A[h];
  h = (h+1)%n;
  return c;
}
```

Goal: enforce data structure invariants



Producer/Consumer Example (2)

```
// invariant:
// data in A[h … t-1]
char A[100];
int h = 0, t = 0;

// producer: add to tail if room
void put(char c) {
  A[t] = c;
  t = (t+1)%n;
}

// consumer: take from head
char get() {
  while (t == h) { };
  char c = A[h];
  h = (h+1)%n;
  return c;
}
```

Goal: enforce data structure invariants

Clicker Q:
What's wrong here?

a) Will lose update to ***t*** and/or ***h***
b) Invariant is not upheld
c) Will produce if full
d) Will consume if empty
e) All of the above

Producer/Consumer Example (3)

```
// invariant:
// data in A[h … t-1]
char A[100];
int h = 0, t = 0;

// producer: add to tail if room
void put(char c) {
  A[t] = c;
  t = (t+1)%n;    ←
}
// consumer: take from head
char get() {
  while (t == h) { }; ←
  char c = A[h];
  h = (h+1)%n;    ←
  return c;
}
```

Goal: enforce data structure invariants

What's wrong here?

- Could miss an update to *t* or *h*
- Breaks invariants: only produce if not full, only consume if not empty

→ *Need to synchronize access to shared data*

Producer/Consumer Example (4)

```
// invariant:
// data in A[h … t-1]
char A[100];
int h = 0, t = 0;

// producer: add to tail if room
void put(char c) {
                    acquire-lock()
  A[t] = c;
  t = (t+1)%n;
                    release-lock()
}
// consumer: take from head
char get() {
                    acquire-lock()
  while (t == h) { };
  char c = A[h];
  h = (h+1)%n;
                    release-lock()
  return c;
}
```

Goal: enforce data structure invariants

Rule of thumb:

all access & updates that can affect the invariant become critical sections

Does this fix work?

Language-level Synchronization

Lots of synchronization variations…

Reader/writer locks

- Any number of threads can hold a read lock
- Only one thread can hold the writer lock

Semaphores

- N threads can hold lock at the same time

Monitors

- Concurrency-safe data structure with 1 mutex
- All operations on monitor acquire/release mutex
- One thread in the monitor at a time