18-447
Computer Architecture
Lecture 28: Multiprocessors

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 4/14/2013

Execution-based Prefetchers (I)

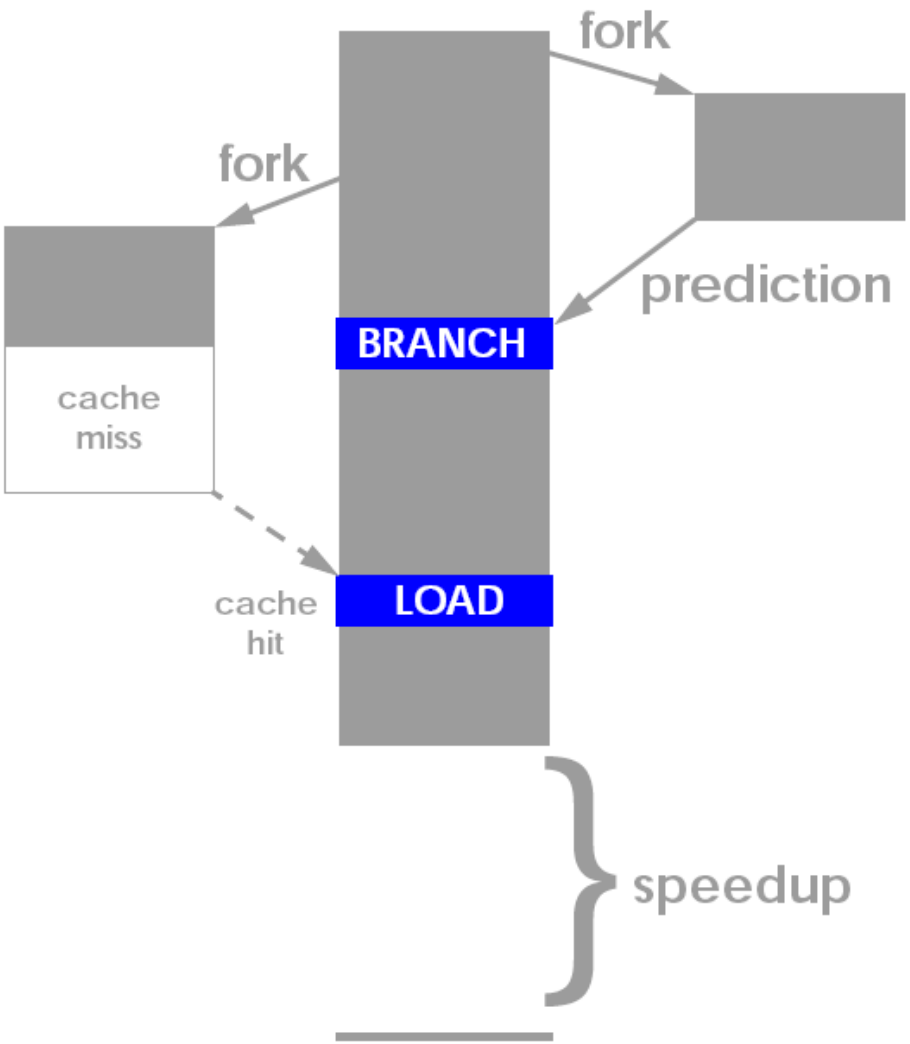
- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses
  - Speculative thread: Pre-executed program piece can be considered a "thread"
  - Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context (think fine-grained multithreading)
  - On the same thread context in idle cycles (during cache misses)

Execution-based Prefetchers (II)

- How to construct the speculative thread:
  - Software based pruning and “spawn” instructions
  - Hardware based pruning and “spawn” instructions
  - Use the original program (no construction), but
    - Execute it faster without stalling and correctness constraints

- Speculative thread
  - Needs to discover misses before the main program
    - Avoid waiting/stalling and/or compute less
  - To get ahead, uses
    - Perform only address generation computation, branch prediction, value prediction (to predict “unknown” values)

Thread-Based Pre-Execution



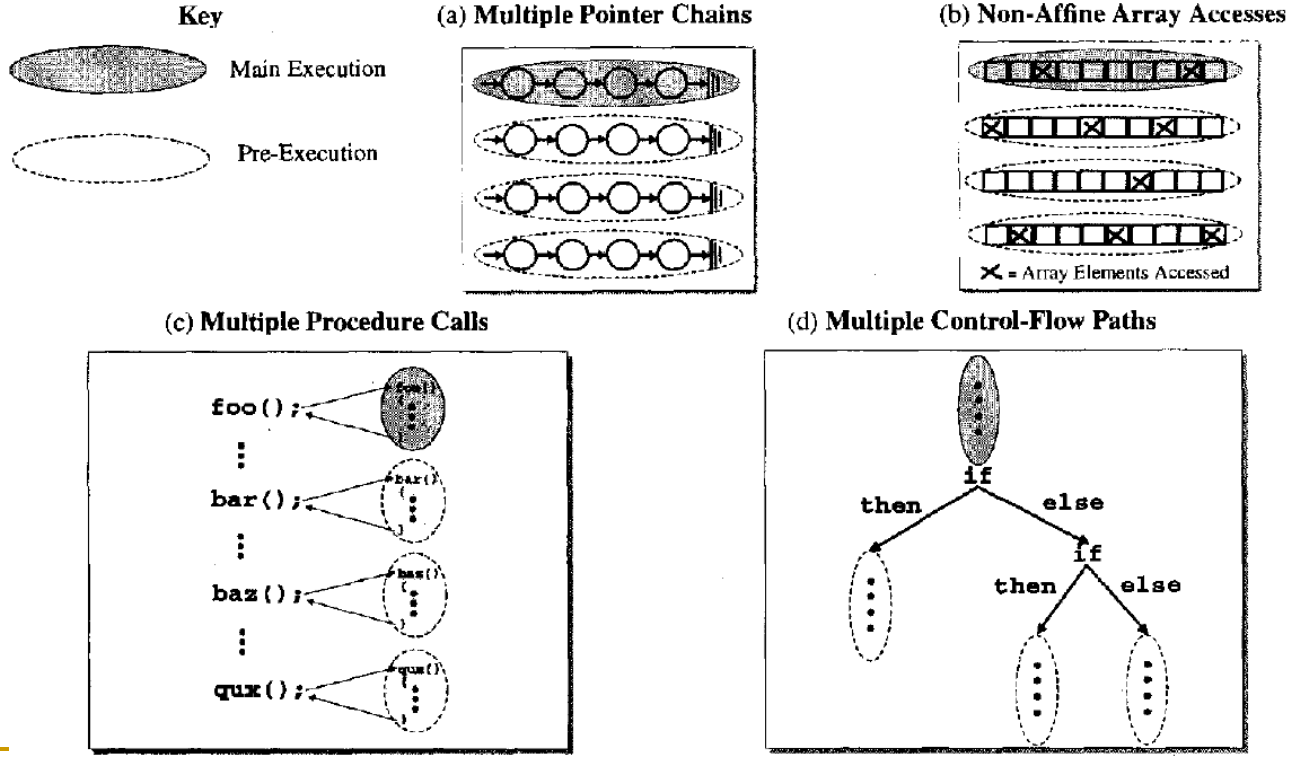
- Dubois and Song, "Assisted Execution," USC Tech Report 1998.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

Thread-Based Pre-Execution Issues

- Where to execute the precomputation thread?
  1. Separate core (least contention with main thread)
  2. Separate thread context on the same core (more contention)
  3. Same core, same context
     - When the main thread is stalled
- When to spawn the precomputation thread?
  1. Insert spawn instructions well before the "problem" load
     - How far ahead?
       - Too early: prefetch might not be needed
       - Too late: prefetch might not be timely
  2. When the main thread is stalled
- When to terminate the precomputation thread?
  1. With pre-inserted CANCEL instructions
  2. Based on effectiveness/contention feedback

Thread-Based Pre-Execution Issues

- Read
  - Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," ISCA 2001.
  - Many issues in software-based pre-execution discussed



An Example

(a) Original Code

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
   // loop over "trips" lists
   if (arcout[1].ident != FIXED) {
      ...
      first_of_sparse_list = arcout + 1;
   }
   ...
   arcin = (arc_t *)first_of_sparse_list
                    →tail→mark;
   // traverse the list starting with
   // the first node just assigned
   while (arcin) {
      tail = arcin→tail;
      ...
      arcin = (arc_t *)tail→mark;
   }
   i++, arcout+=3;
}
```

(b) Code with Pre-Execution

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
   // loop over "trips" lists
   if (arcout[1].ident != FIXED) {
      ...
      first_of_sparse_list = arcout + 1;
   }
   ...
   // invoke a pre-execution starting
   // at END_FOR
   PreExecute_Start(END_FOR);
   arcin = (arc_t *)first_of_sparse_list
            →tail→mark;
   // traverse the list starting with
   // the first node just assigned
   while (arcin) {
      tail = arcin→tail;
      ...
      arcin = (arc_t *)tail→mark;
   }
   // terminate this pre-execution after
   // prefetching the entire list
   PreExecute_Stop();
END_FOR:
   // the target address of the pre-
   // execution
   i++, arcout+=3;
}
// terminate this pre-execution if we
// have passed the end of the for-loop
PreExecute_Stop();
```

Figure 2. Abstract versions of an important loop nest in the Spec2000 benchmark `mcf`. Loads that incur many cache misses are underlined.

The Spec2000 benchmark `mcf` spends roughly half of its execution time in a nested loop which traverses a set of linked lists. An abstract version of this loop is shown in Figure 2(a), in which the for-loop iterates over the lists and the while-loop visits the elements of each list. As we observe from the figure, the first node of each list is assigned by dereferencing the pointer `first_of_sparse_list`, whose value is in fact determined by `arcout`, an induction variable of the for-loop. Therefore, even when we are still working on the current list, the first and the remaining nodes on the next list can be loaded speculatively by pre-executing the next iteration of the for-loop.

Figure 2(b) shows a version of the program with pre-execution code inserted (shown in boldface). **END_FOR** is simply a label to denote the place where `arcout` gets updated. The new instruction **PreExecute_Start(END_FOR)** initiates a pre-execution thread, say T, starting at the PC represented by **END_FOR**. Right after the pre-execution begins, T's registers that hold the values of `i` and `arcout` will be updated. Then `i`'s value is compared against `trips` to see if we have reached the end of the for-loop. If so, thread T will exit the for-loop and encounters a **PreExecute_Stop()**, which will terminate the pre-execution and free up T for future use. Otherwise, T will continue pre-executing the body of the for-loop, and hence compute the first node of the next list automatically. Finally, after traversing the entire list through the while-loop, the pre-execution will be terminated by another **PreExecute_Stop()**. Notice that any **PreExecute_Start()** instructions encountered during pre-execution are simply ignored as we do not allow nested pre-execution in order to keep our design simple. Similarly, **PreExecute_Stop()** instructions cannot terminate the main thread either.

Example ISA Extensions

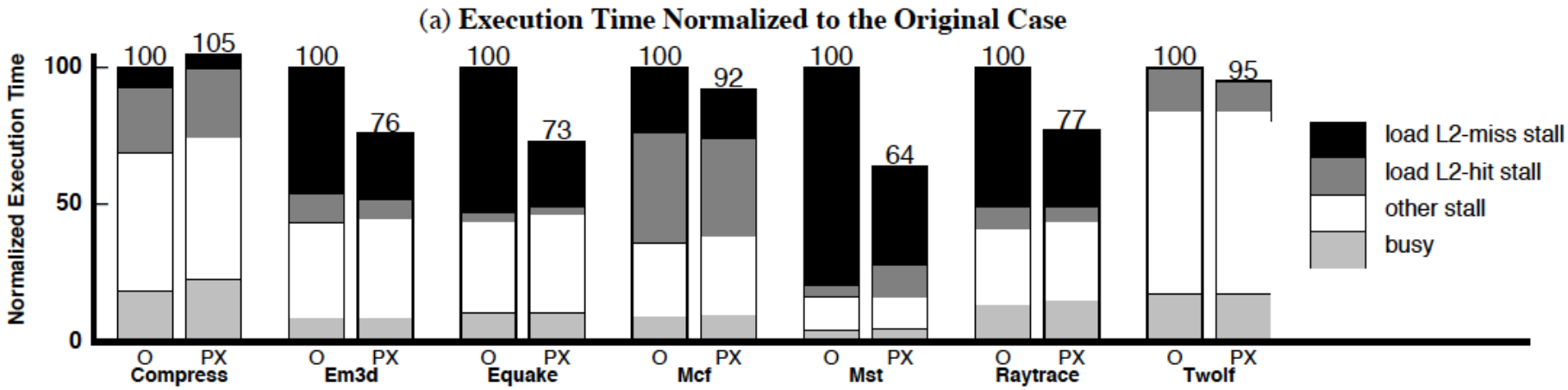
$Thread\_ID$ = **PreExecute_Start**($Start\_PC$, Max_Insts):
Request for an idle context to start pre-execution at $Start\_PC$ and stop when Max_Insts instructions have been executed; $Thread\_ID$ holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

PreExecute_Stop(): The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

PreExecute_Cancel($Thread\_ID$): Terminate the pre-execution thread with $Thread\_ID$. This instruction has effect only if it is executed by the main thread.

Figure 4. Proposed instruction set extensions to support pre-execution. (C syntax is used to improve readability.)

Results on a Multithreaded Processor



Problem Instructions

- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.
- Zilles and Sohi, "Understanding the backward slices of performance degrading instructions," ISCA 2000.

Figure 2. *Example problem instructions from heap insertion routine in* ***vpr****.*

```
struct s_heap **heap; // from [1..heap_size]
int heap_size; // # of slots in the heap
int heap_tail; // first unused slot in heap

  void add_to_heap (struct s_heap *hptr) {
     ...
1.   heap[heap_tail] = hptr;
2.   int ifrom = heap_tail;
3.   int ito = ifrom/2;
4.   heap_tail++;
5.   while ((ito >= 1) &&
6.         (heap[ifrom]->cost < heap[ito]->cost))
7.      struct s_heap *temp_ptr = heap[ito];
8.      heap[ito] = heap[ifrom];
9.      heap[ifrom] = temp_ptr;
10.     ifrom = ito;
11.     ito = ifrom/2;
     }
  }
```

branch misprediction

cache miss

Fork Point for Prefetching Thread

Figure 3. *The* **`node_to_heap`** *function, which serves as the fork point for the slice that covers* **`add_to_heap`**.

```
void node_to_heap (..., float cost, ...) {
    struct s_heap *hptr;  <------ fork point
    ...
    hptr = alloc_heap_data();
    hptr->cost = cost;
    ...
    add_to_heap (hptr);
}
```

Pre-execution Thread Construction

Figure 4. *Alpha assembly for the* **`add_to_heap`** *function. The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.*

```
node_to_heap:
     ... /* skips ~40 instructions */
2    lda     s1, 252(gp)    # &heap_tail
2    ldl     t2, 0(s1)      # ifrom = heap_tail
1    ldq     t5, -76(s1)    # &heap[0]
3    cmplt   t2, 0, t4      # see note
4    addl    t2, 0x1, t6    # heap_tail ++
1    s8addq  t2, t5, t3     # &heap[heap_tail]
4    stl     t6, 0(s1)      # store heap_tail
1    stq     s0, 0(t3)      # heap[heap_tail]
3    addl    t2, t4, t4     # see note
3    sra     t4, 0x1, t4    # ito = ifrom/2
5    ble     t4, return     # (ito < 1)
loop:
6    s8addq  t2, t5, a0     # &heap[ifrom]
6    s8addq  t4, t5, t7     # &heap[ito]
11   cmplt   t4, 0, t9      # see note
10   move    t4, t2         # ifrom = ito
6    ldq     a2, 0(a0)      # heap[ifrom]
6    ldq     a4, 0(t7)      # heap[ito]
11   addl    t4, t9, t9     # see note
11   sra     t9, 0x1, t4    # ito = ifrom/2
6    lds     $f0, 4(a2)     # heap[ifrom]->cost
6    lds     $f1, 4(a4)     # heap[ito]->cost
6    cmptlt  $f0,$f1,$f0    # (heap[ifrom]->cost
6    fbeq    $f0, return    #  < heap[ito]->cost)
8    stq     a2, 0(t7)      # heap[ito]
9    stq     a4, 0(a0)      # heap[ifrom]
5    bgt     t4, loop       # (ito >= 1)
return:
     ... /* register restore code & return */
```

note: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.

Figure 5. *Slice constructed for example problem instructions. Much smaller than the original code, the slice contains a loop that mimics the loop in the original code.*

```
slice:
1      ldq     $6, 328(gp)     # &heap
2      ldl     $3, 252(gp)     # ito = heap_tail
slice_loop:
3,11   sra     $3, 0x1, $3     # ito /= 2
6      s8addq  $3, $6, $16     # &heap[ito]
6      ldq     $18, 0($16)     # heap[ito]
6      lds     $f1, 4($18)     # heap[ito]->cost
6      cmptle  $f1,$f17,$f31   # (heap[ito]->cost
                               #  < cost) PRED
       br      slice_loop

## Annotations
fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```

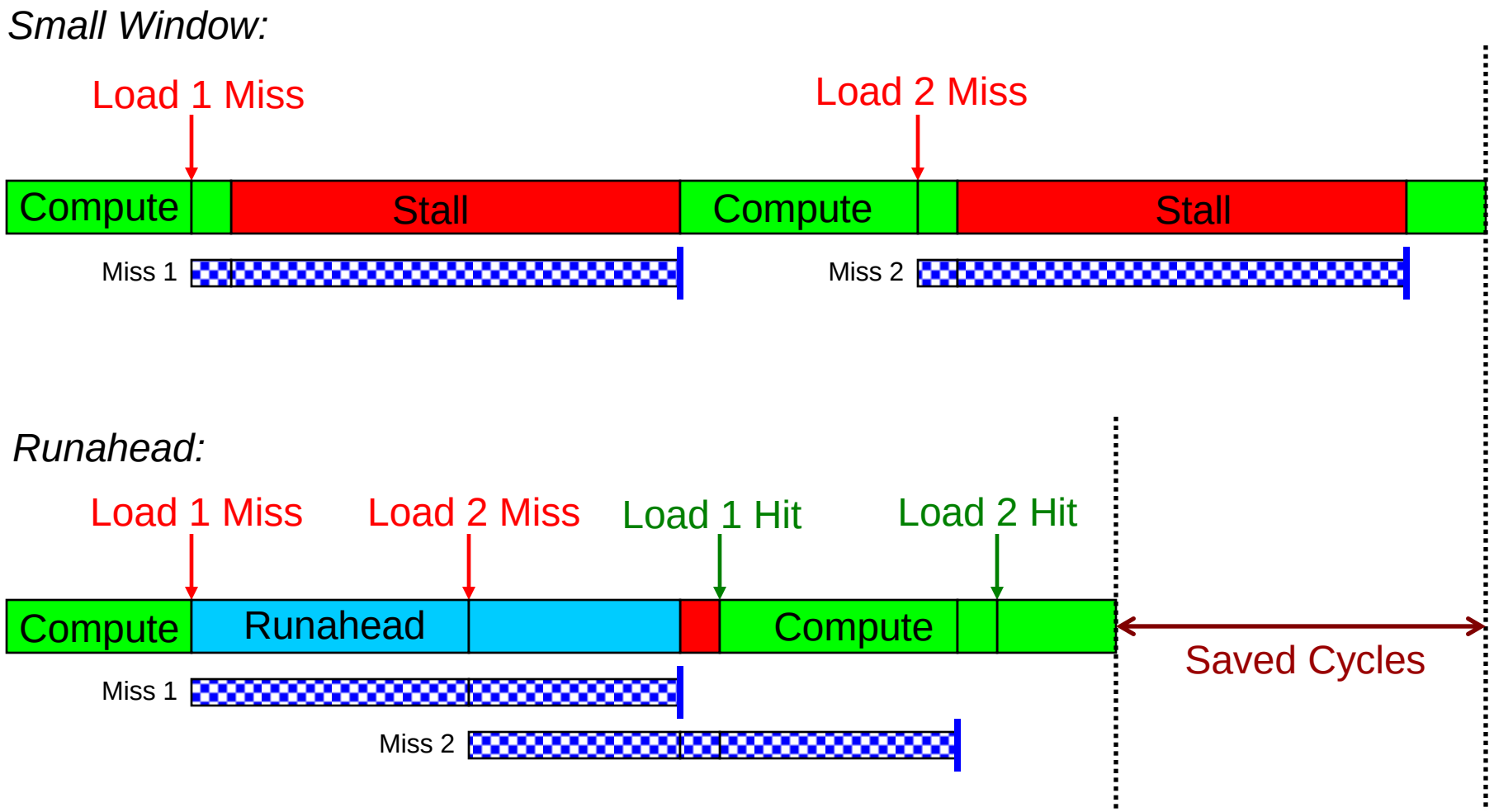
Review: Runahead Execution

- A simple pre-execution method for prefetching purposes

- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode
- In runahead mode:
  - Speculatively pre-execute instructions
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped
- Runahead mode ends when the original miss returns
  - Checkpoint is restored and normal execution resumes

- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.

Review: Runahead Execution (Mutlu et al., HPCA 2003)



Multiprocessors and Issues in Multiprocessing

Readings: Multiprocessing

- Required
  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
  - Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- Recommended
  - Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966
  - Hill, Jouppi, Sohi, "Multiprocessors and Multicomputers," pp. 551-560 in Readings in Computer Architecture.
  - Hill, Jouppi, Sohi, "Dataflow and Multithreading," pp. 309-314 in Readings in Computer Architecture.

Readings: Cache Coherence

- Required
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)

- Recommended:
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

Remember: Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- SISD: Single instruction operates on single data element
- SIMD: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- MISD: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

Why Parallel Computers?

- Parallelism: Doing multiple things at a time
- Things: instructions, operations, tasks

- Main Goal
  - Improve performance (Execution time or task throughput)
    - Execution time of a program governed by Amdahl's Law

- Other Goals
  - Reduce power consumption
    - (4N units at freq F/4) consume less power than (N units at freq F)
    - Why?
  - Improve cost efficiency and scalability, reduce complexity
    - Harder to design a single unit that performs as well as N simpler units
  - Improve dependability: Redundant execution in space

Types of Parallelism and How to Exploit Them

- Instruction Level Parallelism
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow

- Data Parallelism
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors

- Task Level Parallelism
  - Different "tasks/threads" can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)

Task-Level Parallelism: Creating Tasks

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively
- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task

Multiprocessing Fundamentals

Multiprocessor Types

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization

Main Issues in Tightly-Coupled MP

- Shared memory synchronization
  - Locks, atomic operations
- Cache consistency
  - More commonly called cache coherence
- Ordering of memory operations
  - What should the programmer expect the hardware to provide?
- Resource sharing, contention, partitioning
- Communication: Interconnection networks
- Load imbalance

Aside: Hardware-based Multithreading

- Coarse grained
  - Quantum based
  - Event based (switch-on-event multithreading)

- Fine grained
  - Cycle by cycle
  - Thornton, "CDC 6600: Design of a Computer," 1970.
  - Burton Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

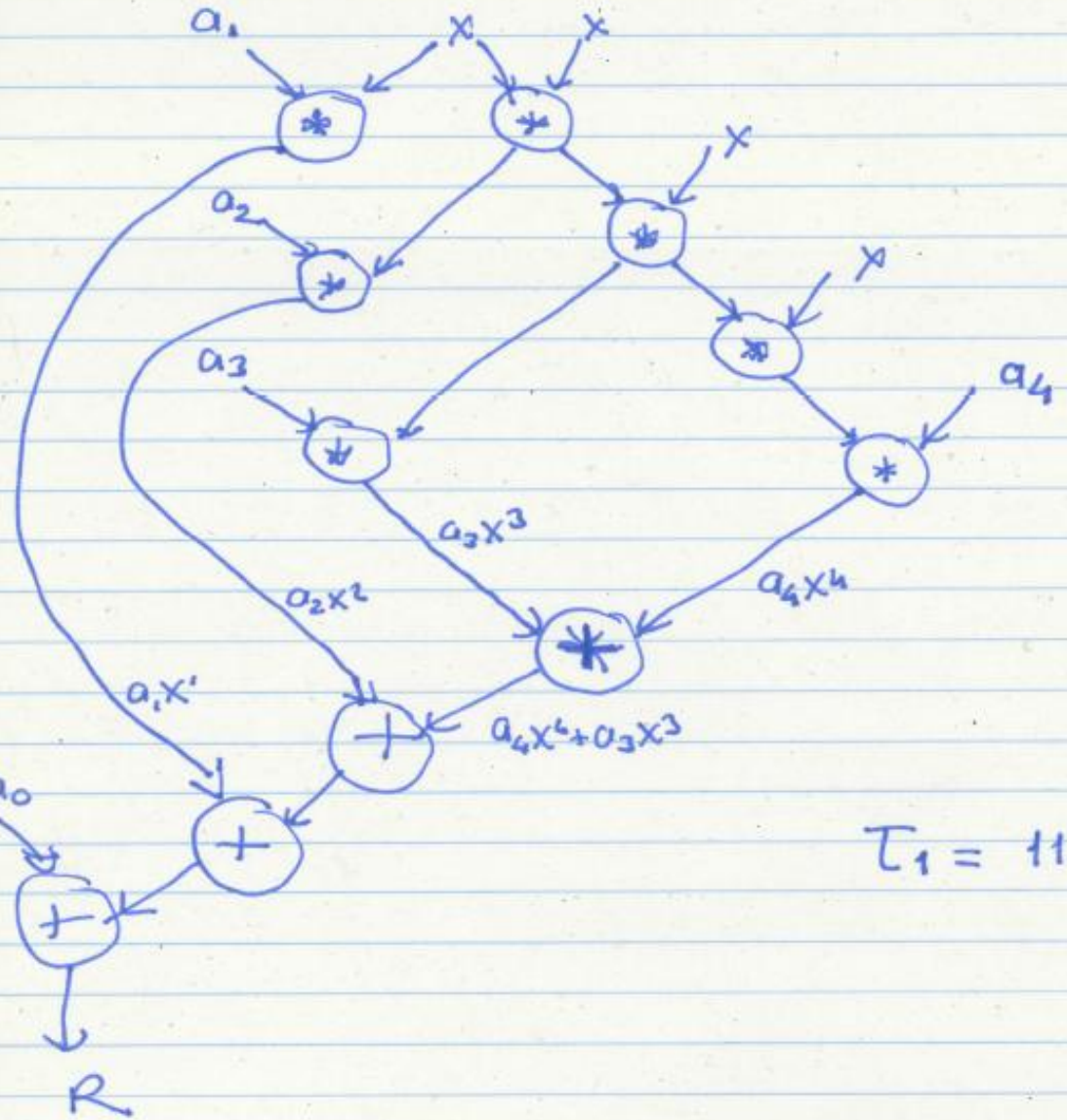
- Simultaneous
  - Can dispatch instructions from multiple threads at the same time
  - Good for improving execution unit utilization

Parallel Speedup Example

- $a4x^4 + a3x^3 + a2x^2 + a1x + a0$
- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor
- How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions
- How fast is this with 3 processors?

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

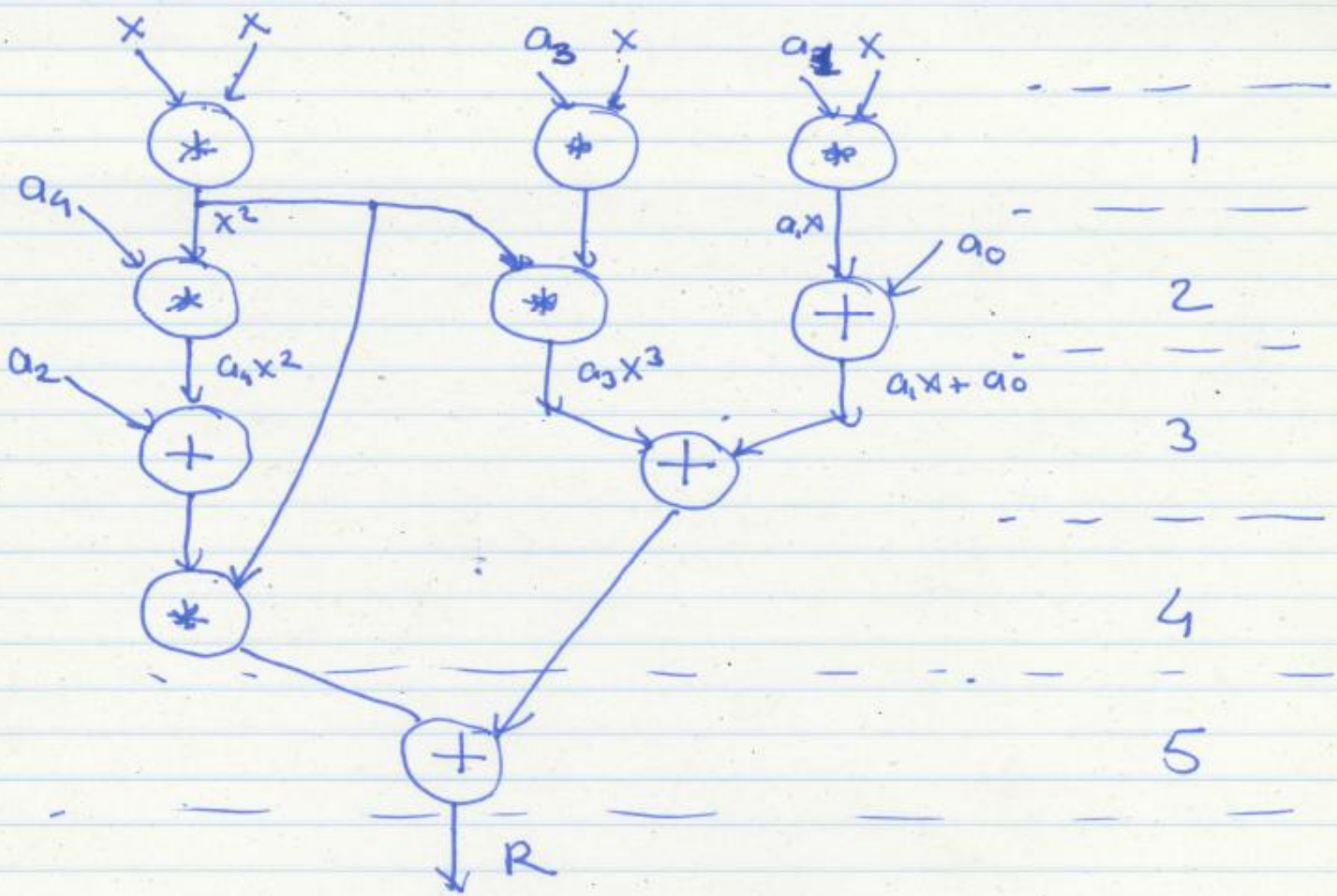
Single processor : 11 operations (DRAW the data flow graph)



$\tau_1 = 11$ cycles

$$R = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Three processors : τ_3 (exec. time with 3 proc.)



$\tau_3 = \underline{5 \text{ cycles}}$

Speedup with 3 Processors

$\tau_3 = \underline{5 \text{ cycles}}$

Speedup with 3 processors $= \dfrac{11}{5} = 2.2$

$\left(\dfrac{\tau_1}{\tau_3}\right)$

<u>Is this a fair comparison?</u>

Revisiting the Single-Processor Algorithm

Revisit τ_1

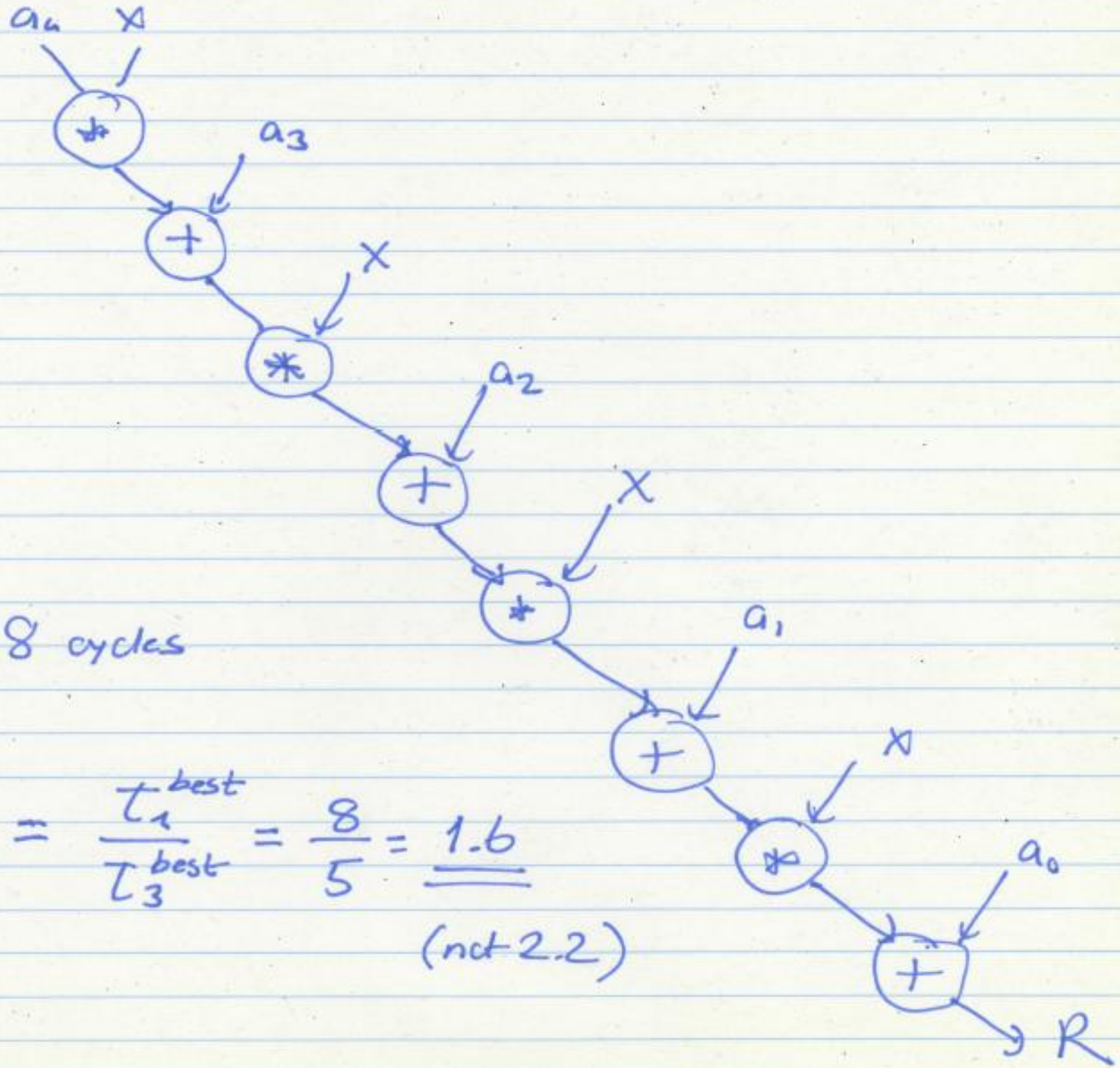
Better single-processor algorithm:

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$$R = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0$$

(Horner's method)

Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.



$\tau_1 = 8$ cycles

Speedup with 3 procs. $= \dfrac{\tau_1^{best}}{\tau_3^{best}} = \dfrac{8}{5} = \underline{\underline{1.6}}$

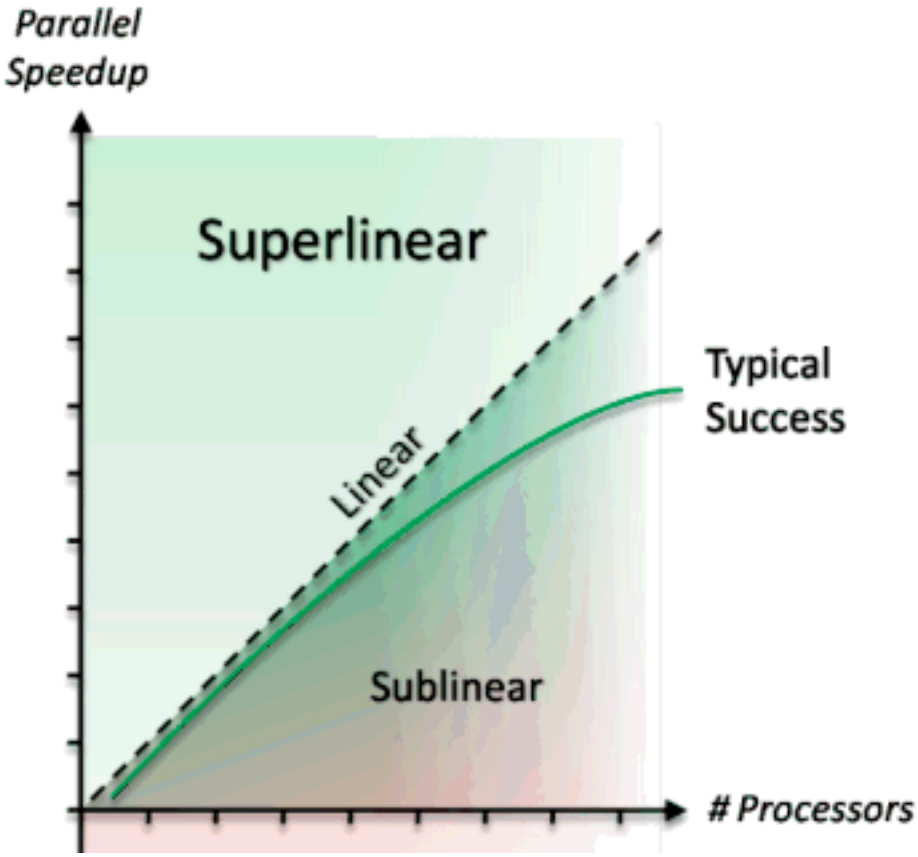
(not 2.2)

Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- Cache effects
- Working set effects

- Happens in two ways:
  - Unfair comparisons
  - Memory effects



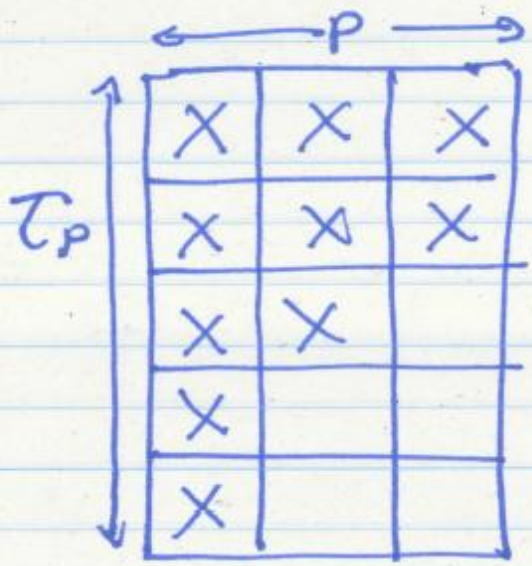
Utilization, Redundancy, Efficiency

- Traditional metrics
  - Assume all P processors are tied up for parallel computation
- Utilization: How much processing capability is used
  - U = (# Operations in parallel version) / (processors x Time)
- Redundancy: how much extra work is done with parallel processing
  - R = (# of operations in parallel version) / (# operations in best single processor algorithm version)
- Efficiency
  - E = (Time with 1 processor) / (processors x Time with P processors)
  - E = U/R

Utilization of a Multiprocessor

Multiprocessor metrics

Utilization : How much processing capability we use



$$U = \frac{\text{10 operations (in parallel version)}}{\text{3 processors} \times \text{5 time units}}$$

$$= \frac{10}{15}$$

$$U = \frac{\text{Ops with p proc.}}{p \times \tau_p}$$

Redundancy: How much extra work due to multiprocessing

$$R = \frac{\text{Ops with } p \text{ proc.}^{best}}{\text{Ops with 1 proc.}^{best}} = \frac{10}{8}$$

R is always ≥ 1

Efficiency: How much resource we use compared to how much resource we can get away with

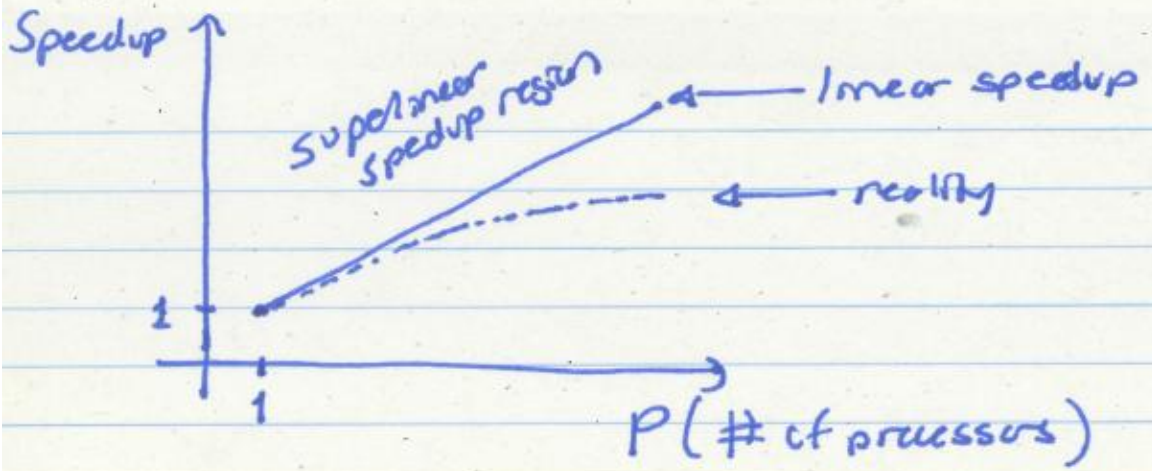
$$E = \frac{1 \cdot \tau_1^{best}}{p \cdot \tau_p^{best}}$$

(tying up 1 proc for τ_p time units)

(tying up p proc. for τ_p time units)

$$= \frac{8}{15} \qquad \left(E = \frac{U}{R} \right)$$

Caveats of Parallelism (I)



Why the reality? (diminishing returns)

$$\tau_p = \underbrace{\alpha \cdot \frac{\tau_1}{p}}_{\text{parallelizable part/fraction of the single-processor program}} + \underbrace{(1-\alpha) \cdot \tau_1}_{\text{non-parallelizable part}}$$

Amdahl's Law

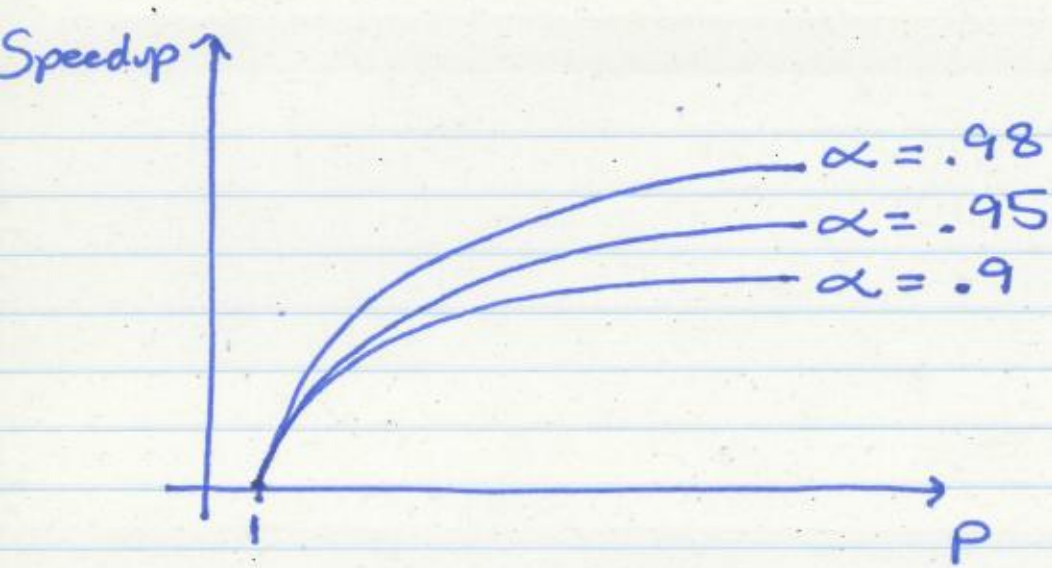
$$\text{Speedup with } p \text{ proc.} = \frac{\tau_1}{\tau_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}$$

$$\text{Speedup as } p \to \infty = \frac{1}{1-\alpha}$$

$\alpha \rightarrow$ bottleneck for parallel Speedup

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

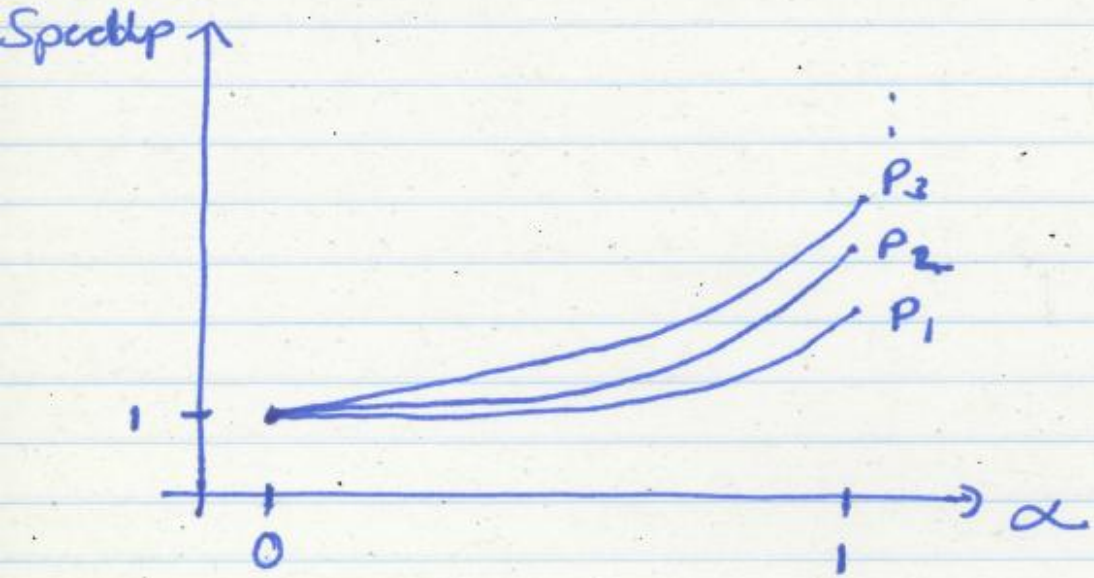
Amdahl's Law Implication 1



Amdahl's Law illustrated

Adding more and more processors gives less & less benefit if $\alpha < 1$

Amdahl's Law Implication 2



The benefit (speedup) is small until $\alpha \approx 1$

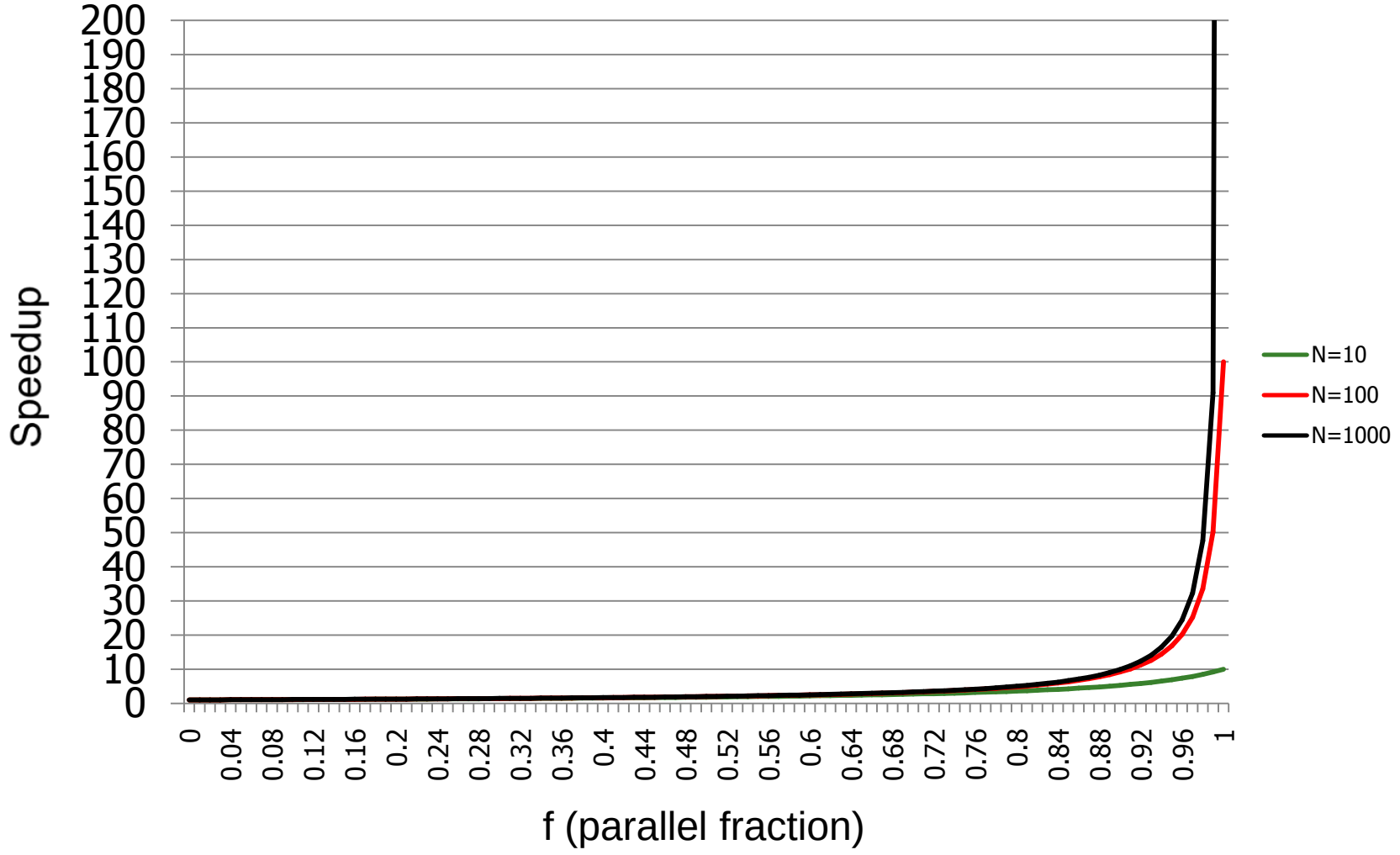
Caveats of Parallelism (II)

- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors

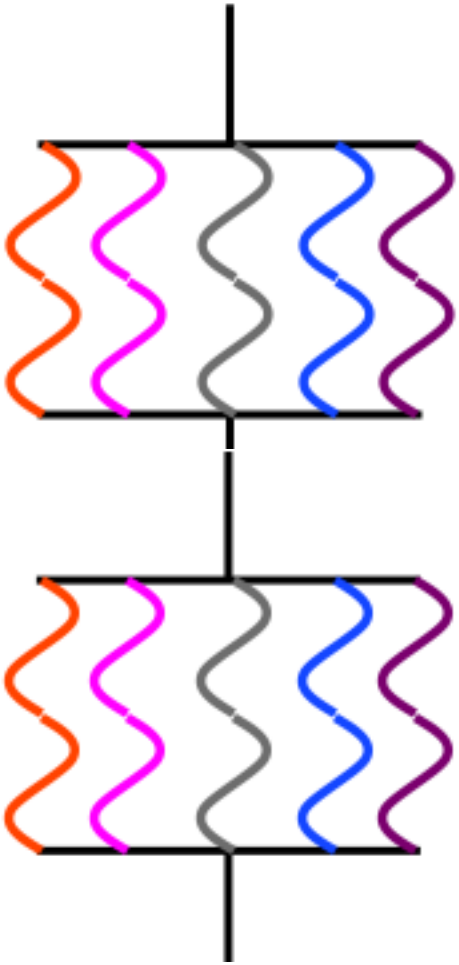
$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
- Maximum speedup limited by serial portion: Serial bottleneck
- Parallel portion is usually not perfectly parallel
  - Synchronization overhead (e.g., updates to shared data)
  - Load imbalance overhead (imperfect parallelization)
  - Resource sharing overhead (contention among N processors)

Sequential Bottleneck



Why the Sequential Bottleneck?



- Parallel machines have the sequential bottleneck
- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)

```
for ( i = 0 ; i < N; i++)
    A[i] = (A[i] + A[i-1]) / 2
```

- Single thread prepares data and spawns parallel tasks (usually sequential)

Another Example of Sequential Bottleneck

LEGEND

A,E: Amdahl's serial part
B: Parallel Portion
C1,C2: Critical Sections
D: Outside critical section

```
InitPriorityQueue(PQ);
SpawnThreads();                                   A

ForEach Thread:

    while (problem not solved)
        Lock (X)
            SubProblem = PQ.remove();             C1
        Unlock(X);
        Solve(SubProblem);
        If(problem solved) break;                 D1        B
        NewSubProblems = Partition(SubProblem);
        Lock(X)
            PQ.insert(NewSubProblems)             ●
        Unlock(X)
                                                  D2
        . . .

PrintSolution();                                  E
```

Bottlenecks in Parallel Portion

- Synchronization: Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - Communication: Tasks may need values from each other

  - Causes thread serialization when shared data is contended

- Load Imbalance: Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects

  - Reduces speedup in parallel portion

- Resource Contention: Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive

  - Additional latency not present when each task runs alone

Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - "Embarrassingly parallel" applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer's job easier in writing correct and high-performance parallel programs

Memory Ordering in Multiprocessors

Ordering of Operations

- Operations: A, B, C, D
  - In what order should the hardware execute (and report the results of) these operations?
- A contract between programmer and microarchitect
  - Specified by the ISA
- Preserving an "expected" (more accurately, "agreed upon") order simplifies programmer's life
  - Ease of debugging; ease of state recovery, exception handling
- Preserving an "expected" order usually makes the hardware designer's life difficult
  - Especially if the goal is to design a high performance processor: Load-store queues in out of order execution

Memory Ordering in a Single Processor

- Specified by the von Neumann model
- Sequential order
  - Hardware executes the load and store operations in the order specified by the sequential program
- Out-of-order execution does not change the semantics
  - Hardware retires (reports to software the results of) the load and store operations in the order specified by the sequential program
- Advantages: 1) Architectural state is precise within an execution. 2) Architectural state is consistent across different runs of the program → Easier to debug programs
- Disadvantage: Preserving order adds overhead, reduces performance

Memory Ordering in a Dataflow Processor

- A memory operation executes when its operands are ready
- Ordering specified only by data dependencies
- Two operations can be executed and retired in any order if they have no dependency
- Advantage: Lots of parallelism → high performance
- Disadvantage: Order can change across runs of the same program → Very hard to debug

Memory Ordering in a MIMD Processor

- Each processor's memory operations are in sequential order with respect to the "thread" running on that processor (assume each processor obeys the von Neumann model)
- Multiple processors execute memory operations concurrently
- How does the memory see the order of operations from all processors?
  - In other words, what is the ordering of operations across different processors?

Why Does This Even Matter?

- Ease of debugging
  - It is nice to have the same execution done at different times have the same order of execution
- Correctness
  - Can we have incorrect execution if the order of memory operations is different from the point of view of different processors?
- Performance and overhead
  - Enforcing a strict "sequential ordering" can make life harder for the hardware designer in implementing performance enhancement techniques (e.g., OoO execution, caches)

Protecting Shared Data

- Threads are not allowed to update shared data concurrently
  - For correctness purposes
- Accesses to shared data are encapsulated inside *critical sections or protected via synchronization constructs (locks, semaphores, condition variables)*
- Only one thread can execute a critical section at a given time
  - Mutual exclusion principle
- A multiprocessor should provide the *correct* execution of synchronization primitives to enable the programmer to protect shared data

Supporting Mutual Exclusion

- Programmer needs to make sure mutual exclusion (synchronization) is correctly implemented
  - We will assume this
  - But, correct parallel programming is an important topic
  - Reading: Dijkstra, "Cooperating Sequential Processes," 1965.
    - http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html
    - See Dekker's algorithm for mutual exclusion

- Programmer relies on hardware primitives to support correct synchronization
- If hardware primitives are not correct (or unpredictable), programmer's life is tough
- If hardware primitives are correct but not easy to reason about or use, programmer's life is still tough

Protecting Shared Data

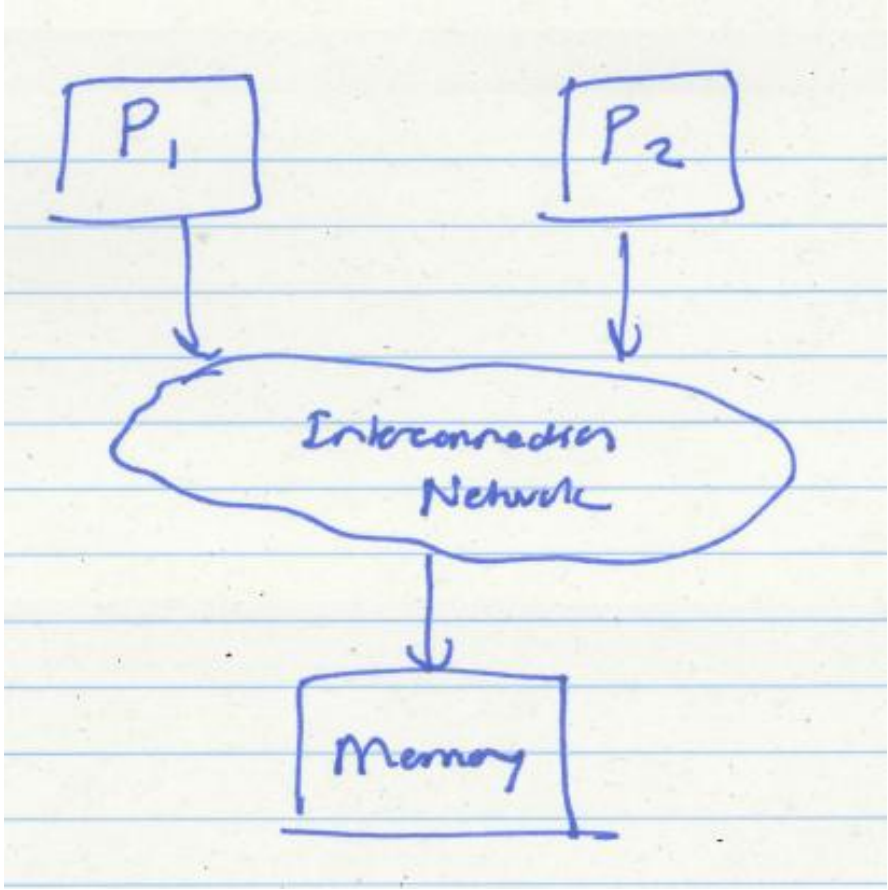
P_1	P_2
$F_1 = \phi$	$F_2 = \phi$
A $F_1 = 1$	X $F_2 = 1$
B IF ($F_2 == \phi$) THEN	Y IF ($F_1 == \phi$) THEN
{Critical section}	{Critical section}
$F_1 \neq \phi$	$F2 = \phi$
ELSE	ELSE
{...}	{....}

Only P_1 or P_2 should be in this section at any given time, not both

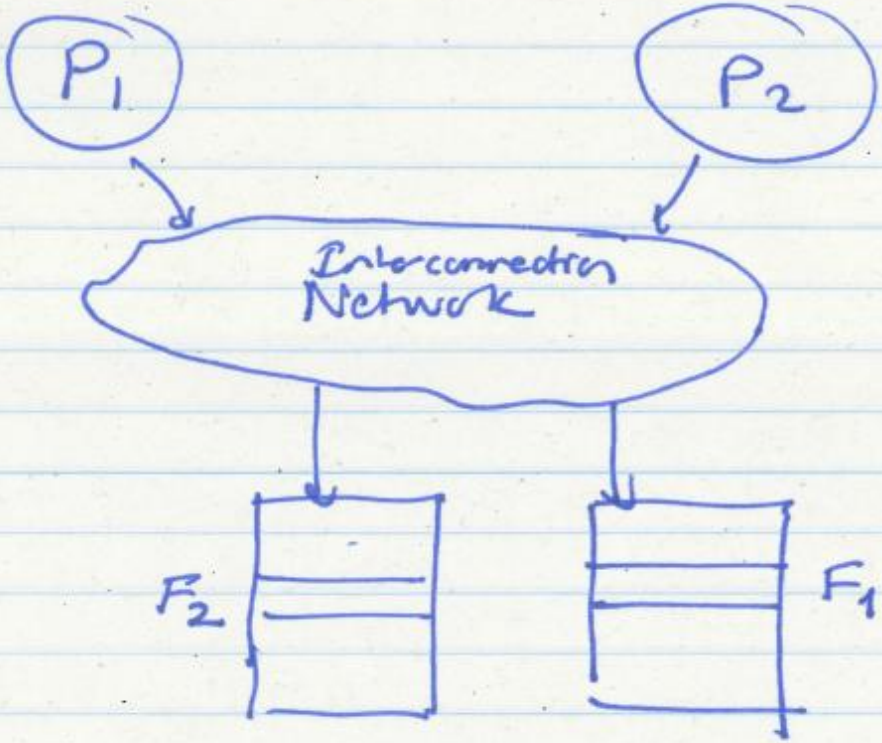
Assume P1 is in critical section.
Intuitively, it must have executed A,
which means F1 must be 1 (as A happens before B),
which means P2 should not enter the critical section.

A Question

- Can the two processors be in the critical section at the same time given that they both obey the von Neumann model?
- Answer: yes



An Incorrect Result (due to an implementation that does not provide sequential consistency)



time 0: P_1 executes A (set $F_1 = 1$) st F_1 complete (from P_1's view)
A is sent to memory

P_2 executes X (set $F_2 = 1$) st F_2 complete (from P_2's view)
X is sent to memory

Both Processors in Critical Section

time 0:	P_1 executes A (set $F_1 = 1$) st F_1 complete (from P_1's view) A is sent to memory	P_2 executes X (set $F_2 = 1$) st F_2 complete (from P_2's view) X is sent to memory
time 1:	P_1 executes B (test $F_2 == 0$) ld F_2 started B is sent to memory	P_2 executes Y (test $F_1 == 0$) ld F_1 started Y is sent to memory
time 50:	Memory sends back to P_1 F_2 (0) ld F_2 complete	Memory sends back to P_2 (F_1 (0) ld F_1 complete
time 51:	P_1 is in critical section ~~execute~~	P_2 is in critical section
time 100:	Memory completes A $F_1 = 1$ in memory (too late!)	Memory completes X $F_2 = 1$ in memory (too late!)

What happened?

P_1's view of mem. ops		P_2's view	
A	($F_1=1$)	X	($F_2=1$)
B	(test $F_2=0$)	Y	(test$_2$ $F_1=0$)
X	($F_2=1$)	A	($F_1=1$)

B executed before X

Y executed before A

Problem!

These two processors did not see the same order of operations in memory

How Can We Solve The Problem?

- Idea: Sequential consistency

- All processors see the same order of operations to memory
- i.e., all memory operations happen in an order (called the global total order) that is consistent across all processors

- Assumption: within this global order, each processor's operations appear in sequential order with respect to its own operations.

Sequential Consistency

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- A multiprocessor system is sequentially consistent if:
  - the result of any execution is the same as if the operations of all the processors were executed in some sequential order

AND

  - the operations of each individual processor appear in this sequence in the order specified by its program

- This is a memory ordering model, or memory model
  - Specified by the ISA

Programmer's Abstraction

- Memory is a switch that services one load or store at a time form any processor
- All processors see the currently serviced load or store at the same time
- Each processor's operations are serviced in program order

Sequentially Consistent Operation Orders

- Potential correct global orders (all are correct):

- A B X Y
- A X B Y
- A X Y B
- X A B Y
- X A Y B
- X Y A B

- Which order (interleaving) is observed depends on implementation and dynamic latencies

Consequences of Sequential Consistency

- Corollaries

1. Within the same execution, all processors see the same global order of operations to memory
   → No correctness issue
   → Satisfies the "happened before" intuition

2. Across different executions, different global orders can be observed (each of which is sequentially consistent)
   → Debugging is still difficult (as order changes across runs)

Issues with Sequential Consistency?

- Nice abstraction for programming, but two issues:
  - Too conservative ordering requirements
  - Limits the aggressiveness of performance enhancement techniques

- Is the total global order requirement too strong?
  - Do we need a global order across all operations and all processors?
  - How about a global order only across all stores?
    - Total store order memory model; unique store order model
  - How about a enforcing a global order only at the boundaries of synchronization?
    - Relaxed memory models
    - Acquire-release consistency model

Issues with Sequential Consistency?

- Performance enhancement techniques that could make SC implementation difficult
- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores
- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors

Weaker Memory Consistency

- The ordering of operations is important when the order affects operations on shared data → i.e., when processors need to synchronize to execute a "program region"

- Weak consistency
  - Idea: Programmer specifies regions in which memory operations do not need to be ordered
  - "Memory fence" instructions delineate those regions
    - All memory operations before a fence must complete before fence is executed
    - All memory operations after the fence must wait for the fence to complete
    - Fences complete in program order
  - All synchronization operations act like a fence

Tradeoffs: Weaker Consistency

- Advantage
  - No need to guarantee a very strict order of memory operations
    - → Enables the hardware implementation of performance enhancement techniques to be simpler
    - → Can be higher performance than stricter ordering
- Disadvantage
  - More burden on the programmer or software (need to get the "fences" correct)
- Another example of the programmer-microarchitect tradeoff

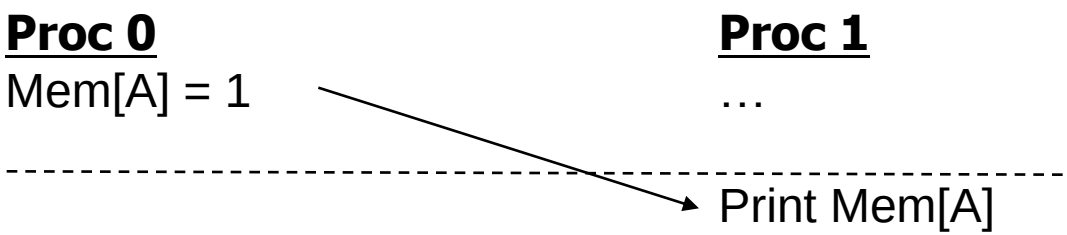
Issues with Sequential Consistency?

- Performance enhancement techniques that could make SC implementation difficult
- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores
- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors

Cache Coherence

Shared Memory Model

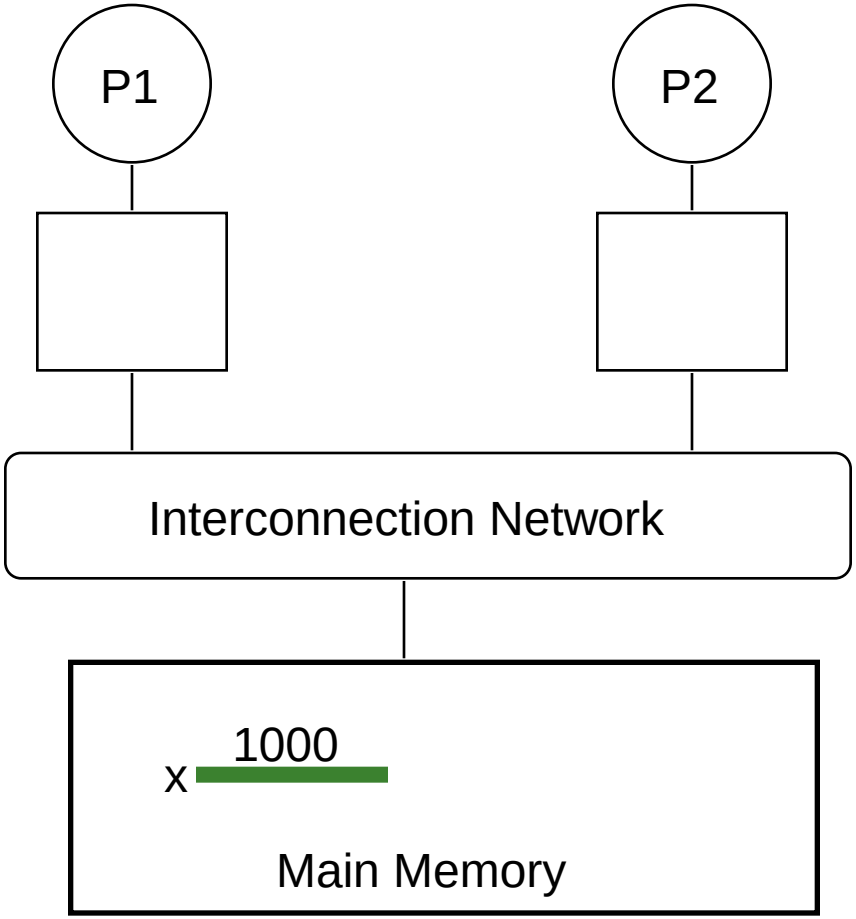
- Many parallel programs communicate through *shared memory*
- Proc 0 writes to an address, followed by Proc 1 reading
  - This implies communication between the two



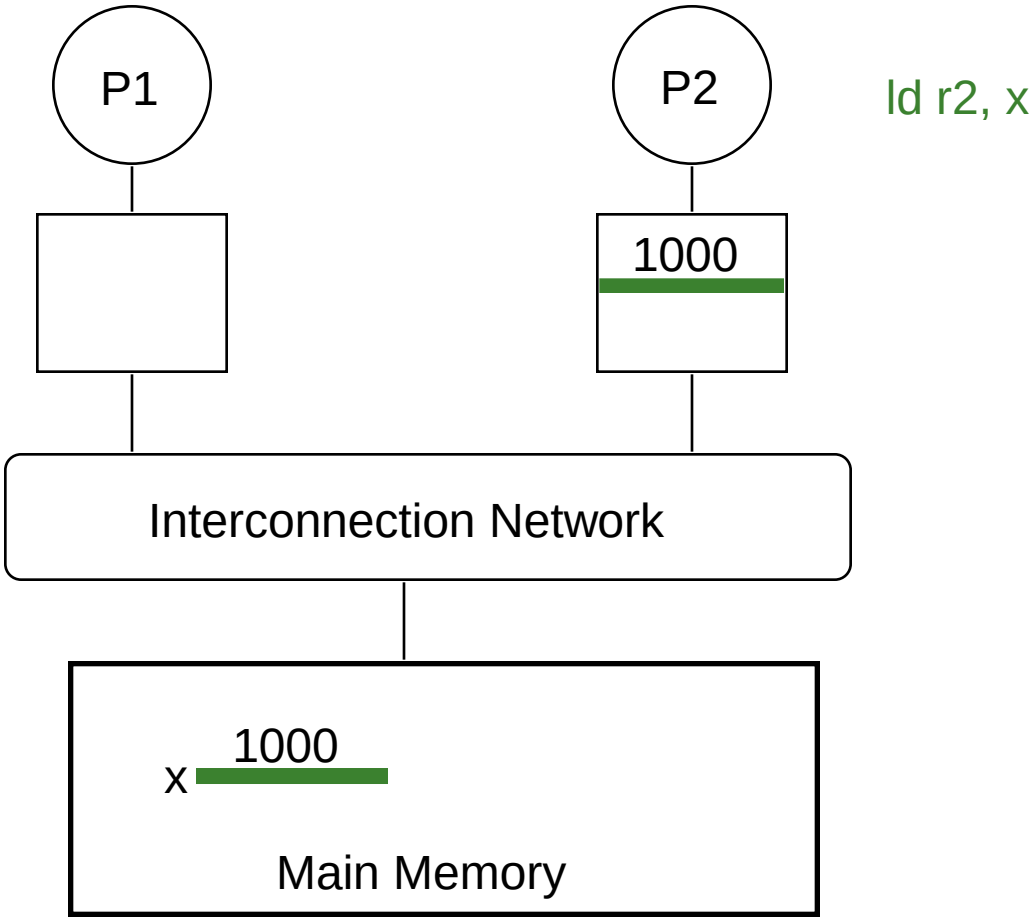
- Each read should receive the value last written by anyone
  - This requires synchronization (what does last written mean?)
- What if Mem[A] is cached (at either end)?

Cache Coherence

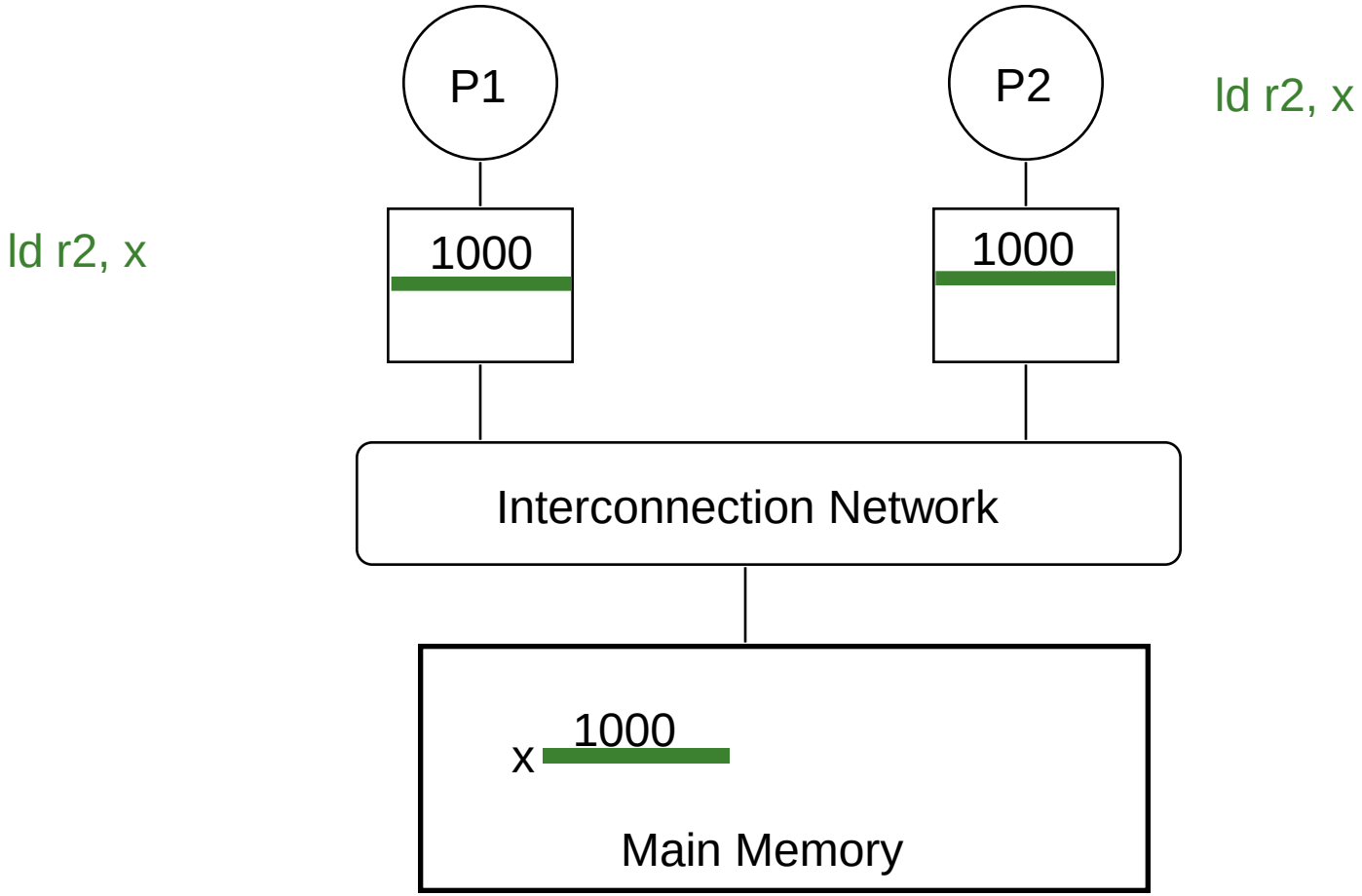
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



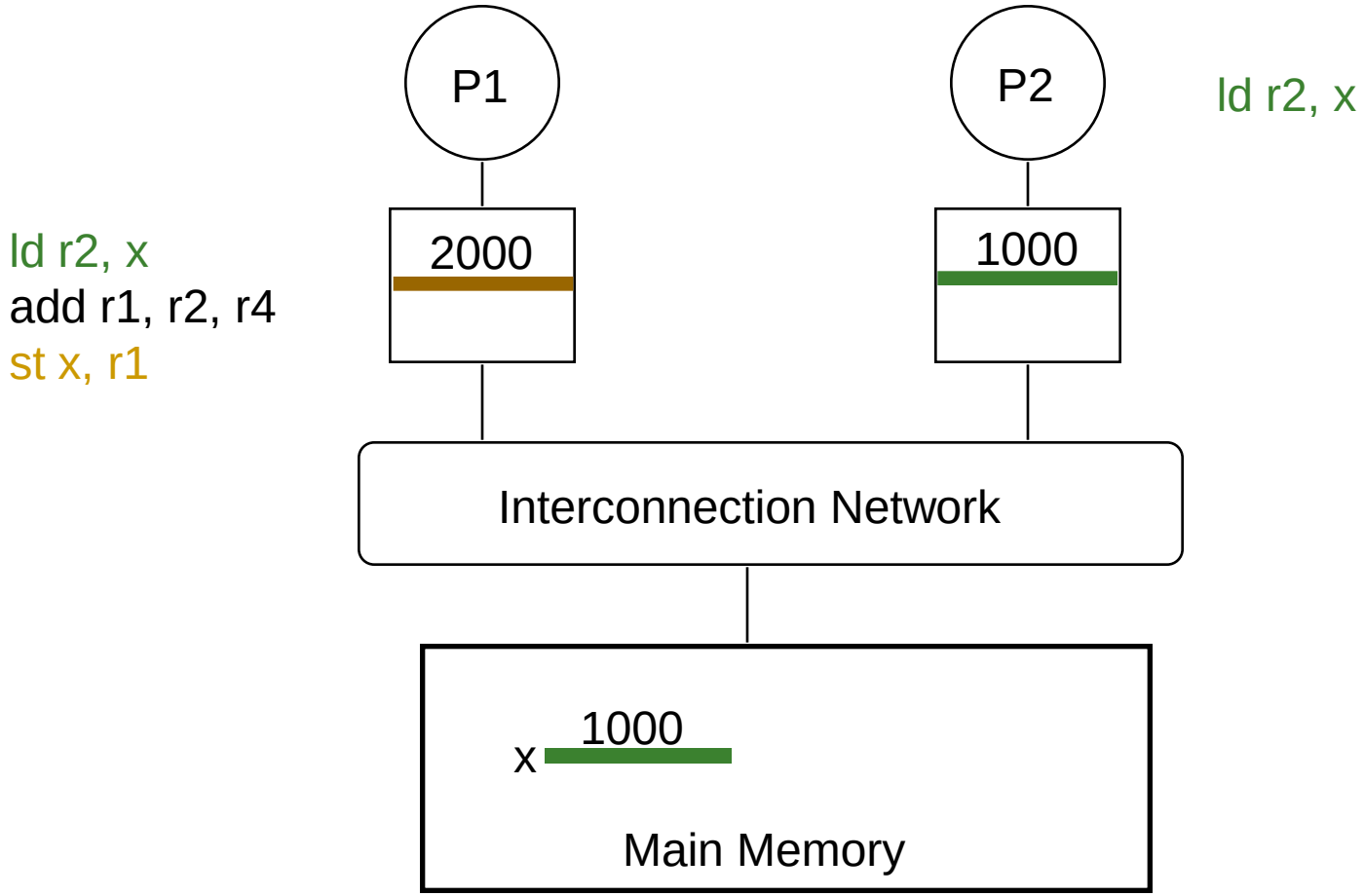
The Cache Coherence Problem



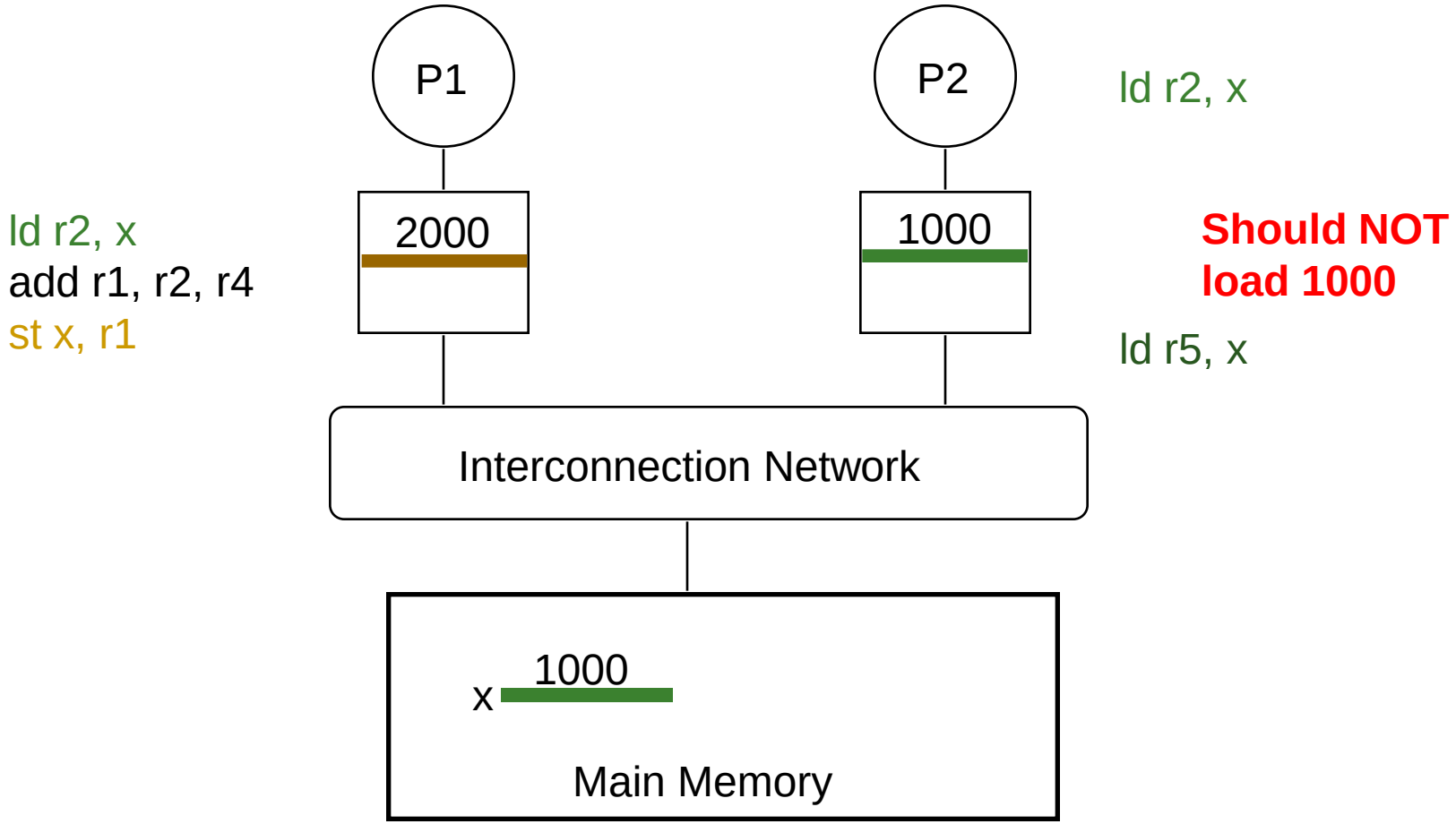
The Cache Coherence Problem



The Cache Coherence Problem



The Cache Coherence Problem



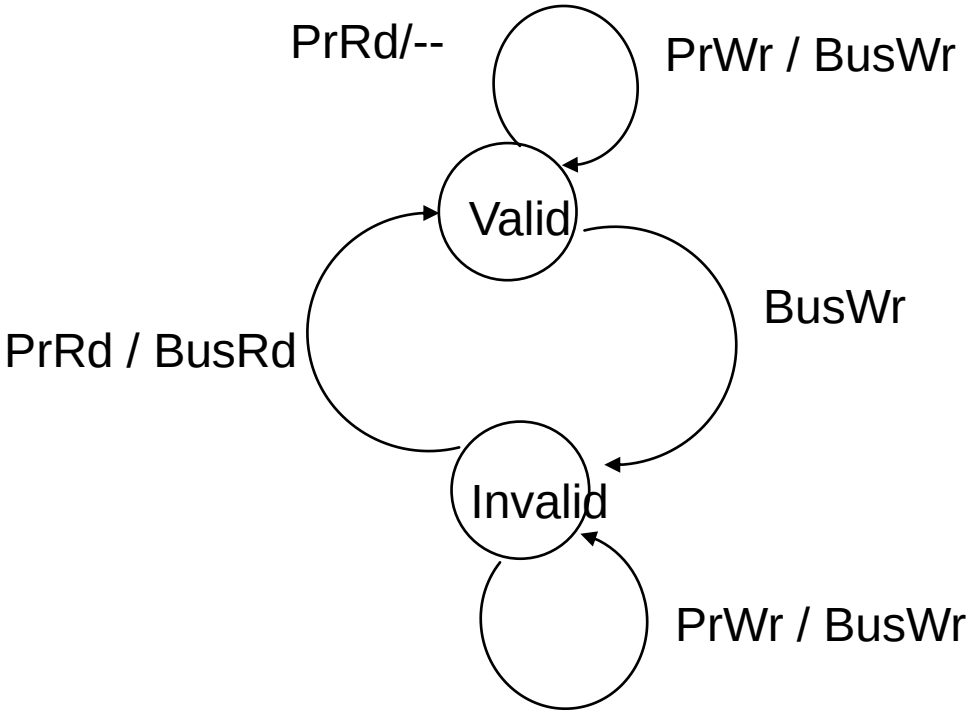
Cache Coherence: Whose Responsibility?

- Software
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided a cache flush instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
    - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.

- Hardware
  - Simplifies software's job
  - One idea: Invalidate all other copies of block A when a processor writes to it

A Very Simple Coherence Scheme

- Caches "snoop" (observe) each other's write/read operations. If a processor writes to a block, all others invalidate it from their caches.
- A simple protocol:



- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr

(Non-)Solutions to Cache Coherence

- No hardware based coherence
  - Keeping caches coherent is software's responsibility

  + Makes microarchitect's life easier

  -- Makes average programmer's life much harder
    - need to worry about hardware caches to maintain program correctness?

  -- Overhead in ensuring coherence in software

- All caches are shared between all processors

  + No need for coherence

  -- Shared cache becomes the bandwidth bottleneck

  -- Very hard to design a scalable system with low-latency cache access this way

Maintaining Coherence

- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location

- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order

- Coherence needs to provide:
  - **Write propagation**: guarantee that updates will propagate
  - **Write serialization**: provide a consistent global order seen by all processors

- Need a global point of serialization for this store ordering

Hardware Cache Coherence

- Basic idea:
  - A processor/cache broadcasts its write/update to a memory location to all other processors
  - Another cache that has the location either updates or invalidates its local copy

Coherence: Update vs. Invalidate

- How can we *safely update replicated data?*
  - Option 1 (Update protocol): push an update to all copies
  - Option 2 (Invalidate protocol): ensure there is only one copy (local), update it

- **On a Read:**
  - If local copy isn't valid, put out request
  - (If another node has a copy, it returns it, otherwise memory does)

Coherence: Update vs. Invalidate (II)

- **On a Write:**
  - Read block into cache as before

Update Protocol:

- Write to block, and simultaneously broadcast written data to sharers
- (Other nodes update their caches if data was present)

Invalidate Protocol:

- Write to block, and simultaneously broadcast invalidation of address to sharers
- (Other nodes clear block from cache)

Update vs. Invalidate Tradeoffs

- Which do we want?
  - Write frequency and sharing behavior are critical
- **Update**
  - \+ If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reacquire (broadcast update pattern)
  - – If data is rewritten without intervening reads by other cores, updates were useless
  - – Write-through cache policy ➔ bus becomes bottleneck
- **Invalidate**
  - \+ After invalidation broadcast, core has exclusive access rights
  - \+ Only cores that keep reading after each write retain a copy
  - – If write contention is high, leads to ping-ponging (rapid mutual invalidation-reacquire)

Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, single point of serialization for all requests
  - Processors observe other processors' actions
    - E.g.: P1 makes "read-exclusive" request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - Single point of serialization *per block*, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks ownership (sharer set) for each block
  - Directory coordinates invalidation appropriately
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1

Directory Based Cache Coherence

Directory Based Coherence

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- An example mechanism:
  - For each cache block in memory, store P+1 bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that a cache has the only copy of the block and can update it without notifying others
  - On a read: set the cache's bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an "exclusive bit" associated with each block in each cache

Directory Based Coherence Example (I)

Example directory based scheme

P=4

P+1 bits: for block A

0	0	0	0	0

Exclusive bit

No cache has the block

① P_1 takes a read miss to block A

0	0	0	0	0

→

0	1	0	0	0

② P_3 takes a read miss

↓

0	1	0	1	0

③ P_2 takes a write miss

→ invalidate P_1 & P_3's caches

→ write request → P_2 has the <u>exclusive copy</u> of the block now. Set the Exclusive bit

0	0	1	0	1

→ P_2 can now update the block without notifying any other processor or the directory

→ P_2 needs to have a bit in its cache indicating it can perform exclusive updates to that block

→ <u>private/exclusive bit per cache block</u>

④ P_3 takes a write miss

→ Mem ~~Ctrller~~ Controller requests ~~the tr~~ block from P_2

→ Mem Controller gives block to P_3

→ P2 invalidates its copy

0	0	0	1	1

⑤ P_2 takes a read miss

→ P_3 supplies it

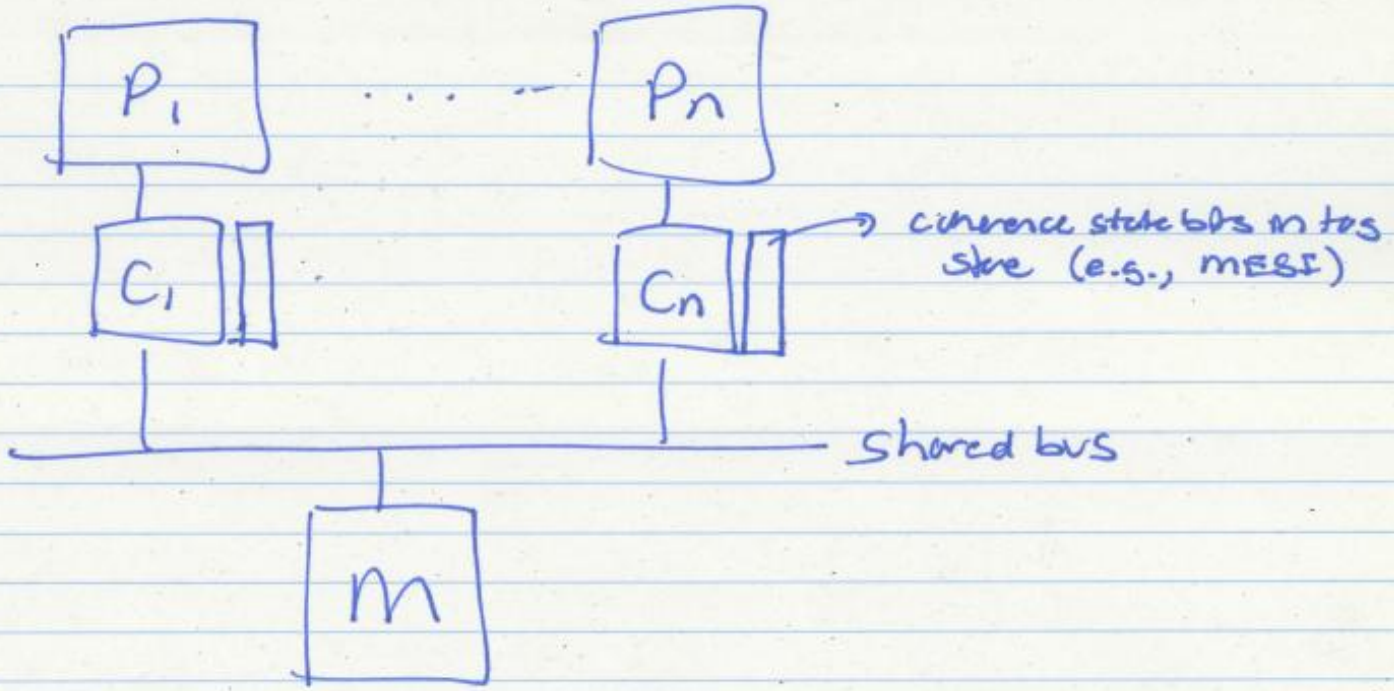
0	0	1	1	0

Snoopy Cache Coherence

Snoopy Cache Coherence

- Idea:
  - All caches "snoop" all other caches' read/write requests and keep the cache block coherent
  - Each cache block has "coherence metadata" associated with it in the tag store of each cache

- Easy to implement if all caches share a common bus
  - Each cache broadcasts its read/write operations on the bus
  - Good for small-scale multiprocessors
  - What if you would like to have a 1000-node multiprocessor?



SNOOPY CACHE

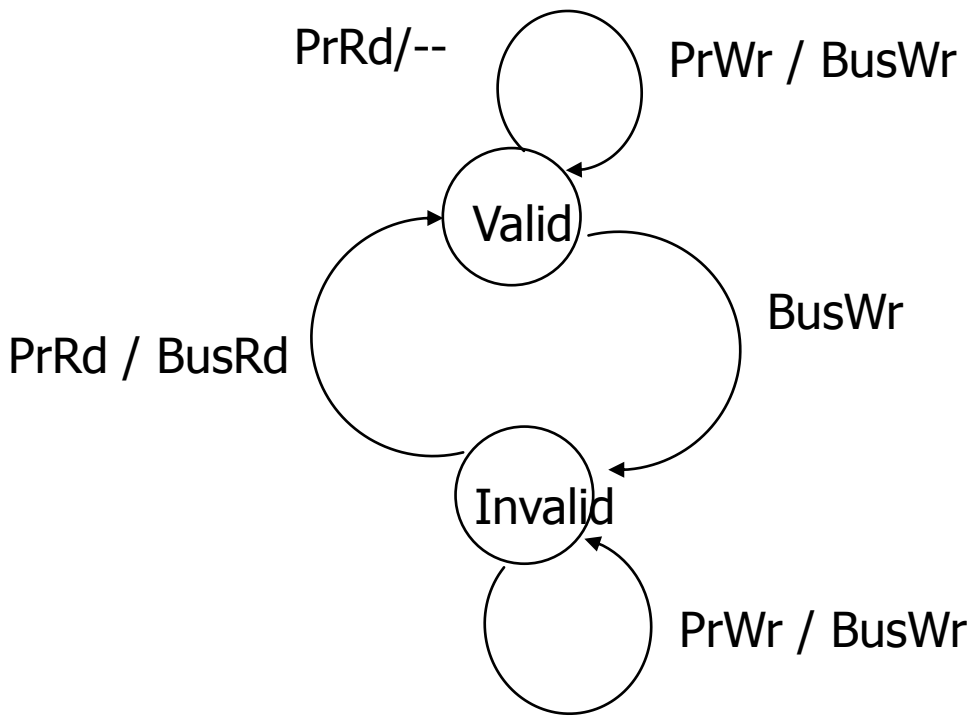
Each Cache observes its own processor & the bus
- Changes the state of the cached block based on observed actions by processor & the bus

Processor actions to a block: PR (Proc. Read)
PW (Proc. Write)

Bus actions to a block (comms from another processor): BR (Bus Read)
BW (Bus Write)
or BRx (Bus Read Exclusive)

A Simple Snoopy Cache Coherence Protocol

- Caches "snoop" (observe) each other's write/read operations
- A simple protocol:



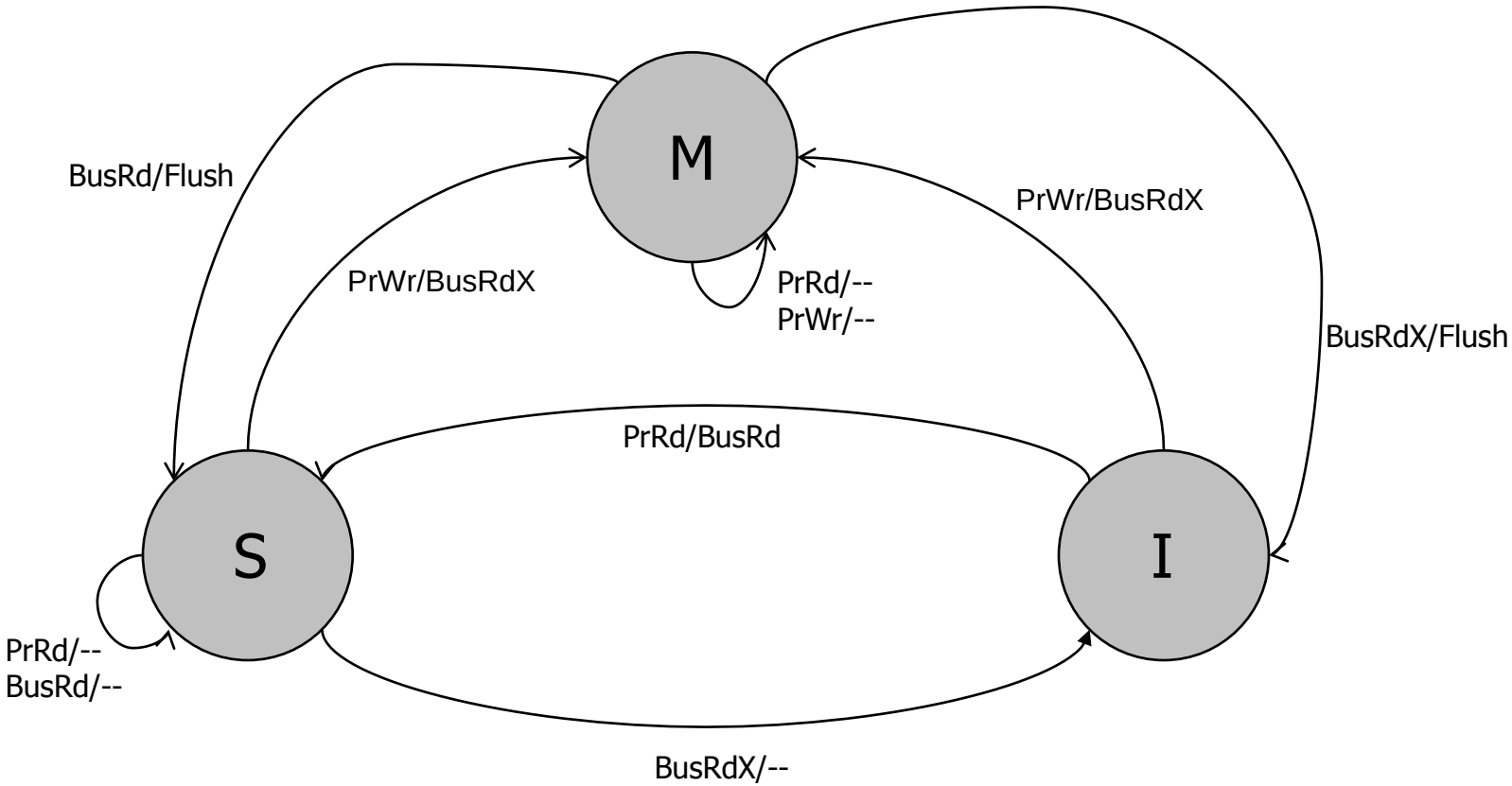
- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr

A More Sophisticated Protocol: MSI

- Extend single valid bit per block to three states:
  - **M**(odified): cache line is only copy and is dirty
  - **S**(hared): cache line is one of several copies
  - **I**(nvalid): not present

- Read miss makes a *Read* request on bus, transitions to **S**
- Write miss makes a *ReadEx* request, transitions to **M** state
- When a processor snoops *ReadEx* from another writer, it must invalidate its own copy (if any)
- S→M upgrade can be made without re-reading data from memory (via *Invalidations)*

MSI State Machine



ObservedEvent/Action

[Culler/Singh96]

The Problem with MSI

- A block is in no cache to begin with
- Problem: On a read, the block immediately goes to "Shared" state although it may be the only copy to be cached (i.e., no other processor will cache it)

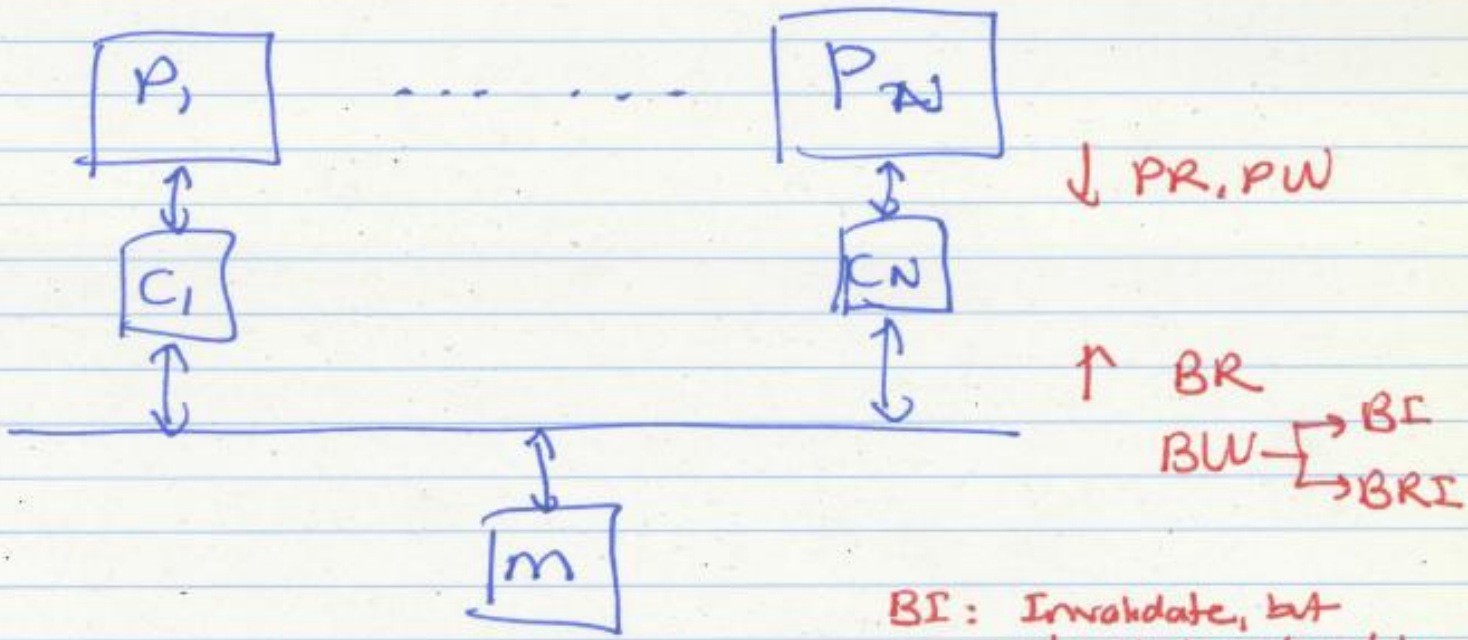
- Why is this a problem?
  - Suppose the cache that read the block wants to write to it at some point
  - It needs to broadcast "invalidate" even though it has the only cached copy!
  - If the cache knew it had the only cached copy in the system, it could have written to the block without notifying any other cache → saves unnecessary broadcasts of invalidations

The Solution: MESI

- Idea: Add another state indicating that this is the only cached copy and it is clean.
  - *Exclusive* state
- Block is placed into the *exclusive* state if, during *BusRd*, no other cache had it
  - Wired-OR "shared" signal on bus can determine this: snooping caches assert the signal if they also have a copy
- Silent transition *Exclusive→Modified* is possible on write!
- MESI is also called the *Illinois protocol* [Papamarcos and Patel, ISCA 1984]

Papamarcos & Patel, ISCA 1984

Illinois Protocol



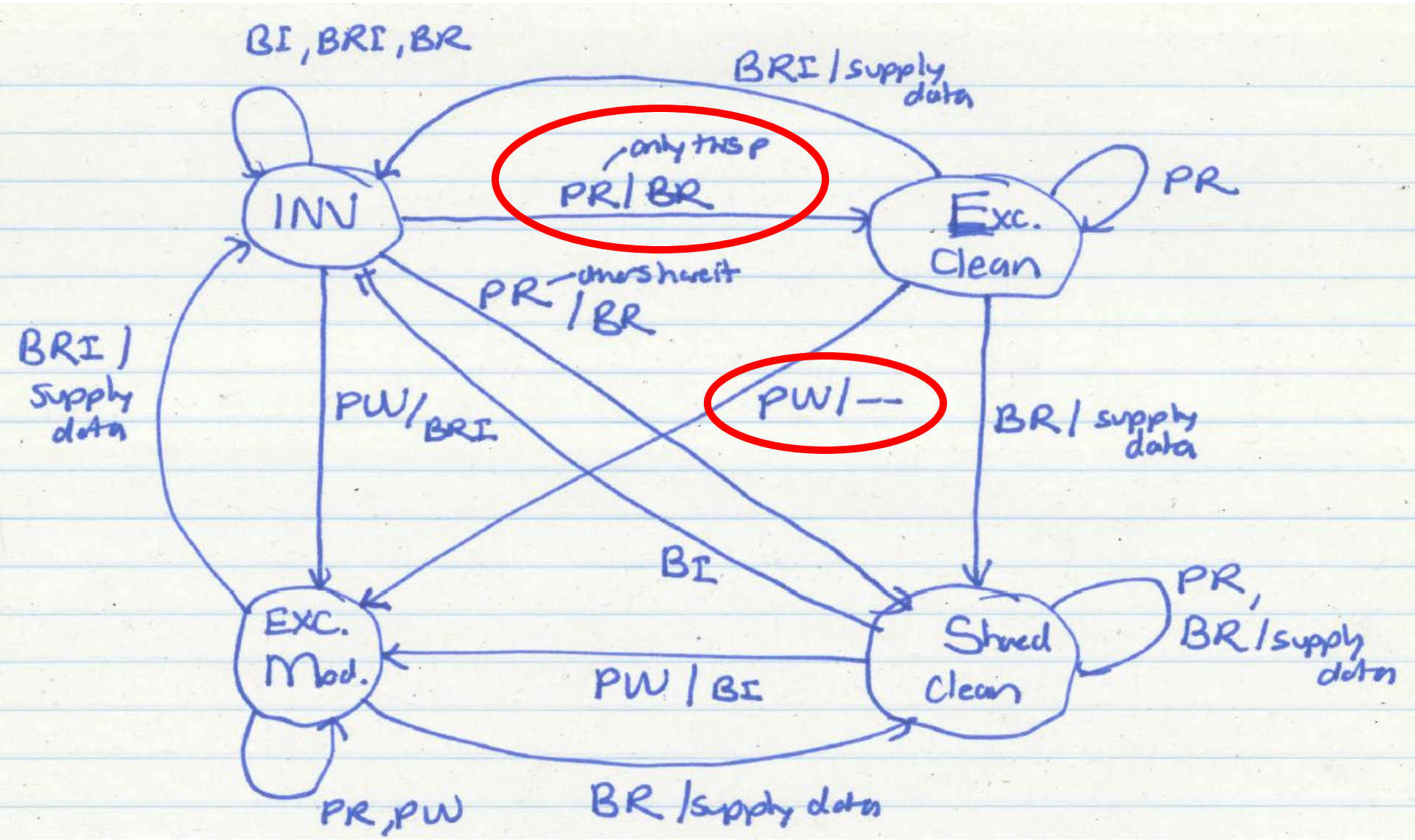
BI: Invalidate, but already have the data (do not supply it)

BRI: Invalidate, but also need the data (supply it)

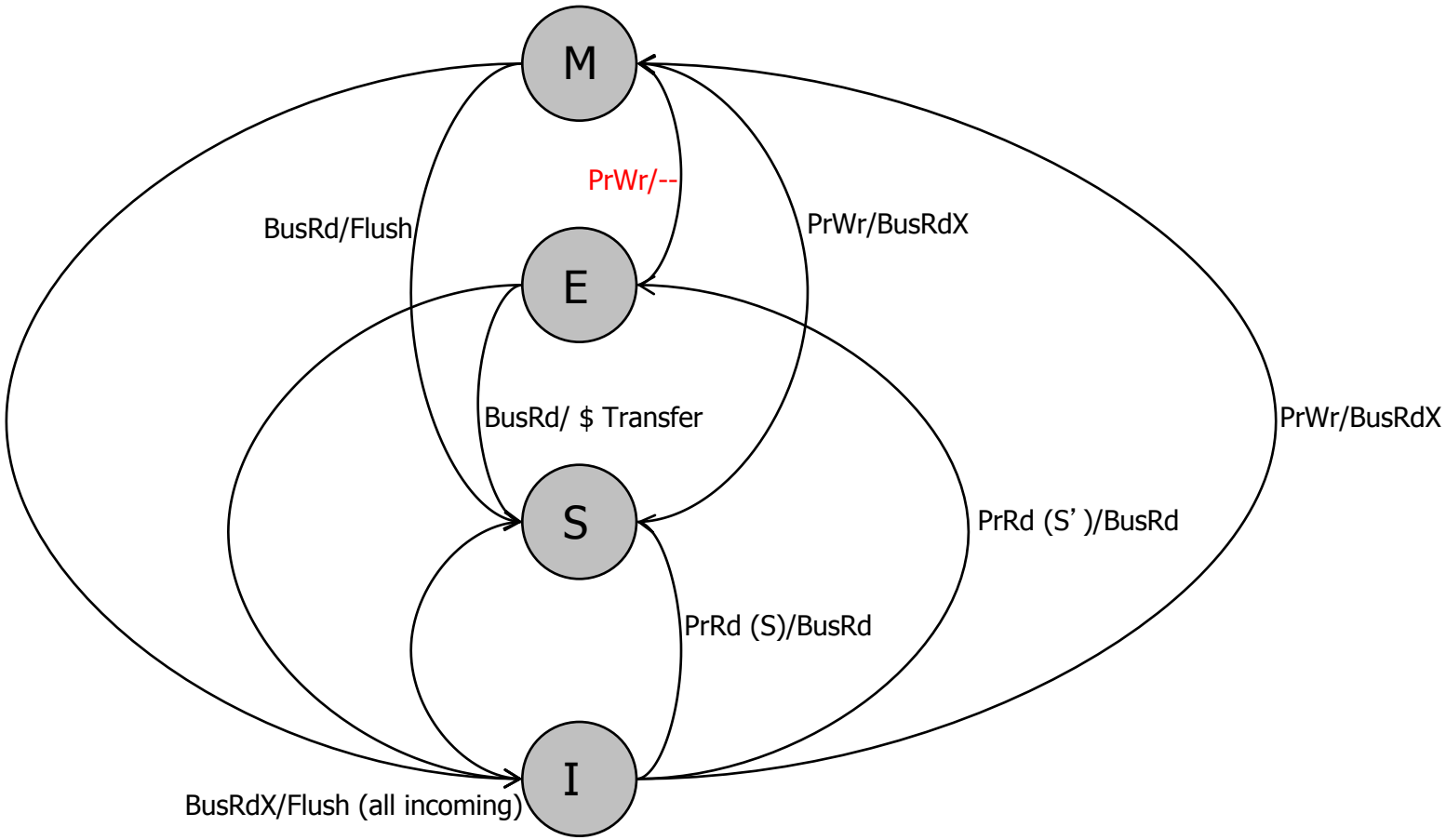
4 States

M: Modified (Exclusive copy, modified)
E: Exclusive (" " , clean)
S: Shared (Shared copy, clean)
I: Invalid

MESI State Machine

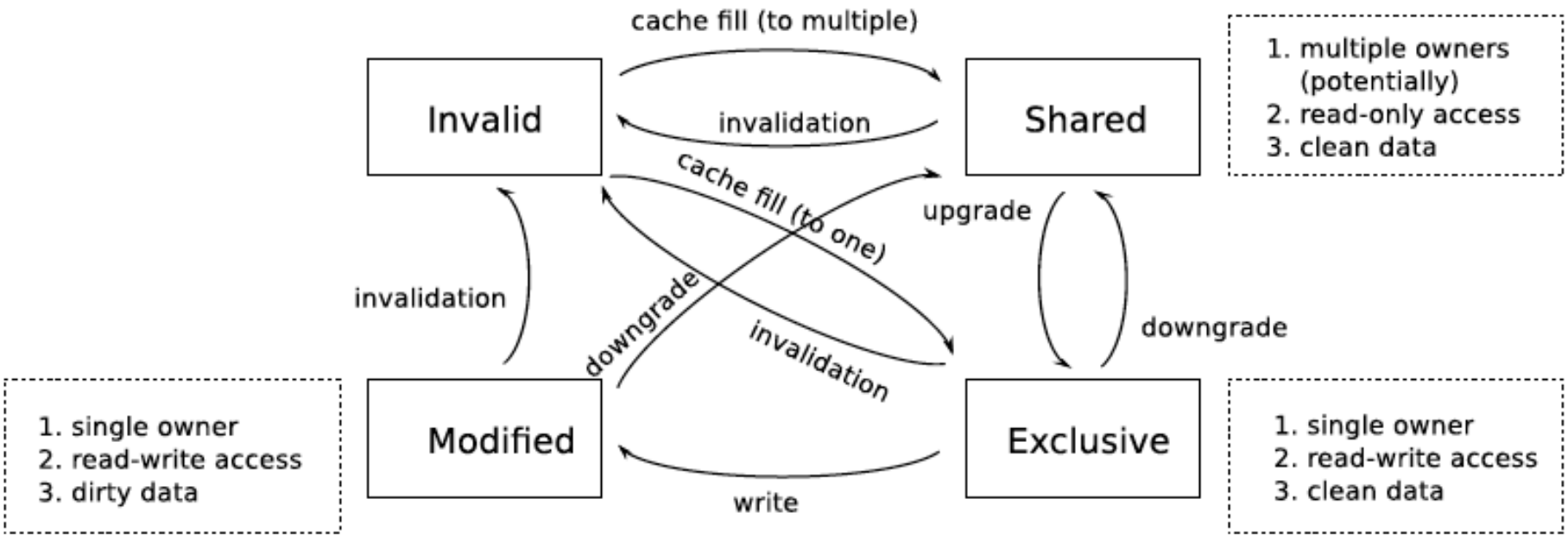


MESI State Machine



[Culler/Singh96]

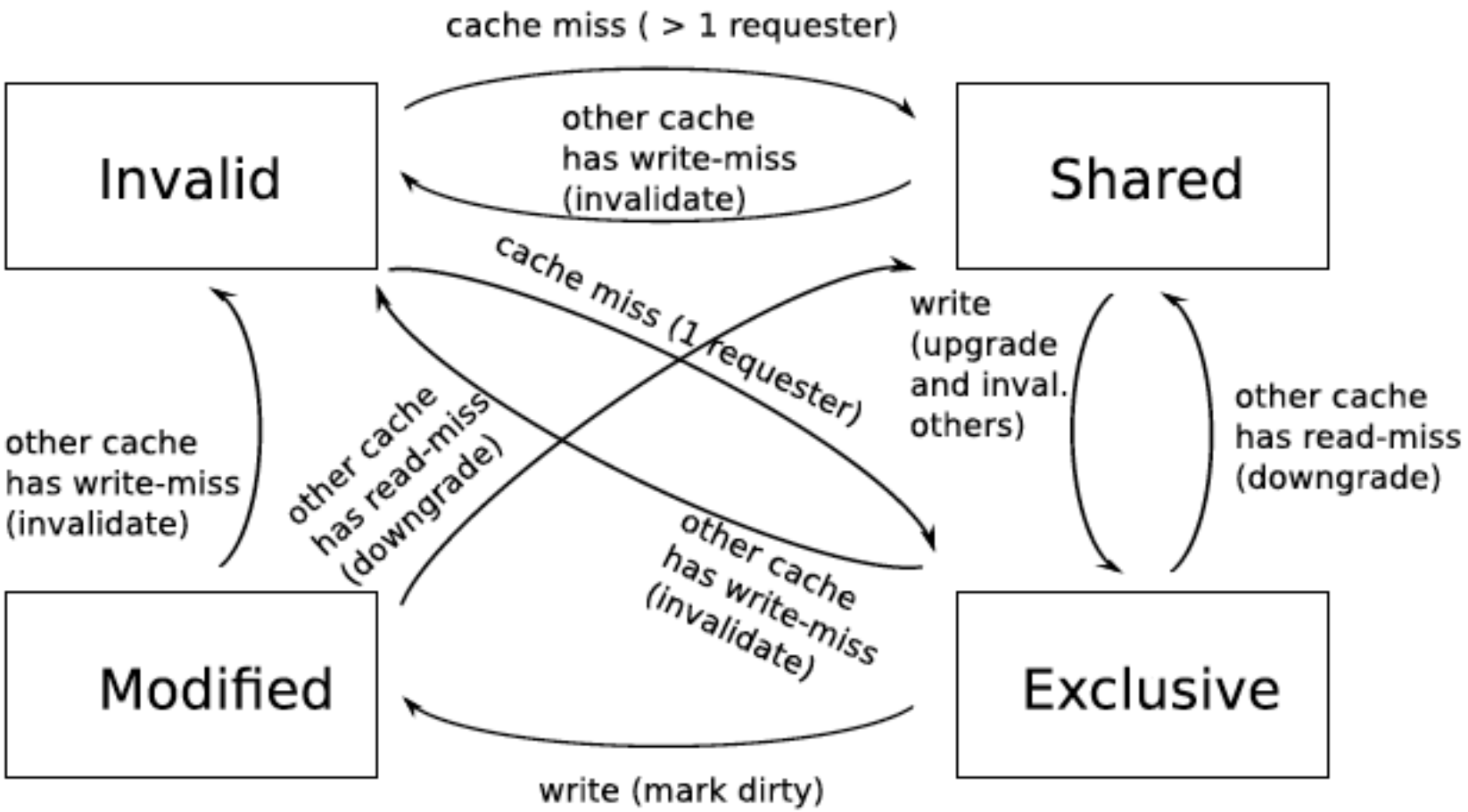
MESI State Machine from Lab 7



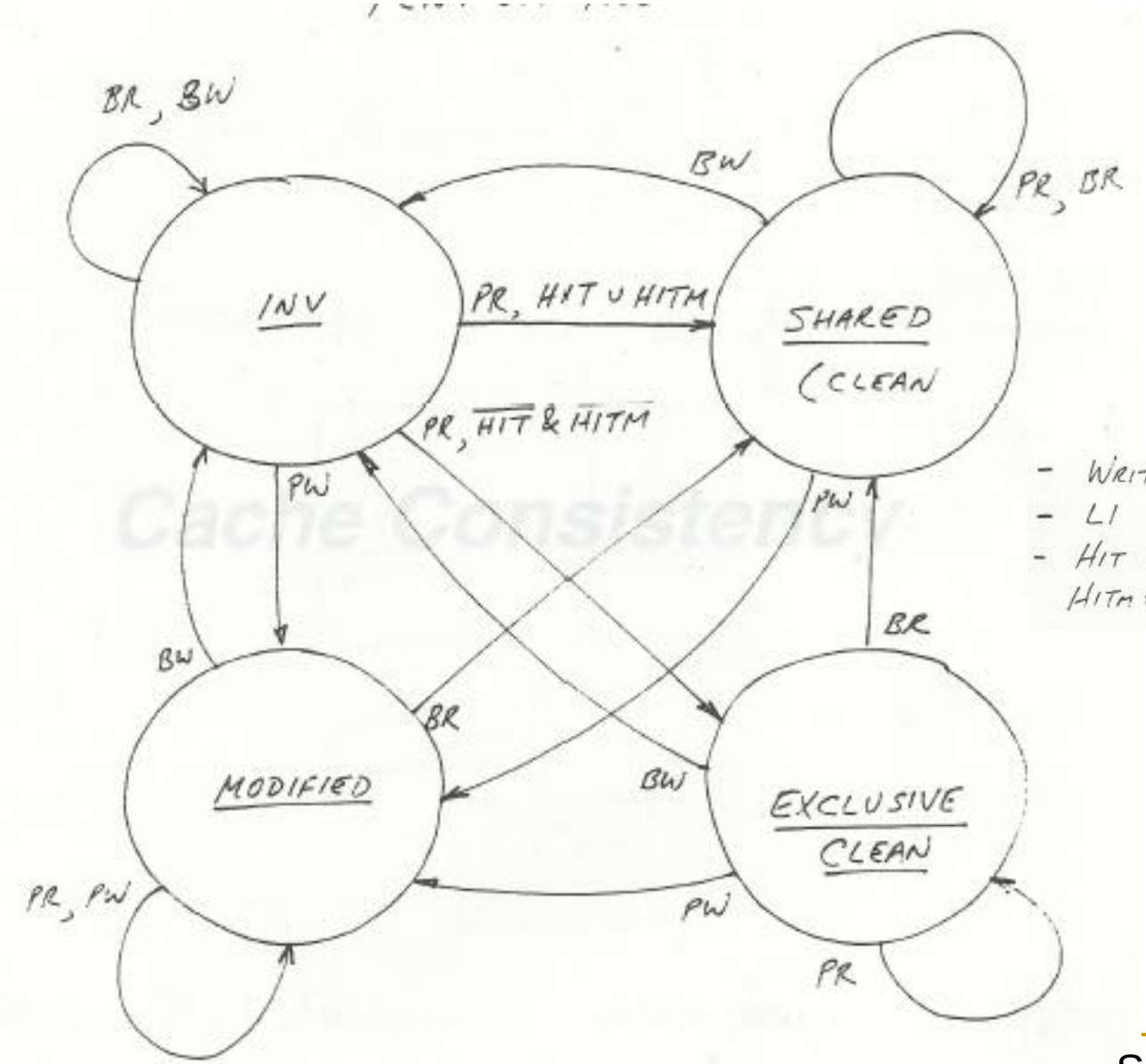
A transition from a single-owner state (Exclusive or Modified) to Shared is called a downgrade, because the transition takes away the owner's right to modify the data

A transition from Shared to a single-owner state (Exclusive or Modified) is called an upgrade, because the transition grants the ability to the owner (the cache which contains the respective block) to write to the block.

MESI State Machine from Lab 7



Intel Pentium Pro



- WRITE ALLOCATE
- L1 CAN HAVE DATA NOT IN L2
- HIT : SOMEONE HAS IT CLEAN
  HITM : SOMEONE HAS IT DIRTY

Slide credit: Yale Patt

Snoopy Invalidation Tradeoffs

- Should a downgrade from M go to S or I?
  - S: if data is likely to be reused (before it is written to by another processor)
  - I: if data is likely to be not reused (before it is written to by another)
- Cache-to-cache transfer
  - On a BusRd, should data come from another cache or memory?
  - Another cache
    - may be faster, if memory is slow or highly contended
  - Memory
    - Simpler: no need to wait to see if cache has data first
    - Less contention at the other caches
    - Requires writeback on M downgrade
- Writeback on Modified->Shared: necessary?
  - One possibility: ***Owner*** (O) state (MOESI protocol)
    - One cache owns the latest data (memory is not updated)
    - Memory writeback happens when all caches evict copies

The Problem with MESI

- Shared state requires the data to be clean
  - i.e., all caches that have the block have the up-to-date copy and so does the memory
- Problem: Need to write the block to memory when BusRd happens when the block is in Modified state

- Why is this a problem?
  - Memory can be updated unnecessarily → some other processor may write to the block while it is cached

Improving on MESI

- Idea 1: Do not transition from M→S on a BusRd. Invalidate the copy and supply the modified block to the requesting processor directly without updating memory

- Idea 2: Transition from M→S, but designate one cache as the owner (O), who will write the block back when it is evicted
  - Now "Shared" means "Shared and potentially dirty"
  - This is a version of the MOESI protocol

Tradeoffs in Sophisticated Cache Coherence Protocols

- The protocol can be optimized with more states and prediction mechanisms to

  + Reduce unnecessary invalidates and transfers of blocks

- However, more states and optimizations

  -- Are more difficult to design and verify (lead to more cases to take care of, race conditions)

  -- Provide diminishing returns

Revisiting Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, single point of serialization for all requests
  - Processors observe other processors' actions
    - E.g.: P1 makes "read-exclusive" request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - Single point of serialization *per block*, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks ownership (sharer set) for each block
  - Directory coordinates invalidation appropriately
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1

Snoopy Cache vs. Directory Coherence

- **Snoopy Cache**
  - \+ Critical path is short: miss → bus transaction to memory
  - \+ Global serialization is easy: bus provides this already (arbitration)
  - \+ Simple: adapt bus-based uniprocessors easily
  - – Relies on broadcast messages to be seen by all caches:

    → single point of serialization (bus): *not scalable*

- **Directory**
  - – Adds indirection to critical path: request → directory → mem
  - – Requires extra storage space to track sharer sets
    - Can be approximate (false positives are OK)
  - – Protocols and race conditions are more complex
  - \+ Exactly as scalable as interconnect and directory storage *(much more scalable than bus)*

Revisiting Directory-Based Cache Coherence

Remember: Directory Based Coherence

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- An example mechanism:
  - For each cache block in memory, store P+1 bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that the cache that has the only copy of the block and can update it without notifying others
  - On a read: set the cache's bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an "exclusive bit" associated with each block in each cache

Remember: Directory Based Coherence

Example directory based scheme

P=4

P+1 bits: for block A

0	0	0	0	0

Exclusive bit

No cache has the block

① P_1 takes a read miss to block A

0	0	0	0	0

→

0	1	0	0	0

② P_3 takes a read miss

↓

0	1	0	1	0

Directory-Based Protocols

- Required when scaling past the capacity of a single bus
- Distributed, *but:*
  - Coherence still requires single point of serialization (for write serialization)
  - Serialization location can be different for every block (striped across nodes)

- We can reason about the protocol for a single block: one *server* (directory node), many *clients* (private caches)

- Directory receives *Read* and *ReadEx* requests, and sends *Inv*l requests: invalidation is explicit (as opposed to snoopy buses)

Directory: Data Structures

0x00	Shared: {P0, P1, P2}
0x04	---
0x08	Exclusive: P2
0x0C	---
…	---

- Key operation to support is *set inclusion test*
  - False positives are OK: want to know which caches *may* contain a copy of a block, and spurious invalidations are ignored
  - False positive rate determines *performance*
- Most accurate (and expensive): full bit-vector
- Compressed representation, linked list, Bloom filters are all possible

Directory: Basic Operations

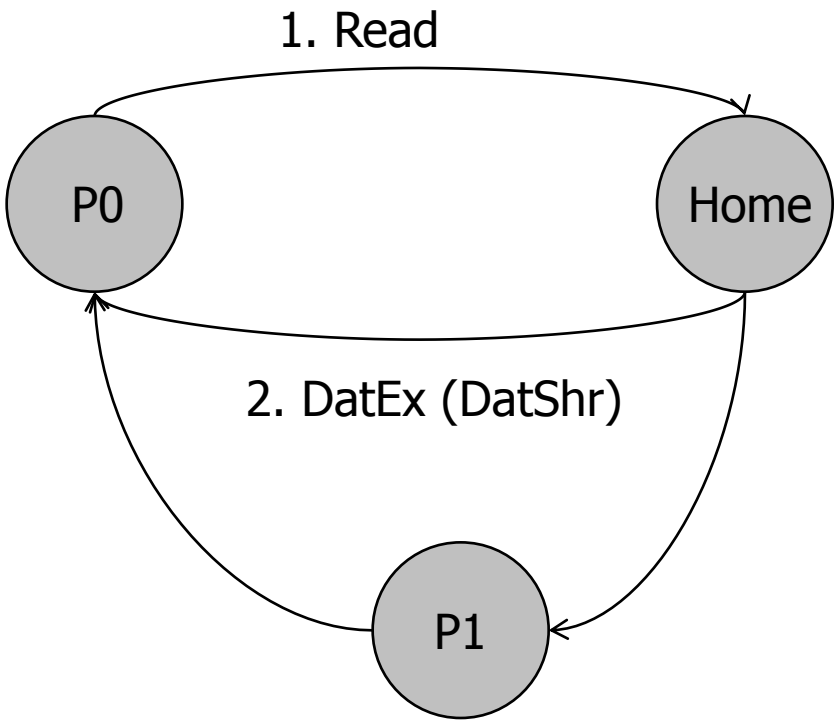
- Follow *semantics* of snoop-based system
  - but with explicit request, reply messages

- Directory:
  - Receives *Read, ReadEx, Upgrade* requests from nodes
  - Sends *Inval/Downgrade* messages to sharers if needed
  - Forwards request to memory if needed
  - Replies to requestor and updates sharing state

- Protocol design is flexible
  - Exact forwarding paths depend on implementation
  - For example, do cache-to-cache transfer?

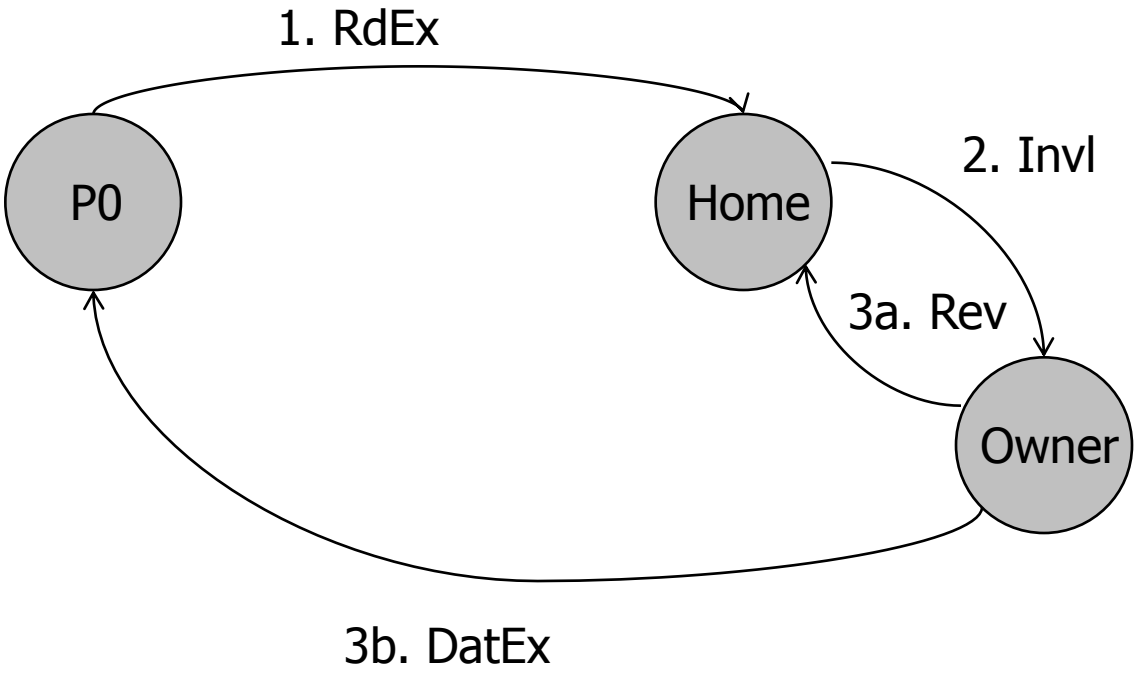
MESI Directory Transaction: Read

P0 acquires an address for reading:

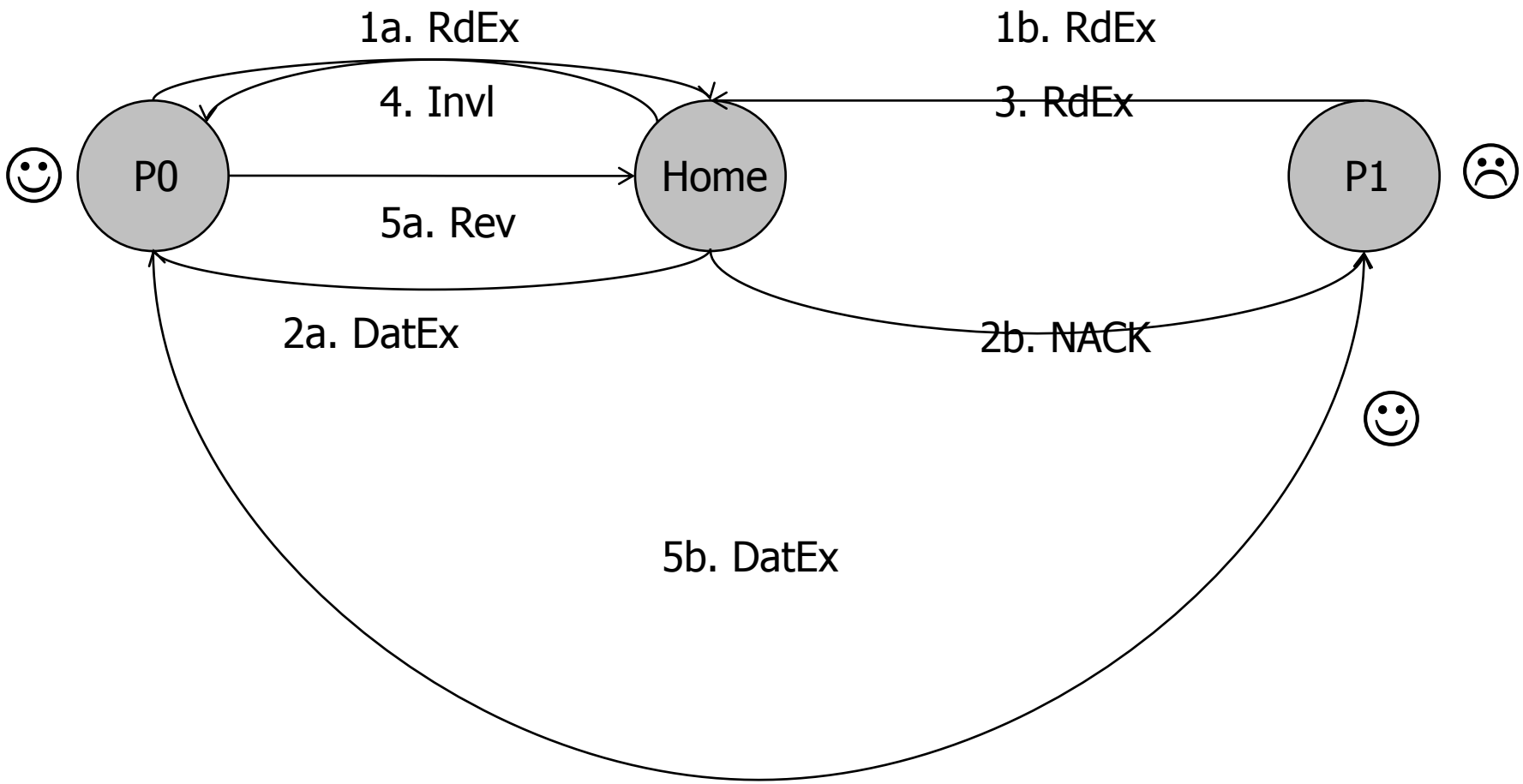


Culler/Singh Fig. 8.16

RdEx with Former Owner



Contention Resolution (for Write)



Issues with Contention Resolution

- Need to escape race conditions by:
  - NACKing requests to busy (pending invalidate) entries
    - Original requestor retries
  - OR, queuing requests and granting in sequence
  - (Or some combination thereof)

- Fairness
  - Which requestor should be preferred in a conflict?
  - Interconnect delivery order, and distance, both matter

- Ping-ponging is a higher-level issue
  - With solutions like combining trees (for locks/barriers) and better shared-data-structure design

Scaling the Directory: Some Questions

- How large is the directory?
- How can we reduce the access latency to the directory?
- How can we scale the system to thousands of nodes?