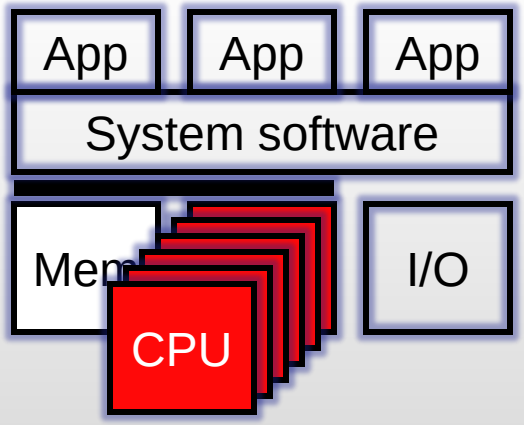
# Thread Level Parallelism I: Multicores

(Shared Memory Multiprocessors)

Readings:

H&P: Chapter 4

# This Unit: Shared Memory Multiprocessors




- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

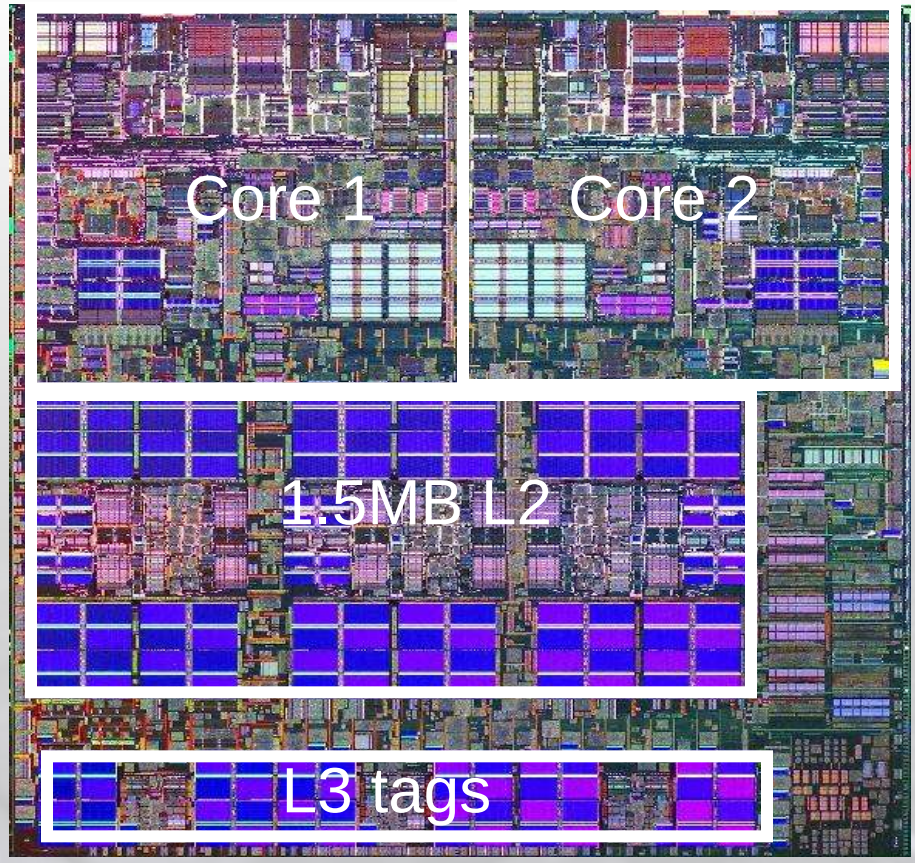
# Multiplying Performance

- A single processor can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism
  - Limited cache hierarchy

- What if we need even more computing power?
  - Use multiple processors!
  - But how?

- High-end example: Sun Ultra Enterprise 25k
  - 72 UltraSPARC IV+ processors, 1.5Ghz
  - 1024 GBs of memory
  - Niche: large database servers
  - $$$

# Multicore: Mainstream Multiprocessors




Why multicore? What else would you do with 500 million transistors?

- **Multicore chips**
- **IBM Power5**
  - Two 2+GHz PowerPC cores
  - Shared 1.5 MB L2, L3 tags
- AMD Quad Phenom
  - Four 2+ GHz cores
  - Per-core 512KB L2 cache
  - Shared 2MB L3 cache
- Intel Core 2 Quad
  - Four cores, shared 4 MB L2
  - Two 4MB L2 caches
  - It is a dual die chip
- Sun Niagara
  - 8 simple cores, each 4-way threaded
  - Shared 2MB L2, shared FP
  - For servers, not desktop

# Roadmap of near-future Multicores

- Intel
  - …/P6/NetBurst/Core/Nehalem/…
  - (tick) Nehalem-EP (Beckton) (45nm): 4 cores per die (up to 8 cores per chip in dual die chips) [now]
  - (tack) Westmere (32nm): 6-8 Cores per die [3Q, 2009]
  - (tick) Sandy Bridge (32nm): 6-8 Cores per die [2010]
  - (tack) Ivy Bridge (22nm): 16 Cores per die? 32 per chip? [2011?]
- AMD:
  - *San Marino*: 6 Cores per chip [2Q, 2009]
  - *Magny-cours*: 8-12 Cores per chip [2010]
  - *Interlagos*: 12-16 Cores per chip [2011]
- Sun Microsystems (aka Oracle ?)
  - Sun Rock: 16 Cores per chip/die (each 4-way threaded) [2009?]

# Application Domains for Multiprocessors

- Scientific computing/supercomputing
  - Examples: weather simulation, aerodynamics, protein folding
  - Large grids, integrating changes over time
  - Each processor computes for a part of the grid
- Server workloads
  - Example: airline reservation database
  - Many concurrent updates, searches, lookups, queries
  - Processors handle different requests
- Media workloads
  - Processors compress/decompress different parts of image/frames
- Desktop workloads…
- Gaming workloads…
- Cloud computing…

**But software must be written to expose parallelism**

# But First, Uniprocessor Concurrency

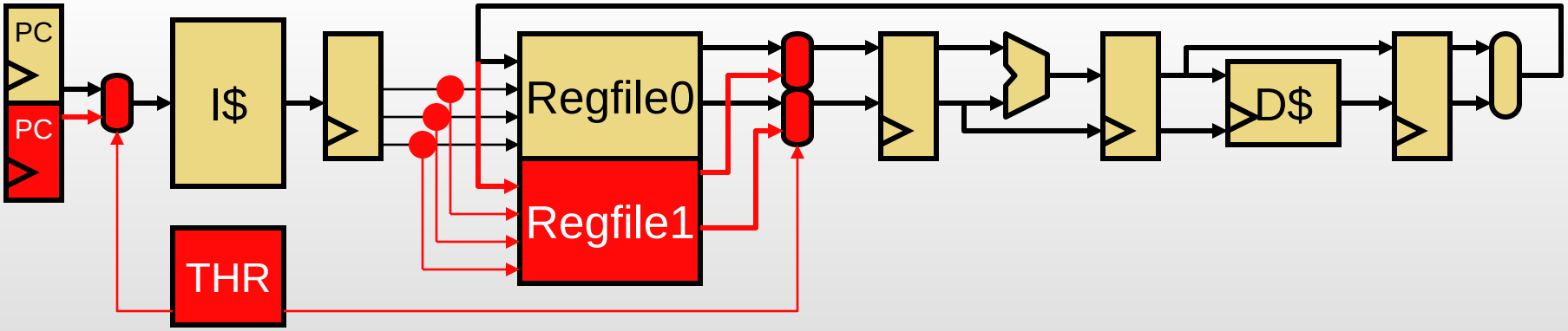
- Software "thread"
  - Independent flow of execution
  - Context state: PC, registers
  - Threads generally share the same memory space
  - "Process" like a thread, but different memory space
  - Java has thread support built in, C/C++ supports P-threads library

- Generally, system software (the O.S.) manages threads
  - "Thread scheduling", "context switching"
  - All threads share the one processor
    - Hardware timer interrupt occasionally triggers O.S.
    - Quickly swapping threads gives illusion of concurrent execution
  - Refresh already known OS topics (¿?)

# Multithreaded Programming Model

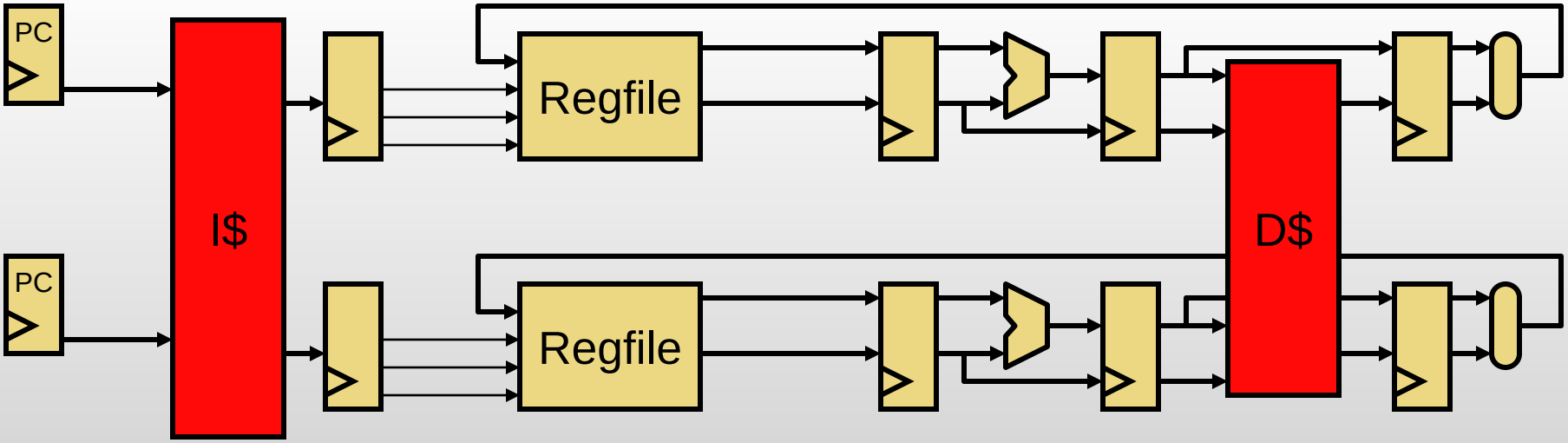
- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
  - Each thread has a private stack frame for local variables
- A "thread switch" can occur at any time
  - Pre-emptive multithreading by OS
- Common uses:
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send network message, wait for response)
  - Expressing parallel work via Thread-Level Parallelism (TLP)

# Aside: Hardware Multithreading




- **Hardware Multithreading (MT)**
  - Multiple threads dynamically share a single pipeline (caches)
  - Replicate thread contexts: PC and register file
  - **Coarse-grain MT**: switch on L2 misses **Why?**
  - **Simultaneous MT**: no explicit switching, fine-grain interleaving
    - Pentium4 is 2-way hyper-threaded, leverages out-of-order core
  + MT Improves utilization and throughput
    - Single programs utilize <50% of pipeline (branch, cache miss)
  - MT does not improve single-thread performance
    - Individual threads run as fast or even slower… especially in Pentium4

# Simplest Multiprocessor




- Replicate entire processor pipeline!
  - Instead of replicating just register file & PC
  - Exception: share caches (we'll address this bottleneck later)
- Same "shared memory" or "multithreaded" model
  - Loads and stores from two processors are interleaved
- Advantages/disadvantages over hardware multithreading?

# Shared Memory Implementations

- **Multiplexed uniprocessor**
  - Runtime system and/or OS occasionally pre-empt & swap threads
  - Interleaved, but no parallelism
- **Hardware multithreading**
  - Tolerate pipeline latencies, higher efficiency
  - Same interleaved shared-memory model
- **Multiprocessing**
  - Multiply execution resources, higher peak performance
  - Same interleaved shared-memory model
  - Foreshadowing: allow private caches, further disentangle cores
- **All have same shared memory programming model**

# Shared Memory Issues

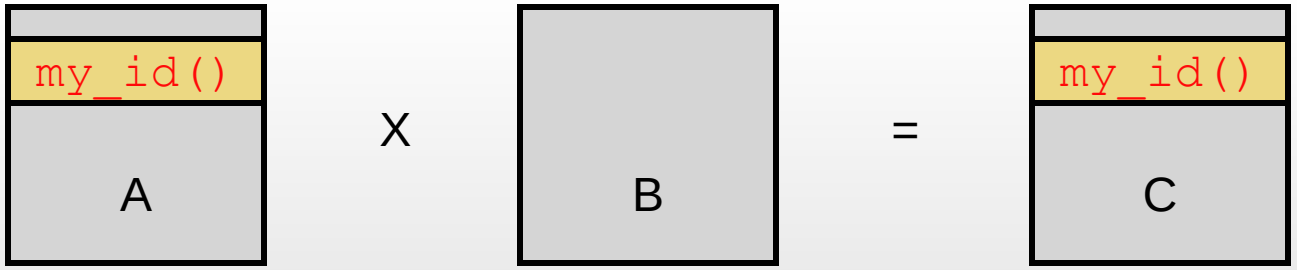
- Three in particular, not unrelated to each other

- Synchronization
  - How to regulate access to shared data?
  - How to implement critical sections?

- Cache coherence
  - How to make writes to one cache “show up” in others?

- Memory consistency model
  - How to keep programmer sane while letting hardware optimize?
  - How to reconcile shared memory with out-of-order execution?

# Example: Parallelizing Matrix Multiply




```
for (I = 0; I < 100; I++)
   for (J = 0; J < 100; J++)
      for (K = 0; K < 100; K++)
         C[I][J] += A[I][K] * B[K][J];
```

- How to parallelize matrix multiply over 100 processors?
- One possibility: give each processor 100 iterations of I

```
for (J = 0; J < 100; J++)
   for (K = 0; K < 100; K++)
      C[my_id()][J] += A[my_id()][K] * B[K][J];
```

- Each processor runs copy of loop above
  - `my_id()` function gives each processor ID from 0 to N
  - Parallel processing library provides this function

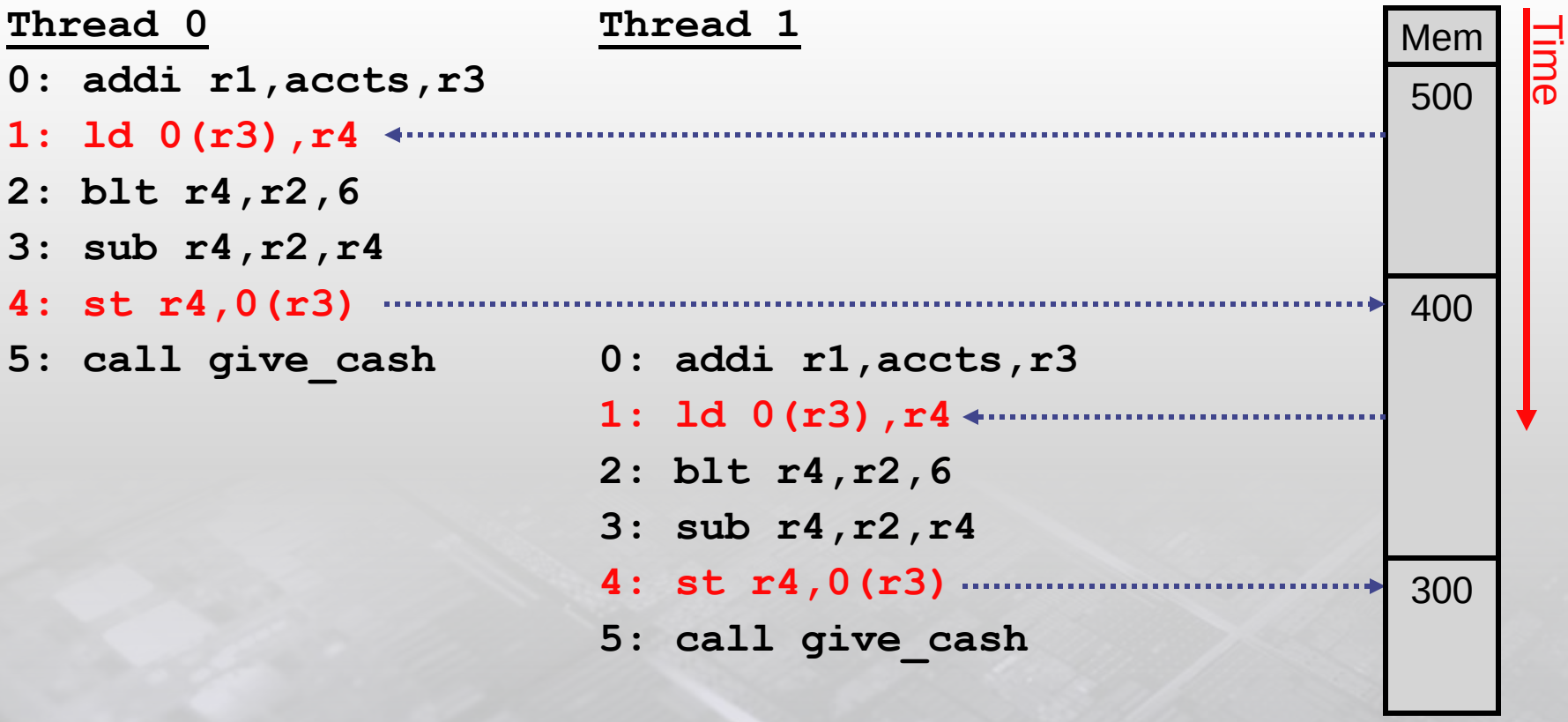
# Example: Thread-Level Parallelism

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id, amt;
if (accts[id].bal >= amt)
{
   accts[id].bal -= amt;
   give_cash();
}
```

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call give_cash
```

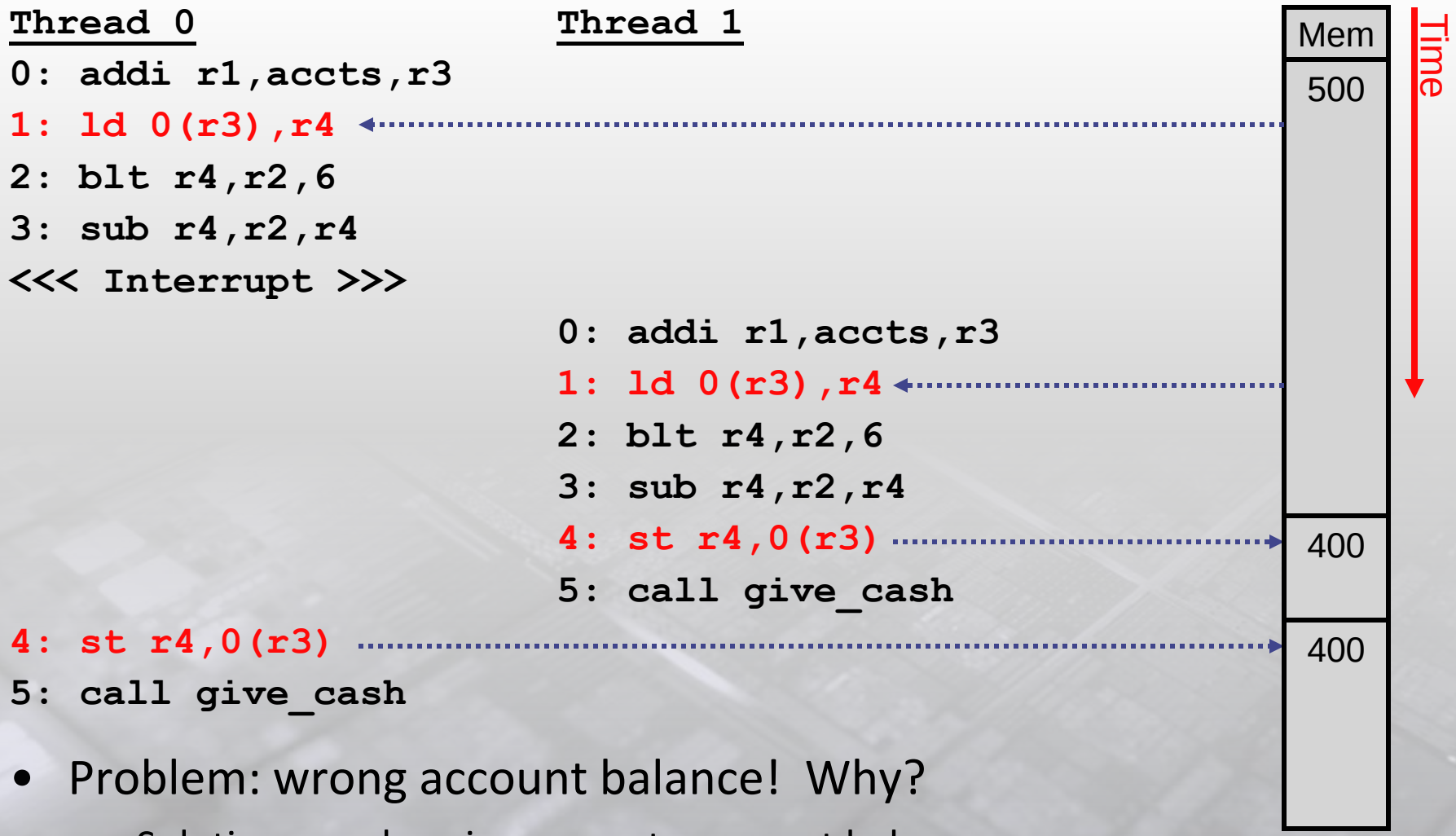
- **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared "loosely" (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
  - **accts** is **shared**, can't register allocate even if it were scalar
  - **id** and **amt** are private variables, register allocated to **r1**, **r2**
- Running example

# An Example Execution




- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)

# A Problem Execution




- Problem: wrong account balance! Why?
  - Solution: synchronize access to account balance

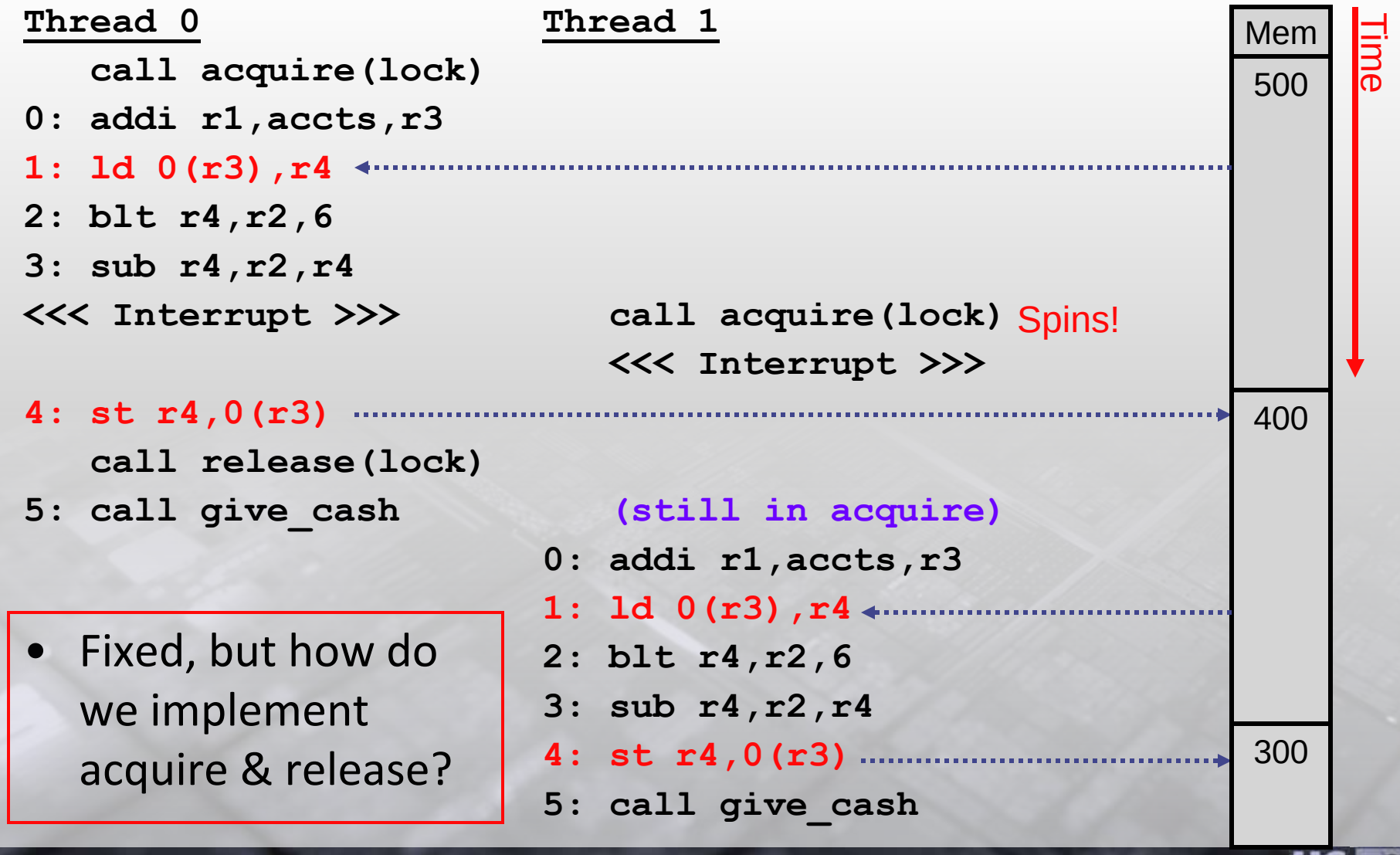
# Synchronization

- **Synchronization**: a key issue for shared memory
  - Regulate access to shared data (mutual exclusion)
  - Software constructs: semaphore, monitor, mutex
  - Low-level primitive: **lock**
    - Operations: **`acquire(lock)`** and **`release(lock)`**
    - Region between **`acquire`** and **`release`** is a **critical section**
    - Must interleave **`acquire`** and **`release`**
    - Interfering **`acquire`** will block

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
shared int lock;
int id, amt;
acquire(lock);
if (accts[id].bal >= amt) {
   accts[id].bal -= amt;
   give_cash(); }
release(lock);
```

// critical section

# A Synchronized Execution




- Fixed, but how do we implement acquire & release?

# Strawman Lock (Incorrect)

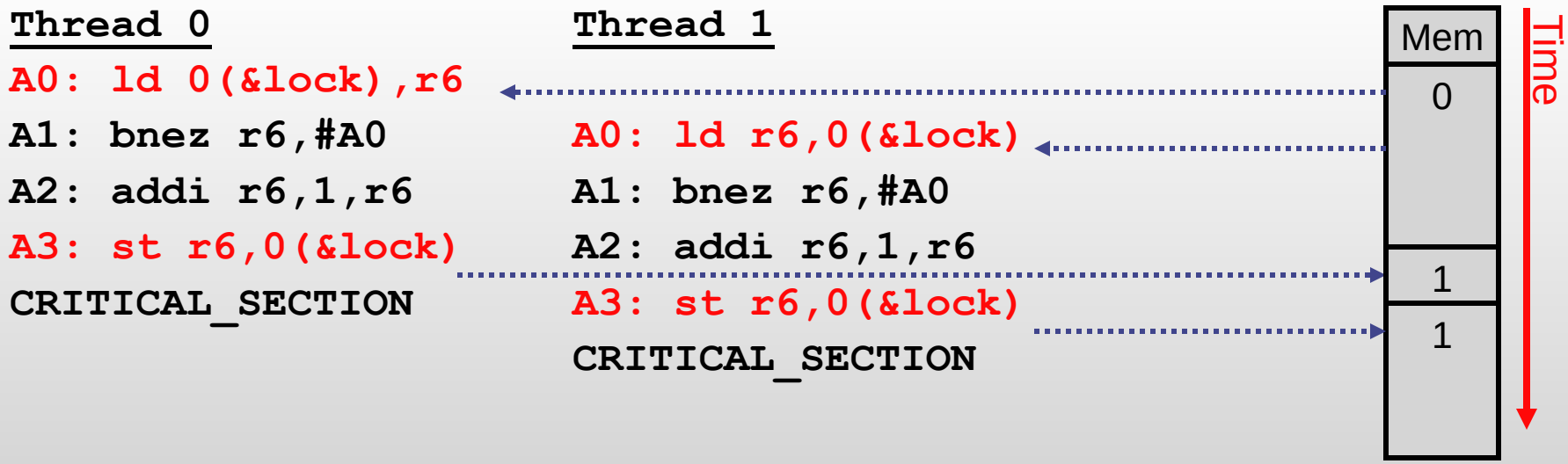
- **Spin lock**: software lock implementation
  - `acquire(lock): while (lock != 0); lock = 1;`
    - "Spin" while lock is 1, wait for it to turn 0

```
A0:  ld 0(&lock),r6
A1:  bnez r6,A0
A2:  addi r6,1,r6
A3:  st r6,0(&lock)
```

  - `release(lock): lock = 0;`

```
R0:  st r0,0(&lock)      // r0 holds 0
```

# Strawman Lock (Incorrect)




- Spin lock makes intuitive sense, but doesn't actually work
  - Loads/stores of two **acquire** sequences can be interleaved
  - Lock **acquire** sequence also not atomic
  - **Same problem as before!**

- Note, **release** is trivially atomic

# A Correct Implementation: SYSCALL Lock

```
ACQUIRE_LOCK:
A1: disable_interrupts          atomic
A2: ld r6,0(&lock)
A3: bnez r6,#A0
A4: addi r6,1,r6
A5: st r6,0(&lock)
A6: enable_interrupts
A7: return
```

- Implement lock in a SYSCALL
  - Only kernel can control interleaving by disabling interrupts
  - \+ Works…
  - – Large system call overhead
  - – **But not in a hardware multithreading or a multiprocessor…**

# Better Spin Lock: Use Atomic Swap

- ISA provides an atomic lock acquisition instruction
  - Example: **atomic swap**

    `swap r1,0(&lock)`

    - Atomically executes:

```
mov r1->r2
ld r1,0(&lock)
st r2,0(&lock)
```

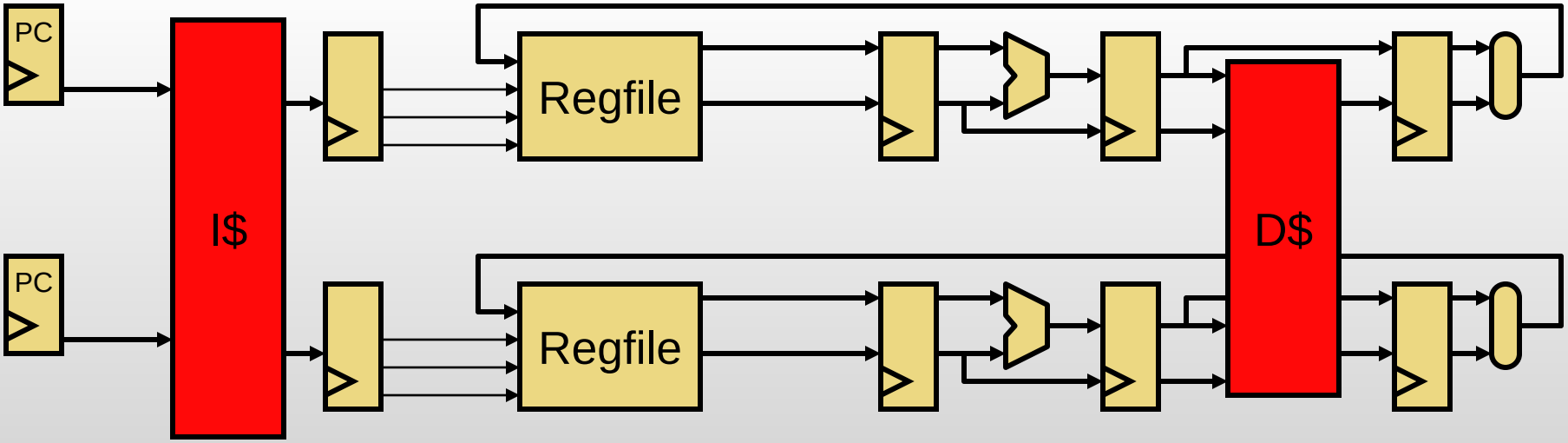
- New acquire sequence

```
    (value of r1 is 1)
A0: swap r1,0(&lock)
A1: bnez r1,A0
```

  - If lock was initially busy (1), doesn't change it, **keep looping**
  - If lock was initially free (0), acquires it (sets it to 1), break loop

- Insures lock held by **at most one thread**
  - Other variants: **exchange**, **compare-and-swap**, **test-and-set (t&s)**, or **fetch-and-add**

# Atomic Update/Swap Implementation




- How is atomic swap implemented?
  - Need to ensure no intervening memory operations
  - Requires blocking access by other threads temporarily (yuck)
- How to pipeline it?
  - Both a load and a store (yuck)
  - Not very RISC-like
  - Some ISAs provide a “load-link” and “store-conditional” insn. pair

# RISC Test-And-Set

- `t&s`: a load and store in one insn is not very "RISC"
  - Broken up into micro-ops, but then how are mops made atomic?

- `ll/sc`: load-locked / store-conditional
  - Atomic load/store pair

```
ll r1,0(&lock)
// potentially other insns
sc r2,0(&lock)
```

  - On `ll`, processor remembers address…
    - …And looks for writes by other processors
    - If write is detected, next `sc` to same address is annulled
      - Sets failure condition
      - Some kind of predication but in reverse direction with the condition (is output not input)

# SWAP Implementation (Example)

- Atomic swap ( between **r4** and **0(r1)**)

```
try:mov  r4,r3
    ll   r2, 0(r1) //toggle internal flags,
                   //address and value
    sc   r3,0(r1)  //if internal flags matches
                   //0(r1) is mod. And r3 is set
    beqz r3,try    //went wrong. Someone wrote
                   //0(r1) in the middle of ll&sc
    mov  r2,r4     //Store (until release) previous
                   // 0(r1) content
```

- **t&s** can be seen as a **swap (1, lock)**

# Lock Correctness

| Thread 0 | Thread 1 |
| --- | --- |
| A0: swap r1,0(&lock) | |
| A1: bnez r1,#A0 | A0: swap r1,0(&lock) |
| CRITICAL_SECTION | A1: bnez r1,#A0 |
| | A0: swap r1,0(&lock) |
| | A1: bnez r1,#A0 |

+ Test-and-set lock actually works…
  - Thread 1 keeps spinning

# Test-and-Set Lock Performance

```
Thread 0                    Thread 1
A0: t&s r1,0(&lock)
A1: bnez r1,#A0             A0: t&s r1,0(&lock)
A0: t&s r1,0(&lock)         A1: bnez r1,#A0
A1: bnez r1,#A0             A0: t&s r1,0(&lock)
                            A1: bnez r1,#A0
```

- …but performs poorly
  - Consider 3 processors rather than 2
  - Processor 2 (not shown) has the lock and is in the critical section
  - But what are processors 0 and 1 doing in the meantime?
    - Loops of `t&s`, each of which includes a `st`
    - Repeated stores by multiple processors costly (more in a bit)
    - Generating a ton of useless interconnect traffic
    - (But … `st` wasn't annulled?)

# Test-and-Test-and-Set Locks

- Solution: **test-and-test-and-set locks**
  - New acquire sequence

```
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1,r1
A3: t&s r1,0(&lock)
A4: bnez r1,A0
```

  - Within each loop iteration, before doing a **t&s**
    - Spin doing a simple test (**ld**) to see if lock value has changed
    - Only do a **t&s** (**st**) if lock is actually free
  - Processors can spin on a busy lock locally (in their own cache)
    - \+ Less unnecessary interconnect traffic
  - Note: test-and-test-and-set is not a new instruction!
    - Just different software

# Queue Locks

- Test-and-test-and-set locks can still perform poorly
  - If lock is contended for by many processors
  - Lock release by one processor, creates "free-for-all" by others
  - – Interconnect gets swamped with `t&s` requests
- **Software queue lock**
  - Each waiting processor spins on a different location (a queue)
  - When lock is released by one processor...
    - Only the next processors sees its location go "unlocked"
    - Others continue spinning locally, unaware lock was released
  - Effectively, passes lock from one processor to the next, in order
  - \+ Greatly reduced network traffic (no mad rush for the lock)
  - \+ Fairness (lock acquired in FIFO order)
  - – Higher overhead in case of no contention (more instructions)
  - – Poor performance if one thread gets swapped out

# Programming With Locks Is Tricky

- Multicore processors are the way of the foreseeable future
  - thread-level parallelism anointed as parallelism model of choice
  - Just one problem…
- Writing lock-based multi-threaded programs is tricky!
- More precisely:
  - Writing programs that are correct is "easy" (not really)
  - Writing programs that are highly parallel is "easy" (not really)
  - **Writing programs that are both correct and parallel is difficult**
    - And that's the whole point, unfortunately
  - Why?
    - **Locking**

# Coarse-Grain Locks: Correct but Slow

- **Coarse-grain locks**: e.g., one lock for entire database
  - \+ Easy to make correct: no chance for unintended interference
  - – Limits parallelism: no two critical sections can proceed in parallel

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
shared int lock;


acquire(lock);
if (accts[id].bal >= amt) {
   accts[id].bal -= amt;
   give_cash(); }
release(lock);
```

# Fine-Grain Locks: Parallel But Difficult

- **Fine-grain locks**: e.g., multiple locks, one per record
  - \+ Fast: critical sections (to different records) can proceed in parallel
  - – Difficult to make correct: easy to make mistakes
    - This particular example is easy
      - Requires only one lock per critical section
    - Consider critical section that requires two locks…

```
struct acct_t { int bal,lock; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;


acquire(accts[id].lock);
if (accts[id].bal >= amt) {
   accts[id].bal -= amt;
   give_cash(); }
release(accts[id].lock);
```

# Multiple Locks

- **Multiple locks**: e.g., acct-to-acct transfer
  - Must acquire both **id_from**, **id_to** locks
  - Running example with accts 241 and 37
  - Simultaneous transfers 241 → 37 and 37 → 241
  - Contrived… but even contrived examples must work correctly too

```
struct acct_t { int bal,lock; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

acquire(accts[id_from].lock);
acquire(accts[id_to].lock);
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
release(accts[id_to].lock);
release(accts[id_from].lock);
```

# Multiple Locks And Deadlock

**Thread 0**

```
id_from = 241;
id_to = 37;

acquire(accts[241].lock);
// wait to acquire lock 37
// waiting…
// still waiting…
```

**Thread 1**

```
id_from = 37;
id_to = 241;

acquire(accts[37].lock);
// wait to acquire lock 241
// waiting…
// …
```

- **Deadlock**: circular wait for shared resources
  - Thread 0 has lock 241 waits for lock 37
  - Thread 1 has lock 37 waits for lock 241
  - Obviously this is a problem
  - The solution is …

# Correct Multiple Lock Program

- **Always acquire multiple locks in same order**
  - Just another thing to keep in mind when programming

```
struct acct_t { int bal,lock; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;
int id_first = min(id_from, id_to);
int id_second = max(id_from, id_to);

acquire(accts[id_first].lock);
acquire(accts[id_second].lock);
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
release(accts[id_second].lock);
release(accts[id_first].lock);
```

# Correct Multiple Lock Execution

**Thread 0**

```
id_from = 241;
id_to = 37;
id_first = min(241,37)=37;
id_second = max(37,241)=241;

acquire(accts[37].lock);
acquire(accts[241].lock);
// do stuff
release(accts[241].lock);
release(accts[37].lock);
```

**Thread 1**

```
id_from = 37;
id_to = 241;
id_first = min(37,241)=37;
id_second = max(37,241)=241;

// wait to acquire lock 37
// waiting…
// …
// …
// …
acquire(accts[37].lock);
```

- Great, are we done? No

# More Lock Madness

- What if…
  - Some actions (e.g., deposits, transfers) require 1 or 2 locks…
  - …and others (e.g., prepare statements) require all of them?
  - Can these proceed in parallel?
- What if…
  - There are locks for global variables (e.g., operation id counter)?
  - When should operations grab this lock?
- What if… what if… what if…

- **So lock-based programming is difficult…**
- **…wait, it gets worse**

# And To Make It Worse…

- **Acquiring locks is expensive…**
  - By definition requires a slow atomic instructions
    - Specifically, acquiring write permissions to the lock
  - Ordering constraints (see soon) make it even slower

- **…and 99% of the time un-necessary**
  - Most concurrent actions don't actually share data
  - You paying to acquire the lock(s) for no reason

- Fixing these problem is an area of active research
  - One proposed solution "Transactional Memory"

# Research: Transactional Memory (TM)

- **Transactional Memory**
  - + Programming simplicity of coarse-grain locks
  - + Higher concurrency (parallelism) of fine-grain locks
    - Critical sections only serialized if data is actually shared
  - + No lock acquisition overhead
  - Hottest thing since sliced bread (or was a few years ago)

- Sun Rock processor has hardware support for it
  - …so, starting to be less **Research** and more **Practice**

# Transactional Memory: The Big Idea

- Big idea I: **no locks, just shared data**
  - Look ma, no locks
- Big idea II: **optimistic (speculative) concurrency**
  - Execute critical section speculatively, abort on conflicts
  - "Better to beg for forgiveness than to ask for permission"

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

# Transactional Memory: Read/Write Sets

- **Read set**: set of shared addresses critical section reads
  - Example: `accts[37].bal`, `accts[241].bal`
- **Write set**: set of shared addresses critical section writes
  - Example: `accts[37].bal`, `accts[241].bal`

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

# Transactional Memory: Begin

- **begin_transaction**
  - Take a local register checkpoint
  - Begin locally tracking read set (remember addresses you read)
    - See if anyone else is trying to write it
  - Locally buffer all of your writes (invisible to other processors)
  - \+ **Local actions only: no lock acquire**

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

# Transactional Memory: End

- `end_transaction`
  - Check read set: is all data you read still valid (i.e., no writes to any)
  - Yes? Commit transactions: commit writes
  - No? Abort transaction: restore checkpoint (lazy conflict detection)
- Lazy / Eager approaches to detect conflicts.

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

# Transactional Memory Implementation

- How are read-set/write-set implemented?
  - Track locations accessed using bits in the cache
- Read-set: additional "transactional read" bit per block
  - Set on reads between begin_transaction and end_transaction
  - Any other write to block with set bit ➔ triggers abort (eager conflict detection) / abort at commit (lazy conflict detection)
  - Flash cleared on transaction abort or commit
- Write-set: additional "transactional write" bit per block
  - Set on writes between begin_transaction and end_transaction
  - Before first write, if dirty, initiate writeback ("clean" the block)
  - Flash cleared on transaction commit
  - On transaction abort: blocks with set bit are invalidated
  - *Aside: Where to write new values? (Version management): Eager (on-place) Lazy (somewhere else)*

# Transactional Execution

**Thread 0**

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
    …
    // write accts[241].bal
    // abort (eager)
```

**Thread 1**

```
id_from = 37;
id_to = 241;

begin_transaction();
if(accts[37].bal > 100) {
    accts[37].bal -= amt;
    acts[241].bal += amt;
}
end_transaction();
// no writes to accts[241].bal
// no writes to accts[37].bal
// commit
```

# Transactional Execution II (More Likely)

**Thread 0**

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
   accts[241].bal -= amt;
   acts[37].bal += amt;
}
end_transaction();
// no write to accts[240].bal
// no write to accts[37].bal
// commit
```

**Thread 1**

```
id_from = 450;
id_to = 118;

begin_transaction();
if(accts[450].bal > 100) {
   accts[450].bal -= amt;
   acts[118].bal += amt;
}
end_transaction();
// no write to accts[450].bal
// no write to accts[118].bal
// commit
```

- Critical sections execute in parallel

# Transaction Semantics - ACI Properties

- Atomicity – All or Nothing
- Consistency – Correct at beginning and end (no matter if the transaction is successful or not)
- Isolation – Partially done work not visible to other threads
- D? – Very different context: this is not a DBMS, although uses loosely related concepts, not the same problem!

# So, Let's Just Do Transactions?

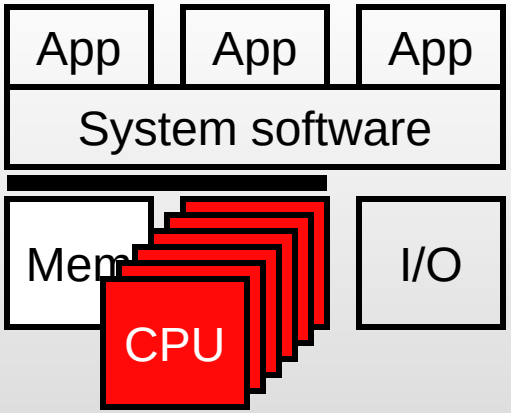
- What if…
  - Read-set or write-set bigger than cache?
  - Transaction gets swapped out in the middle?
  - Transaction wants to do I/O or SYSCALL (not-abortable)?
  - Transaction nesting?
- How do we transactify existing lock based programs?
  - Replace **`acquire`** with **`begin_trans`** does not always work
  - Weak automaticity ⇎ strong atomicity
- Several different kinds of transaction semantics
  - Are transactions atomic relative to code outside of transactions?
- Do we want transactions in hardware or in software?
  - What we just saw is **hardware transactional memory (HTM)**
- That's what these research groups are looking at

# In The Meantime: Do SLE

```
Processor 0

acquire(accts[37].lock); // don't actually set lock to 1
// begin tracking read/write sets
// CRITICAL_SECTION
// check read set
//   no conflicts? Commit, don't actually set lock to 0
//   conflicts? Abort, retry by acquiring lock
release(accts[37].lock);
```

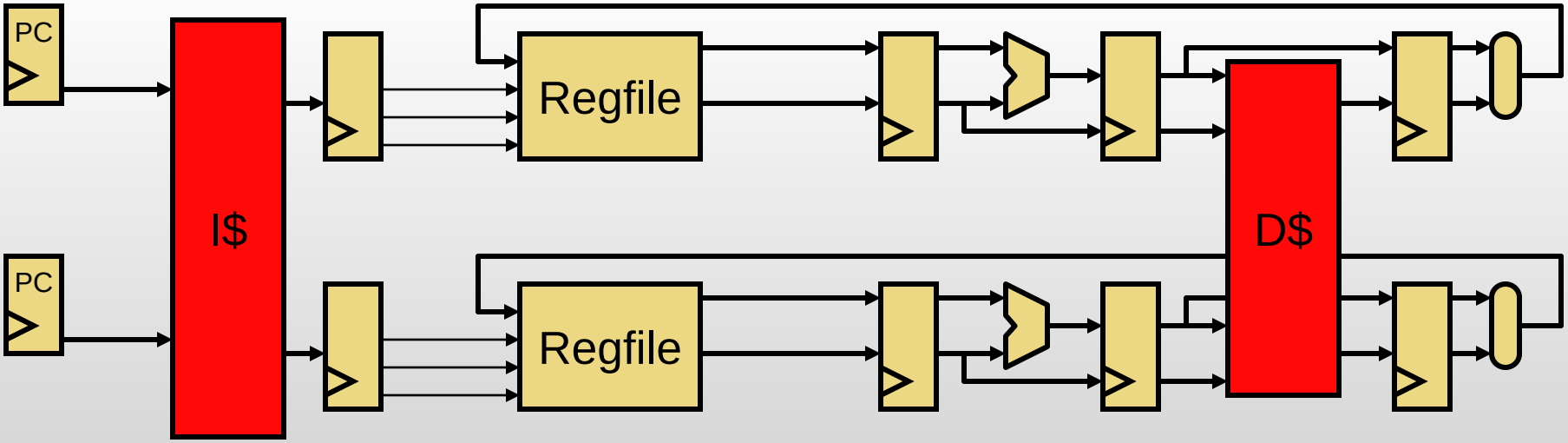
- Until TM interface solidifies…
- … speculatively transactify lock-based programs in hardware
  - **Speculative Lock Elision (SLE)** [Rajwar+, MICRO'01]
  - \+ No need to rewrite programs
  - \+ Can always fall back on lock-based execution (overflow, I/O, etc.)
  - Modified rumor: this is what Sun's Rock actually does

# Unit Checkpoint




- Thread-level parallelism (TLP)
  - Shared memory model
    - Multiplexed uniprocessor
    - Hardware multihreading
      - Multiprocessing
  - Synchronization
    - Lock implementation
      - Locking gotchas
  - Cache coherence
    - Bus-based protocols
    - Directory protocols
- Memory consistency models

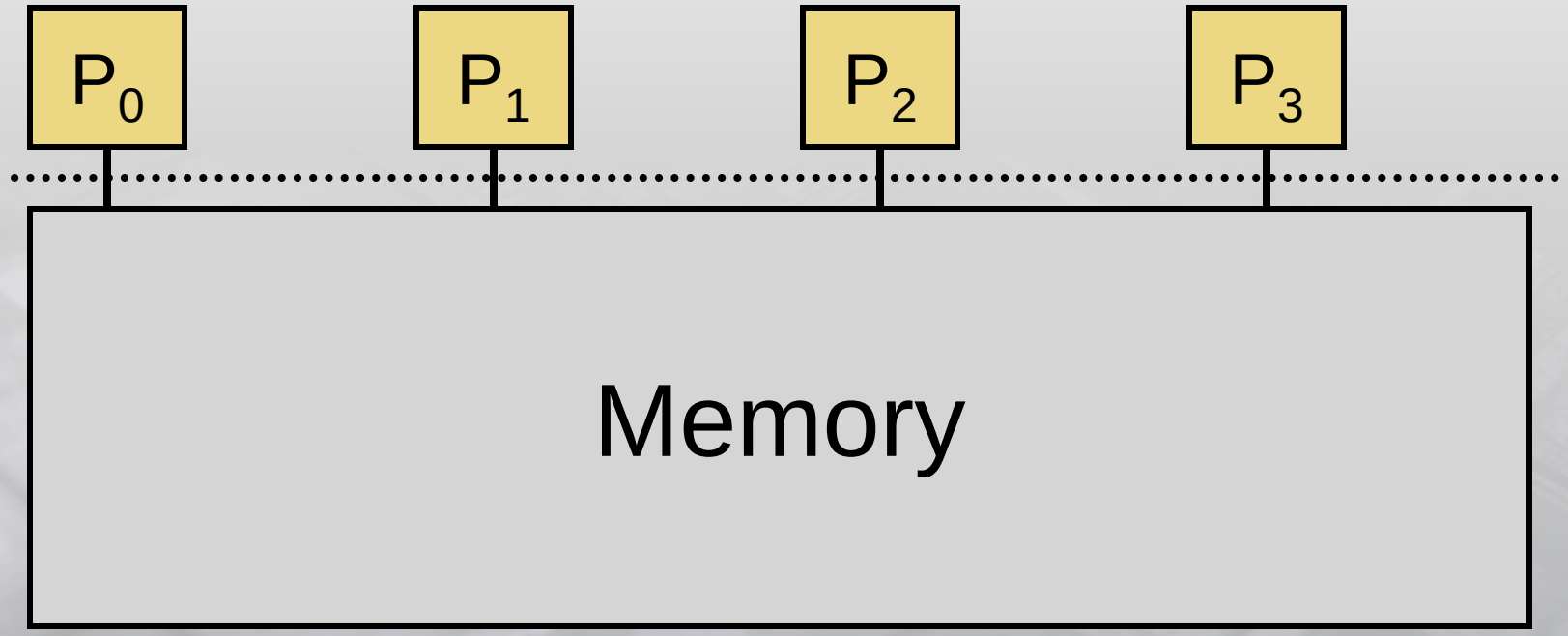
# Recall: Simplest Multiprocessor




- What if we don't want to share the L1 caches?
  - Bandwidth and latency issue
- Solution: use per-processor ("private") caches
  - Coordinate them with a *Cache Coherence Protocol*

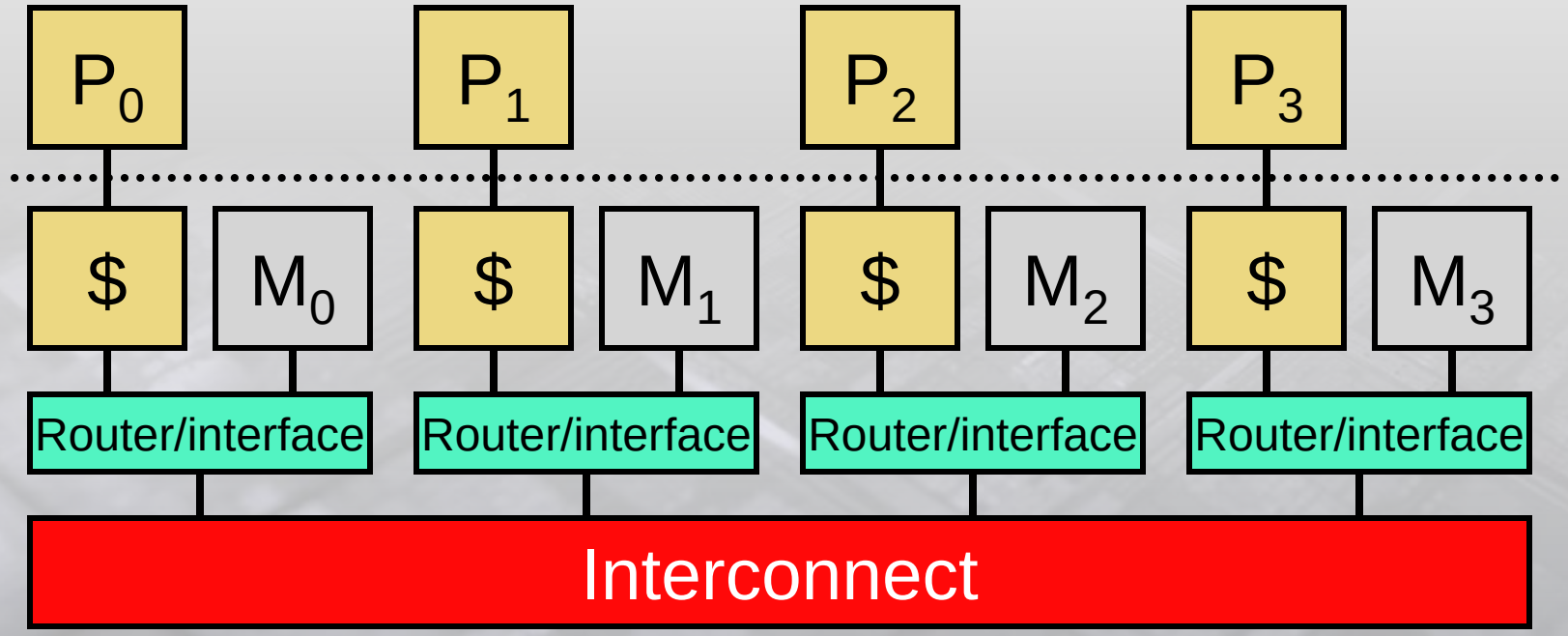
# Shared-Memory Multiprocessors

- **Conceptual model**
  - The shared-memory abstraction
  - Familiar and feels natural to programmers
  - Life would be easy if systems actually looked like this…



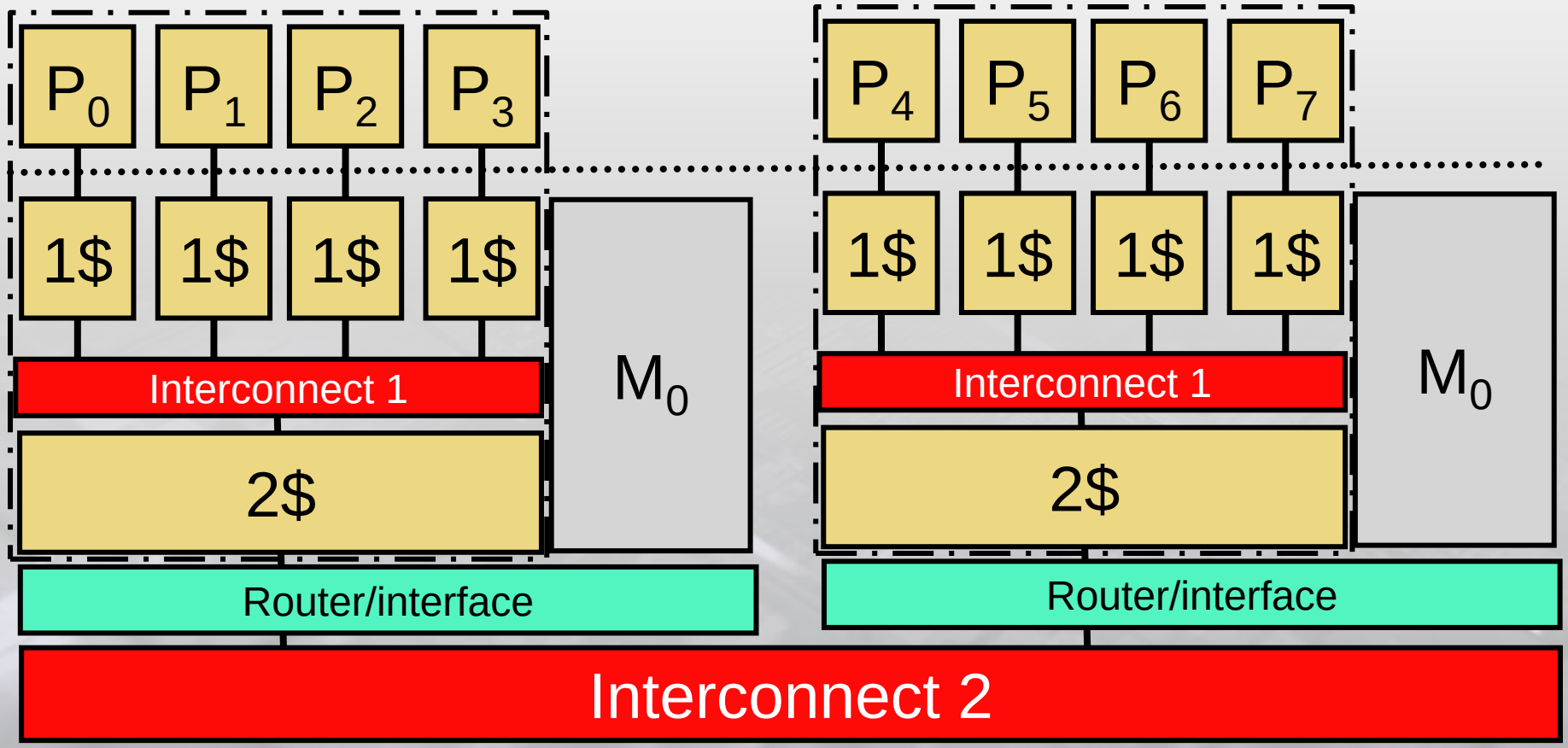
# Shared-Memory Multiprocessors

- …but systems actually look more like this
  - Processors have caches
  - Memory may be physically distributed
  - Arbitrary interconnect



# Chip Multiprocessors (CMP)

- Add another level to the hierarchy



# Revisiting Our Motivating Example

CPU0 CPU1 Mem

**Processor 0**

```
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash
```

critical section (locks not shown)

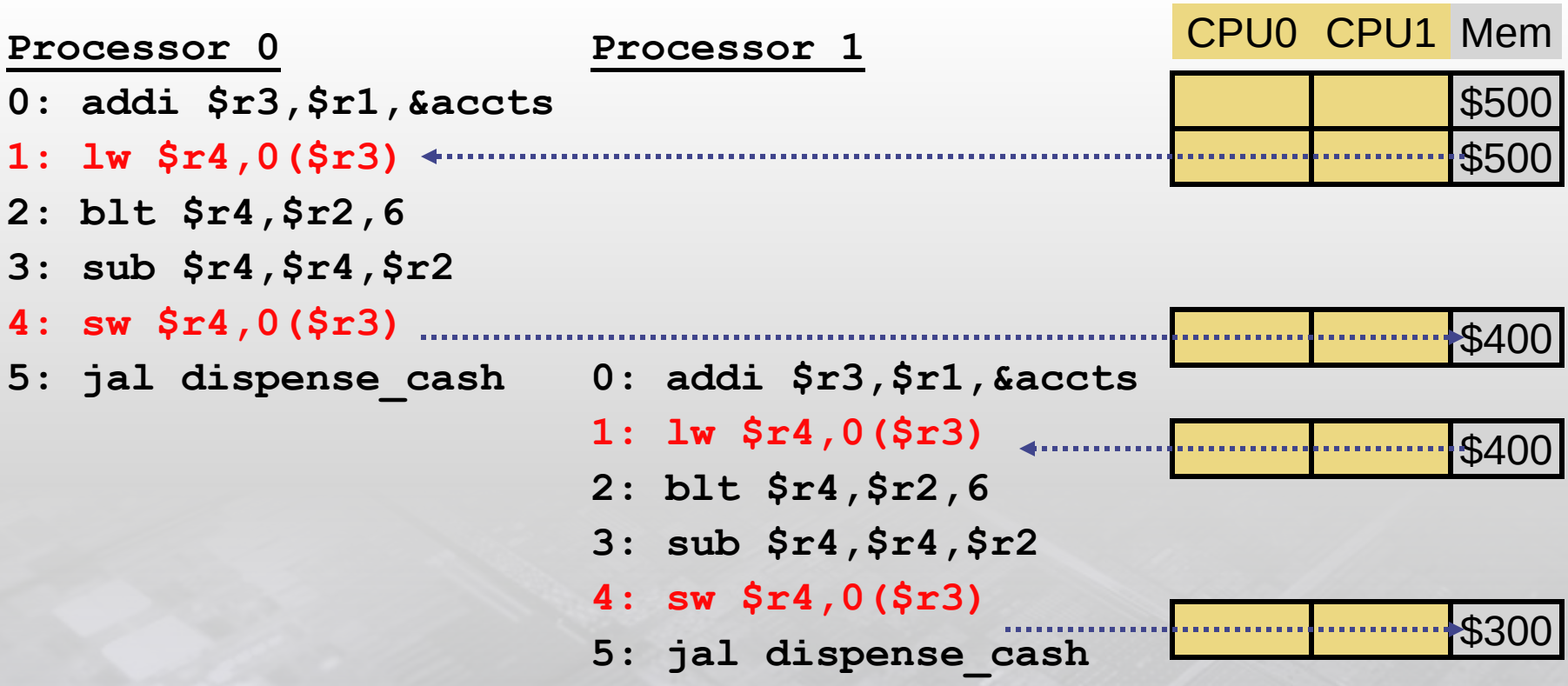
**Processor 1**

```
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash
```

critical section (locks not shown)

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `$r3`)

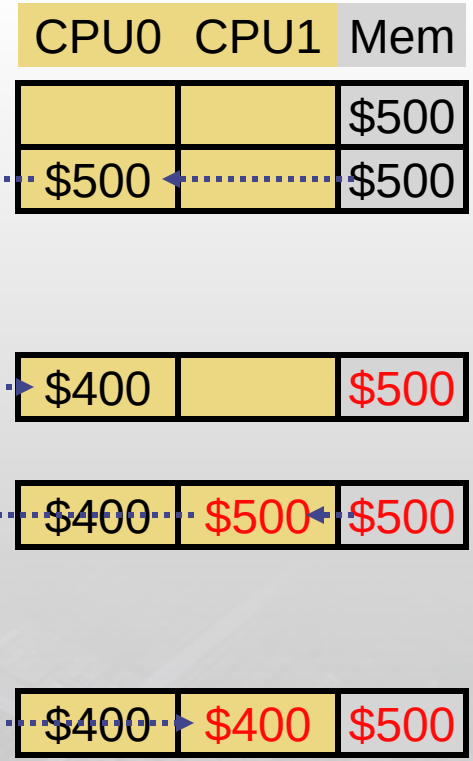
# No-Cache, No-Problem




- Scenario I: processors have no caches
  - No problem

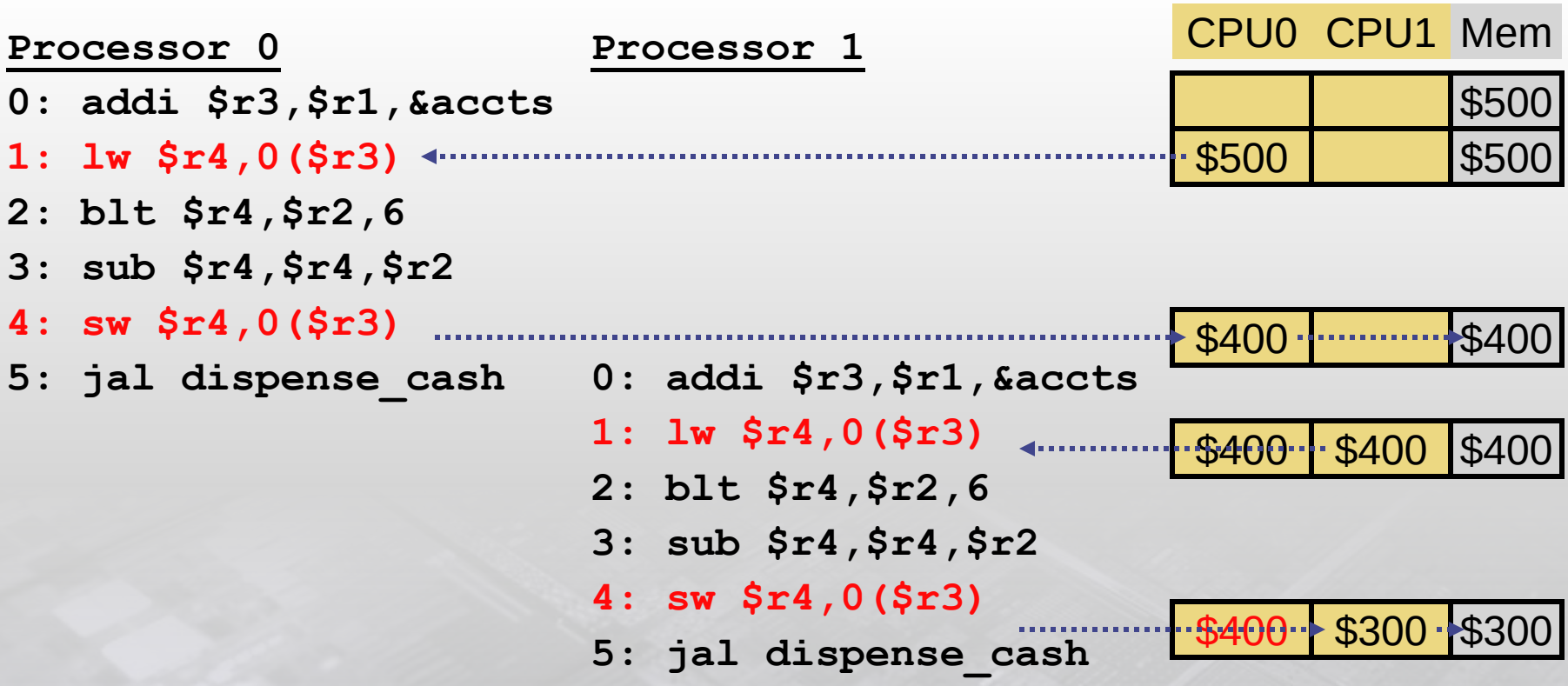
# Cache Incoherence

```
Processor 0                    Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash           0: addi $r3,$r1,&accts
                               1: lw $r4,0($r3)
                               2: blt $r4,$r2,6
                               3: sub $r4,$r4,$r2
                               4: sw $r4,0($r3)
                               5: jal dispense_cash
```




- Scenario II(a): processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (inconsistent)

# Write-Through Doesn't Fix It




- Scenario II(b): processors have write-through caches
  - This time only 2 (different) copies of `accts[241].bal`
  - No problem? What if another withdrawal happens on processor 0?

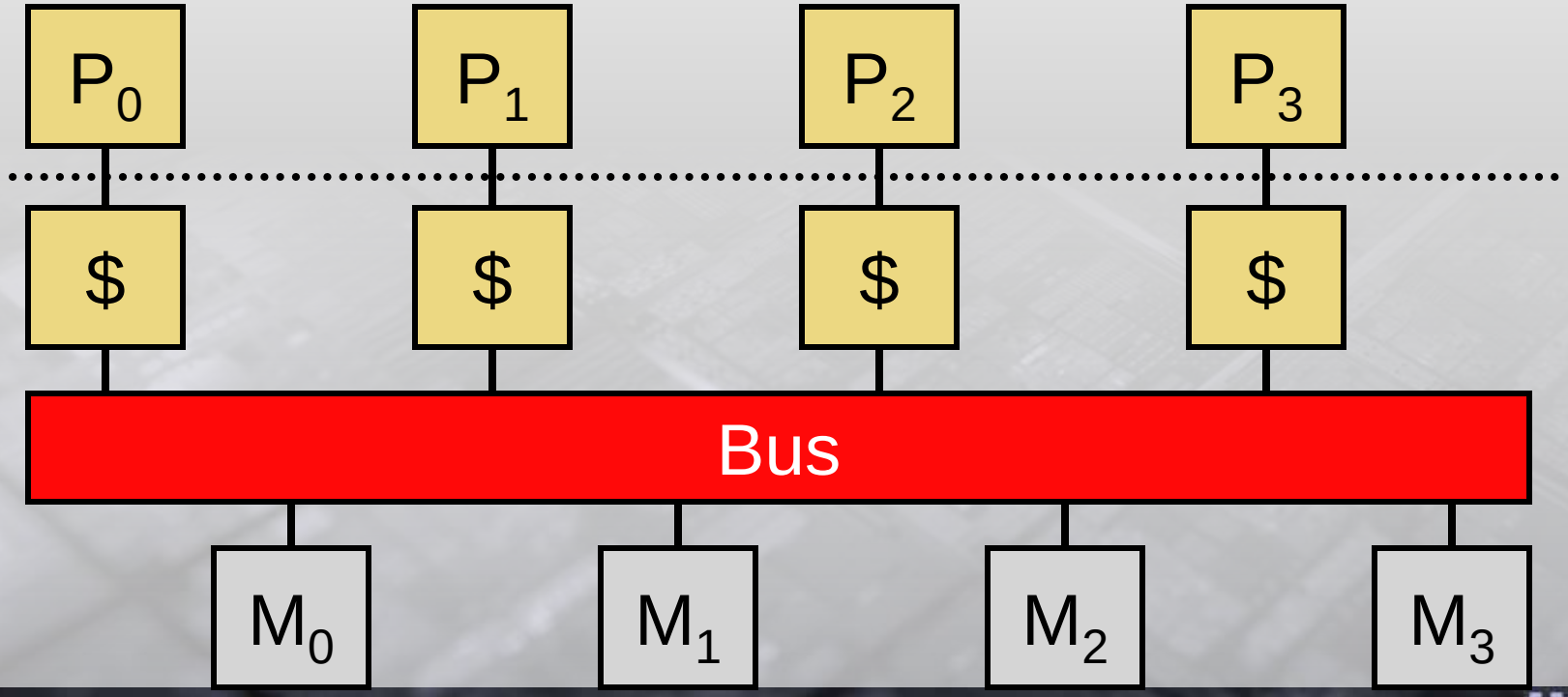
# What To Do?

- No caches?
  - Slow
- Make shared data uncachable?
  - Faster, but still too slow
  - Entire **`accts`** database is technically "shared"
    - Definition of "loosely shared"
    - Data only really shared if two ATMs access same acct at once
- Flush all other caches on writes to shared data?
  - May as well not have caches

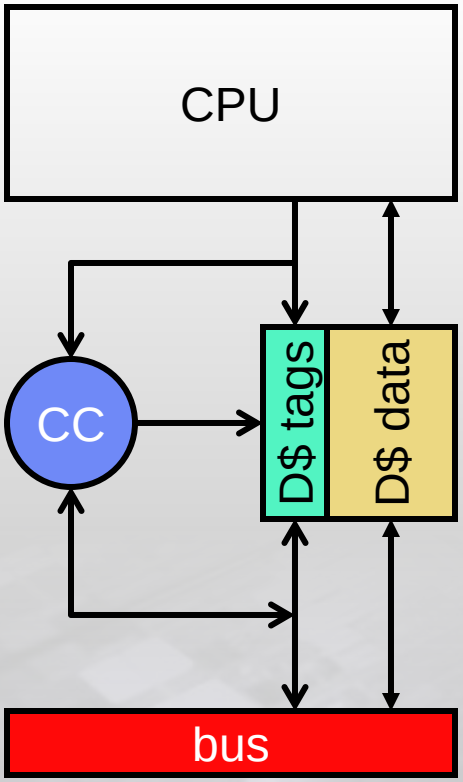
- **Hardware cache coherence**
  - Rough goal: all caches have same data at all times
  + Minimal flushing, maximum caching → best performance

# Bus-based Multiprocessor

- Simple multiprocessors use a bus
  - **All** processors see all requests at the **same time**, same order
- Memory
  - Single memory module, **-or-**
  - Banked memory module



# 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
  - **R**: read **W**: write **WB**: write-back
- Two remote-initiated events
  - **BR**: bus-read, read miss from ***another*** processor
  - **BW**: bus-write, write miss from ***another*** processor
- Three responses event:
  - **SD:** send data **BR:** ask to read **BW:** ask to write

# 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 valid bit
- Protocol diagram (left)
  - Convention: event⇒generated-event
  - Summary
    - If anyone 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

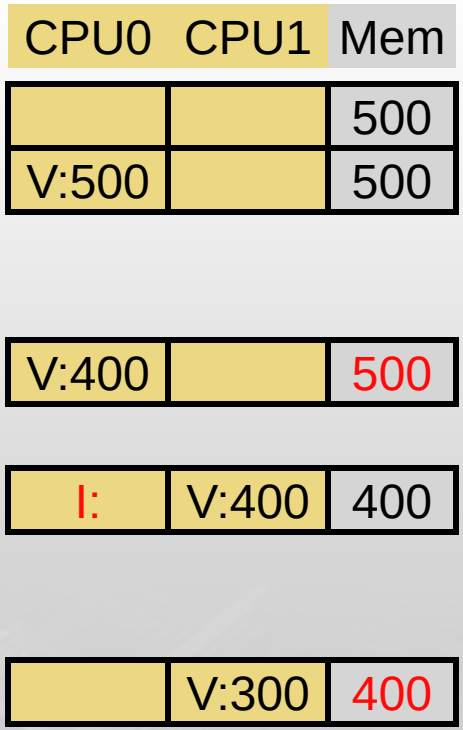
# VI Protocol State Transition Table

| State | *This Processor* | | *Other Processor* | |
|---|---|---|---|---|
| | Load | Store | Load Miss | Store Miss |
| Invalid (I) | Miss(BR) → V | Miss (BW) → V | --- | --- |
| Valid (V) | Hit | Hit | Send Data → I | Send Data → I |

- Rows are "states"
  - I vs V
- Columns are "events"
  - Writeback events not shown
- Memory controller not shown
  - Responds when no other processor would respond
  - Memory controller has his own table (to know when has to respond)
- If multiple Cache levels present => each one has his own CC (protocol and state per block)

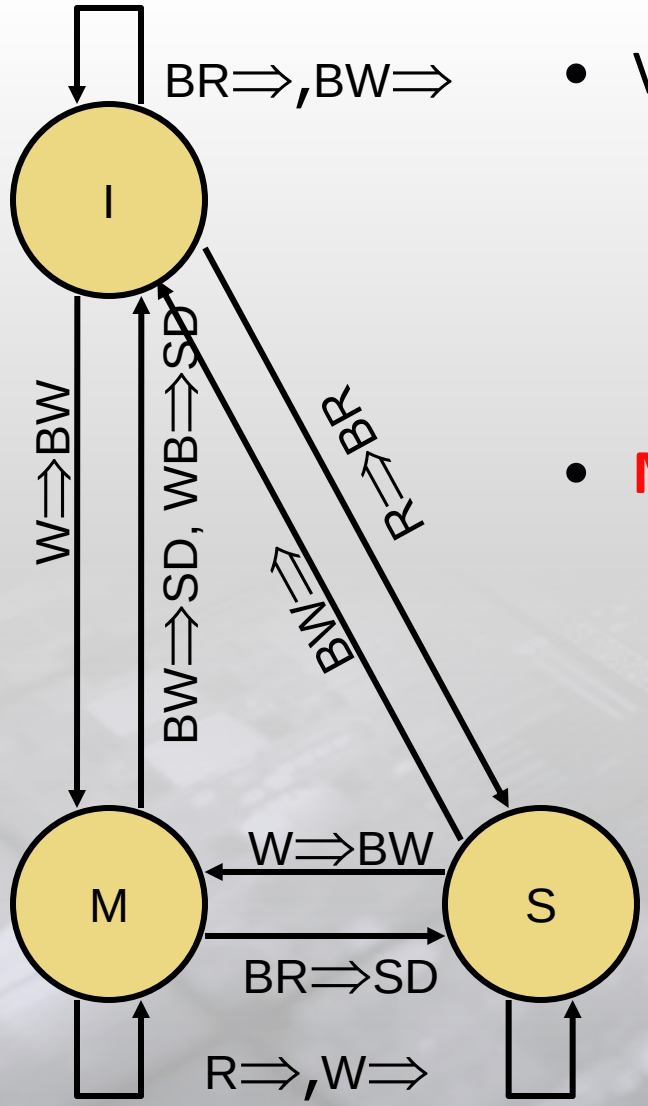
# VI Protocol (Write-Back Cache)

```
Processor 0                    Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash           0: addi $r3,$r1,&accts
                               1: lw $r4,0($r3)
                               2: blt $r4,$r2,6
                               3: sub $r4,$r4,$r2
                               4: sw $r4,0($r3)
                               5: jal dispense_cash
```




- `lw` by processor 1 generates a BR (bus read)
  - processor 0 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 State Transition Table

| State | *This Processor* | | *Other Processor* | |
|---|---|---|---|---|
| | 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 also updates memory
  - After which, memory willl respond (as all processors will be in S)

# MSI Protocol (Write-Back Cache)

```
Processor 0                    Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash           0: addi $r3,$r1,&accts
                               1: lw $r4,0($r3)
                               2: blt $r4,$r2,6
                               3: sub $r4,$r4,$r2
                               4: sw $r4,0($r3)
                               5: jal dispense_cash
```

| CPU0 | CPU1 | Mem |
|---|---|---|
| | | 500 |
| S:500 | | 500 |
| M:400 | | 500 |
| S:400 | S:400 | 400 |
| I: | M:300 | 400 |

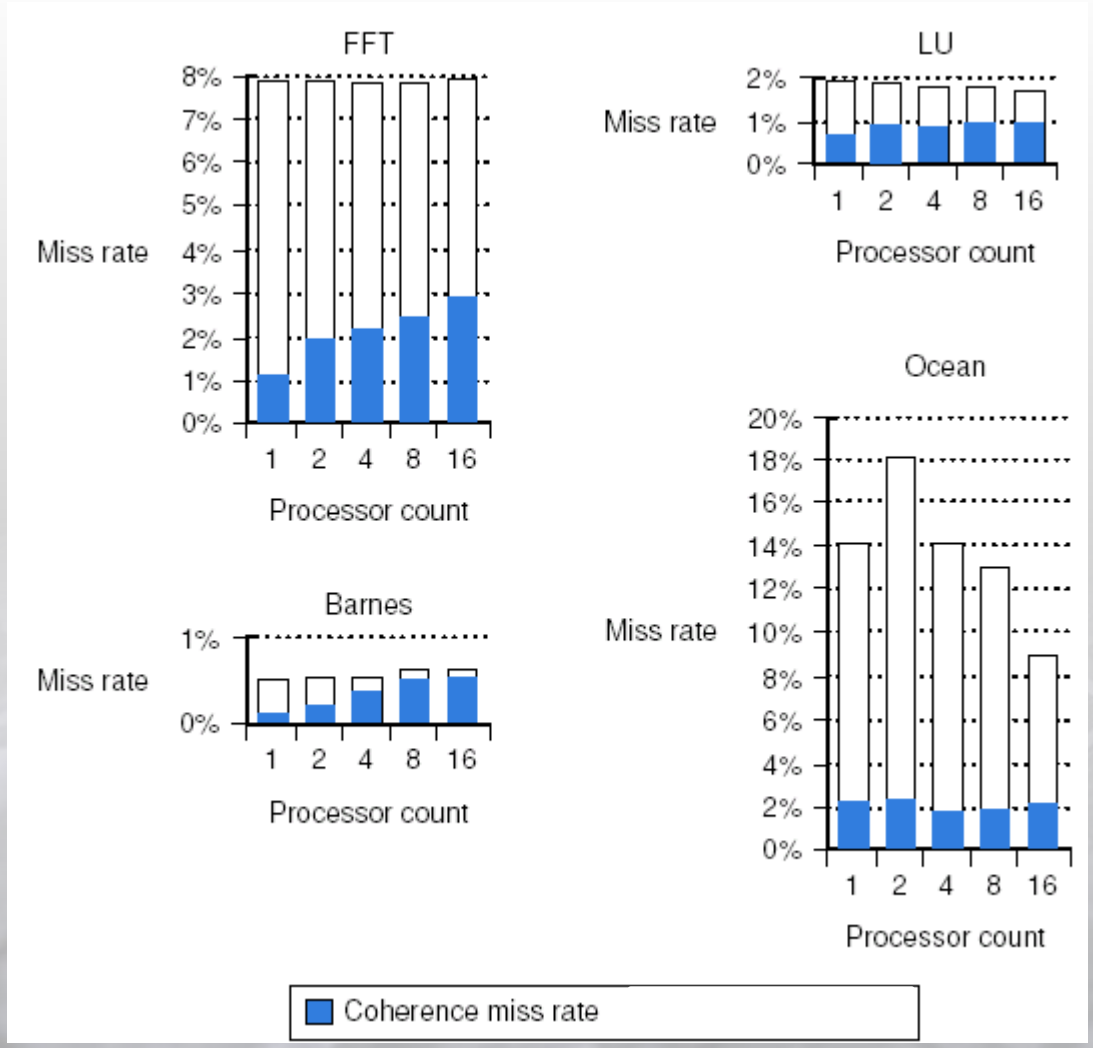
- `lw` by processor 1 generates a BR
  - Processor 0 responds by sending its dirty copy, transitioning to **S**
- `sw` by processor 1 generates a BW
  - Processor 0 responds by transitioning to **I**

# Cache Coherence and Cache Misses

- Coherence introduces two new kinds of cache misses
  - **Upgrade miss**: delay to acquire write permission to read-only block
  - **Coherence miss**: miss to a block evicted by bus event
  - Example: direct-mapped 4B cache, 2B blocks

| Cache contents (prior to access) | | Request | Outcome |
|---|---|---|---|
| **TT0B** | **TT1B** | | |
| ----\|----:I | ----\|----:I | **1100 R** | Compulsory miss |
| **1100**\|1101:**S** | ----\|----:I | **1100 W** | **Upgrade miss** |
| 1100\|1101:**M** | ----\|----:I | **0010 BW** | - (no action) |
| 1100\|1101:**M** | ----\|----:I | **1101 BW** | - (evict) |
| ----\|----:**I** | ----\|----:I | **1100 R** | **Coherence miss** |
| **1100**\|1101:**S** | ----\|----:I | **0000 R** | Compulsory miss |
| **0000**\|0001:**S** | ----\|----:I | **1100 W** | Conflict miss |

# 4th C


FFT
Miss rate
8%
7%
6%
5%
4%
3%
2%
1%
0%
1 2 4 8 16
Processor count
LU
Miss rate
2%
1%
0%
1 2 4 8 16
Processor count
Barnes
Miss rate
1%
0%
1 2 4 8 16
Processor count
Ocean
Miss rate
20%
18%
16%
14%
12%
10%
8%
6%
4%
2%
0%
1 2 4 8 16
Processor count
Coherence miss rate

# Cache Parameters and Coherence Misses

- Larger capacity: more coherence misses
  - But the effect offset (by far) by reduction in capacity misses
- Increased block size: more coherence misses
  - **False sharing**: "sharing" a cache line without sharing data
  - Creates pathological "ping-pong" behavior
  - Careful data placement may help, but is difficult

| Cache contents (prior to access) | | Request | Outcome |
|---|---|---|---|
| **TT0B** | **TT1B** | | |
| ----\|----:I | ----\|----:I | **1100 R** | Compulsory miss |
| **1100**\|**1101**:**S** | ----\|----:I | **1100 W** | Upgrade miss |
| **1100\|1101:M** | ----\|----:I | **1101 BW** | - (evict) |
| **----\|----:I** | ----\|----:I | **1100 R** | **Coherence miss (false sharing)** |

- More processors: (usually) also more coherence misses

# *Exclusive Clean* Protocol Optimization

**Processor 0**

```
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash
```

**Processor 1**

```
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash
```

| CPU0 | CPU1 | Mem |
|---|---|---|
| | | 500 |
| E:500 | | 500 |

(No miss)

| | | |
|---|---|---|
| M:400 | | 500 |
| S:400 | S:400 | 400 |
| I: | M:300 | 400 |

- Most modern protocols also include **E (exclusive)** state
  - Interpretation: "I have the only cached copy, and it's a **clean** copy"
  - Why would this state be useful?

# MESI Protocol State Transition Table

| State | *This Processor* Load | *This Processor* Store | *Other Processor* Load Miss | *Other Processor* Store Miss |
|---|---|---|---|---|
| Invalid (I) | Miss → **S or E** | Miss → M | --- | --- |
| Shared (S) | Hit | Upg Miss → M | --- | → I |
| Exclusive (E) | Hit | Hit → M | Send Data → S | Send Data → I |
| Modified (M) | Hit | Hit | Send Data → S | Send Data → I |

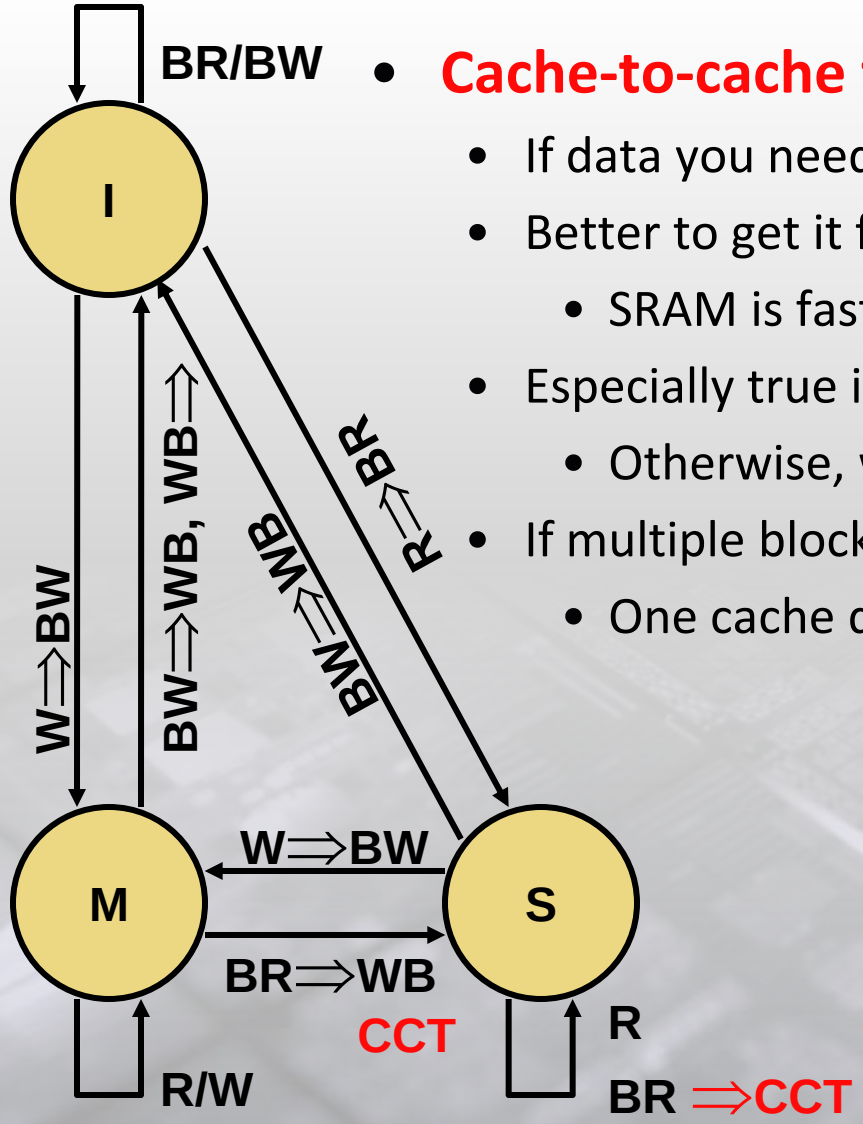
- Load misses lead to "E" if no other processors is caching the block

# MESI Protocol and Cache Misses

- MESI protocol reduces upgrade misses
  - And miss traffic.

| Cache contents (prior to access) | | Request | Outcome |
|---|---|---|---|
| **TT0B** | **TT1B** | | |
| ----\|----:I | ----\|----:I | **1100 R** | Compulsory miss (block from memory) |
| **1100**\|1101:**E** | ----\|----:I | **1100 W** | **- (no upgrade miss)** |
| 1100\|1101:**M** | ----\|----:I | **0010 BW** | - (no action) |
| 1100\|1101:**M** | ----\|----:I | **1101 BW** | - (evict) |
| ----\|----:**I** | ----\|----:I | **1100 R** | Coherence miss |
| **1100**\|1101:**E** | ----\|----:I | **0000 R** | Compulsory miss |
| **0000**\|0001:**S** | ----\|----:I | **1100 W** | Conflict miss **(no writeback)** |

# Another Protocol Optimization




- **Cache-to-cache transfers (CCT)**
  - If data you need is in both memory and other cache…
  - Better to get it from the other cache
    - SRAM is faster than DRAM
  - Especially true if cache block is dirty
    - Otherwise, writeback followed by memory read
  - If multiple blocks have copies, who does CCT?
    - One cache designated as “owner”

# MOESI Protocol State Transition Table

| State | *This Processor* | | *Other Processor* | |
|---|---|---|---|---|
| | Load | Store | Load Miss | Store Miss |
| Invalid (I) | Miss → S or E | Miss → M | --- | --- |
| Shared (S) | Hit | Upg Miss → M | --- | → I |
| Exclusive (E) | Hit | Hit → M | Send Data → O | Send Data → I |
| Owner (O) | Hit | Upg Miss → M | Send Data | Send Data → I |
| Modified (M) | Hit | Hit | Send Data → O | Send Data → I |

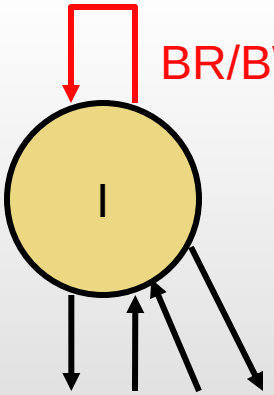
# Snooping Bandwidth Requirements

- Coherence events generated on…
  - L2 misses (and writebacks)
- Some parameters
  - 2 GHz CPUs, 2 IPC, 33% memory operations,
  - 2% of which miss in the L2, 64B blocks, 50% dirty
  - (0.33 * 0.02 * 1.5) = 0.01 events/insn
  - 0.01 events/insn * 2 insn/cycle * 2 cycle/ns = 0.04 events/ns
  - Address request: 0.04 events/ns * 4 B/event = 0.16 GB/s
  - Data response: 0.04 events/ns * 64 B/event = 2.56 GB/s
- That's 2.5 GB/s … per processor
  - With 16 processors, that's 40 GB/s!
  - With 128 processors, that's 320 GB/s!!
  - … Increasing the number of processor doesn't increase the available BW

# More Snooping Bandwidth Problems

- Bus bandwidth is not the only problem
- Also **processor snooping bandwidth**
  - 0.01 events/insn * 2 insn/cycle = 0.02 events/cycle per processor
  - 16 processors: 0.32 bus-side tag lookups per cycle
    - Add 1 port to cache tags? Sure
  - 128 processors: 2.56 bus-side tag lookups per cycle!
    - Add 3 ports to cache tags? Oy vey!
  - Implementing **inclusion** (L1 is strict subset of L2) helps a little
    - 2 additional ports on L2 tags only
    - Processor doesn't use existing tag port most of the time
    - If L2 doesn't care about bus-side transactions (99% of the time), no need to bother L1
    - Still kind of bad though
- What if bus/snooping bandwidth is not enough?
  - **Contention**

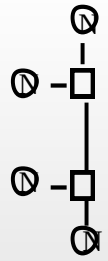
# Scalable Cache Coherence



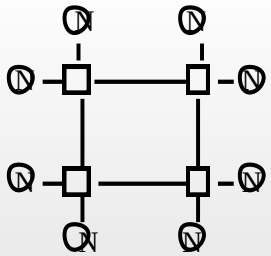

- Part I: **bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)…
  - …with scalable **interconnection network** (one that scale his bandwidth when you increase the count of processor)

- Part II: **processor snooping bandwidth**
  - Most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)…
  - …with scalable **directory protocol** (only notify processors that care)

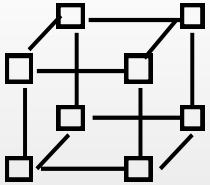
# Part I: How is a Scalable Network?



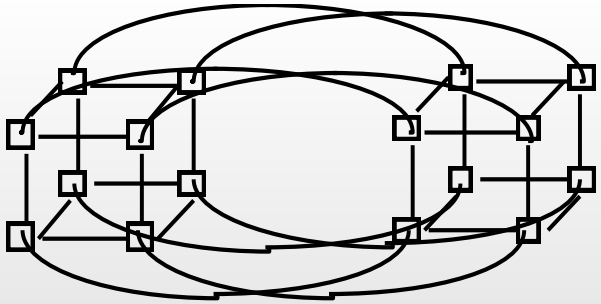
(b) 4-node



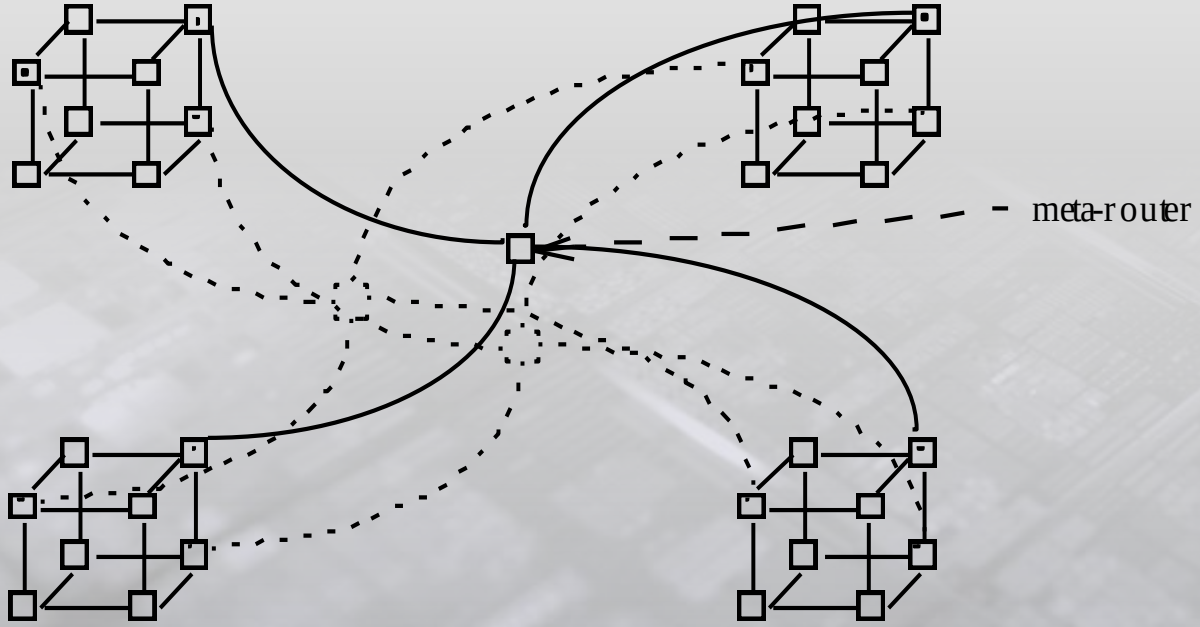
(c) 8-node



(d) 16-node



(d) 32-node

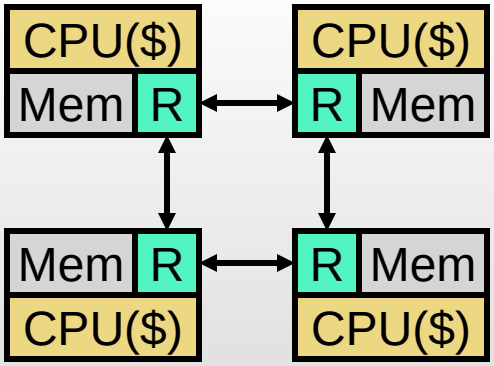


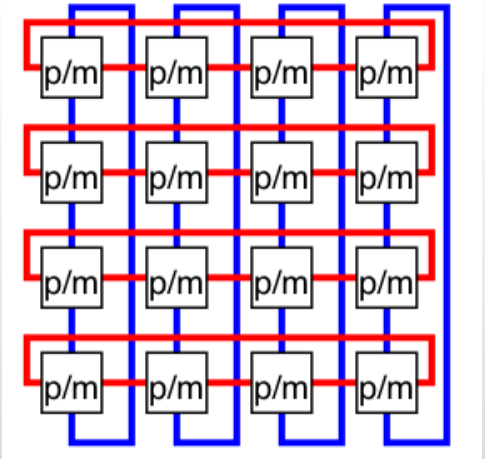

(e) 64-node

# Elements

- Network links
  - The wires used by the information to travel from source to destination
  - Could be optical or electrical
- Routers
  - The element that interconnect network links
  - Similar to LAN/WAN routers, but usually only best-effort
- Network interfaces
  - Responsible to regulate the access to network
  - Similar to Network Card, but in most cases no software
- More processors => more routers => more links => more BW

# Part II: Scalable Cache Coherence



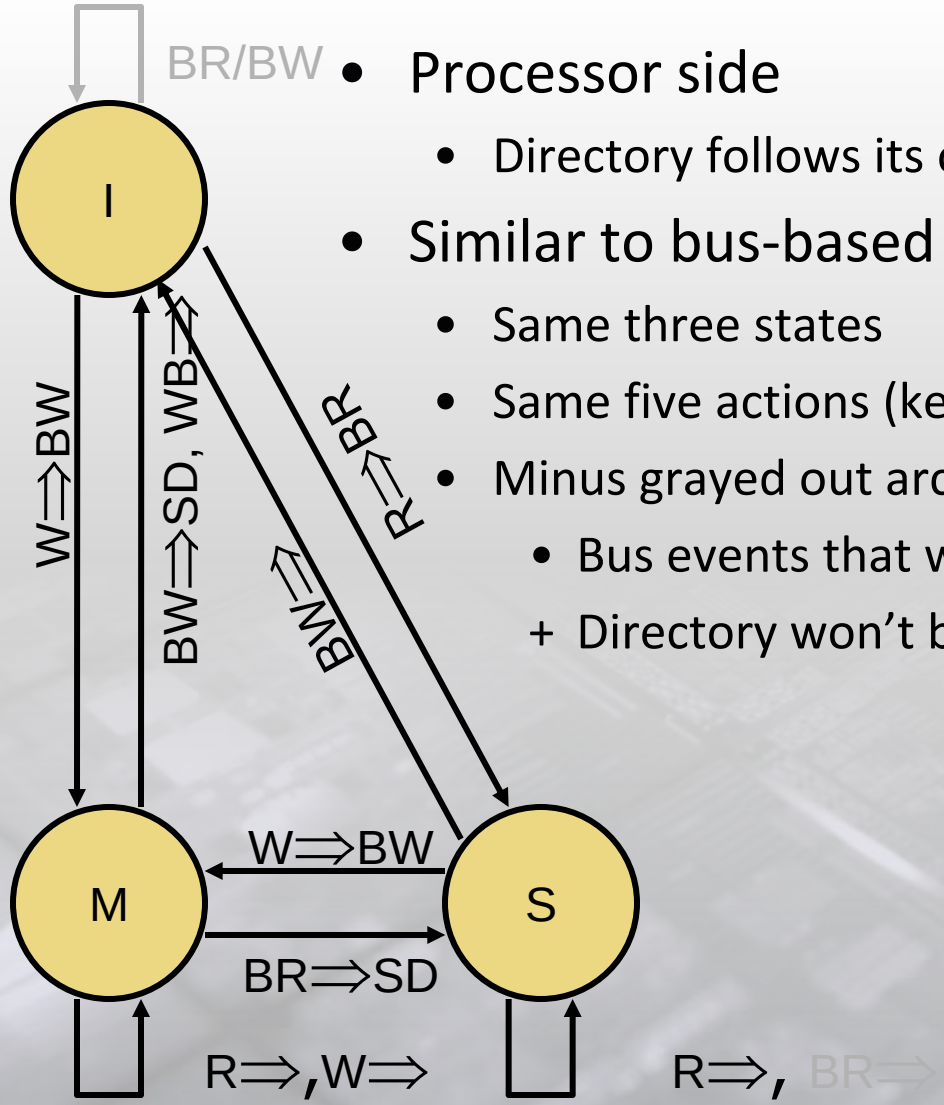




- Point-to-point interconnects
  - **Massively parallel processors (MPPs)**
    - + Can be arbitrarily large: 10000's of processors
    - Non uniform view of the memory -or- Non-cache coherent
  - **Scalable multi-processors**
    - Companies have much smaller systems: 32–64 processors
    - Intel Nehalem/ AMD Opteron – point-to-point, glueless, broadcast
- Distributed memory: non-uniform memory architecture (NUMA)

# NUMA & Directory Coherence Protocols

- Observe: address space statically partitioned
  - + Can easily determine which memory module holds a given line
    - That memory module sometimes called "**home**"
  - – Can't easily determine which processors have line in their caches
  - Bus-based protocol: broadcast events to all processors/caches
    - ± Simple and fast, but non-scalable
- **Directories**: non-broadcast coherence protocol
  - Extend memory to track caching information
  - For each physical cache line whose home this is, track:
    - **Owner**: which processor has a dirty copy (I.e., M state)
    - **Sharers**: which processors have clean copies (I.e., S state)
  - Processor sends coherence event to home directory
    - Home directory only sends events to processors that care

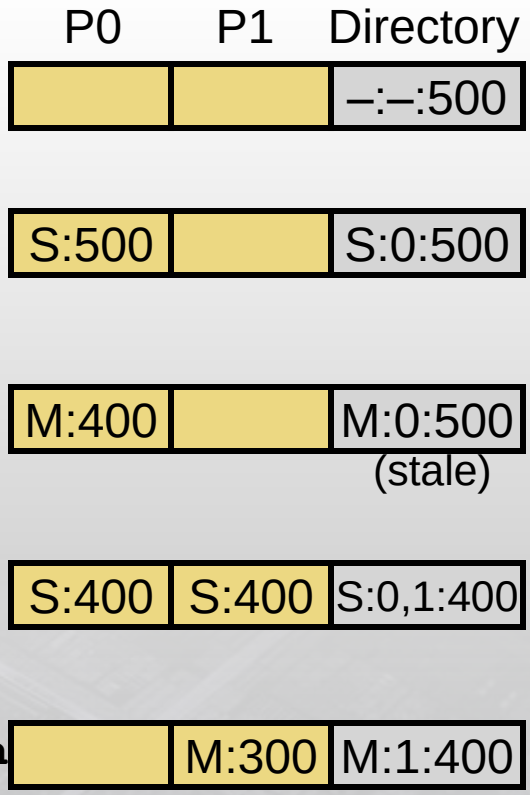
# MSI Directory Protocol




- Processor side
  - Directory follows its own protocol (obvious in principle)
- Similar to bus-based MSI
  - Same three states
  - Same five actions (keep BR/BW names)
  - Minus grayed out arcs/actions
    - Bus events that would not trigger action anyway
    - \+ Directory won't bother you unless you need to act

# Directory MSI Protocol

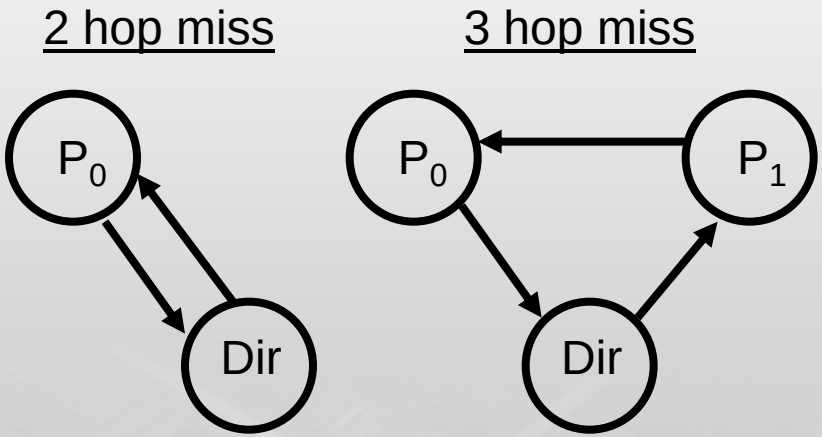
```
Processor 0                 Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call dispense_cash       0: addi r1,accts,r3
                            1: ld 0(r3),r4
                            2: blt r4,r2,6
                            3: sub r4,r2,r4
                            4: st r4,0(r3)
                            5: call dispense_cash
```




- `ld` by P1 sends BR to directory
  - Directory sends BR to P0, P0 sends P1 data, does WB, goes to **S**
- `st` by P1 sends BW to directory
  - Directory sends BW to P0, P0 goes to **I**

# Directory Flip Side: Latency

- Directory protocols
  - + Lower bandwidth consumption → more scalable
  - – Longer latencies




- Two read miss situations
- Unshared: get data from memory
  - Snooping: 2 hops (P0→memory→P0)
  - Directory: 2 hops (P0→memory→P0)
- Shared or exclusive: get data from other processor (P1)
  - Assume cache-to-cache transfer optimization
  - Snooping: 2 hops (P0→P1→P0)
  - – Directory: **3 hops** (P0→memory→P1→P0)
  - Common, with many processors high probability someone has it

# Directory Flip Side: Complexity

- Latency not only issue for directories
  - Subtle correctness issues as well
  - Stem from unordered nature of underlying inter-connect
- Individual requests to single cache must be ordered
  - Bus-based Snooping: all processors see all requests in same order
    - Ordering automatic
  - Point-to-point network: requests may arrive in different orders
    - Directory has to enforce ordering explicitly
    - Cannot initiate actions on request B…
    - Until all relevant processors have completed actions on request A
    - Requires directory to collect acks, queue requests, etc.
- Directory protocols
  - Obvious in principle
  - Complicated in practice

# Best of Both Worlds?

- Ignore processor snooping bandwidth for a minute
- Can we combine best features of snooping and directories?
  - From snooping: fast two-hop cache-to-cache transfers
  - From directories: scalable point-to-point networks
  - In other words…

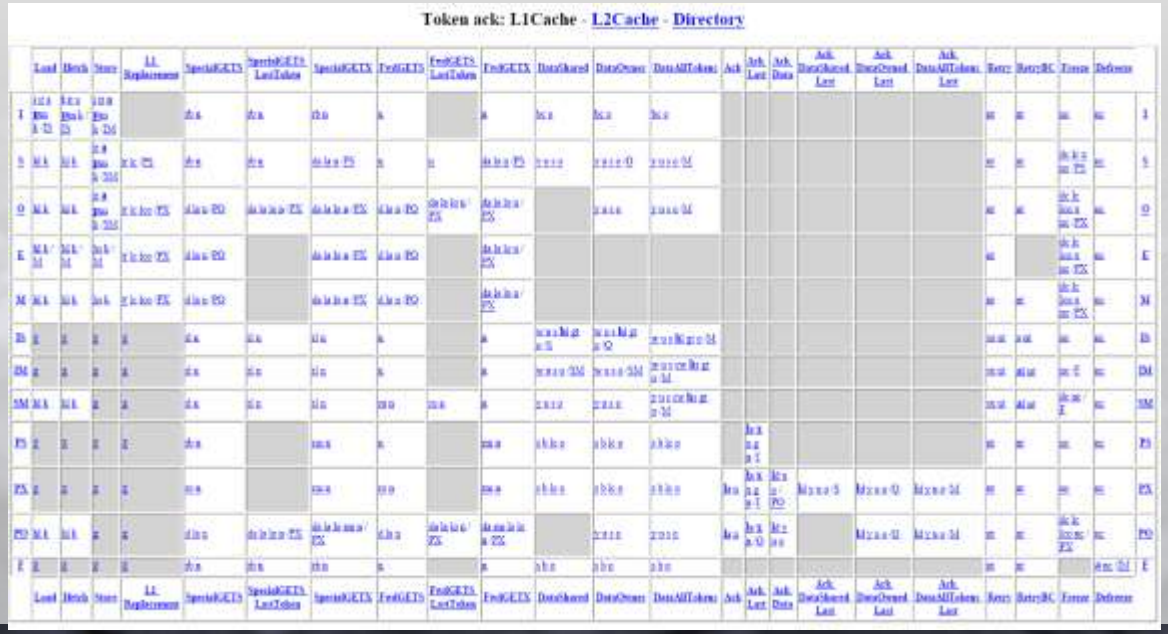
- Can we use broadcast on an unordered network?
  - Yes, and most of the time everything is fine
  - But sometimes it isn't … **protocol race**

- Research Proposal: **Token Coherence (TC)**
  - An unordered broadcast snooping protocol … without data races

# What means complex?

**Token ack: L1Cache - L2Cache - Directory**

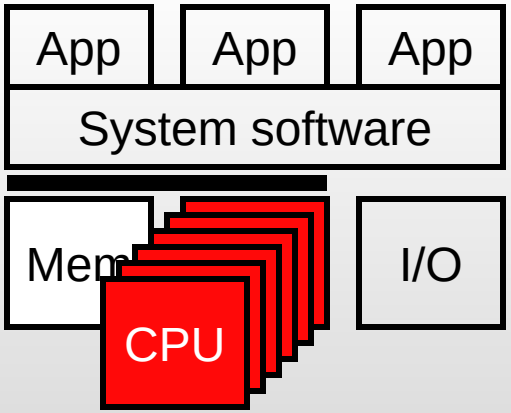
| | L1GETS | L1GETX | L1GETX Done | L1GETX DL Acks | L1GETX DL NoAcks | L2 Replacement Request | L2 Replacement Writeback | L2 Replacement Control | SpecialL1GETS | SpecialL1GETX | DataShared | DataOwned | DataAllTokens | Ack | Ack Last | AckData | AckData Owned | AckData Shared Last | AckData Owned Last | AckData AllTokens Last | CompleteFirst | CompleteOther | CompleteLast Acks | CompleteLast NoAcks | Complete NoFifo | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | n | i ca n /IM | | | | | | | rb n | n | i s u w nw /S | i s u w nw /O | i s u w nw /M | n | | | | | | | | | | | i ce nt /IM | I |
| S | n | da ca la n /IM | | | | r lm /PX | r lm /PX | r lm /PX | rb n | n | s u y nw | s u y nw /O | s u y nw /M | n | | | | | | | | | | | i ce da la nt /IM | S |
| O | da la lo n /PX | da ca la lo n /IM | | | | r lm lmo /PX | r lm lmo /PX | r lm lmo /PX | da la lo n /PX | n | | s u y nw | s u y nw /M | n | | | | | | | | | | | i ce da la lo nt /IM | O |
| M | da la lo n /PX | da ca la lo n /IM | | | | r lm lmo /PX | r lm lmo /PX | r lm lmo /PX | da la lo n /PX | n | | | | n | | | | | | | | | | | i ce da la lo nt /IM | M |
| PT | ro n | da ca la n /IM | | | | zr | z | zc | ro n | n | s u y nw | s u y nw /PO | s u y nw /PO | la n | la x n /S | ld u y nw | ld u y nw /PO | ld u y x nw /S | ld u y x nw /O | ld u y x nw /M | | | | | i ce da la nt /IM | PT |
| PX | ro n | rm ca n /IM | | | | zr | z | zc | ro n | n | s u w nw /PT | s u w nw /PO | s u w nw /PO | la n | la x x g n /I | ld u w nw | ld u w nw /PO | ld u w x nw /S | ld u w x nw /O | ld u w x nw /M | | | | | i ce nt /IM | PX |
| PO | da la lo n /PX | rm da ca la lo n /IM | | | | zr | z | zc | da la lo n /PX | n | | s u y nw | s u y nw | la n | la x n /O | ld u y nw | ld u y nw | | ld u y x nw /O | ld u y x nw /M | | | | | i ce da la lo nt /IM | PO |
| IM | rf n | f ca n | crd k n | crd lo n /PX | crd x x g n /I | zr | z | zc | rf n | n | s bf nw | s bf nw | s bf nw | la n | la n | ld cra bf nw | | ld cra u w nw /S | ld cra u w nw /O | ld cra u w nw /M | cr k nt | cr kc nt | cr loc nt /PX | cr x x g nt /I | i ce nt | IM |
| | L1GETS | L1GETX | L1GETX Done | L1GETX DL Acks | L1GETX DL NoAcks | L2 Replacement Request | L2 Replacement Writeback | L2 Replacement Control | SpecialL1GETS | SpecialL1GETX | DataShared | DataOwned | DataAllTokens | Ack | Ack Last | AckData | AckData Owned | AckData Shared Last | AckData Owned Last | AckData AllTokens Last | CompleteFirst | CompleteOther | CompleteLast Acks | CompleteLast NoAcks | Complete NoFifo | |



# Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
  - Can be easily combined to form multi-processors
    - E.g., Intel Pentium4 Xeon
  - Multi-core
- Larger scale (directory) systems built from smaller MPs
  - E.g., Sun Wildfire, NUMA-Q, IBM Summit
- Some shared memory machines are **not cache coherent**
  - E.g., CRAY-T3D/E, Cell Broadband Engine
  - Shared data is uncachable
  - If you want to cache shared data, copy it to private data section
  - Basically, cache coherence implemented in software
    - Have to really know what you are doing as a programmer

# Unit Checkpoint




- Thread-level parallelism (TLP)
  - Shared memory model
    - Multiplexed uniprocessor
    - Hardware multihreading
      - Multiprocessing
  - Synchronization
    - Lock implementation
      - Locking gotchas
  - Cache coherence
    - Bus-based protocols
    - Directory protocols
- Memory consistency models

# Hiding Store Miss Latency

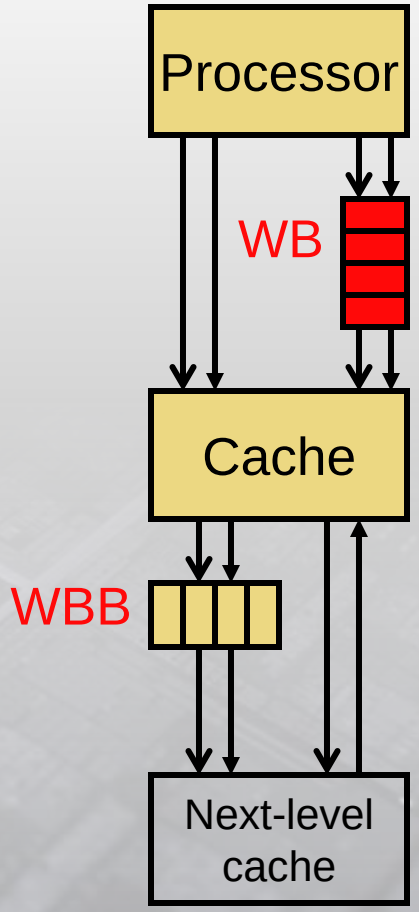
- Recall (back from caching unit)
  - Hiding store miss latency
  - How? Write buffer
- Said it would complicate multiprocessors
  - Yes. It does.

# Recall: Write Misses and Write Buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?

- **Write buffer**: a small buffer
  - Stores put address/value to write buffer, **keep going**
  - Write buffer writes stores to D$ in the background
  - Loads must search write buffer (in addition to D$)
  - \+ Eliminates stalls on write misses (mostly)
  - – Creates some problems (later)

- Write buffer vs. writeback-buffer
  - Write buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks



# Memory Consistency

- **Memory coherence**
  - Creates globally uniform (consistent) view…
  - Of **a single memory location** (in other words: cache line)
  - Not enough
    - Cache lines A and B can be individually consistent…
    - But inconsistent with respect to each other

- **Memory consistency**
  - Creates globally uniform (consistent) view…
  - Of **all memory locations relative to each other**

- Who cares? Programmers
  - Globally inconsistent memory creates mystifying behavior

# Coherence vs. Consistency

```
            A=flag=0;
Processor 0             Processor 1
A=1;                    while (!flag); // spin
flag=1;                 print A;
```

- **Intuition says**: P1 prints A=1
- **Coherence says**: absolutely nothing
  - P1 can see P0's write of **`flag`** before write of **`A`**!!! How?
    - Maybe coherence event of **`A`** is delayed somewhere in network
    - **Or P0 has a coalescing write buffer that reorders writes**

- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **Real systems** act in this strange manner

# Sequential Consistency (SC)

```
          A=flag=0;
Processor 0          Processor 1
A=1;                 while (!flag); // spin
flag=1;              print A;
```

- **Sequential consistency (SC)**
  - **Formal definition of memory view programmers expect**
  - Processors see their own loads and stores in program order
    - \+ Provided naturally, even with out-of-order execution
  - But also: processors see others' loads and stores in program order
  - And finally: all processors see same global load/store ordering
    - – Last two conditions not naturally enforced by coherence
- **Lamport definition**: multiprocessor ordering…
  - Corresponds to some sequential interleaving of uniprocessor orders
  - **Indistinguishable from multi-programmed uni-processor**

# SC Doesn't "Happen Naturally" Why?

- What is consistency concerned with?
  - P1 doesn't actually view P0's commitd loads and stores
  - Views their **coherence events** instead
  - "***Consistency model***": how observed order of coherence events relates to order of committed insns
- What does SC say?
  - Coherence event order must match committed insn order
    - And be identical for all processors
  - Let's go SC and forget this!
  - Not so easy: Let's see what that implies

# Enforcing SC

- What does it take to enforce SC?
  - Definition: all loads/stores globally ordered
  - Use ordering of coherence events to order all loads/stores
- **When do coherence events happen naturally?**
  - On cache access
  - For stores: commitment → in-order → good
    - No write buffer? Yikes, but OK with write-back D$
  - For loads: execution → out-of-order → bad
    - No out-of-order execution? Double yikes
- Is it true that multi-processors cannot be out-of-order?
  - That would be really bad
  - Out-of-order is needed to hide cache miss latency
  - And multi-processors not only have more misses…
  - … but miss handling takes longer (coherence actions)

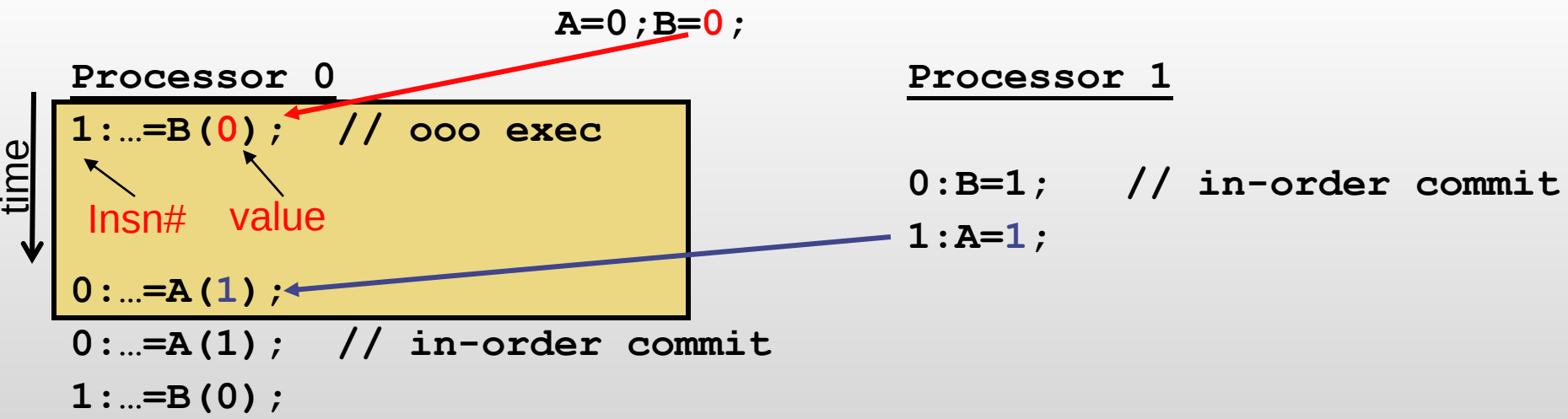
# SC + Out-of-Order

- Recall: opportunistic load scheduling in a uni-processor
  - **Loads issue speculatively relative to older stores**
  - Stores scan for younger loads to same address have issued (at LQ)
  - Find one? Ordering violation → flush and restart
  - In-flight loads effectively "snoop" older stores from same process (at SQ)

- SC + OOO can be reconciled using **same technique**
  - "Invalidation" requests from other processors snoop in-flight loads (at LQ)
  - Think of load/store queue as extension of the cache hierarchy
  - MIPS R10K does this

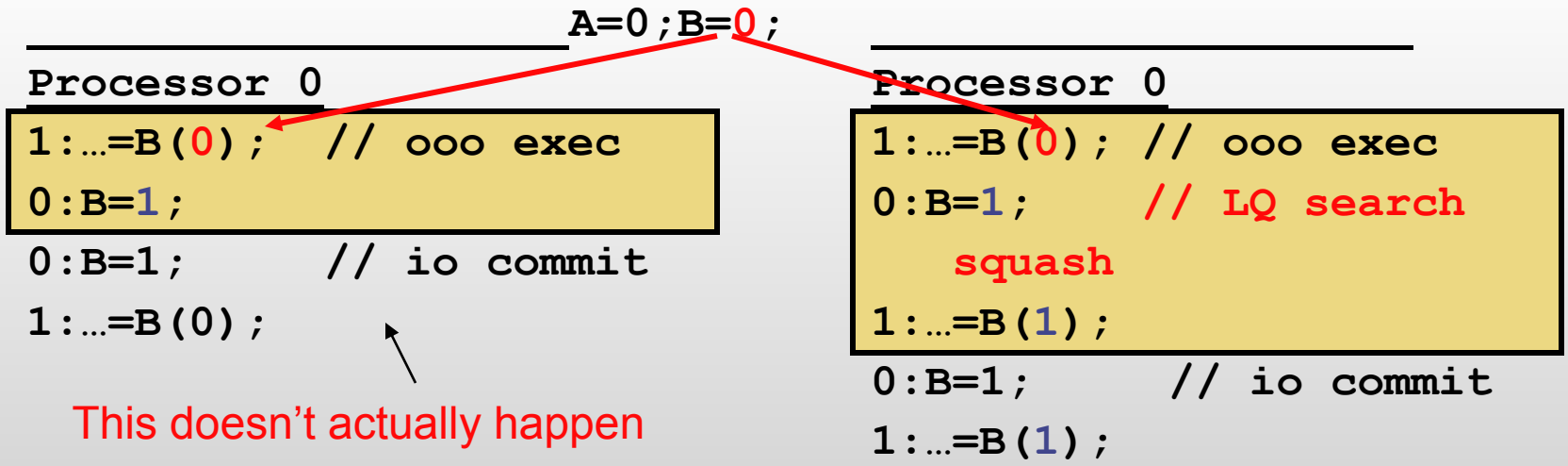
- SC implementable, but overheads still remain:
  - Write buffer issues
  - Complicated load/store queue

# SC + Out-of-Order



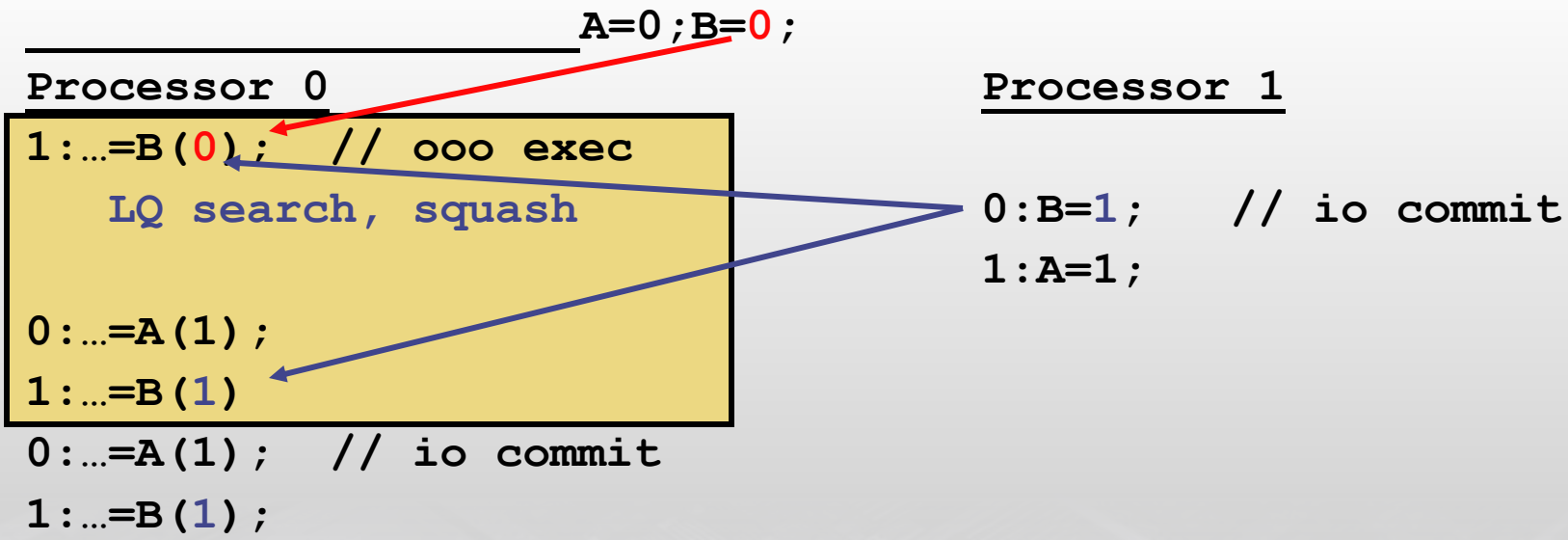

- What is this?
  - P0 sees P1's write of A
  - P0 should also see P1's write of B (older than write of A)
  - But doesn't because it read of B out-of-order w.r.t. read of A
  - Does this mean no out-of-order? (that would be bad)
  - Fortunately, there is a way

# SC + Out-of-Order




- What would happen if…
  - P0 executed read of B out-of-order w.r.t. its own write of B?
  - Would read of B get the wrong value?
  - No, write of B searches LQ, discovers read of B went early, flushes
- Same solution here…
  - Commit of B on P1 searches load queue (**LQ**) of **P0**

# SC + Out-of-Order



# SC + Write Buffers

- Store misses are slow
  - Global acquisition of M state (write permission)
  - Multiprocessors have more store misses than uniprocessors
    - **Upgrade miss**: I have block in S, require global upgrade to M
- Apparent solution: **write buffer**
  - Commit store to write buffer, let it absorb store miss latency
  - But a write buffer means…
  - I see my own stores commit before everyone else sees them
- Orthogonal to out-of-order execution
  - Even in-order processors have write buffers

# SC + Write Buffers

```
                    A=0; B=0;
Processor 0                          Processor 1
A=1;      // in-order to WrBu        B=1;      // in-order to WrBu
if(B==0)  // in-order commit        if(A==0)  // in-order commit
A=1;      // in-order to D$          B=1;      // in-order to D$
```

- Possible for both (**B==0**) and (**A==0**) to be true
- Because **B=1** and **A=1** are just sitting in the write buffers
  - Which is wrong
  - So does SC mean no write buffer?
  - Yup, and that hurts
- Research direction: use deep speculation to hide latency
  - Beyond the out-of-order window, looks like transactional memory: BulkSC

# Is SC Really Necessary?

- SC
  - \+ Most closely matches programmer's intuition (don't under-estimate)
  - – Restricts optimization by compiler, CPU, memory system
  - Supported by MIPS, HP PA-RISC

- Is full-blown SC really necessary? What about…
  - All processors see others' loads/stores in program order
  - But not all processors have to see same global order
  - \+ Allows processors to have in-order write buffers
  - – Doesn't confuse programmers too much
    - Synchronized programs (e.g., our example) work as expected
  - **Processor Consistency (PC)**: e.g., Intel/AMD x86, SPARC

# Weak Memory Ordering

- For properly synchronized programs…
- …only **`acquires/releases`** must be strictly ordered
- Why? **`acquire-release`** pairs define **critical sections**
  - Between critical-sections: data is private
    - Globally unordered access OK
  - Within critical-section: access to shared data is exclusive
    - Globally unordered access also OK
  - Implication: compiler or dynamic scheduling is OK
    - As long as re-orderings do not cross synchronization points

- **Weak Ordering (WO)**: Alpha, Itanium, ARM, PowerPC
  - ISA provides **`fence`** to indicate scheduling barriers
  - Proper use of fences is somewhat subtle
  - **Use synchronization library, don't write your own**

# Fences aka Memory Barriers

- **Fences (memory barriers)**: special insns
  - Ensure that loads/stores don't cross acquire release boundaries
  - Very roughly

```
acquire
fence
critical section
fence
release
```

- How do they work?
  - `fence` insn must commit before any younger insn dispatches
    - This also means write buffer is emptied
  - Makes lock acquisition and release slow(er)
- **Use synchronization library, don't write your own: Portability**

# Shared Memory Summary

- **Synchronization**: regulated access to shared data
  - Key feature: atomic lock acquisition operation (e.g., `t&s`)
  - Performance optimizations: test-and-test-and-set, queue locks
- **Coherence**: consistent view of individual cache lines
  - Absolute coherence not needed, relative coherence OK
  - VI and MSI protocols, cache-to-cache transfer optimization
  - Implementation? snooping, directories
- **Consistency**: consistent view of all memory locations
  - Programmers intuitively expect sequential consistency (SC)
    - Global interleaving of individual processor access streams
    - Not always naturally provided, may prevent optimizations
  - Weaker ordering: consistency only for synchronization points

# Wrap-up
## + Power
## - Programming

# Recall: Reducing Dynamic Power

- Target each component: $P_{dynamic} \sim N * C * V^2 * f * A$
- **Reduce number of transistors** (N)
  - Use fewer transistors/gates
- **Reduce capacitance** (C)
  - Smaller transistors (Moore's law)
- **Reduce voltage** (V)
  - Quadratic reduction in energy consumption!
  - But also slows transistors (transistor speed is ~ to V)
- **Reduce frequency** (f)
  - Slower clock frequency (reduces power but not energy) ***Why?***
- **Reduce activity** (A)
  - "Clock gating" disable clocks to unused parts of chip
  - Don't switch gates unnecessarily

# Recall: Reducing Static Power

- Target each component: $P_{static} \sim N * V * e^{-Vt}$
- **Reduce number of transistors** (N)
  - Use fewer transistors/gates
- **Reduce voltage** (V)
  - Linear reduction in static energy consumption
  - But also slows transistors (transistor speed is ~ to V)
- **Disable transistors** (also targets N)
  - "Power gating" disable power to unused parts (long latency to power up)
  - Power down units (or entire cores) not being used
- **Dual $V_t$** – use a mixture of high and low $V_t$ transistors
  - Use slow, low-leak transistors in SRAM arrays
  - Requires extra fabrication steps (cost)
- **Low-leakage transistors**
  - High-K/Metal-Gates in Intel's 45nm process
- Note: reducing frequency can actually hurt static energy. ***Why?***

# Recall: Voltage/Frequency Scaling

| | Mobile PentiumIII "**SpeedStep**" | Transmeta 5400 "LongRun" | Intel X-Scale (StrongARM2) |
|---|---|---|---|
| f (MHz) | 300–1000 (step=50) | 200–700 (step=33) | 50–800 (step=50) |
| V (V) | 0.9–1.7 (step=0.1) | 1.1–1.6V (cont) | 0.7–1.65 (cont) |
| High-speed | 3400MIPS @ 34W | 1600MIPS @ 2W | 800MIPS @ 0.9W |
| Low-power | 1100MIPS @ 4.5W | 300MIPS @ 0.25W | 62MIPS @ 0.01W |

- Dynamic voltage/frequency scaling
  - **Favors parallelism**

# Multiprocessing & Power Consumption

- Multiprocessing can be very power efficient

- Recall: dynamic voltage and frequency scaling
  - Performance vs power is NOT linear
  - Example: Intel's Xscale
    - 1 GHz $\rightarrow$ 200 MHz reduces energy used by 30x

- Impact of parallel execution
  - What if we used 5 Xscales at 200Mhz?
  - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
    - 5 cores * 1/30th = 1/6th

- Assumes parallel programming ( a difficult task)
- Assumes parallel speedup (even more difficult task)
  - Remember Ahmdal's law

# Shared Memory Programming

- Standard Alternatives
  - Posix Threads
    - Implicit communication
    - Explicit synchronization
  - **OpenMP**
    - Implicit communication
    - Implicit synchronization
  - …
- Non-standard Alternatives
  - SysV
  - IRIX Sprocs
  - WIN32
  - …

# What is OpenMP?

- Characteristics
  - Shared-memory Open standard for HPC programming
  - Highly portable
- Components
  - Preprocessor directives (bindings for C/C++/F77/F90/Ada…)
  - Libraries
  - Runtime
  - Environment
- Advantages over pthreads (OS)
  - \+ Easy to parallelize  many HPC codes (in some cases is fully automatic)
  - \+ Many nasty constructs and calls to threads library are hidden
  - \+ Better portability (??)
  - \- Specific for HPC

# Hello.c

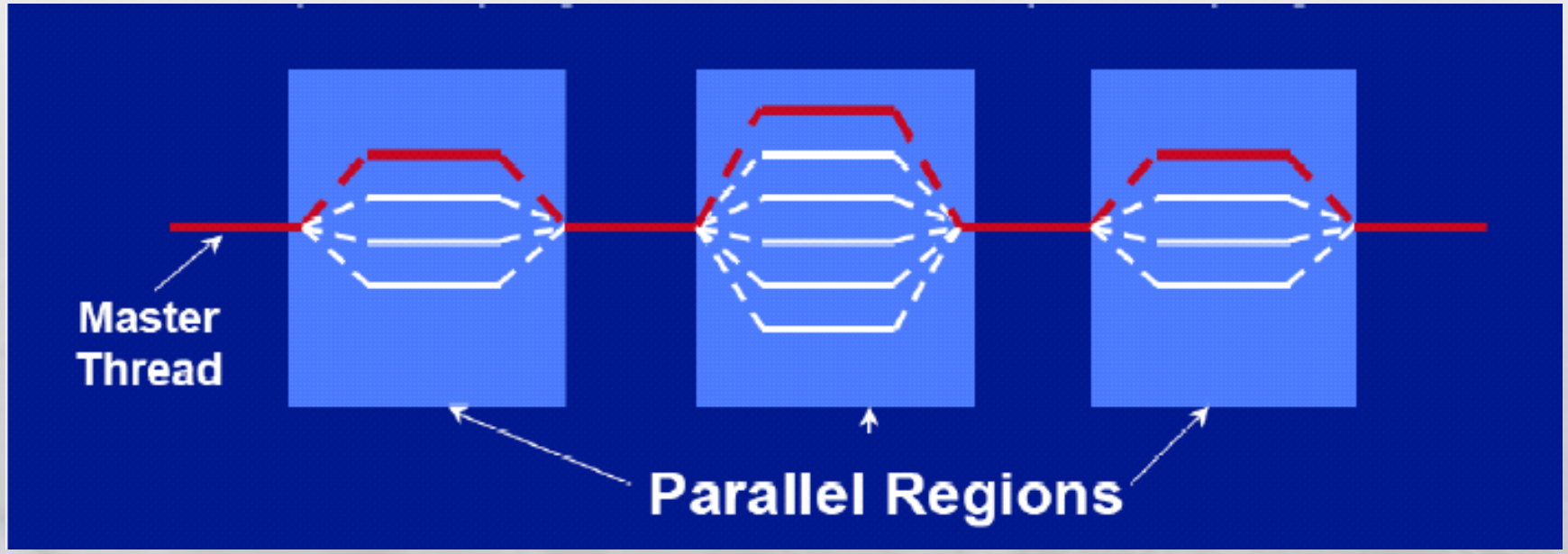
```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main( int argc, char * argv[]) {

  int nthreads, nprocs;

  // Ask for the num of idle
  // processors

  nprocs = omp_get_num_procs();

   printf("Hello World! There is%i
    processor \n",nprocs);

  // Overwrtiable with env. Variable
  //OMP_NUM_THREADS  (gcc doesn'work?)
 //http://gcc.gnu.org/onlinedocs/gcc-4.4.0/libgomp/

  omp_set_num_threads(nprocs);
```

```
// parallel region
  #pragma omp parallel
  {
    printf("Hello from %i\n", omp_get_thread_num());

    if(omp_get_thread_num() == 0)
    {
      // How many threads here?.
      nthreads = omp_get_num_threads();
      printf("Número de threads =    %i\n",nthreads);

    }
  } //join
  omp_set_num_threads(2);
  #pragma omp parallel
  {
   printf("Ahora somos dos threads y to soy %i\n",
                omp_get_thread_num());

  }
  return 0;
}
```

With gcc4.2>=

**gcc –fopenmp hello.c**

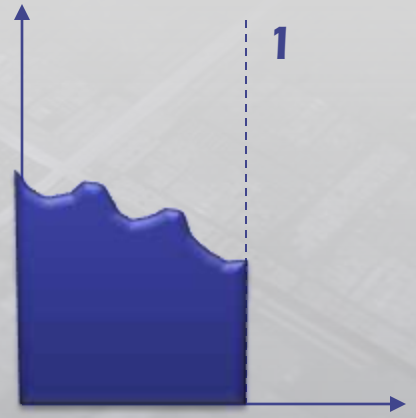
# Parallel regions and execution flow



# Ejemplo: π (sec.)

```
static long num_steps = 1000000000;
double step;
void main ()
{
  int i;
  double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0;i< num_steps; i++)
  {
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

$$\int_0^1 \frac{dx}{1+x^2} = \left[ tg^{-1}x \right]_0^1 = \frac{\pi}{4} - 0$$



# Example : π (par.)

```
#include <omp.h>
static long num_steps = 1000000000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
          double x; int id;
          id = omp_get_thread_num();
          sum[id]=0.0;
          for (i=id*num_steps/NUM_THREADS, i< (id+1)*num_steps/NUM_THREADS; i++)
          {
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
          }
    }
    for(i=0, pi=0.0;i<NUM_THREADS;i++) pi += sum[i] * step;
}
```

# Example: π (loop)

```
#include <omp.h>
static long num_steps = 1000000000;
double step;
#define NUM_THREADS 2
void main ()
{
   int i; double x, pi, sum[NUM_THREADS];
   step = 1.0/(double) num_steps;
   for(i=0;i<NUM_THREADS;i++) sum[i]=0.0;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel for private(x)
   for (i=0;i< num_steps; i++)
   {
        x = (i+0.5)*step;
        sum[omp_get_thread_num()] += 4.0/(1.0+x*x);
   }
   for(i=0, pi=0.0;i<NUM_THREADS;i++)
        pi += sum[i] * step;
}
```

# #pragma omp critical

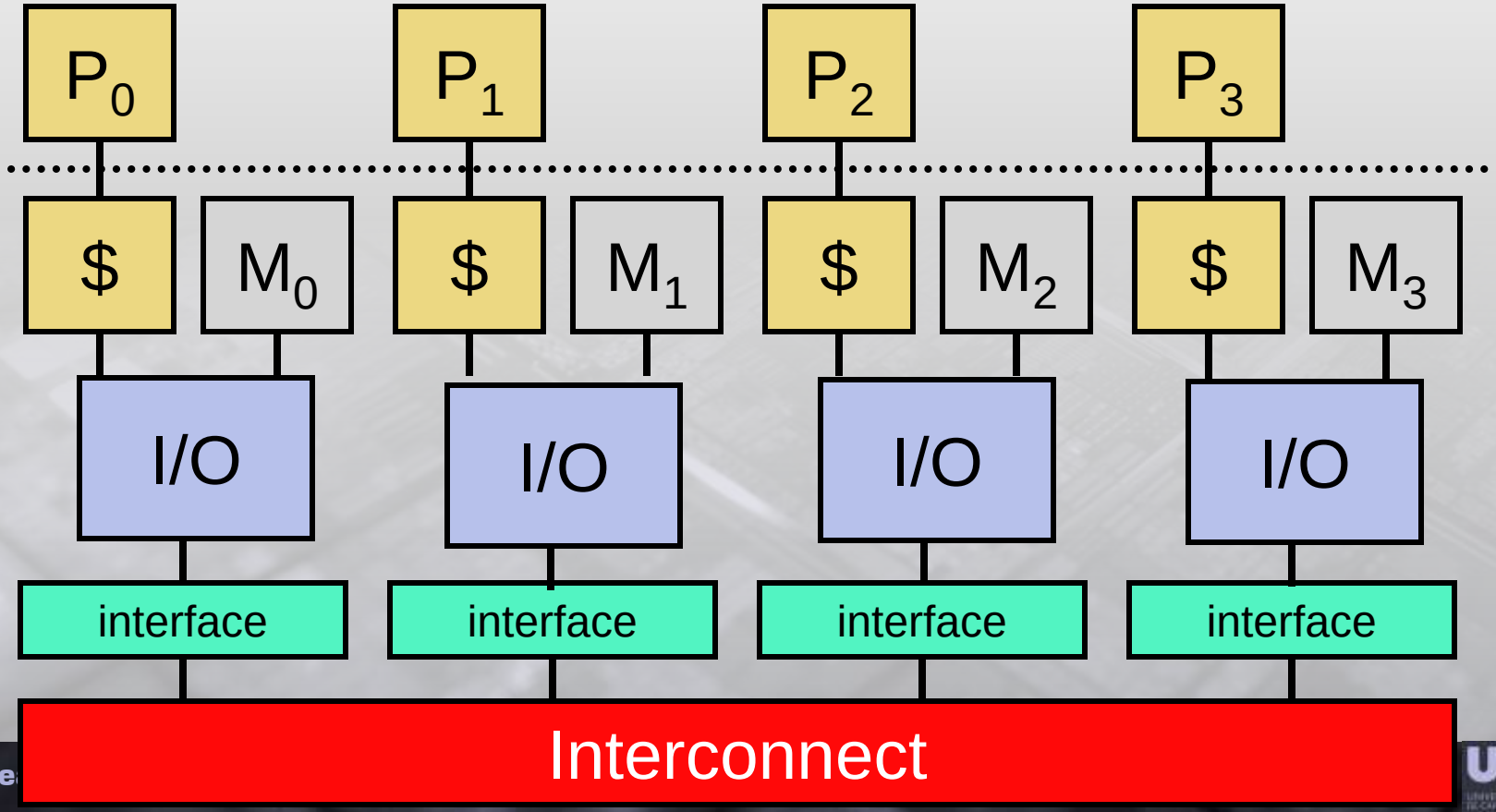
```
#include <omp.h>
static long num_steps = 1000000000;
double step;
#define NUM_THREADS 2
void main ()
{
   int i;
   double x, pi, sum;
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel for private(x) shared (sum)
   for (i=0;i< num_steps; i++)
   {
        x = (i+0.5)*step;
   #pragma omp critical
        sum+= 4.0/(1.0+x*x);
   }
   pi = sum * step;
}
```

# Strongly Specific

```
#include <omp.h>
static long num_steps = 1000000000;
double step;
#define NUM_THREADS 2
void main ()
{
   int i;
   double x, pi, sum;
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel for private(x) reduction(+:sum)
   for (i=0;i< num_steps; i++)
   {
        x = (i+0.5)*step;
        sum+= 4.0/(1.0+x*x);
   }
   pi = sum * step;
}
```

# Multi-computers

- Non-shared-memory architecture, slower network
- Message-passing paradigm… even harder than shared memory
- More next year…



# Example: π with MPI

```
#include <stdio.h>
#include "mpi.h"
double pieza_de_pi(int,int,long);
void main(int argc, char ** argv)
{
    long intervalos=160000000;
    int miproc, numproc;
    double pi, di;
    int i;
    MPI_Status status;
    double t3, t4;
    t3 = MPI_Wtime(); /* tiempo de inicio */
    MPI_Init (&argc, &argv); /* Inicializar MPI */
    MPI_Comm_rank(MPI_COMM_WORLD,&miproc); MPI_Comm_size(MPI_COMM_WORLD,&numproc);
    MPI_Barrier(MPI_COMM_WORLD);
    if (miproc == 0)
    {
        for (i = 1; i < numproc; i++)
            MPI_Send(&intervalos, 1, MPI_LONG, i, 98, MPI_COMM_WORLD);
    }
    else
        MPI_Recv(&intervalos, 1, MPI_LONG, 0, 98, MPI_COMM_WORLD, &status);

    /* cada proceso ejecuta pieza_de_pi */
    pi = pieza_de_pi(miproc, numproc, intervalos);

    MPI_Barrier (MPI_COMM_WORLD); /* sección de sincronización */
    if (miproc == 0) /* si es maestro recoge los resultados, suma y los imprime */
    {
        for (i = 1; i < numproc; i++)
        {
            MPI_Recv (&di, 1, MPI_DOUBLE, i, 99, MPI_COMM_WORLD, &status);
            pi += di;
        }
MPI_Barrier(MPI_COMM_WORLD);

        t4 = MPI_Wtime(); /* tiempo final */
        printf("Valor de pi: %lf \n",pi);
        printf("Tiempo de ejecucion: %.3lf seg\n",t4-t3)

    }
    else
    { /* el esclavo envia los resultados al maestro */
    MPI_Send(&pi, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
      MPI_Barrier (MPI_COMM_WORLD);
    }
    MPI_Finalize ();

}

double pieza_de_pi(int idproc, int nproc, long intervalo
{

    double ancho, x, localsum;
    long j;


    ancho = 1.0 / intervalos; /* peso de la muestra */
    localsum = 0.0;
    for (j = idproc; j < intervalos; j += nproc)

        {
        x = (j + 0.5) * ancho;
        localsum += 4 / (1 + x * x);

    }
    return(localsum * ancho);


}
```

# Parallel Programming

- Parallel programming is also hard because:
  - Thread scheduling & load balancing
  - Lack of deterministic behavior
  - …
- Thread scheduling load balance
  - What if thread creation/destruction is “mostly” free?
  - Already happening in GPU assisted processing (nVidia CUDA, ATI Stream)
  - General purpose processing?
  - Could synchronization be avoidable too?
- Lack of deterministic behavior
  - Record, via hardware, winners in races-for-data
  - Use that information to “replay” execution in program debugging
  - Deterministic back-track (as we are used to see in sequential programming). *One of the uses of VM*

# Delegate the problem?

- Some advocates for add another level to the stack: specialized languages (called Domain Specific Languages) plus runtime
  - Only runtime programmer should deal with the problem
  - Conventional programmer has to know only the DSL interface, runtime does the “nasty” work.
  - Examples:
    - SQL (data parallelism)
    - Matlab (scientific)
    - Ruby/Rails (Web)
- Will be productive the “classic way” in 100+ core age?

# Scalability of parallel programming: Amdahl law

- Queremos obtener un speedup de 80 para una aplicación que debe ser ejecutada en un computador paralelo de 100 procesadores: ¿Qué porcentaje del código deberá ser susceptible de paralelizar?

$$\text{Speedup} = \frac{\text{Tejec}_{1p}}{\text{Tejec}_{100p}} = 80$$

$$\text{Tejec}_{1p} = \text{Tejec}_{1p}(\text{Fracción}_{\text{paralela}} + \text{Fracción}_{\text{secuencia}}) =$$
$$= \text{Tejec}_{1p}(\text{Fracción}_{\text{paralela}} + 1 - \text{Fracción}_{\text{paralela}})$$

$$\text{Tejec}_{100p} = \text{Tejec}_{1p}\left(\frac{\text{Fracción}_{\text{paralela}}}{100} + 1 - \text{Fracción}_{\text{paralela}}\right)$$

$$\text{Speedup} = \frac{1}{\frac{\text{Fracción}_{\text{paralela}}}{100} + 1 - \text{Fracción}_{\text{paralela}}} = 80 \Rightarrow \text{Fracción}_{\text{paralela}} = 0.9975$$

**Solo el 0.25% puede ser secuencial!!!**

# Acknowledgments


- Slides developed by Amir Roth of University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.
- Slides enhanced by Milo Martin and Mark Hill with sources that included Profs. Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Sohi, Vijaykumar, and Wood
- Slides re-adapted by V. Puente of University of Cantabria