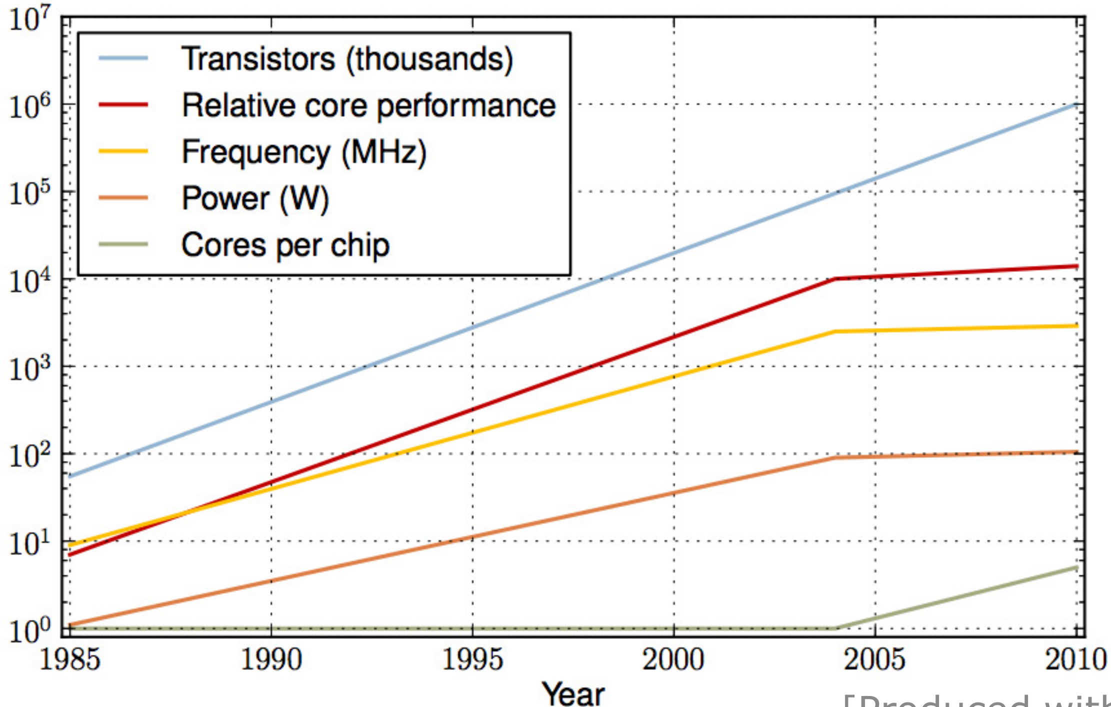
Cache Coherence

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

http://www.csg.csail.mit.edu/6.823

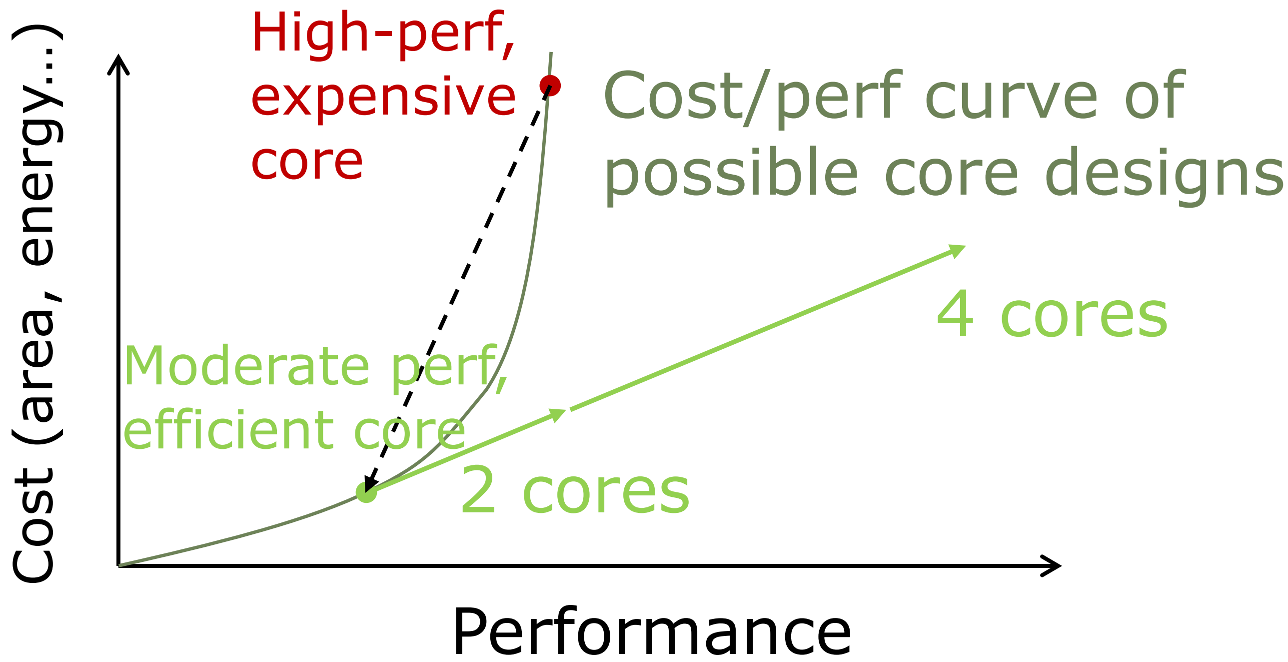
The Shift to Multicore



[Produced with CPUDB, cpudb.stanford.edu]

- Since 2005, improvements in system performance mainly due to increasing cores per chip
- Why? Limited instruction-level parallelism
  Technology scaling

Multicore Performance



What factors may limit multicore performance?

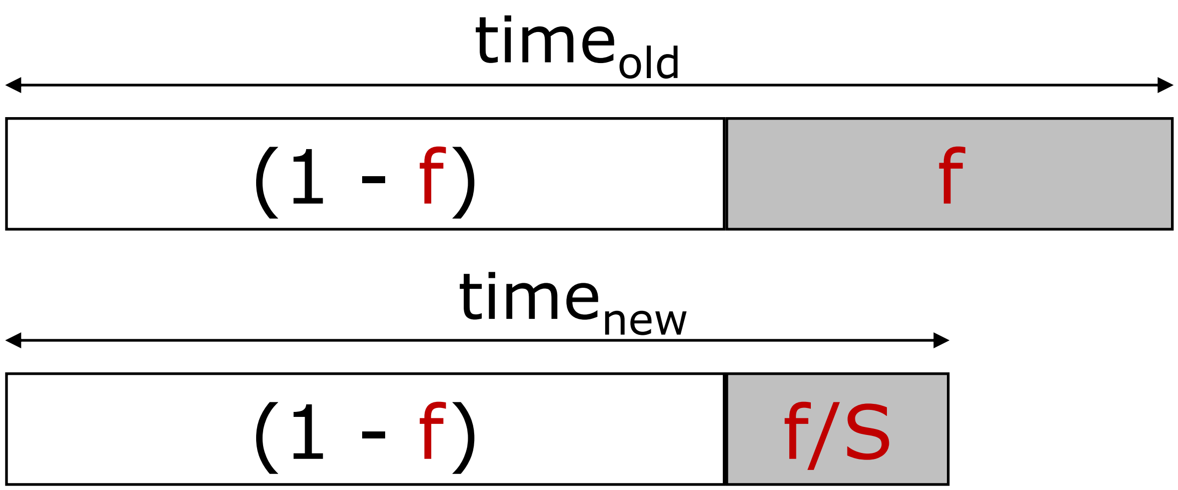
Limited application parallelism
Memory accesses and inter-core communication
Programming complexity

Amdahl's Law

- Speedup= $time_{without\ enhancement}$ / $time_{with\ enhancement}$
- Suppose an enhancement speeds up a fraction f of a task by a factor of S

$$time_{new} = time_{old} \cdot ((1-f) + f/S)$$

$$S_{overall} = 1 / ((1-f) + f/S)$$



Corollary: Make the common case fast

Amdahl's Law and Parallelism

- Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential
- What is the maximum speedup you can get by running on a multicore machine?

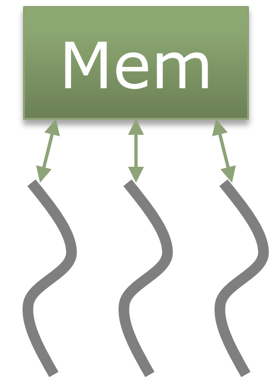
$$S_{overall} = 1 / ((1-f) + f/S)$$

$$f = 0.9, S=\infty \rightarrow S_{overall} = 10$$

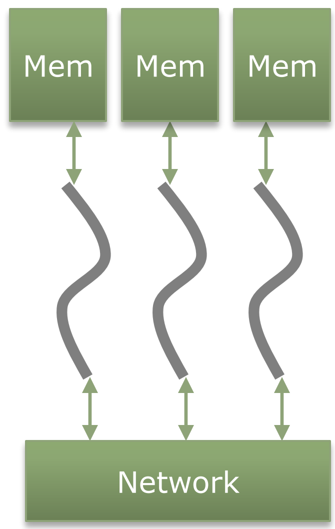
What f do you need to use a 1000-core machine well?

Communication Models

- Shared memory:
  - Single address space
  - Implicit communication by reading/writing memory
    - Data
    - Control (semaphores, locks, barriers, …)
  - Low-level programming model: threads



- Message passing:
  - Separate address spaces
  - Explicit communication by send/rcv messages
    - Data & control (blocking msgs, barriers, …)
  - Low-level programming model: processes + inter-process communication (e.g., MPI)

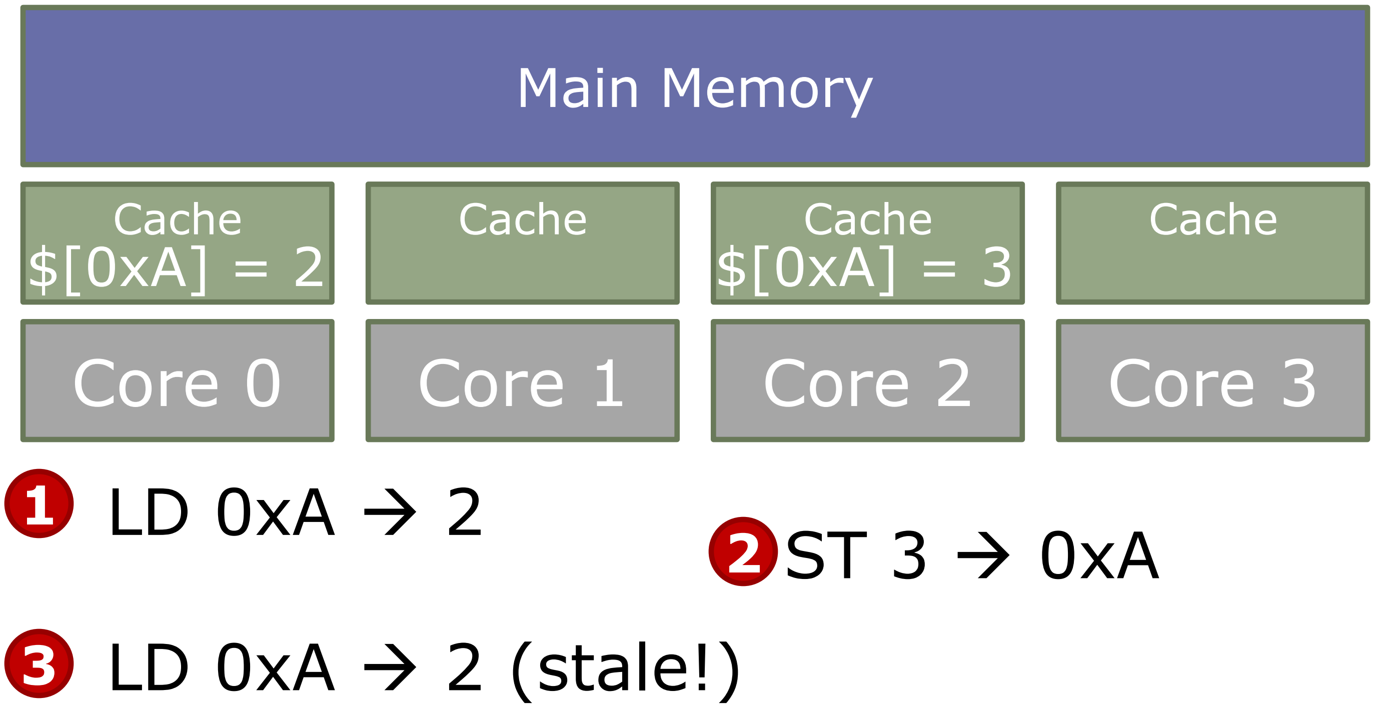


- Pros/cons of each model?

Coherence & Consistency

- Shared memory systems:
  - Have multiple private caches for performance reasons
  - Need to provide the illusion of a single shared memory
- Intuition: A read should return the most recently written value
  - What is "most recent"?
- Formally:
  - Coherence: What values can a read return?
    - Concerns reads/writes to a single memory location
  - Consistency: When do writes become visible to reads?
    - Concerns reads/writes to multiple memory locations

Cache Coherence Avoids Stale Data

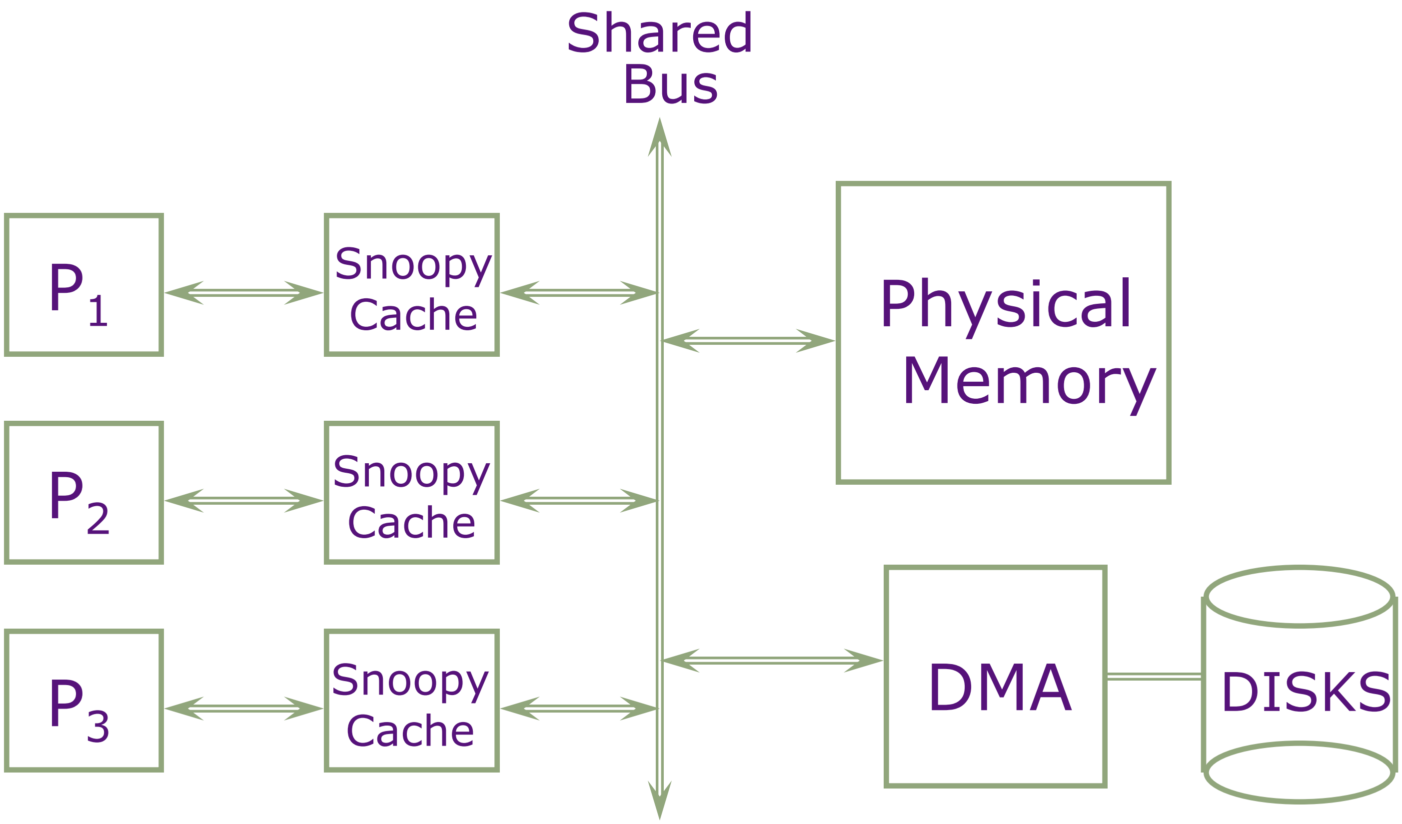


- A cache coherence protocol controls cache contents to avoid stale cache lines

Implementing Cache Coherence

- Coherence protocols must enforce two rules:
  - Write propagation: Writes eventually become visible to all processors
  - Write serialization: Writes to the same location are serialized (all processors see them in the same order)
- How to ensure write propagation?
  - Write-invalidate protocols: Invalidate all other cached copies before performing the write
  - Write-update protocols: Update all other cached copies after performing the write
- How to track sharing state of cached data and serialize requests to the same address?
  - Snooping-based protocols: All caches observe each other's actions through a shared bus
  - Directory-based protocols: A coherence directory tracks contents of private caches and serializes requests

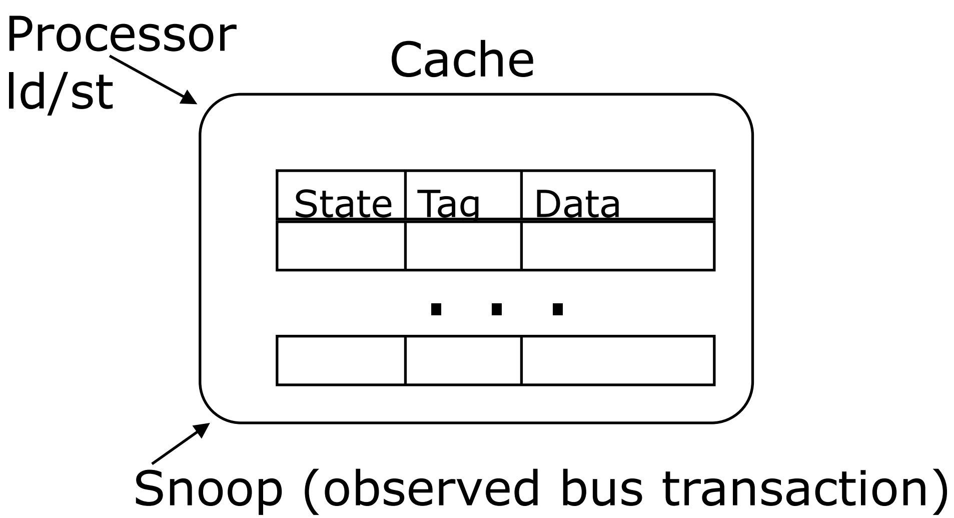
Snooping-Based Coherence [Goodman 1983]



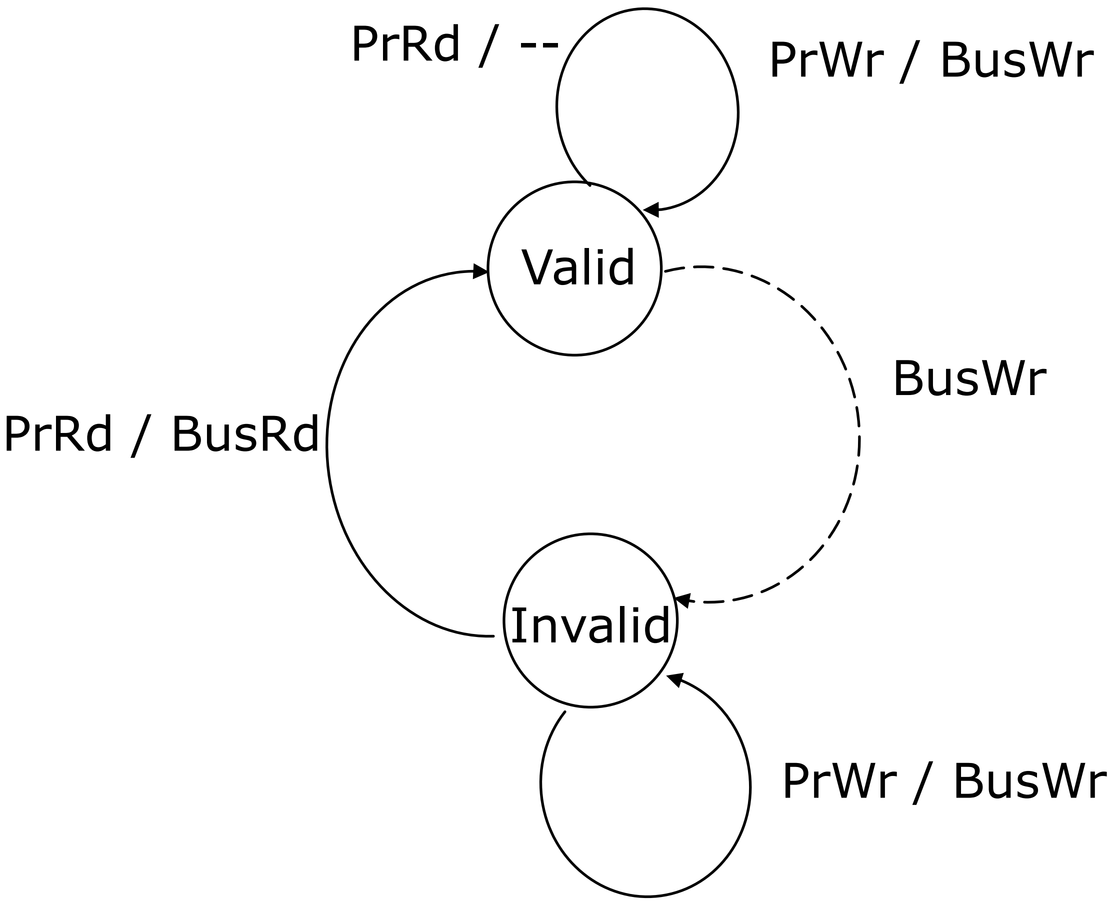
Caches watch (snoop on) bus to keep all processors' view of memory coherent

Snooping-Based Coherence

- Bus provides serialization point
  - Broadcast, totally ordered
  - Each cache controller "snoops" all bus transactions
  - Controller updates state of cache in response to processor and snoop events and generates bus transactions
- Snoopy protocol (FSM)
  - State-transition diagram
  - Actions
- Handling writes:
  - Write-invalidate
  - Write-update



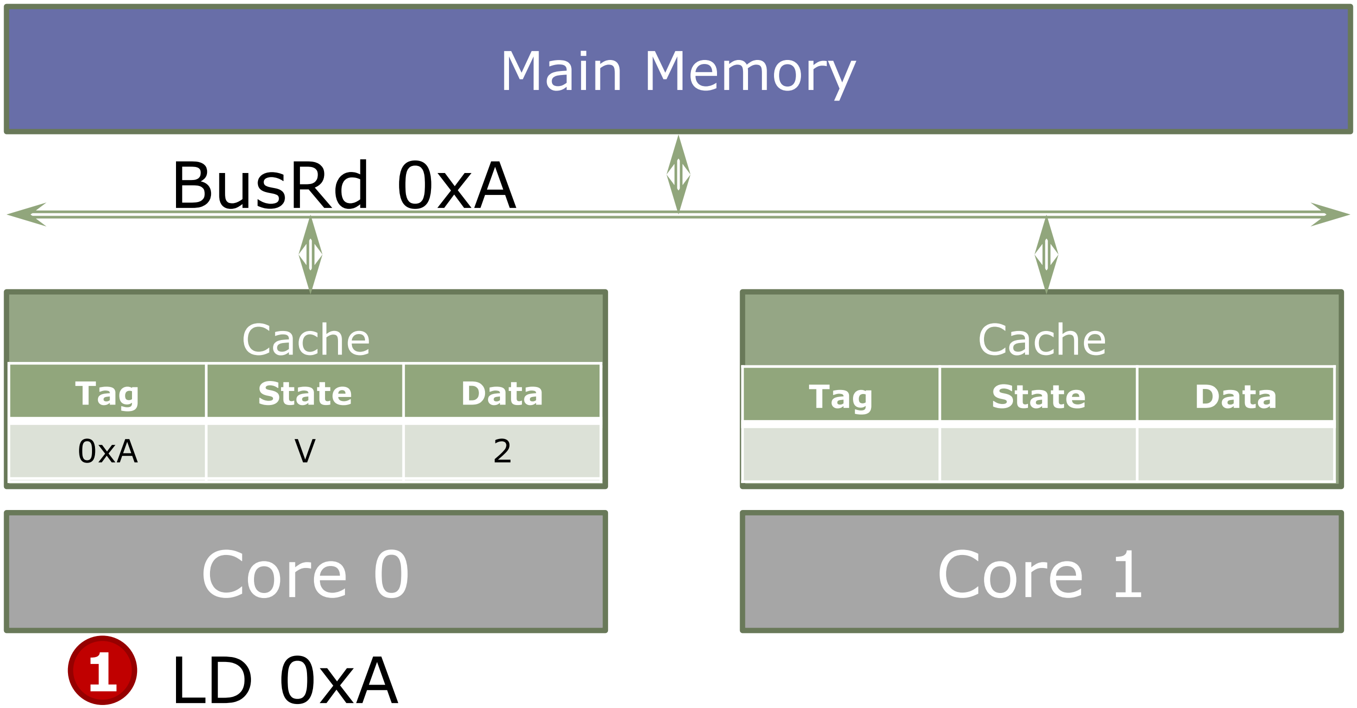
A Simple Protocol: Valid/Invalid (VI)



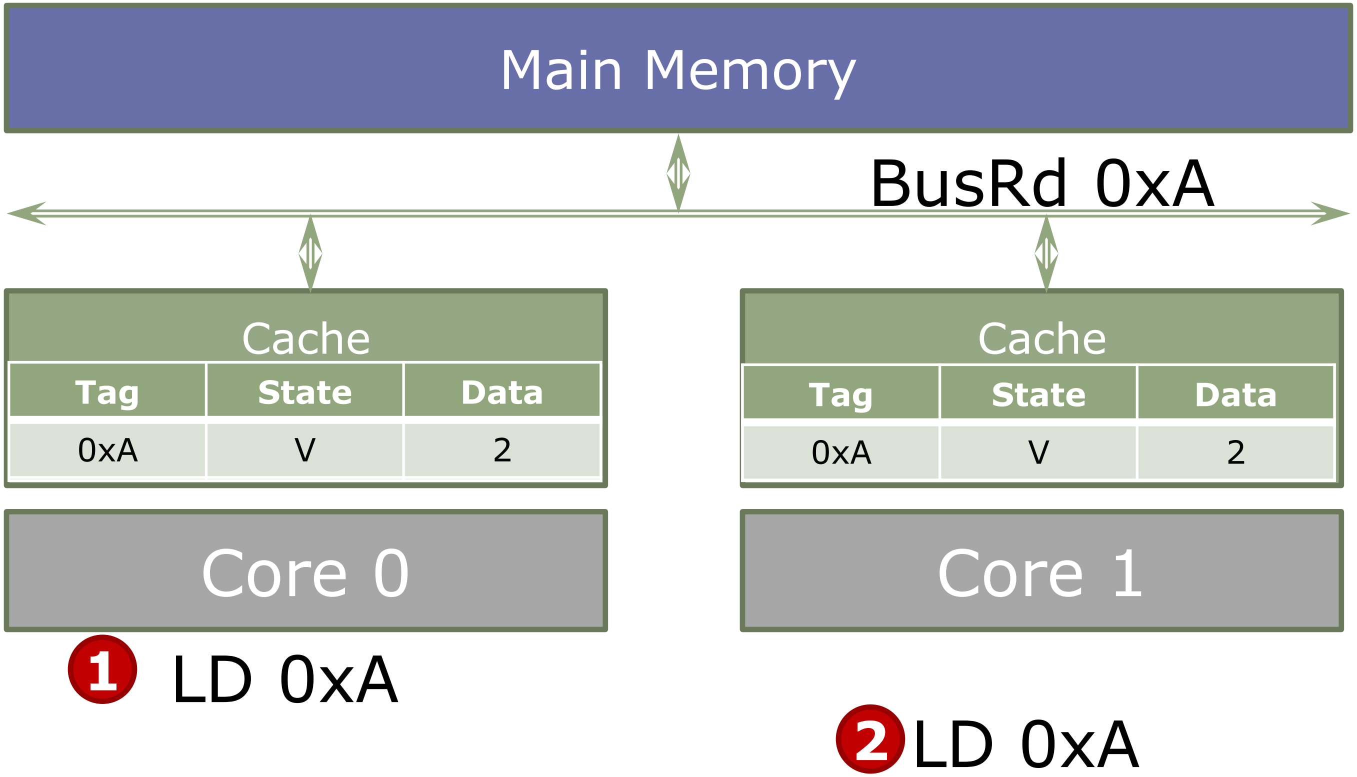
- Assume write-through caches

Actions
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Write (BusWr)

Valid/Invalid Example

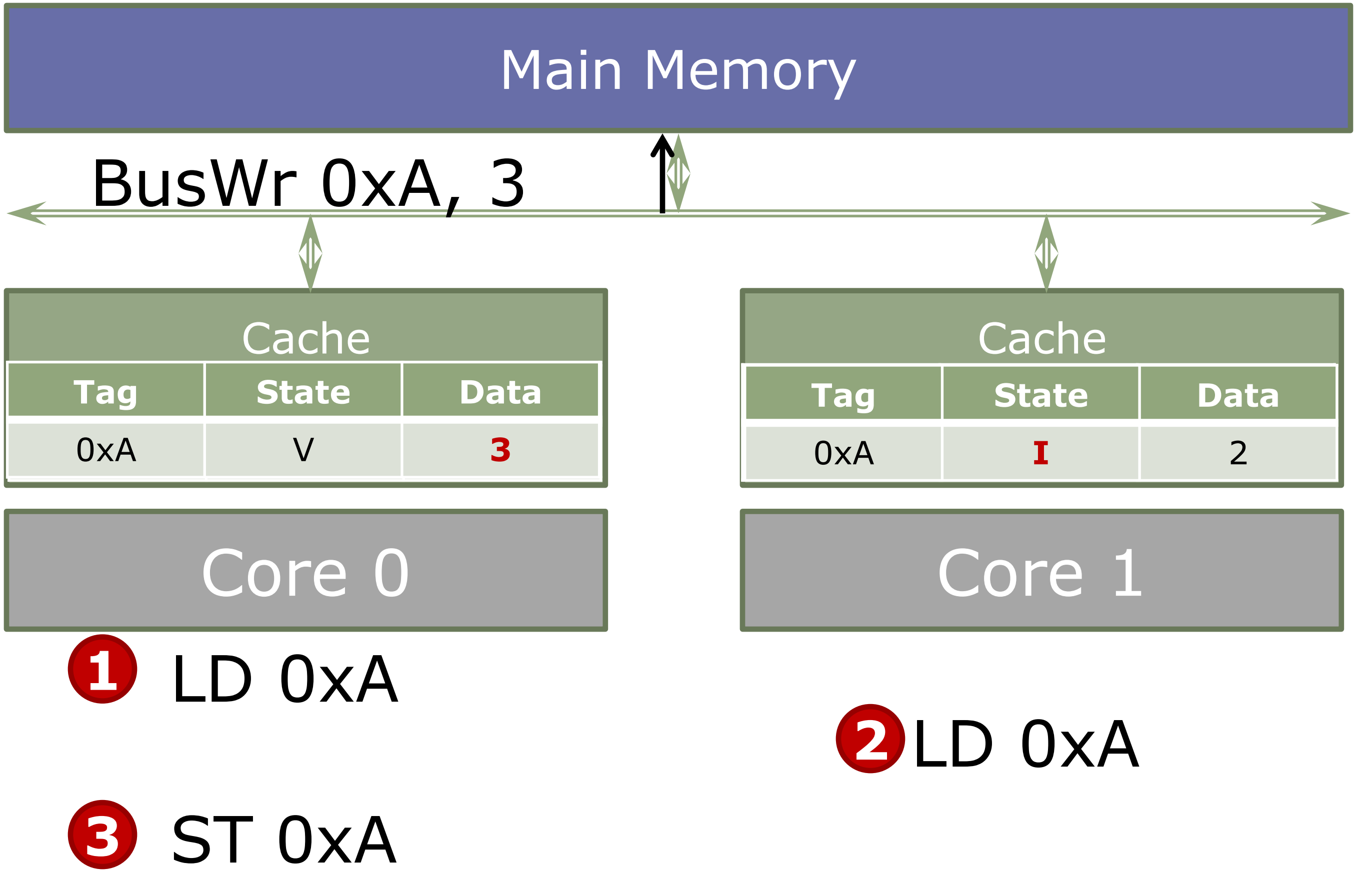


Valid/Invalid Example

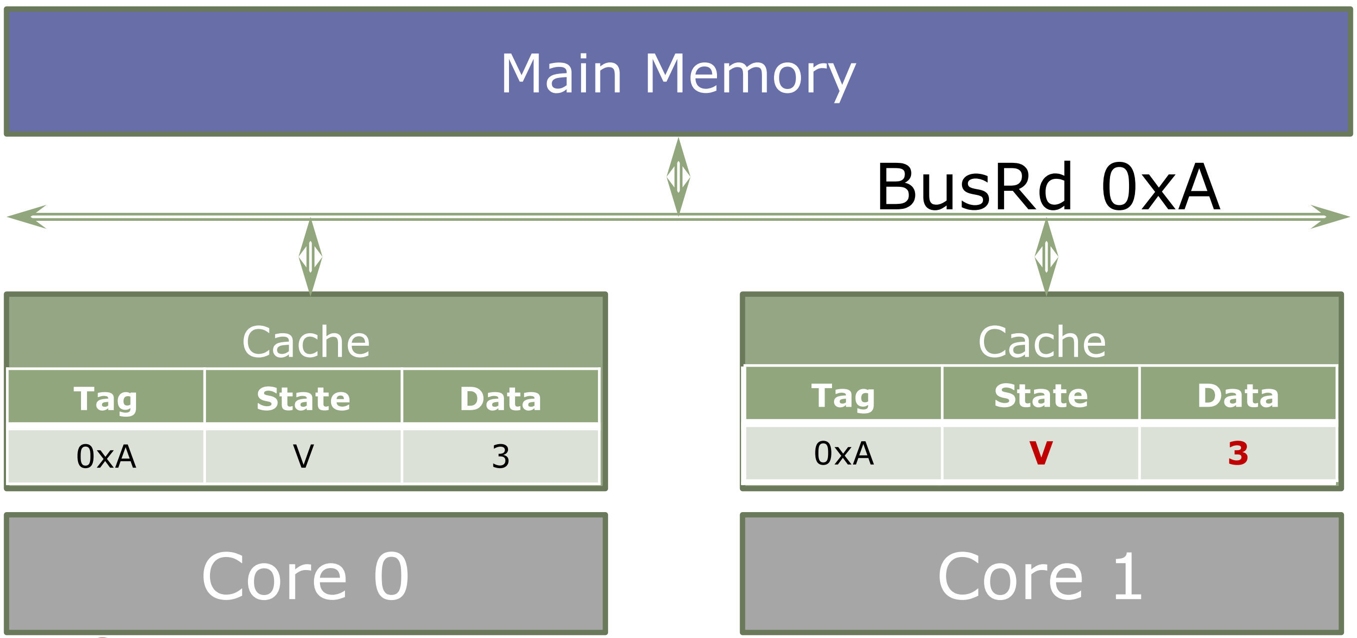


Additional loads satisfied locally, without BusRd

Valid/Invalid Example



Valid/Invalid Example



1 LD 0xA

2 LD 0xA

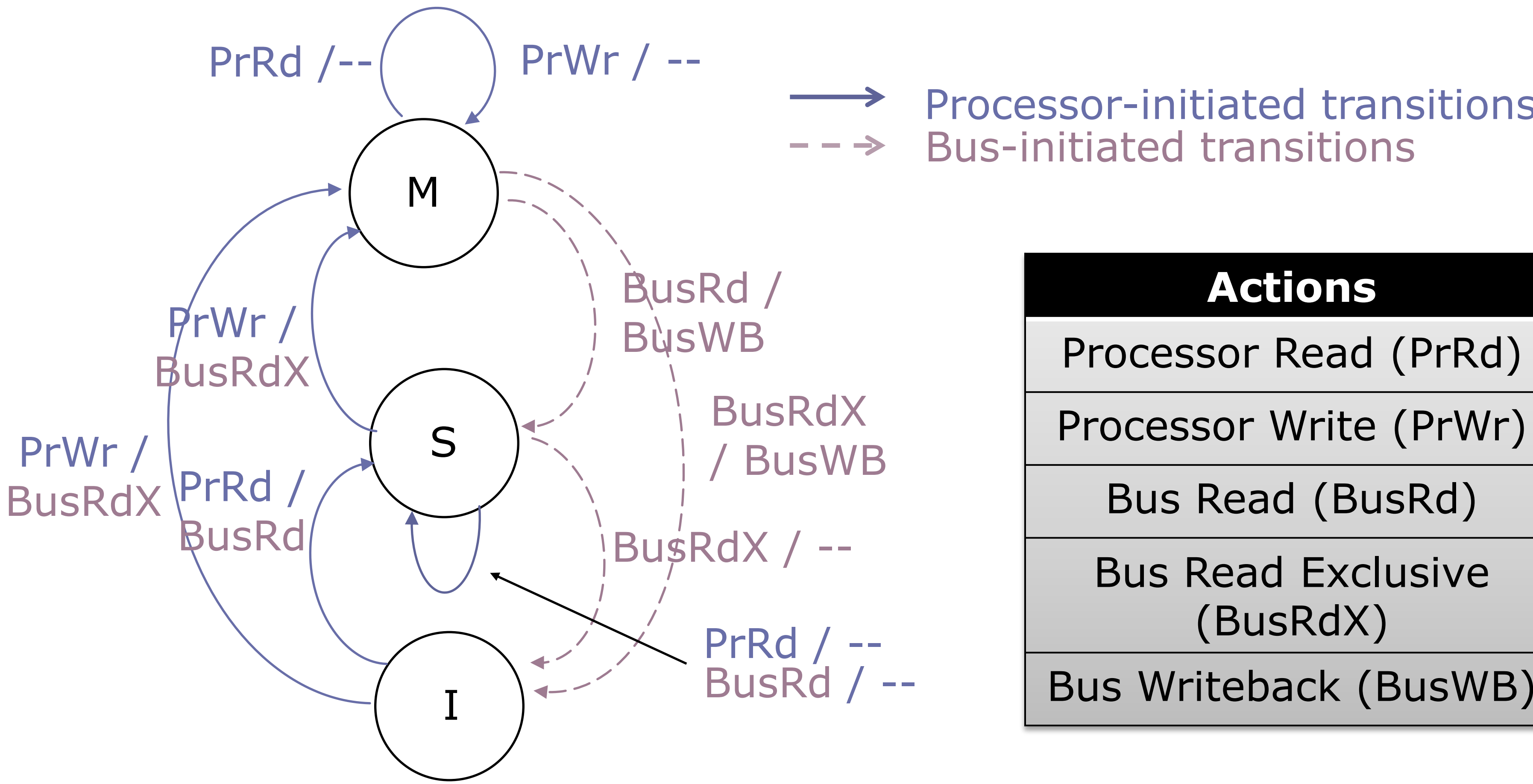
3 ST 0xA

4 LD 0xA

VI Problems? Every write updates main memory
Every write requires broadcast & snoop

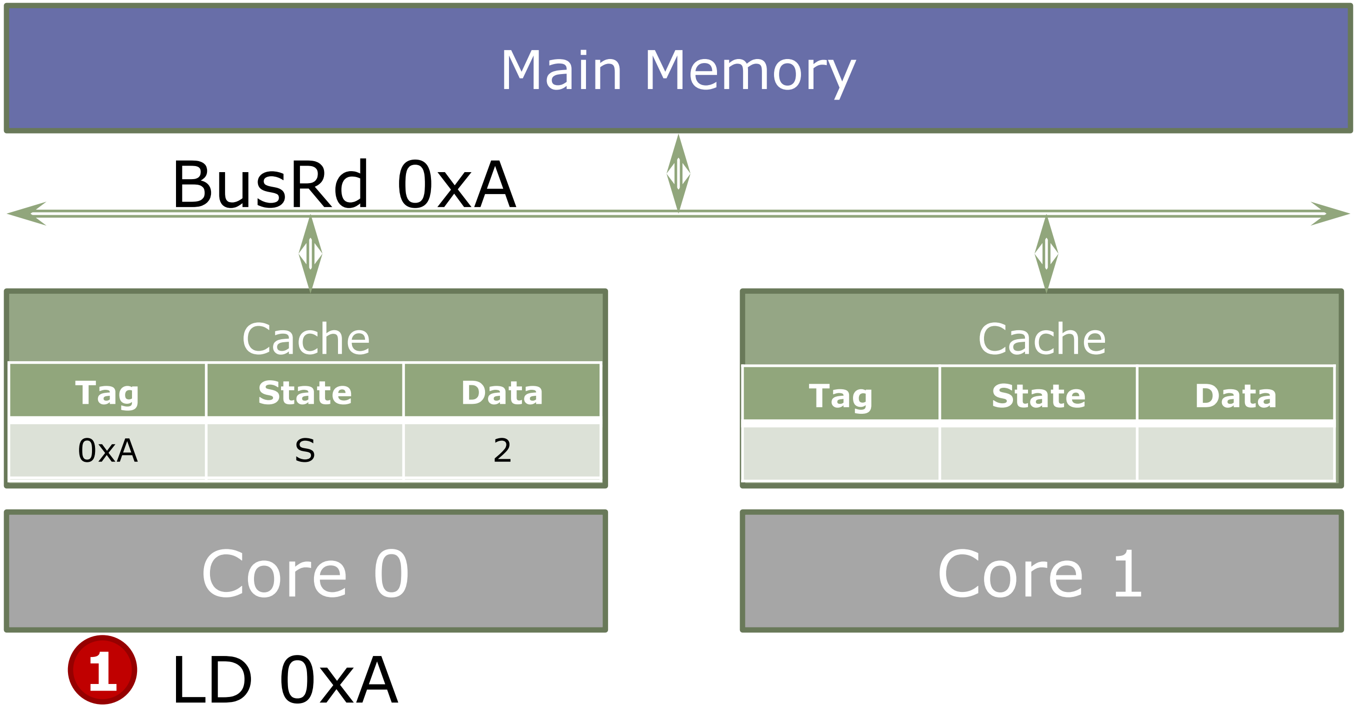
Modified/Shared/Invalid (MSI) Protocol

- Allows writeback caches + satisfying writes locally

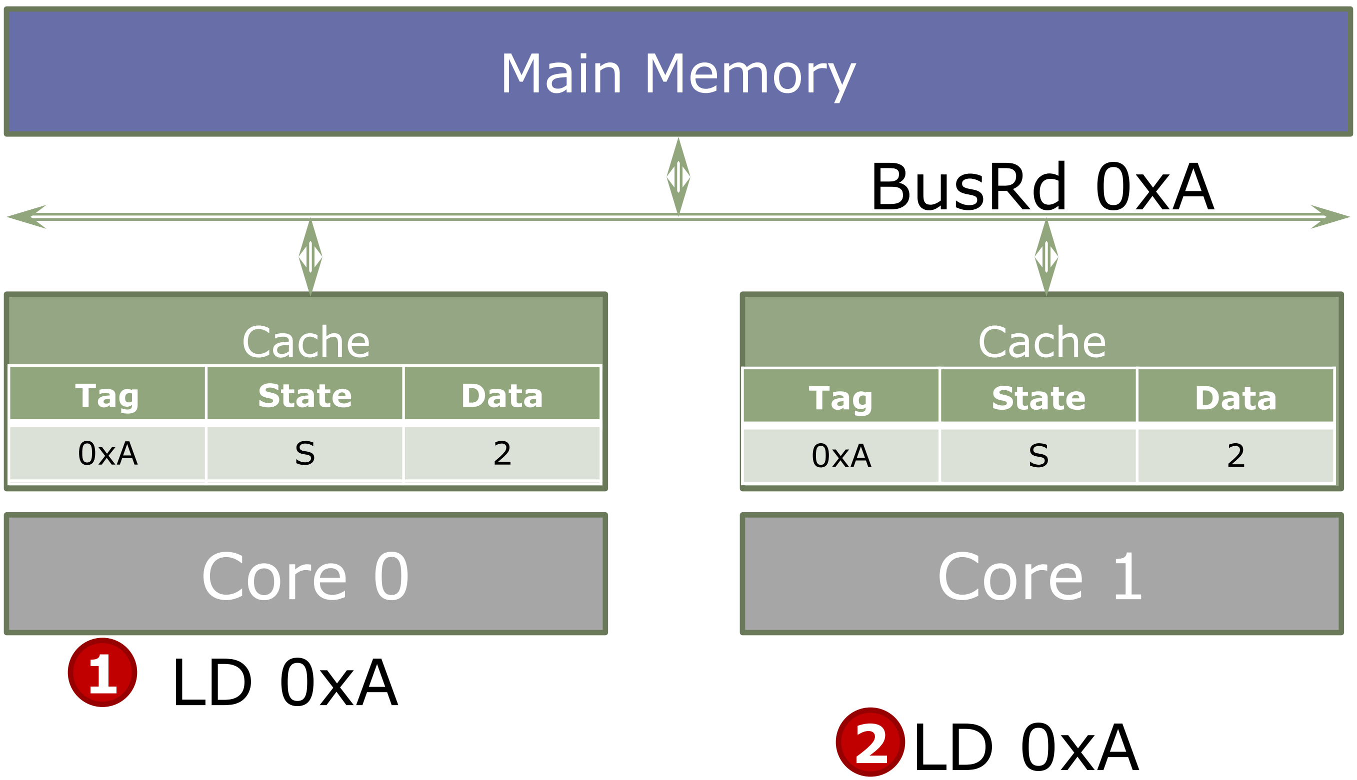


Actions
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Read Exclusive (BusRdX)
Bus Writeback (BusWB)

MSI Example

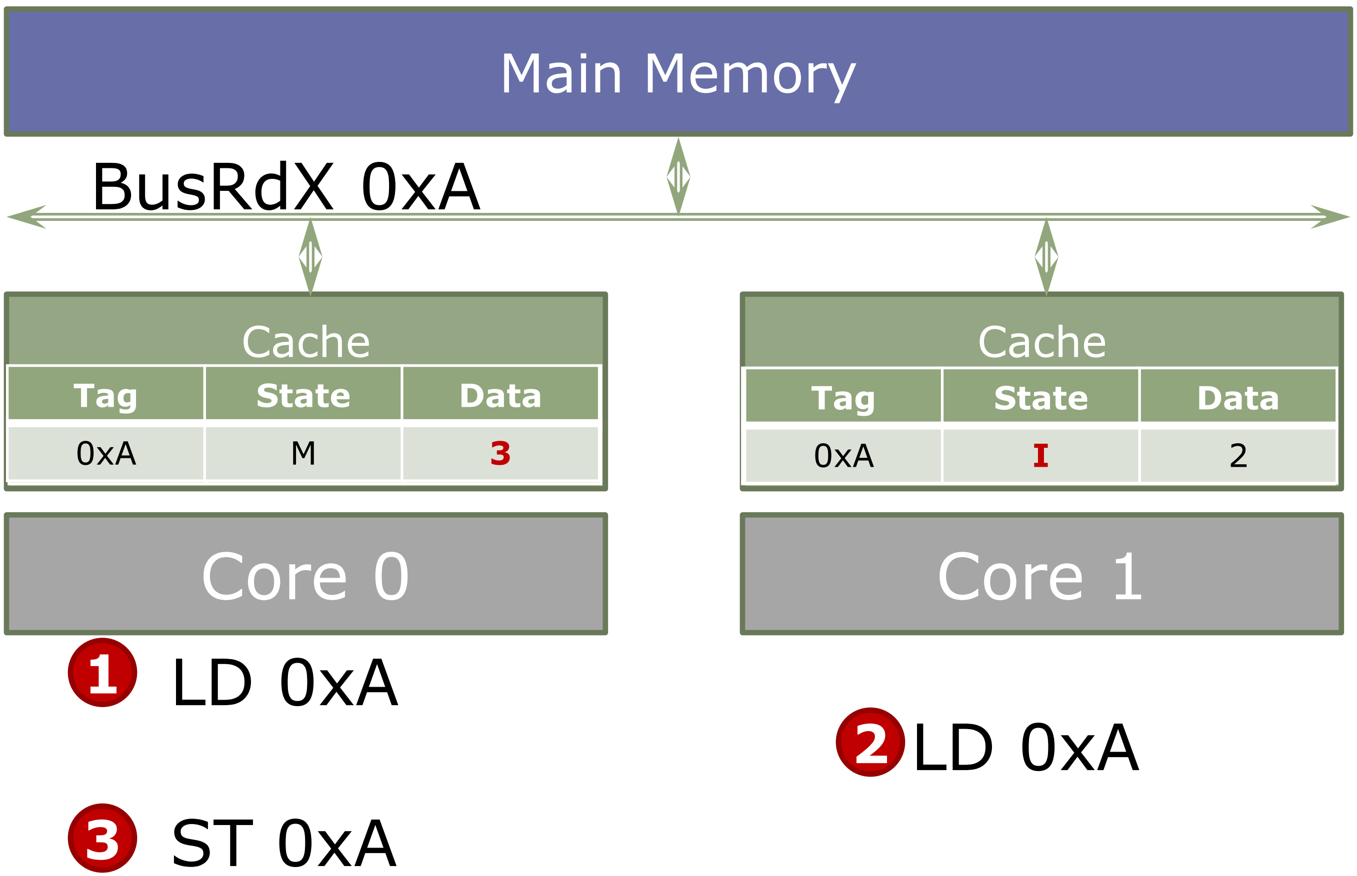

Main Memory
BusRd 0xA
Cache
Tag
State
Data
0xA
S
2
Cache
Tag
State
Data
Core 0
Core 1
1
LD 0xA

MSI Example



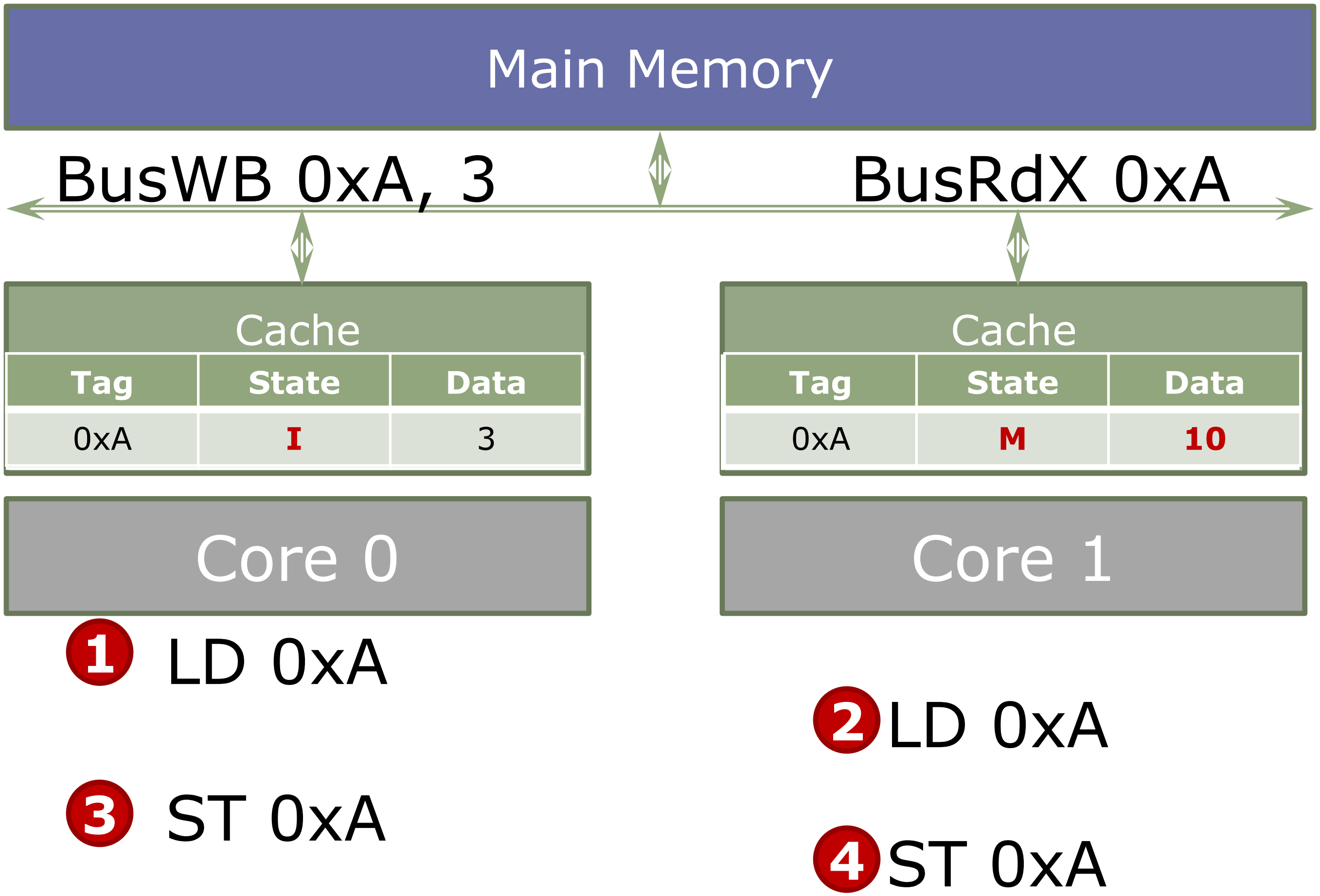
Additional loads satisfied locally, without BusRd (like in VI)

MSI Example

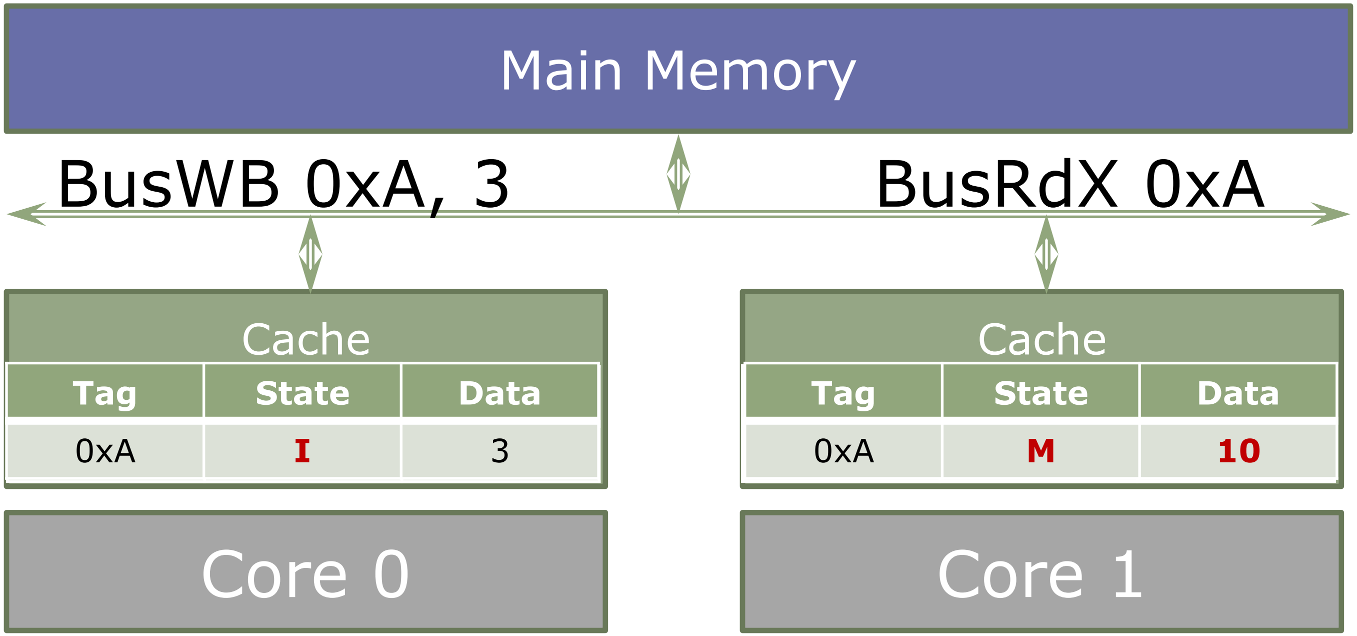


Additional loads *and stores* from core 0 satisfied locally, without bus transactions (unlike in VI)

MSI Example

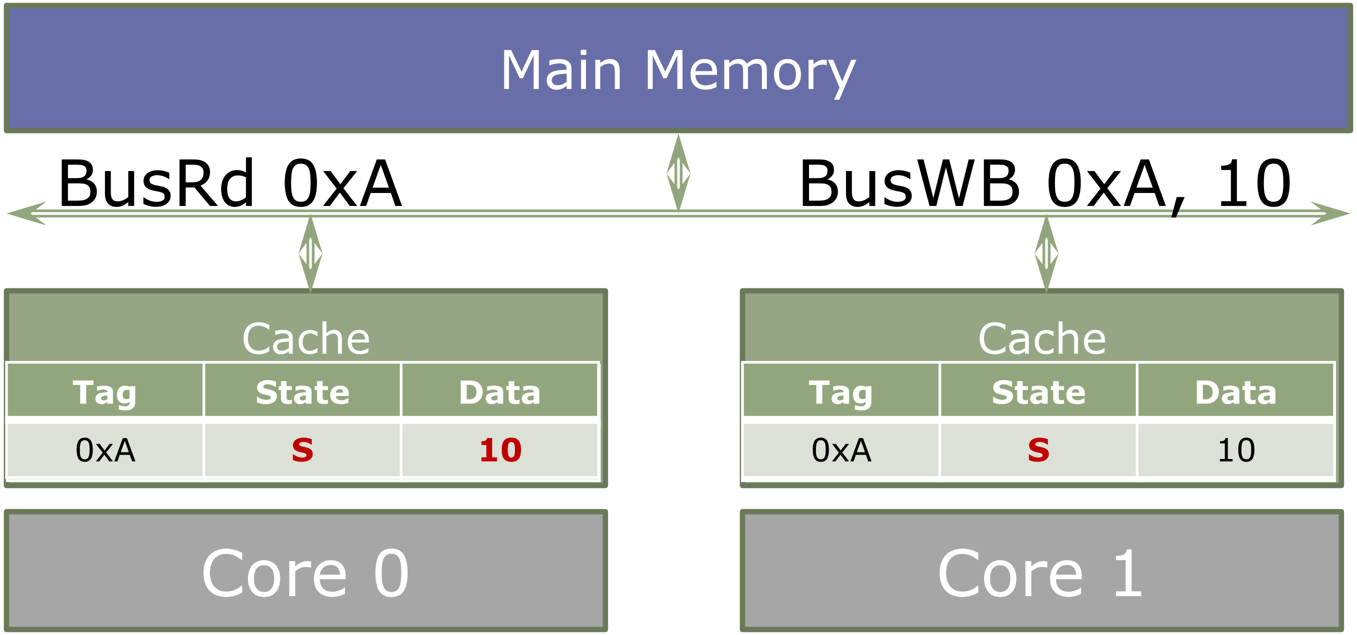

Main Memory
BusWB 0xA, 3
BusRdX 0xA
Cache
Tag
State
Data
0xA
I
3
Cache
Tag
State
Data
0xA
M
10
Core 0
Core 1
1 LD 0xA
2 LD 0xA
3 ST 0xA
4 ST 0xA

Cache interventions



- MSI allows caches to serve writes without updating memory, so main memory can have stale data
  - Core 0’s cache needs to supply data
  - But main memory may also respond!
- Cache must override response from main memory

MSI Example



Core 0 cache:

Tag	State	Data
0xA	**S**	**10**

Core 1 cache:

Tag	State	Data
0xA	**S**	10

Core 0:
1. LD 0xA
3. ST 0xA
5. LD 0xA

Core 1:
2. LD 0xA
4. ST 0xA

MSI Optimizations: Exclusive State

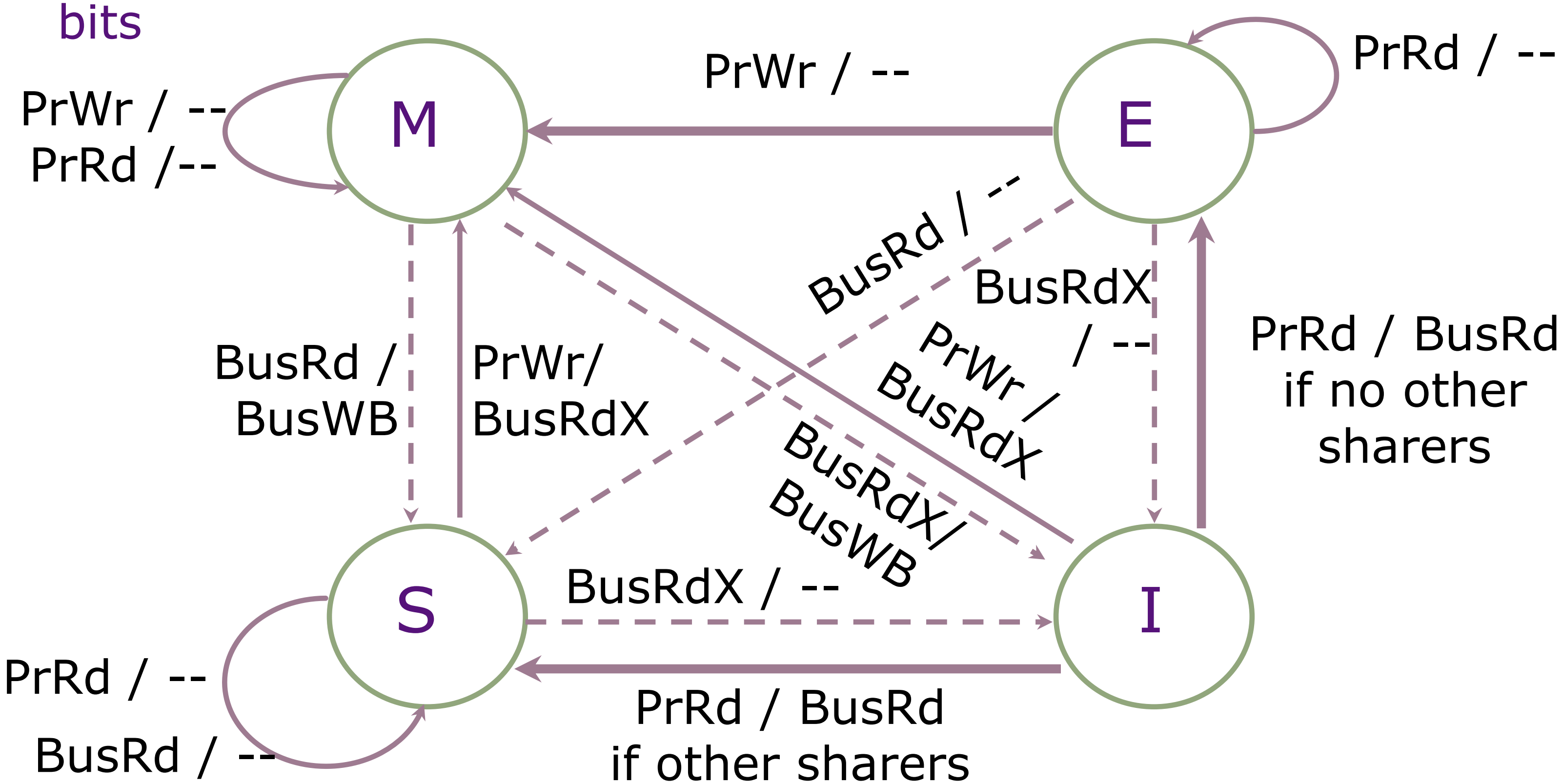
- Observation: Doing read-modify-write sequences on private data is common
  - What's the problem with MSI?

- Solution: E state (exclusive, clean)
  - If no other sharers, a read acquires line in E instead of S
  - Writes silently cause E→M (exclusive, dirty)

MESI: An Enhanced MSI protocol
increased performance for private read-write data

Each cache line has a tag

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



MSI Optimizations: Owner State

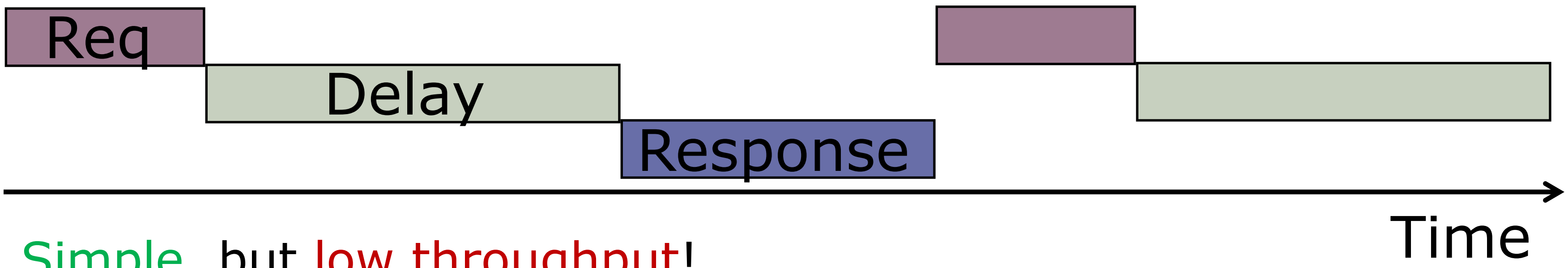
- Observation: On M→S transitions, must write back line!
  - What happens with frequent read-write sharing?
  - Can we defer the write after S?

- Solution: O state (Owner)
  - O = S + responsibility to write back
  - On M→S transition, one sharer (typically the one who had the line in M) retains the line in O instead of S
  - On eviction, O writes back line (or another sharer does S→O)

- MSI, MESI, MOSI, MOESI…
  - Typically E if private read-write >> shared read-only (common)
  - Typically O only if writebacks are expensive (main mem vs L3)

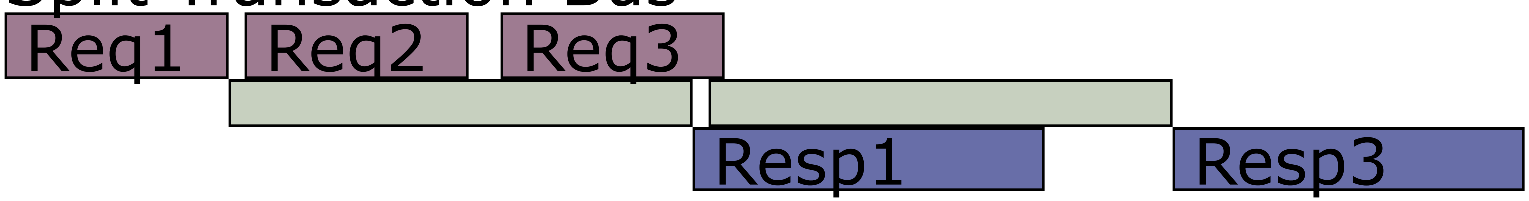
Split-Transaction and Pipelined Buses

Atomic Transaction Bus



Simple, but low throughput!

Split-Transaction Bus

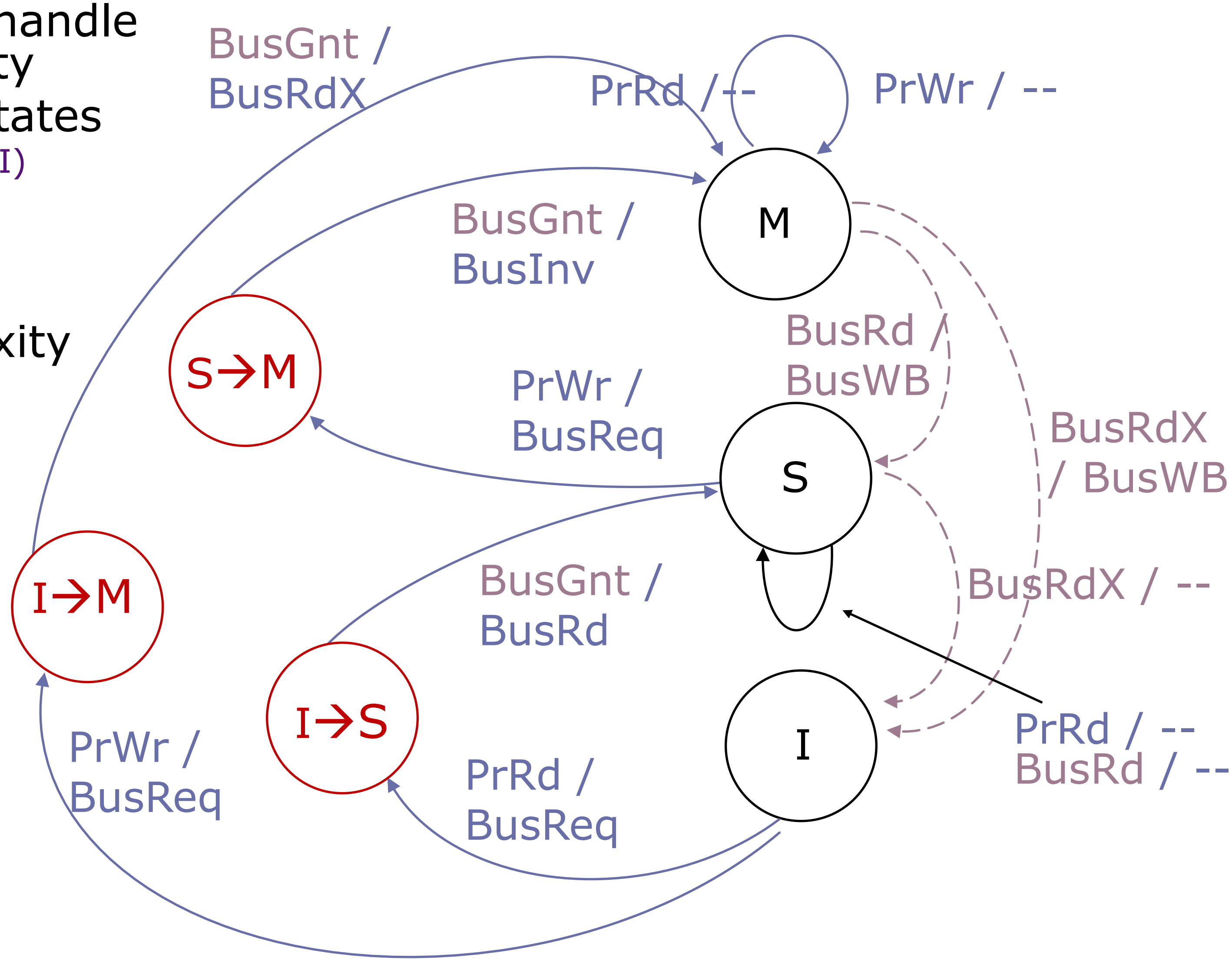


- Supports multiple simultaneous transactions
  - Higher throughput
  - Responses may arrive out of order
- Often implemented as multiple buses (req+resp)

Non-Atomicity → Transient States

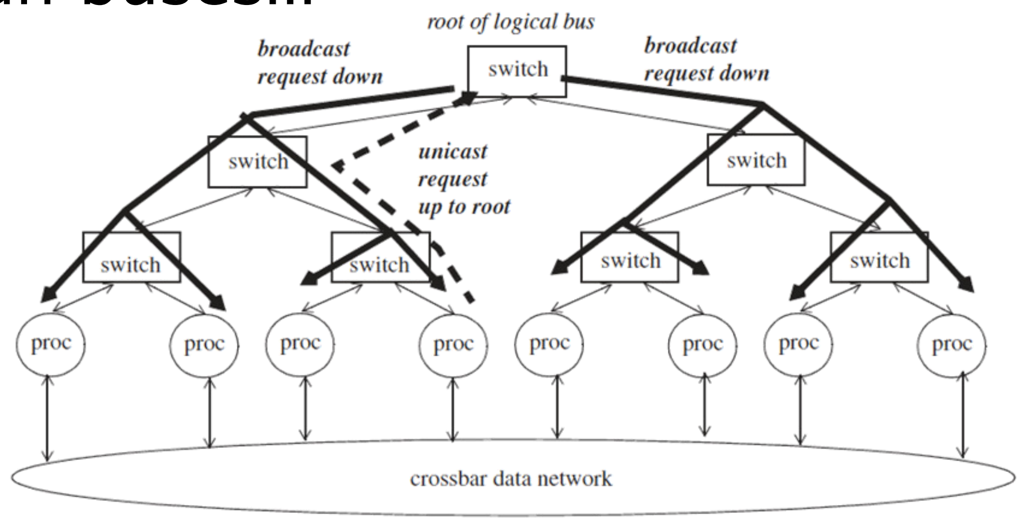
- Protocol must handle lack of atomicity
- Two types of states
  - Stable (e.g. MSI)
  - Transient
- Split + race transitions
- Higher complexity

Actions
Bus Request (BusReq)
Bus Grant (BusGnt)



Scaling Cache Coherence

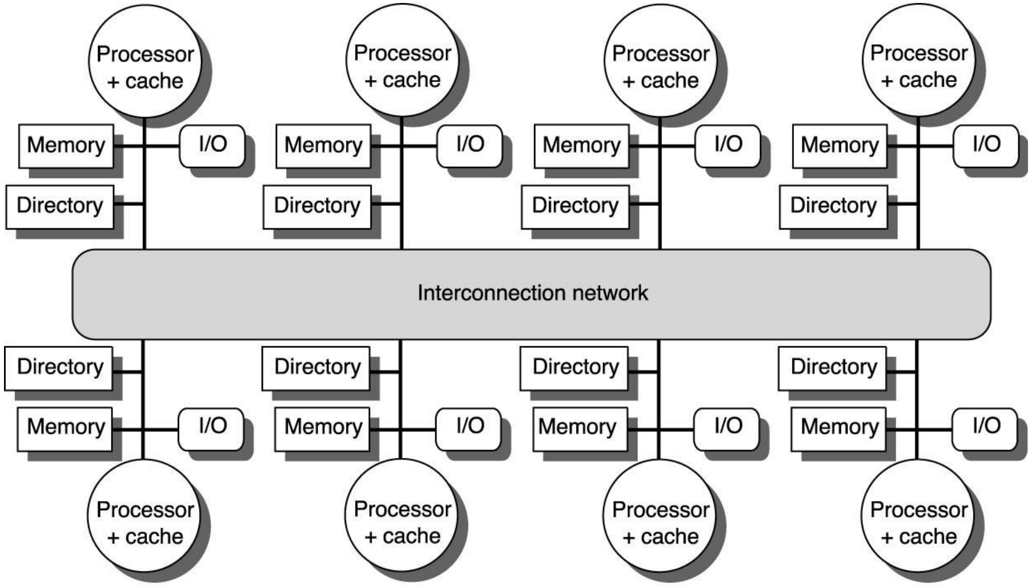
- Can implement ordered interconnects that scale better than buses…



Starfire E10000 (drawn with only eight processors for clarity). A coherence request is unicast up to the root, where it is serialized, before being broadcast down to all processors

- … but broadcast is fundamentally unscalable
  - Bandwidth, energy of transactions with 100s of cache snoops?

Directory-Based Coherence



- Route all coherence transactions through a directory
  - Tracks contents of private caches → No broadcasts
  - Serves as ordering point for conflicting requests → Unordered networks

(more on next lecture)

CC and False Sharing

Performance Issue - 1

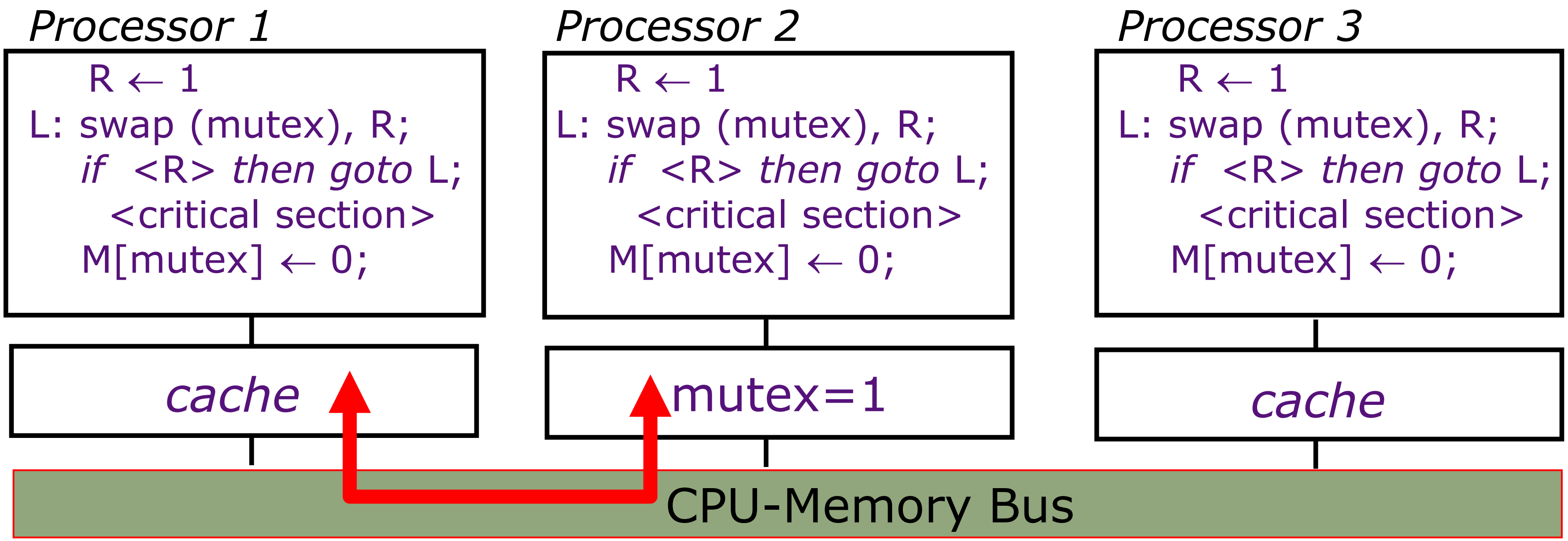
state	blk addr	data0	data1	...	dataN

A cache block contains more than one word and cache coherence is done at the block-level and not word-level

Suppose P_1 writes $word_i$ and P_2 writes $word_k$ and both words have the same block address.

What can happen? The block may be invalidated (ping-pong) many times unnecessarily because addresses are in the same block.

CC and Synchronization
Performance Issue - 2



Cache coherence protocols will cause mutex to *ping-pong* between P1’s and P2’s caches.

Ping-ponging can be reduced by first reading the mutex location *(non-atomically)* and executing a swap only if it is found to be zero (test&test&set).

CC and Bus Occupancy
Performance Issue - 3

In general, an atomic *read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors

In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation

$\Rightarrow$ expensive for simple buses

$\Rightarrow$ *very expensive* for split-transaction buses

modern processors use

load-reserve

store-conditional

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, (a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional (a), R:
    if <flag, adr> == <1, a>
    then  cancel other procs'
          reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to 0

- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

Performance:
Load-reserve & Store-conditional

The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases bus utilization* (and reduces processor stall time), especially in split-transaction buses
- *reduces cache ping-pong effect* because processors trying to acquire a mutex do not have to perform stores each time