18-742

Lecture 5

Symmetric Multiprocessors

Spring 2005

Prof. Babak Falsafi

http://www.ece.cmu.edu/~ece742

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Reinhardt, Smith, and Singh of University of Illinois, Carnegie Mellon University, University of Wisconsin, Duke University, University of Michigan, and Princeton University.

Homework & Reading

Projects handout

- **On Friday**
- **Form teams, groups of two**

Homework

- **Homework 2 due by Friday**
- **Homework 3 will be review questions only**

Reading

- **Chapter 5**
- **James Archibald and Jean-Loup Baer, *Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model*, ACM Transaction on Computer Systems, Vol. 4, No. 4, pp 273-298, Nov. 1986.**

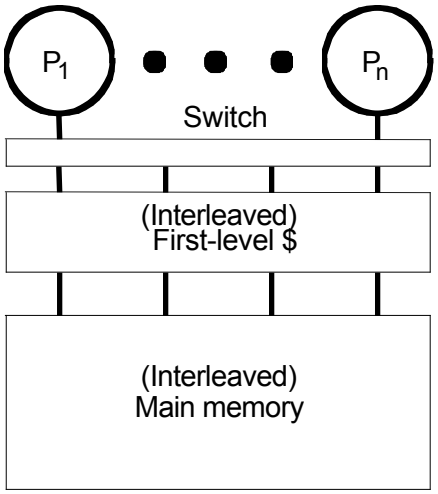
Outline

- **Motivation**
- **Coherence**
- **Coherence Tradeoffs**
- **Memory Consistency**
- **Synchronizaton**

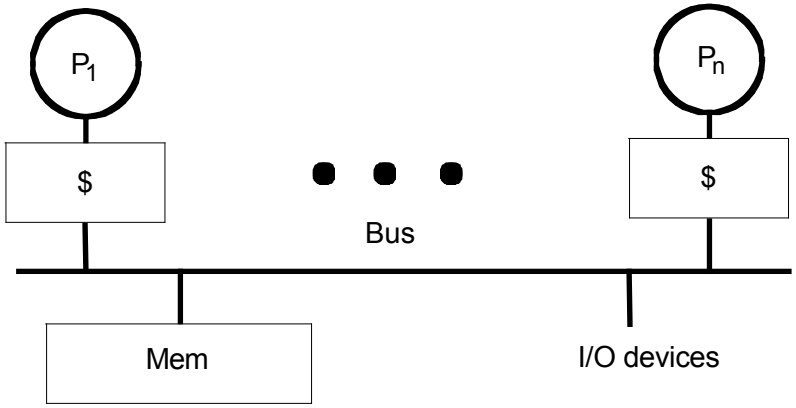
What is (Hardware) Shared Memory?

- **Take multiple (micro-)processors**
- **Implement a memory system with a single global physical address space (usually)**
- **Minimize memory latency (co-location & caches)**
- **Maximize memory bandwidth (parallelism & caches)**

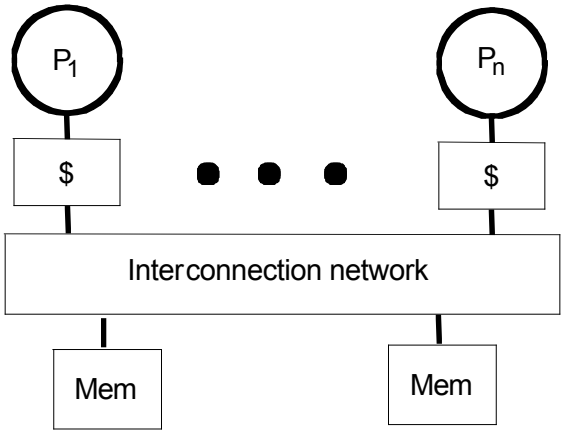
Some Memory System Options



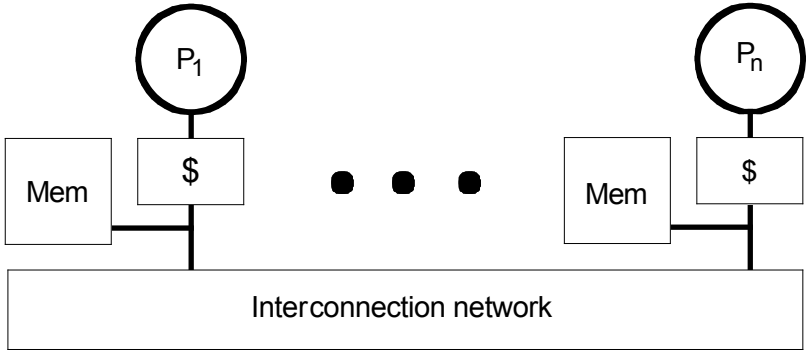
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall



(d) Distributed-memory

Why Shared Memory?

- **Pluses**
  - For applications looks like multitasking uniprocessor
  - For OS only evolutionary extensions required
  - Easy to do communication without OS
  - Software can worry about correctness first then performance
- **Minuses**
  - Proper synchronization is complex
  - Communication is implicit so harder to optimize
  - Hardware designers must implement
- **Result**
  - Symmetric Multiprocessors (SMPs) are the most success parallel machines ever
  - And the first with multi-billion-dollar markets

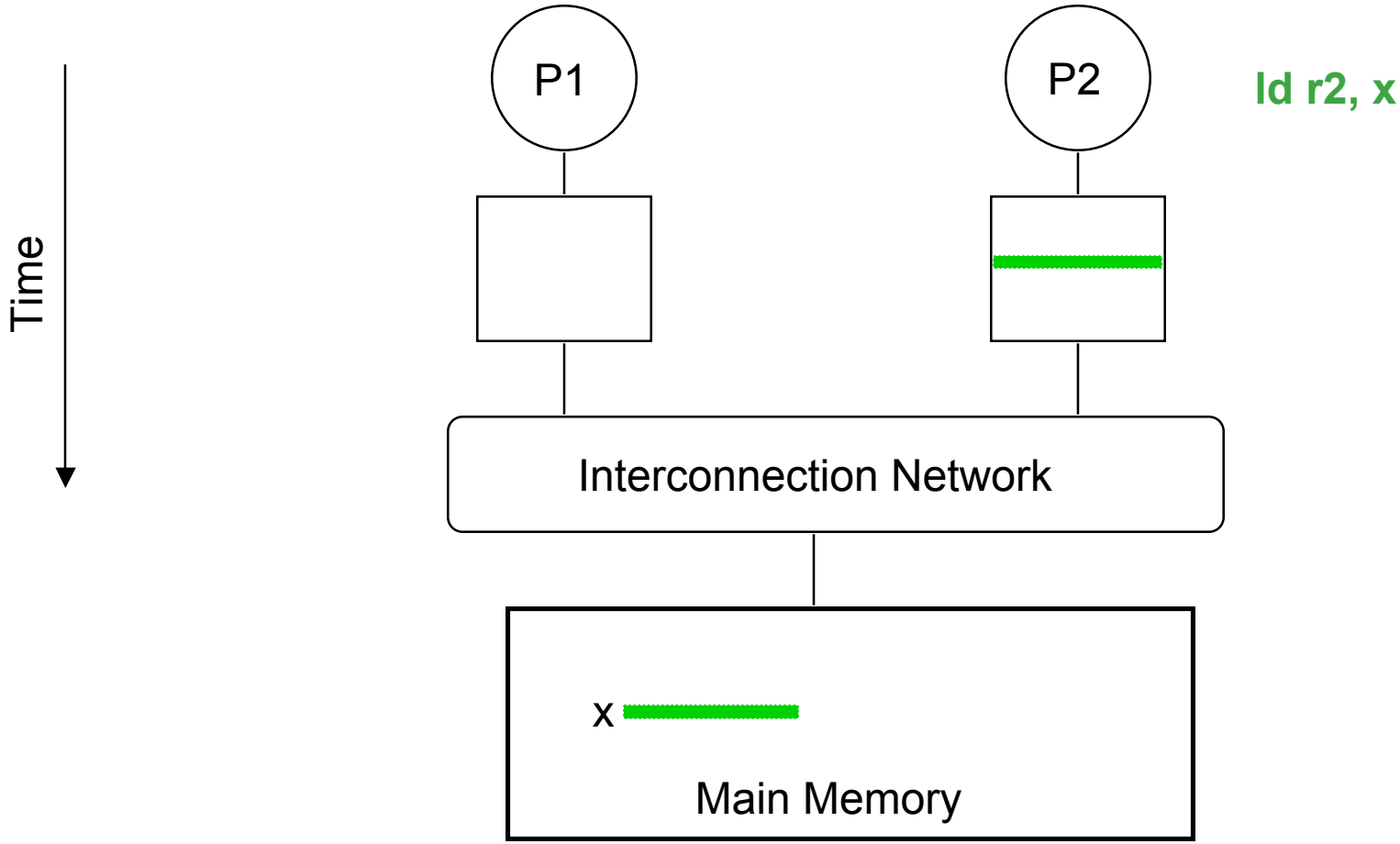
In More Detail

- **Efficient Naming**
  - virtual to physical using TLBs
  - ability to name relevent portions of objects
- **Ease and efficiency of caching**
  - caching is natural and well understood
  - can be done in HW automatically
- **Communication Overhead**
  - low since protection is built into memory system
  - easy for HW to packetize requests / replies
- **Integration of latency tolerance**
  - demand-driven: consistency models, prefetching, multithreaded
  - Can extend to push data to PEs and use bulk transfer

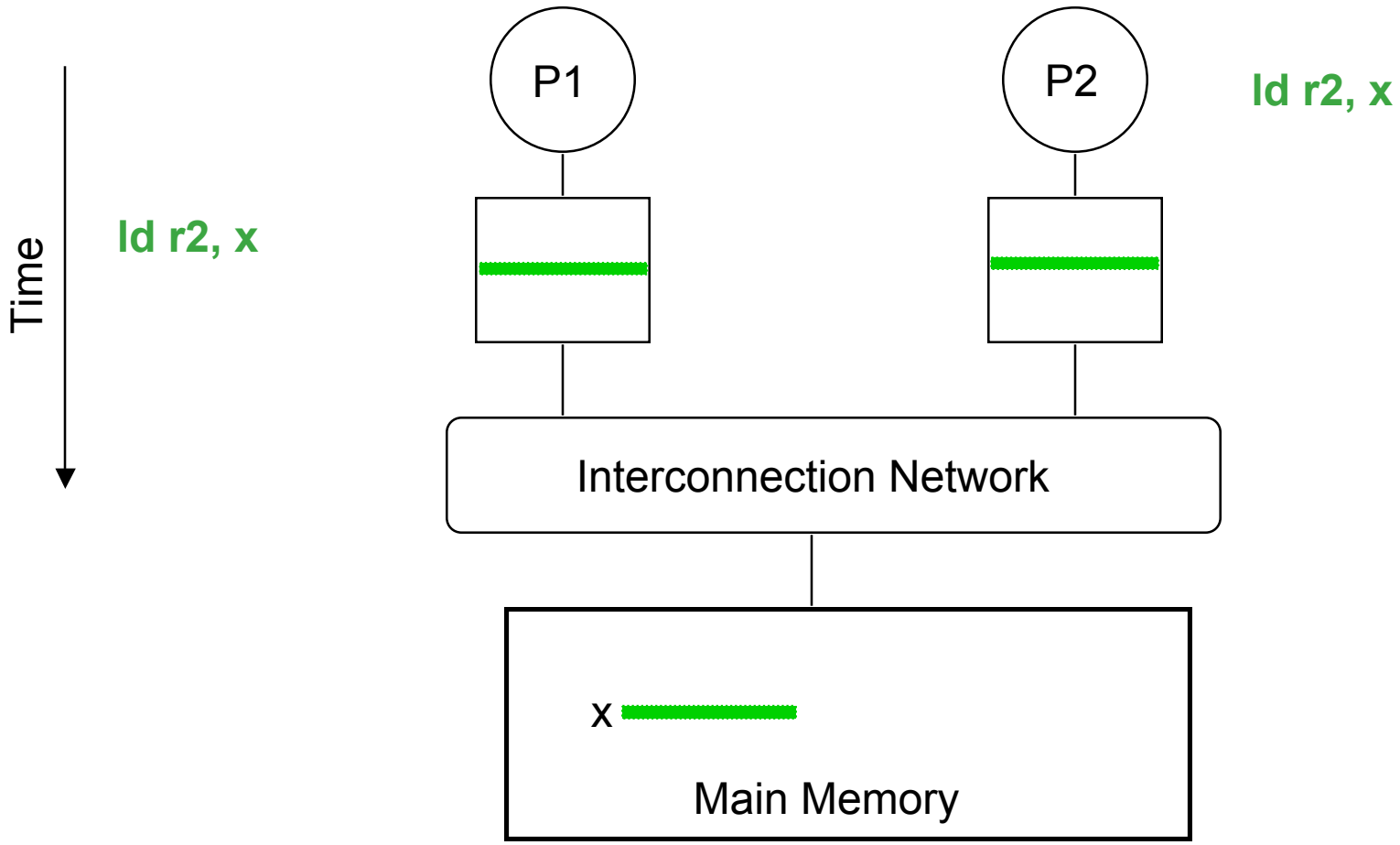
Symmetric Multiprocesors (SMP)

- **Multiple (micro-)processors**
- **Each has cache (today a cache hierarchy)**
- **Connect with logical bus (totally-ordered broadcast)**
- **Implement Snooping Cache Coherence Protocol**
  - **Broadcast all cache "misses" on bus**
  - **All caches "snoop" bus and may act**
  - **Memory responds otherwise**

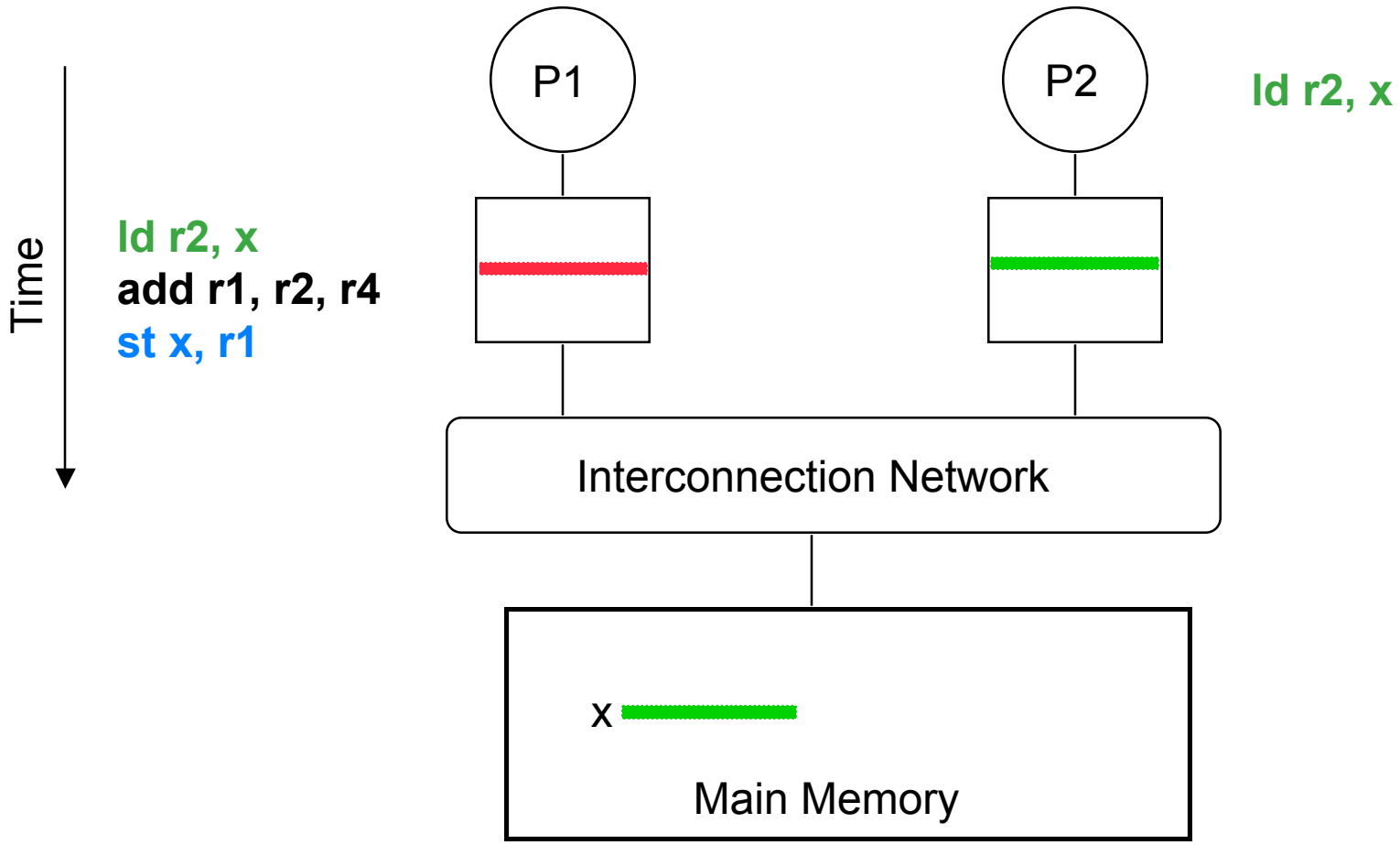
Cache Coherence Problem (Step 1)



Cache Coherence Problem (Step 2)



Cache Coherence Problem (Step 3)

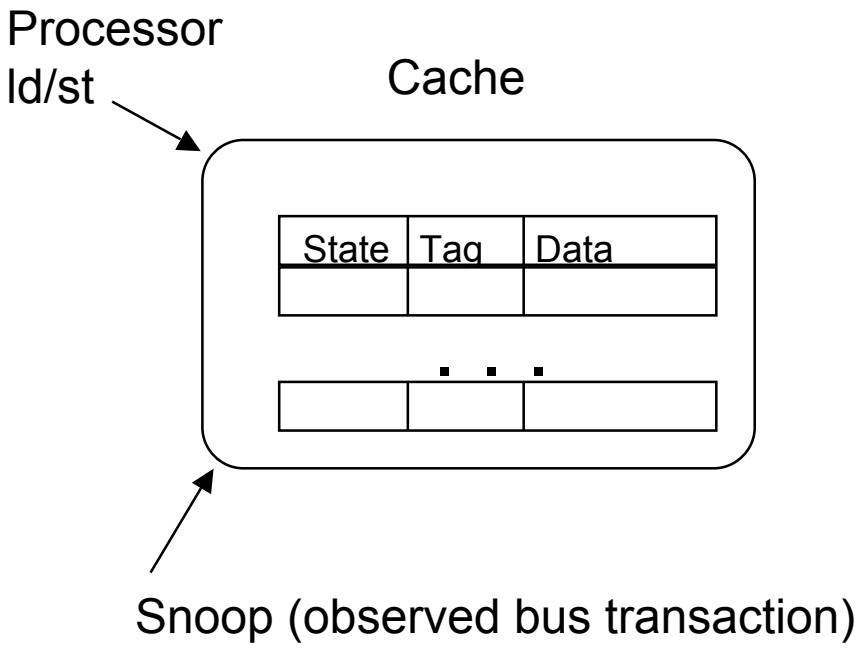


Snooping Cache-Coherence Protocols

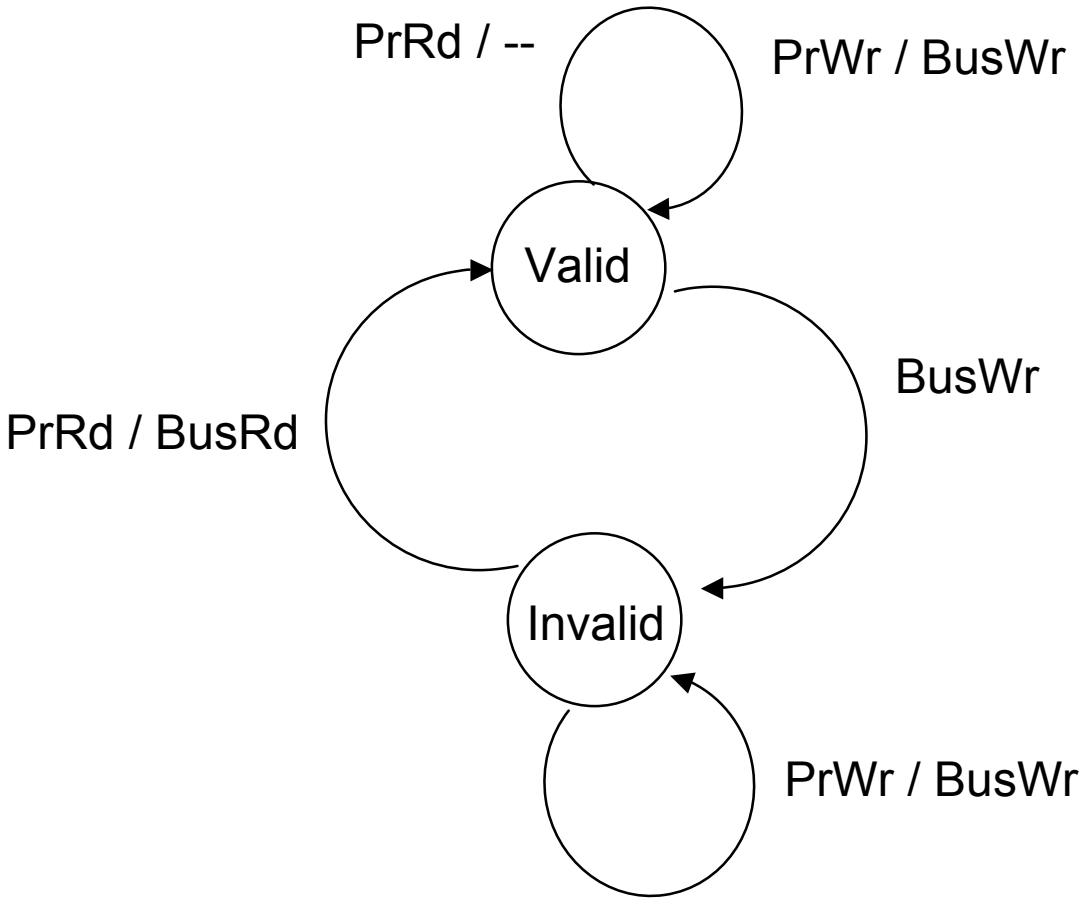
- **Bus provides serialization point (more on this later)**
- **Each cache controller "snoops" all bus transactions**
  - **relevant transactions if for a block it contains**
  - **take action to ensure coherence**
    - » **invalidate**
    - » **update**
    - » **supply value**
  - **depends on state of the block and the protocol**
- **Simultaneous Operation of Independent Controllers**

Snooping Design Choices

- **Controller updates state of blocks in response to processor and snoop events and generates bus xactions**
- **Often have duplicate cache tags**
- **Snoopy protocol**
  - **set of states**
  - **state-transition diagram**
  - **actions**
- **Basic Choices**
  - **write-through vs. write-back**
  - **invalidate vs. update**



The Simple Invalidate Snooping Protocol



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

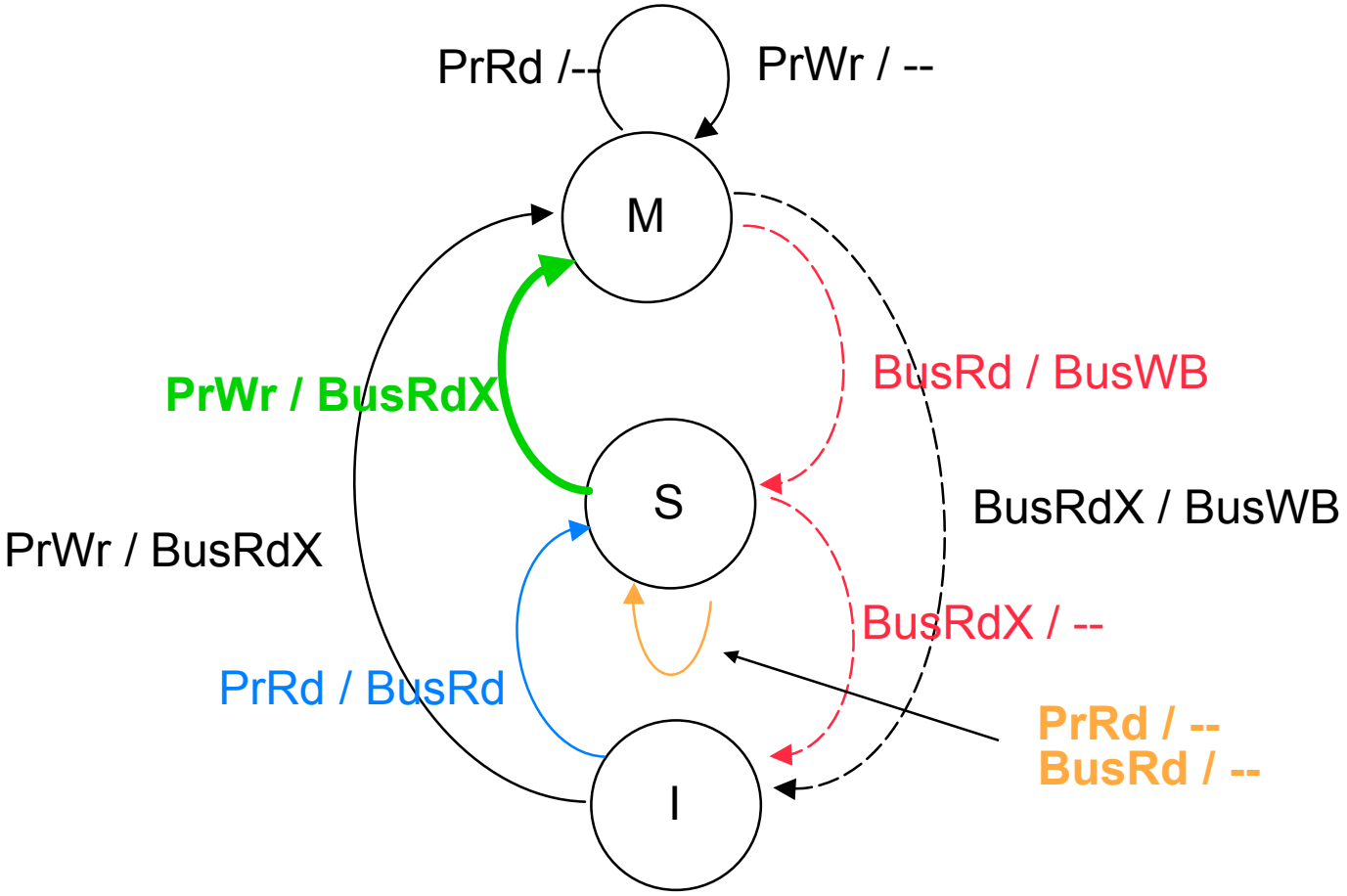
A 3-State Write-Back Invalidation Protocol

- **2-State Protocol**
  - **+ Simple hardware and protocol**
  - **– Bandwidth (every write goes on bus!)**
- **3-State Protocol (MSI)**
  - **Modified**
    - **» one cache has valid/latest copy**
    - **» memory is stale**
  - **Shared**
    - **» one or more caches have valid copy**
  - **Invalid**
- **Must invalidate all other copies before entering modified state**
- **Requires bus transaction (order and invalidate)**

MSI Processor and Bus Actions

- **Processor:**
  - **PrRd**
  - **PrWr**
  - **Writeback on replacement of modified block**
- **Bus**
  - **Bus Read (BusRd) Read without intent to modify, data could come from memory or another cache**
  - **Bus Read-Exclusive (BusRdX) Read with intent to modify, must invalidate all other caches copies**
  - **Writeback (BusWB) cache controller puts contents on bus and memory is updated**
  - **Definition: cache-to-cache transfer occurs when another cache satisfies BusRd or BusRdX request**
- **Let's draw it!**

MSI State Diagram



An example

Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read u	S	--	--	BusRd	Memory
2. P3 read u	S	--	S	BusRd	Memory
3. P3 write u	I	--	M	BusRdX	Memory or not
4. P1 read u	S	--	S	BusRd	P3's cache
5. P2 read u	S	S	S	BusRd	Memory

- Single writer, multiple reader protocol
- Why Modified to Shared?
- What if not in any cache?
  - **Read, Write produces 2 bus transactions!**

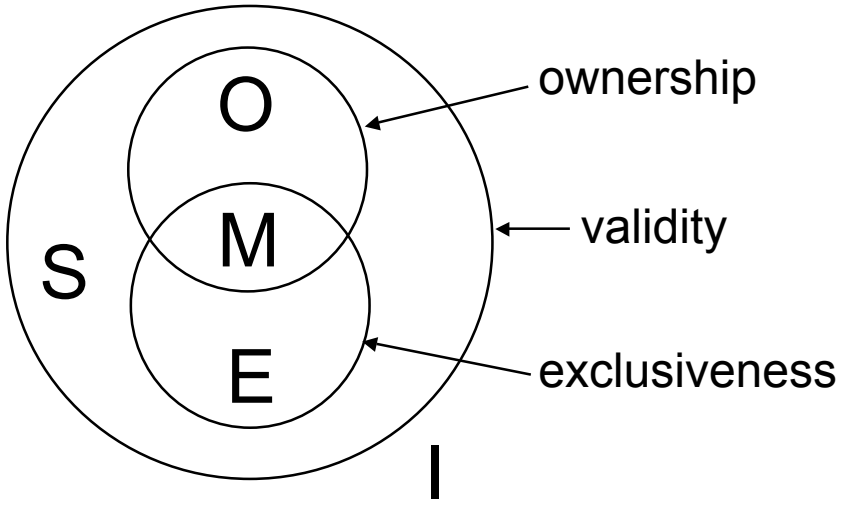
4-State (MESI) Invalidation Protocol

- Often called the Illinois protocol
- Modified (dirty)
- Exclusive (clean unshared) only copy, not dirty
- Shared
- Invalid
- Requires shared signal to detect if other caches have a copy of block
- Cache Flush for cache-to-cache transfers
  - Only one can do it though
- What does state diagram look like?

More Generally: MOESI

- **[Sweazey & Smith ISCA86]**
- **M - Modified (dirty)**
- **O - Owned (dirty but shared) WHY?**
- **E - Exclusive (clean unshared) only copy, not dirty**
- **S - Shared**
- **I - Invalid**

- **Variants**
  - **MSI**
  - **MESI**
  - **MOSI**
  - **MOESI**



4-State Write-back Update Protocol

- Dragon (Xerox PARC)
- States
  - Exclusive (E): one copy, clean, memory is up-to-date
  - Shared-Clean (SC): could be two or more copies, memory unknown
  - Shared-Modified (SM): could be two or more copies, memory stale
  - Modified (M)
- Adds Bus Update Transaction
- Adds Cache Controller Update operation
- Must obtain bus before updating local copy
- What does state diagram look like?
  - let's look at the actions first

Dragon Actions

- **Processor**
  - **PrRd**
  - **PrWr**
  - **PrRdMiss**
  - **PrWrMiss**
  - **Update in response to BusUpd**
- **Bus Xactions**
  - **BusRd**
  - **BusUpd**
  - **BusWB**

Tradeoffs in Protocol Design

- New State Transitions
- What Bus Transactions
- Cache block size
- Workload dependence
- Compute bandwidth, miss rates, from state transitions

Computing Bandwidth

- **Why bandwidth?**
- **How do I compute it?**
- **Monitor State Transitions**
  - **tells me bus transactions**
  - **I know how many bytes each bus transaction requires**

MESI State Transitions and Bandwidth

FROM/TO	NP	I	E	S	M
NP	--	--	BusRd 6+64	BusRd 6+64	BusRdX 6+64
I	--	--	BusRd 6+64	BusRd 6+64	BusRdX 6+64
E	--	--	--	--	--
S	--	--	NA	--	BusUpgr 6
M	BusWB 6 + 64	BusWB 6+64	NA	BusWB 6 + 64	--

Bandwidth of MSI vs. MESI

- 200 MIPS/MFLOPS processor
  - use with measured state transition counts to obtain transitions/sec
- Compute state transitions/sec
- Compute bus transactions/sec
- Compute bytes/sec
- What is BW savings of MESI over MSI?
- Difference between protocols is Exclusive State
  - Add BusUpgr for E->M transtion
- Result is very small benefit!
  - Small number of E->M transitions
  - Only 6 bytes on bus

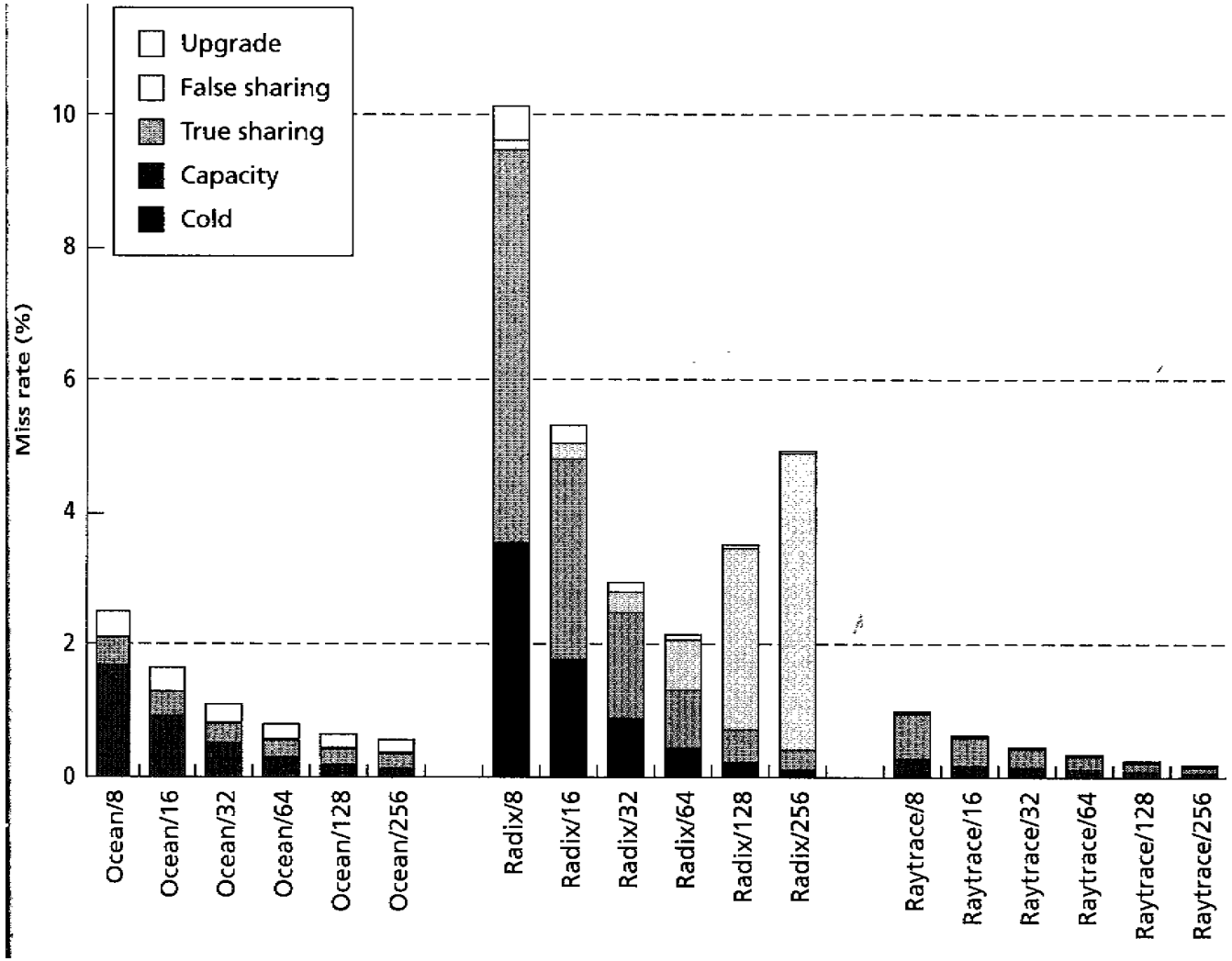
MSI BusUpgrd vs. BusRdX

- **MSI S->M Transition Issues BusUpgrd**
  - **could have block invalidated while waiting for BusUpgrd response**
  - **adds complexity to detect this**
- **Instead just issue BusRdX**
  - **from MESI put BusRdX in E->M and S->M**
- **Result is 10% to 20% Improvement**
  - **application dependent**

Cache Block Size

- **Block size is unit of transfer and of coherence**
  - **Doesn't have to be, could have coherence smaller [Goodman]**
- **Uniprocessor 3C's**
  - **(Compulsory, Capacity, Conflict)**
- **SM adds Coherence Miss Type**
  - **True Sharing miss fetches data written by another processor**
  - **False Sharing miss results from independent data in same coherence block**
- **Increasing block size**
  - **Usually fewer 3C misses but more bandwidth**
  - **Usually more false sharing misses**
- **P.S. on increasing cache size**
  - **Usually fewer capacity/conflict misses (& compulsory don't matter)**
  - **No effect on true/false "coherence" misses (so may dominate)**

Cache Block Size: Miss Rate



Invalidate vs. Update

- **Pattern 1:**

```
for i = 1 to k
   P1(write, x);        // one write before reads
   P2--PN-1(read, x);
end for i
```

- **Pattern 2:**

```
for i = 1 to k
   for j = 1 to m
      P1(write, x);   // many writes before reads
   end for j
   P2(read, x);
end for i
```

Invalidate vs. Update, cont.

- **Pattern 1 (one write before reads)**
  - **N = 16, M = 10, K = 10**
  - **Update**
    - » **Iteration 1: N regular cache misses (70 bytes)**
    - » **Remaining iterations: update per iteration (14 bytes; 6 cntrl, 8 data)**
  - **Total Update Traffic = 16*70 + 9*14 = 1246 bytes**
    - » **book assumes 10 updates instead of 9…**
  - **Invalidate**
    - » **Iteration 1: N regular cache misses (70 bytes)**
    - » **Remaining: P1 generates upgrade (6), 15 others Read miss (70)**
  - **Total Invalidate Traffic = 16*70 + 9*6 + 15*9*17 = 10,624 bytes**
- **Pattern 2 (many writes before reads)**
  - **Update = 1400 bytes**
  - **Invalidate = 824 bytes**

Invalidate vs. Update, cont.

- **What about real workloads?**
  - **Update can generate too much traffic**
  - **Must limit (e.g., competitive snooping)**
- **Current Assessment**
  - **Update very hard to implement correctly (c.f., consistency discussion coming next)**
  - **Rarely done**
- **Future Assessment**
  - **May be same as current or**
  - **Chip multiprocessors may revive update protocols**
    - » **More intra-chip bandwidth**
    - » **Easier to have predictable timing paths?**

Qualitative Sharing Patterns

- [Weber & Gupta, ASPLOS3]
- Read-Only
- Migratory Objects
  - Maniputalated by one processor at a time
  - Often protected by a lock
  - Usually a write causes only a single invalidation
- Synchronization Objects
  - Often more processors imply more invalidations
- Mostly Read
  - More processors imply more invalidations, but writes are rare
- Frequently Read/Written
  - More processors imply more invalidations