

Beyond ILP

In Search of More Parallelism

Nima Honarmand

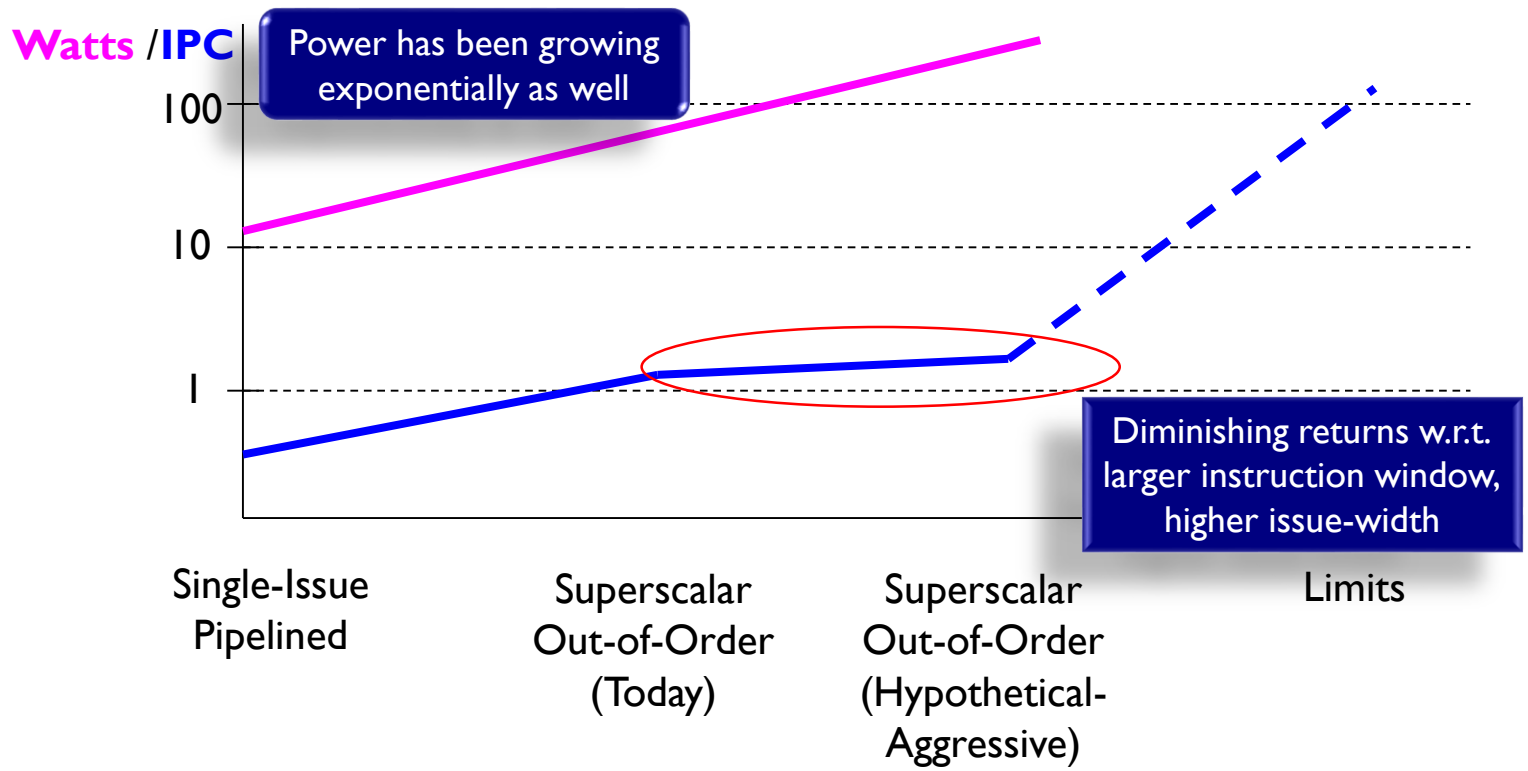
Getting More Performance

- OoO superscalars extract ILP from sequential programs
 - Hardly more than 1-2 IPC on real workloads
 - Although some studies suggest ILP degrees of 10's-100's
- In practice, IPC is limited by:
 - Limited BW
 - From memory and cache
 - Fetch/commit bandwidth
 - Renaming (must find dependences among all insns dispatched in a cycle)
 - Limited HW resources
 - # renaming registers, ROB, RS and LSQ entries, functional units
 - True data dependences
 - Coming from algorithm and compiler
 - Branch prediction accuracy
 - Imperfect memory disambiguation

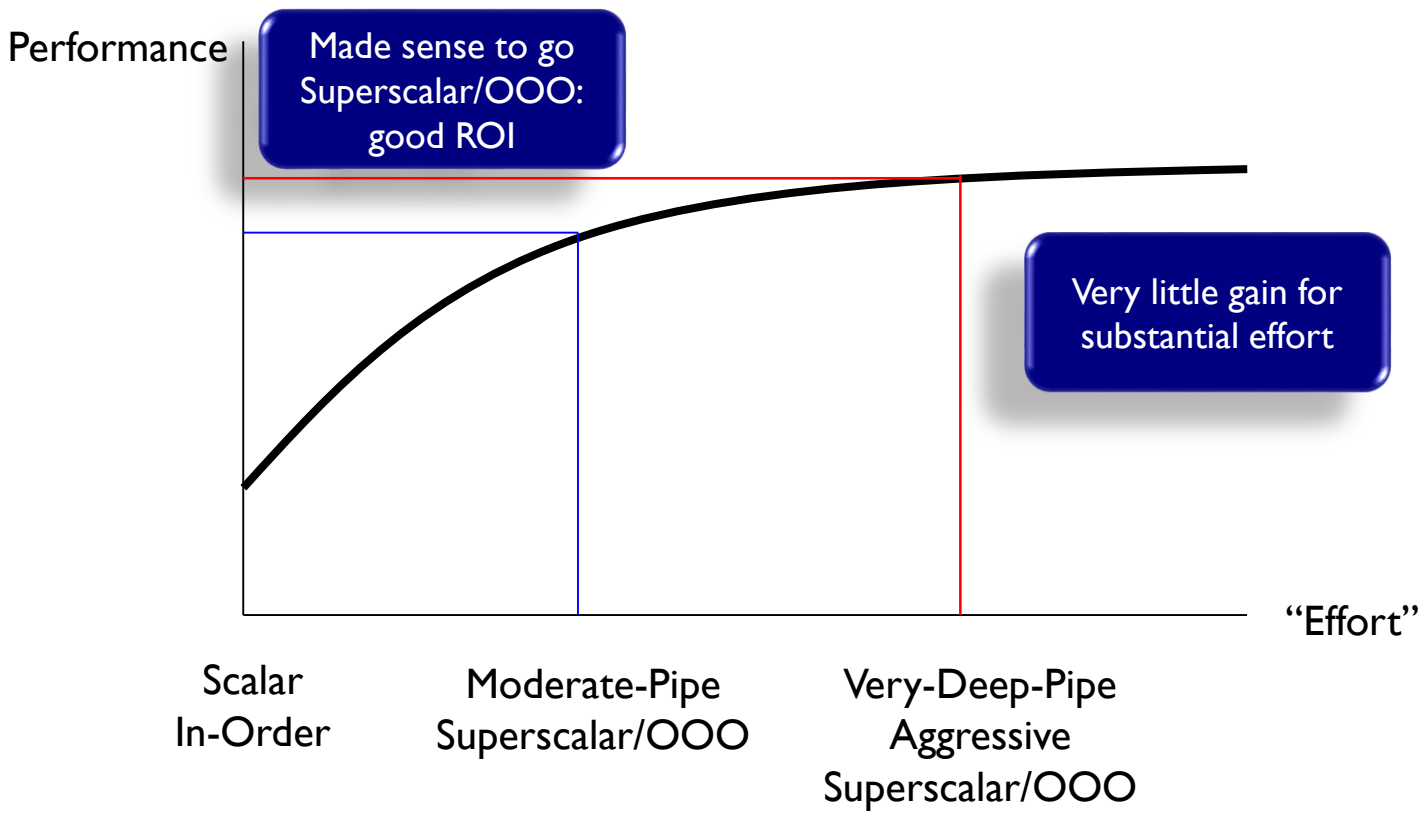
Getting More Performance

- Keep pushing IPC and/or frequency
 - Design complexity (time to market)
 - Cooling (cost)
 - Power delivery (cost)
 - ...
- Possible, but too costly

Bridging the Gap



Higher Complexity not Worth Effort



User Visible/Invisible (1/2)

- Problem: HW is in charge of finding parallelism
 - **User-invisible parallelism**
 - Most of what of what we discussed in the class so far!
- Users got “free” performance just by buying a new chip
 - No change needed to the program (same ISA)
 - Higher frequency & higher IPC (different micro-arch)
 - But this was not sustainable...

User Visible/Invisible (2/2)

- Alternative: **User-visible parallelism**
 - User (developer) responsible for finding and expressing parallelism
 - HW does not need to find parallelism → Simpler, more efficient HW
- Common forms
 - ***Thread-Level Parallelism (TLP)***: Multiprocessors, Hardware Multithreading
 - ***Data-Level Parallelism (DLP)***: Vector processors, SIMD extensions, GPUs
 - ***Request-Level Parallelism (RLP)***: Data centers

Thread-Level Parallelism (TLP)

Sources of TLP

- Different applications
 - MP3 player in background while you work in Office
 - Other background tasks: OS/kernel, virus check, etc...
 - Piped applications
 - `gunzip -c foo.gz | grep bar | perl some-script.pl`
- Threads within the same application
 - Explicitly coded multi-threading
 - `pthread`s
 - Parallel languages and libraries
 - OpenMP, Cilk, TBB, etc...

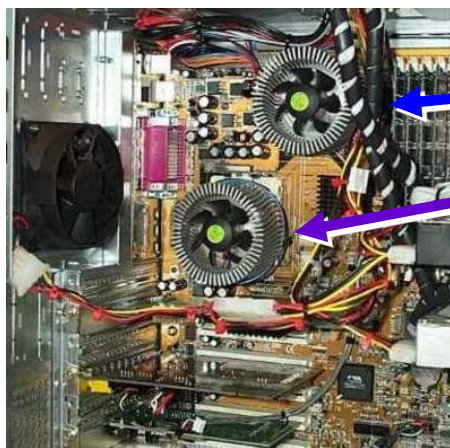
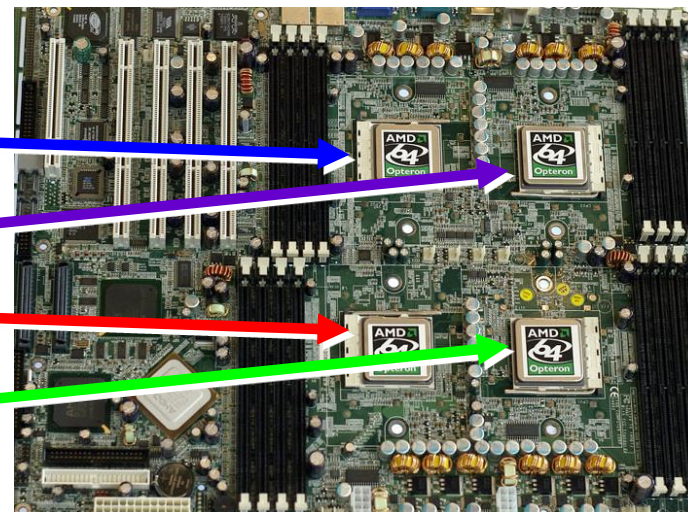
Architectures to Exploit TLP

- ***Multiprocessors (MP)***: Different threads run on different processors
 - Symmetric Multiprocessors (SMP)
 - Chip Multiprocessors (CMP)
- ***Hardware Multithreading (MT)***: Multiple threads share the same processor pipeline
 - Coarse-grained MT (CGMT)
 - Fine-grained MT (FMT)
 - Simultaneous MT (SMT)

Multiprocessors (MP)

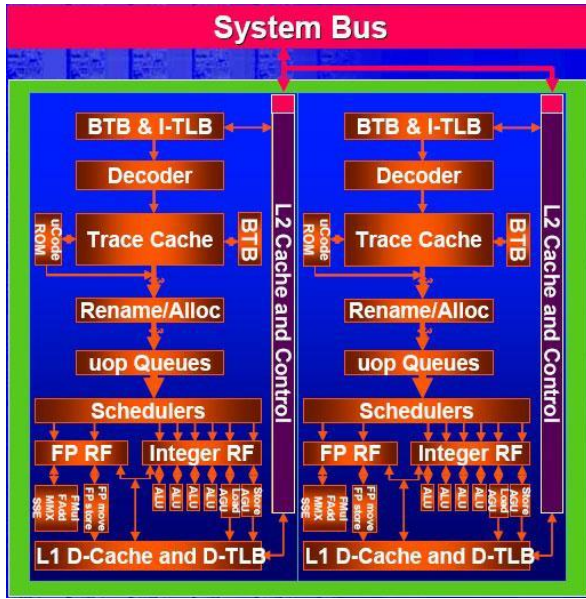
SMP Machines

- SMP = Symmetric Multi-Processing
 - Symmetric = All CPUs are the same and have “equal” access to memory
 - All CPUs are treated as similar by the OS
 - E.g.: no master/slave, no bigger or smaller CPUs, ...
- OS sees multiple CPUs
 - Runs one process (or thread) on each CPU

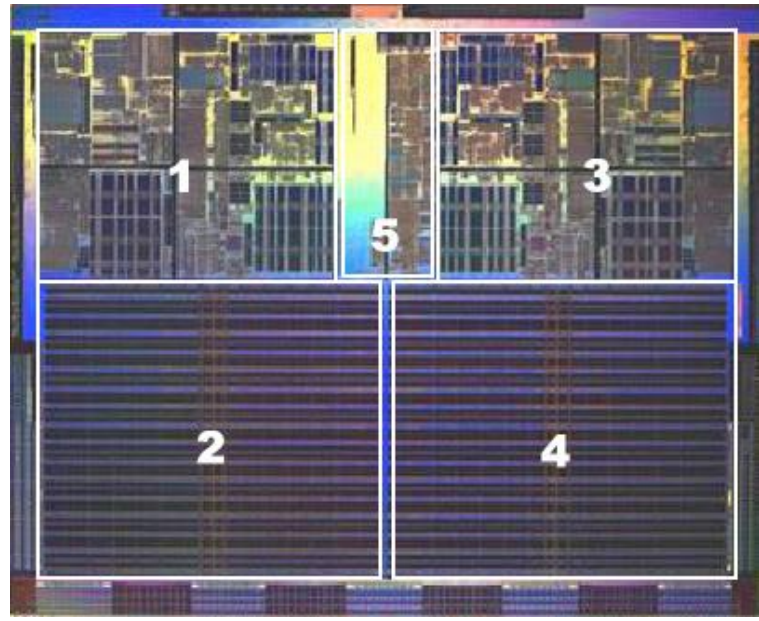
CPU₀CPU₁CPU₂CPU₃

Chip-Multiprocessing (CMP)

- Simple SMP on the same chip
 - CPUs now called “cores” by hardware designers
 - OS designers still call these “CPUs”



Intel “Smithfield” (Pentium D) Block Diagram



AMD Dual-Core Athlon FX

Benefits of CMP

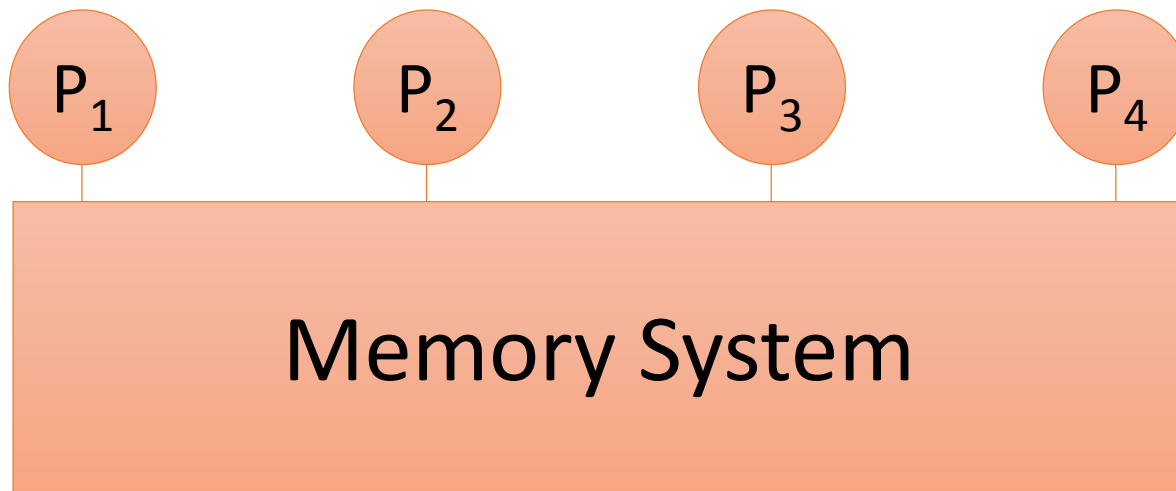
- Cheaper than multi-chip SMP
 - All/most interface logic integrated on chip
 - Fewer chips
 - Single CPU socket
 - Single interface to memory
 - Less power than multi-chip SMP
 - Communication on die uses less power than chip to chip
- Efficiency
 - Use transistors for multiple cores (instead of wider/more aggressive OoO)
 - Potentially better use of hardware resources

CMP Performance vs. Power

- 2x CPUs not necessarily equal to 2x performance
- 2x CPUs \rightarrow $\frac{1}{2}$ power for each
 - Maybe a little better than $\frac{1}{2}$ if resources can be shared
- Back-of-the-Envelope calculation:
 - 3.8 GHz CPU at 100W
 - Dual-core: 50W per Core
 - $P \propto V^3$: $V_{\text{orig}}^3 / V_{\text{CMP}}^3 = 100\text{W} / 50\text{W} \rightarrow V_{\text{CMP}} = 0.8 V_{\text{orig}}$
 - $f \propto V$: $f_{\text{CMP}} = 3.0\text{GHz}$

Shared-Memory Multiprocessors

- Multiple threads use shared memory (address space)
 - “System V Shared Memory” or “Threads” in software
- Communication implicit via loads and stores
 - Opposite of explicit message-passing multiprocessors



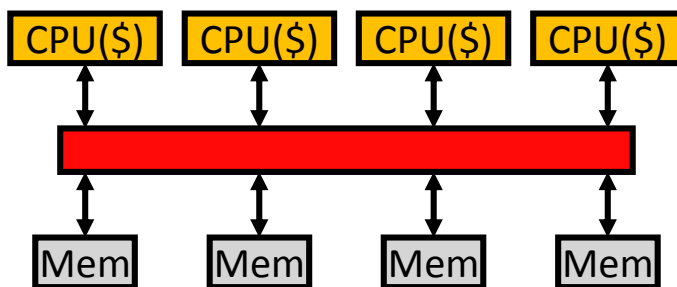
Why Shared Memory?

- Pluses
 - + Programmers don't need to learn about explicit communications
 - Because communication is implicit (through memory)
 - + Applications similar to the case of multitasking uniprocessor
 - Programmers already know about synchronization
 - + OS needs only evolutionary extensions
- Minuses
 - Communication is hard to optimize
 - Because it is implicit
 - Not easy to get good performance out of shared-memory programs
 - Synchronization is complex
 - Over-synchronization → bad performance
 - Under-synchronization → incorrect programs
 - **Very** difficult to debug
 - Hard to implement in hardware

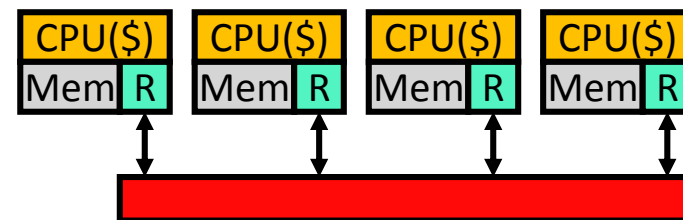
Result: the most popular form of parallel programming

Uniformity in Memory Access

- **Uniform Memory Access (UMA)**: Equal latency to memory from all processors
 - Simpler software, doesn't matter where you put data
 - Lower peak performance
 - Common in bus-based systems
- **Non-Uniform Memory Access (NUMA)**: Local memory access faster than remote
 - More complex software: where you put data matters
 - Higher peak performance: assuming proper data placement



UMA Example

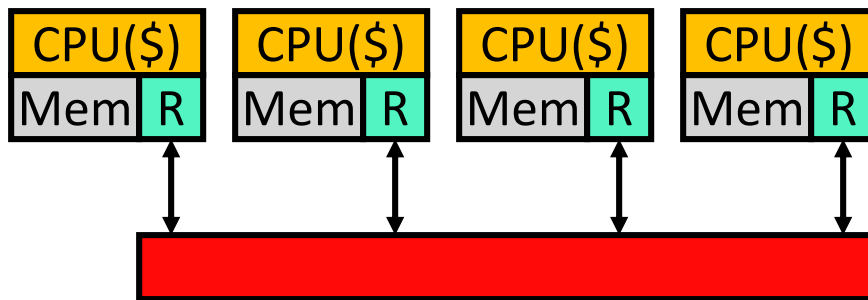


NUMA Example

Shared vs. Point-to-Point Networks

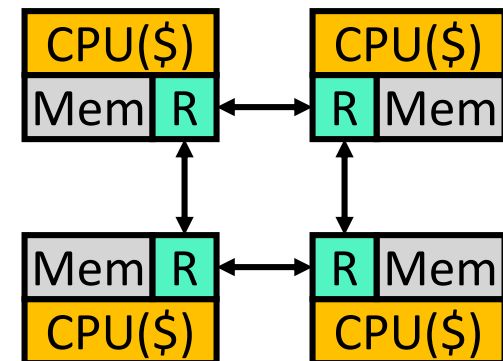
- Shared network

- Example: bus
- Low latency
- Low bandwidth
 - Doesn't scale > ~16 cores
- Simpler cache coherence



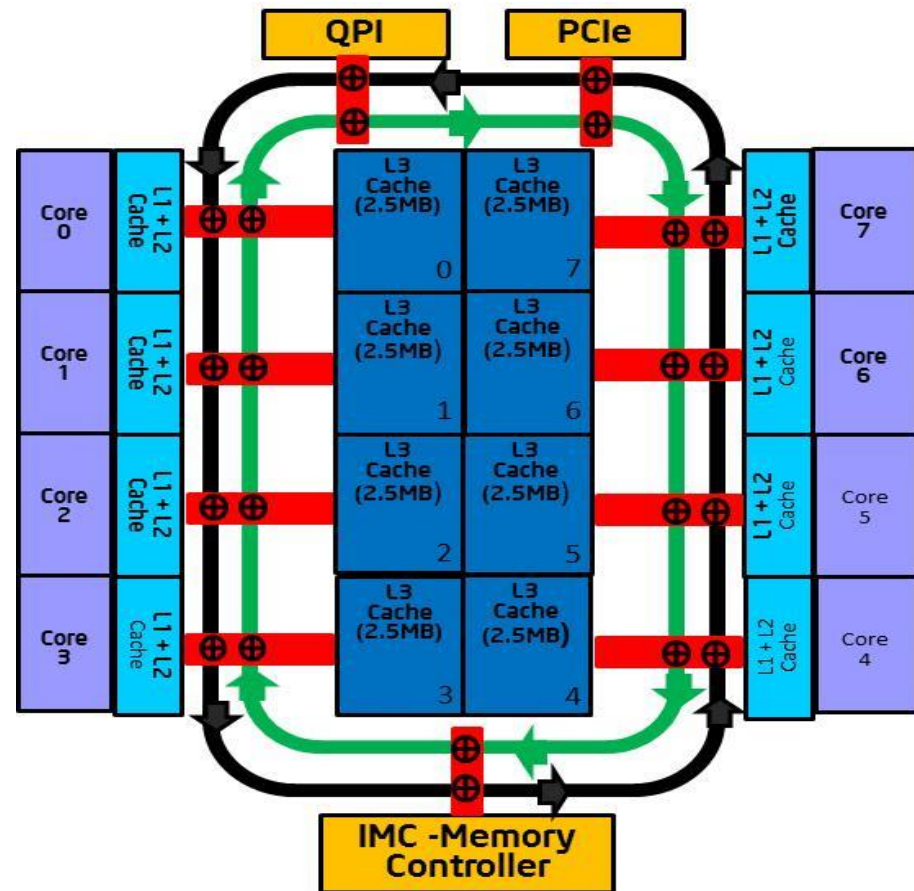
- Point-to-point network:

- Example: mesh, ring
- High latency (many *“hops”*)
- Higher bandwidth
 - Scales to 1000s of cores
- Complex cache coherence



Example: Shared Interconnect

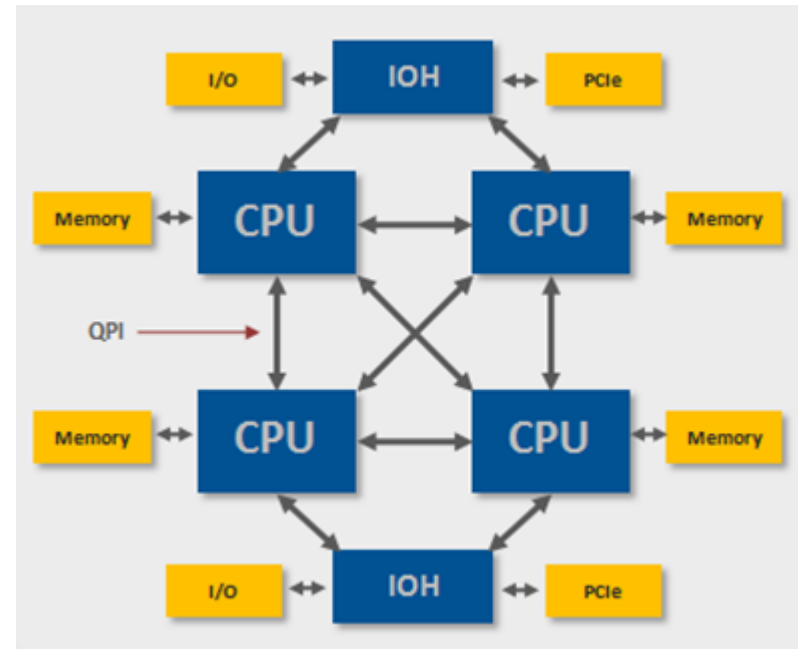
- Intel Xeon[®] E5-2600 family
 - Multi-ring interconnect
 - Connecting 8 cores and 8 banks of L3
 - On-Chip interconnect



Source: <https://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview>

Example: P2P Interconnect

- Intel Quick Path Interconnect (QPI)
 - Off-chip interconnect
 - Fully connected
 - Connecting processor sockets to each other and IO hubs
 - Memory directly connected to processor sockets using a memory bus



Source: <http://www.ni.com/white-paper/11266/en/>

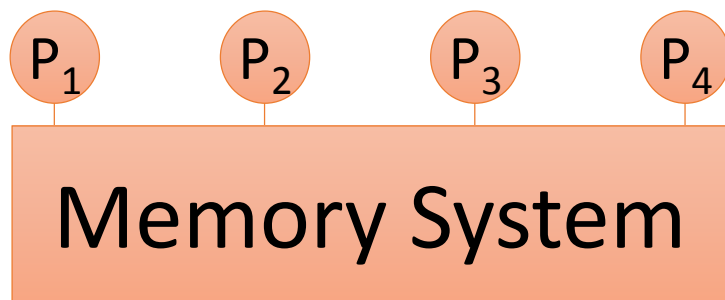
Issues for Shared Memory Systems

- Two big ones
 - *Cache coherence*
 - *Memory consistency model*
- Closely related
 - *But often confused*
- Covered in great detail in CSE 610

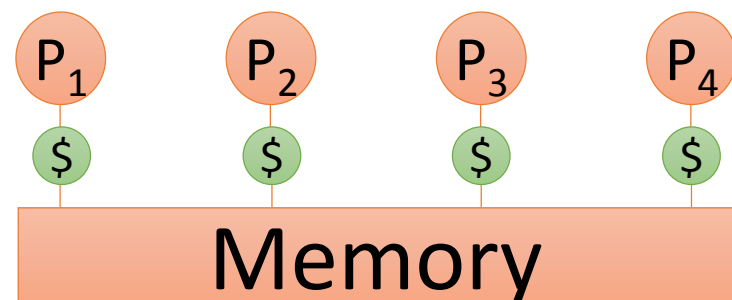
Cache Coherence

Cache Coherence: The Problem (1/3)

- Multiple copies of each cache block
 - One in main memory
 - Up to one in each cache
- Multiple copies can get inconsistent when writes happen
 - Should make sure all processors have a consistent view of memory



Logical View

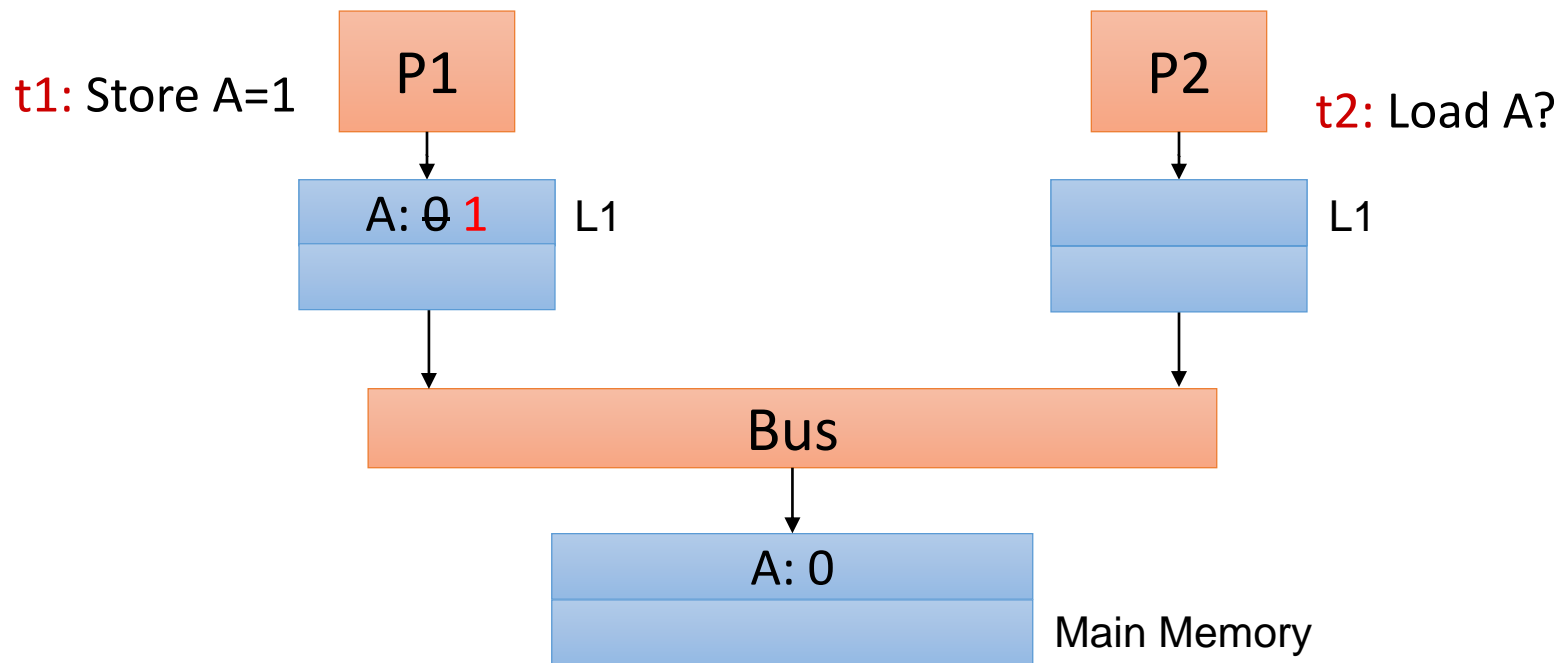


Reality (more or less!)

Should propagate one processor's write to others

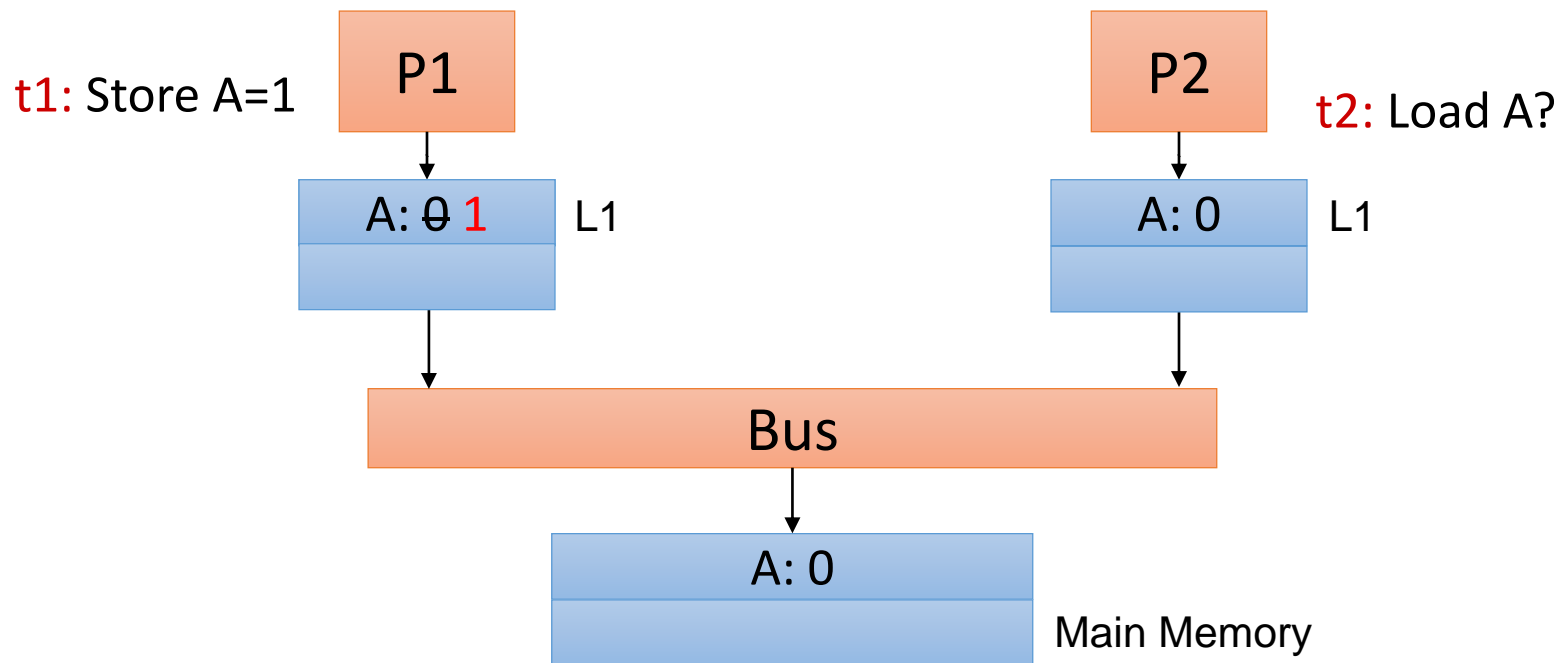
Cache Coherence: The Problem (2/3)

- Variable A initially has value 0
- P1 stores value 1 into A
- P2 loads A from memory and sees old value 0



Cache Coherence: The Problem (3/3)

- P1 and P2 both have variable A (value 0) in their caches
- P1 stores value 1 into A
- P2 loads A from its cache and sees old value 0



Need to do something to keep P2's cache coherent

Software Cache Coherence

- Software-based solutions
 - Mechanisms:
 - Add “Flush” and “Invalidate” instructions
 - “Flush” writes all (or some specified) dirty lines in my \$ to memory
 - “Invalidate” invalidate all (or some specified) valid lines in my \$
 - Could be done by compiler or run-time system
 - Should know what memory ranges are shared and which ones are private (i.e., only accessed by one thread)
 - Should properly use “invalidate” and “flush” instructions at “communication” points
 - Difficult to get perfect
 - Can induce a lot of unnecessary “flush”es and “invalidate”s → reducing cache effectiveness

Hardware Cache Coherence

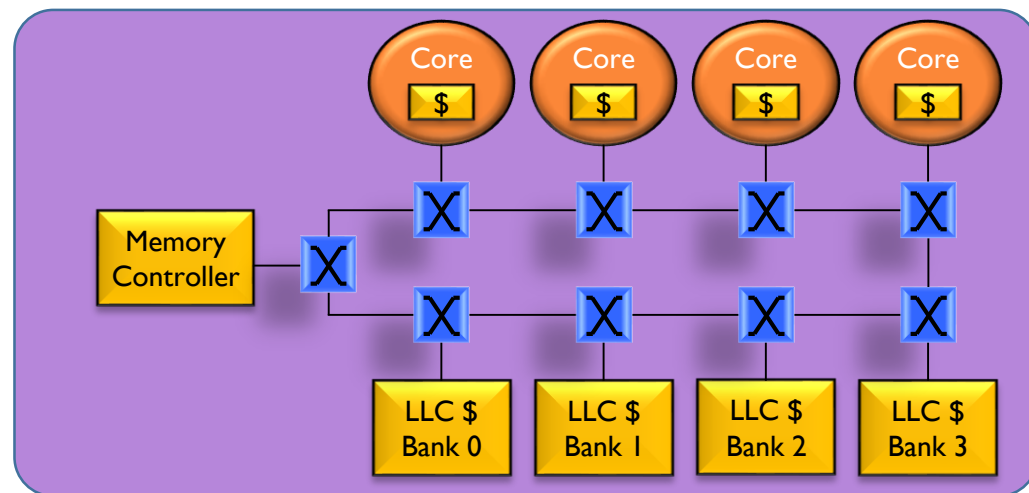
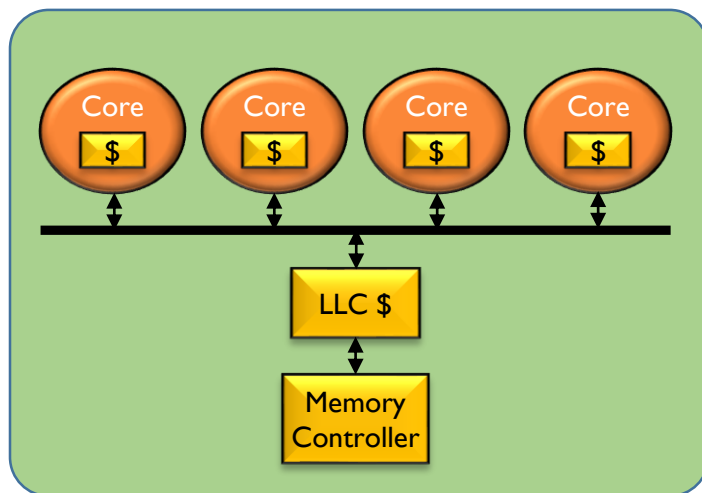
- Hardware solutions are far more common
 - System ensures everyone always sees the latest value

Two important aspects

- **Update vs. Invalidate**: on a write
 - update other copies, or
 - invalidate other copies
- **Broadcast vs. multicast**: send the update/invalidate...
 - to all other processors (aka **snoopy coherence**) , or
 - only those that have a cached copy of the line (aka **directory coherence** or **scalable coherence**)
- Invalidation protocols are far more common (our focus)

Snoopy Protocols

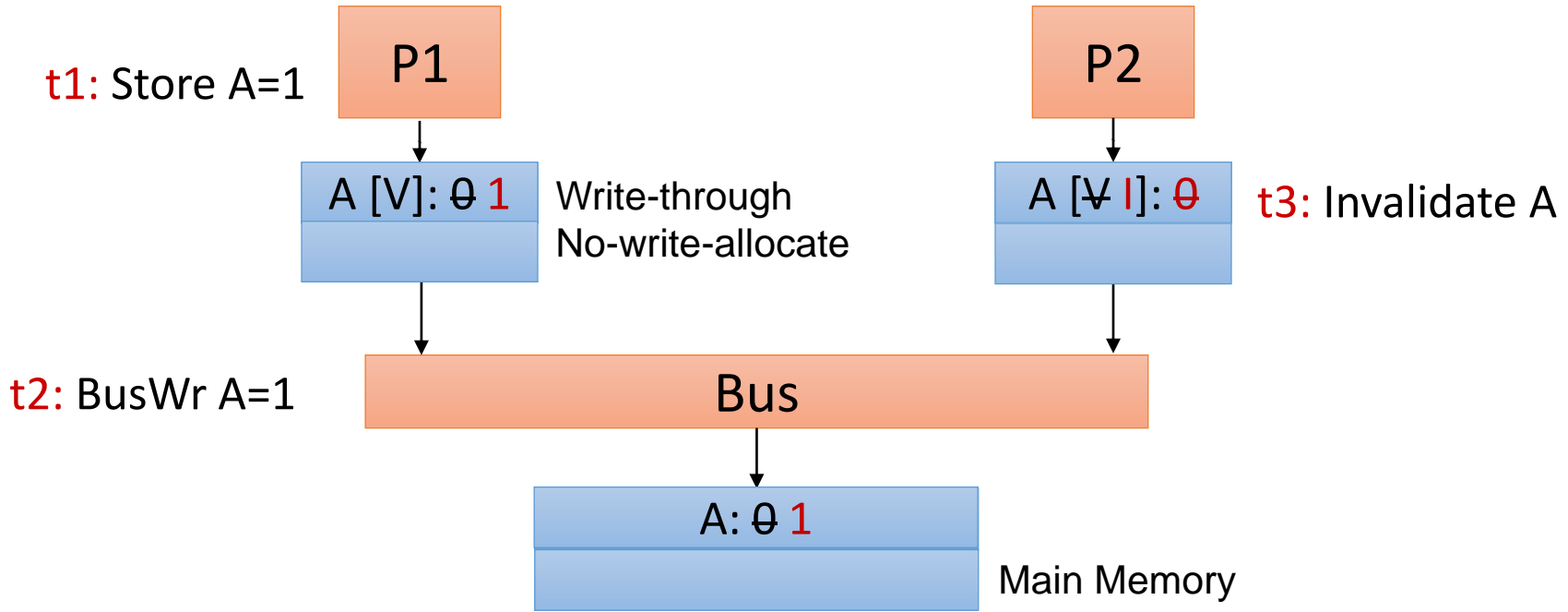
- Rely on broadcast based interconnection network between caches
 - Typically **Bus** or **Ring**



- **All caches** must monitor (aka “snoop”) **all traffic**
 - And keep track of cache line states based on the observed traffic

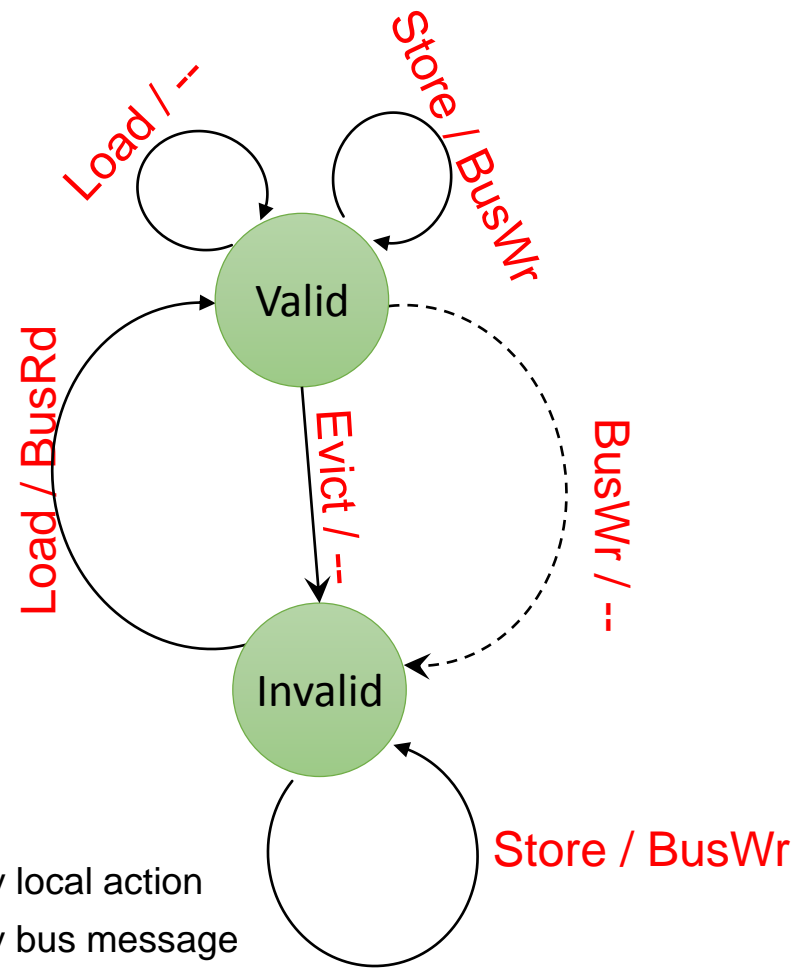
Example 1: Snoopy w/ Write-through \$

- Assume Write-through, no-write-allocate cache
- Allows multiple readers, but writes through to bus
- Simple state machine for each cache frame



Valid/Invalid Snooping Protocol

- 1 bit to tack coherence state per cache frame
 - Valid/Invalid
- Processor Actions
 - Ld, St, Evict
- Bus Messages
 - BusRd, BusWr



Example 2: Supporting Write-Back \$

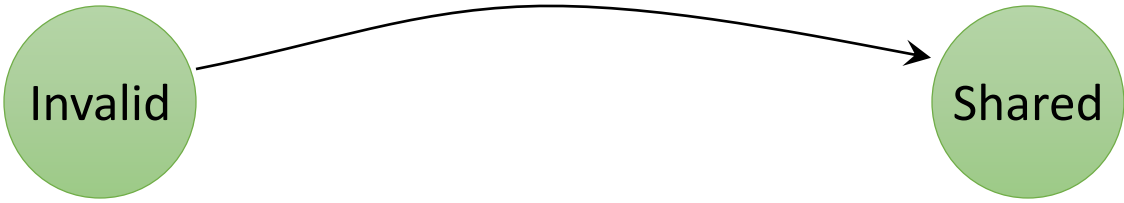
- Write-back caches are good
 - Drastically reduce bus write bandwidth
- Add notion of “ownership” to Valid/Invalid
 - The “owner” has the only replica of a cache block
 - Can update it freely
 - On a read, system must check if there is an owner
 - If yes, take away ownership and owner becomes a sharer
 - The reader becomes another sharer
 - Multiple sharers are ok
 - None is allowed to write without gaining ownership

Modified/Shared/Invalid (MSI) States

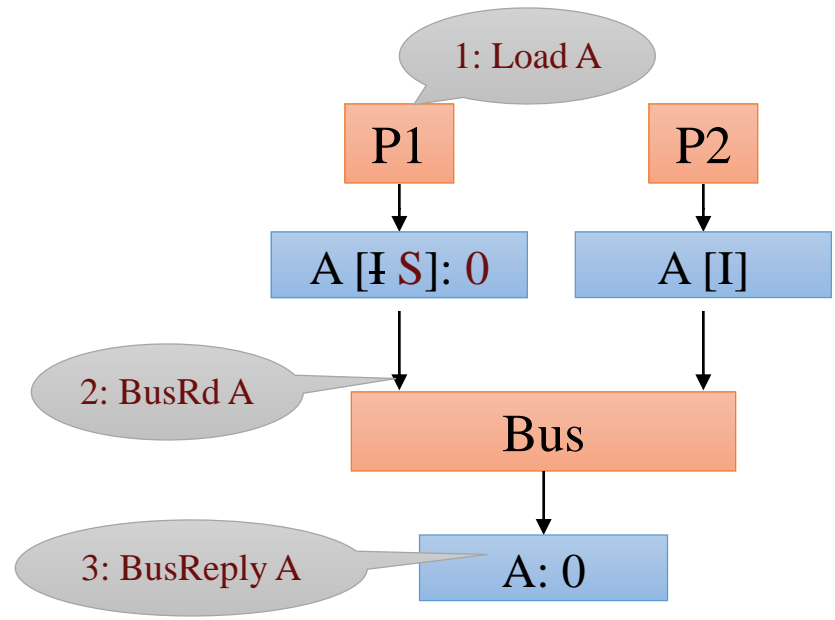
- Track 3 states per cache frame
 - Invalid: cache does not have a copy
 - Shared: cache has a read-only copy; clean
 - Clean: memory (or later caches) is up to date
 - Modified: cache has the only valid copy; writable; dirty
 - Dirty: memory (or lower-level caches) out of date
- Processor Actions
 - Load, Store, Evict
- Bus Messages
 - BusRd, BusRdX, BusInv, BusWB, BusReply
(Here for simplicity, some messages can be combined)

Simple MSI Protocol (1/9)

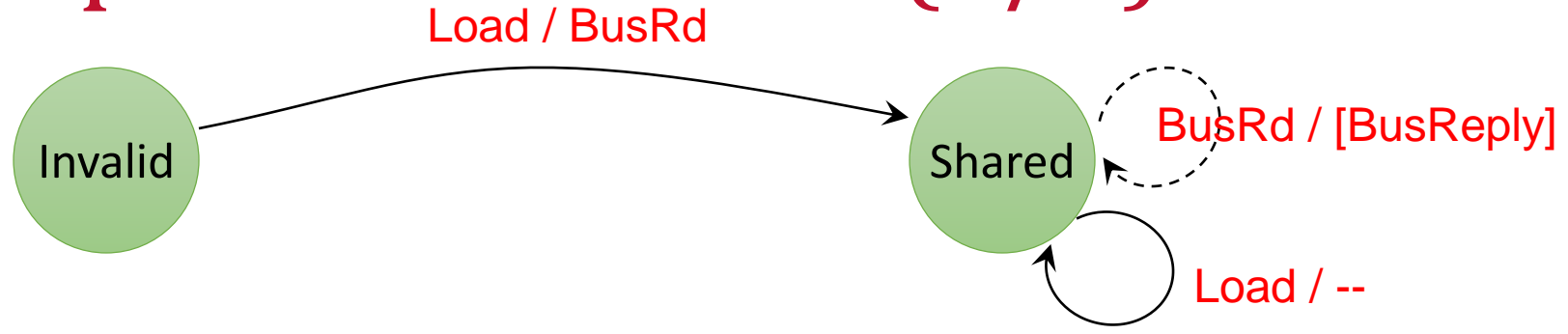
Load / BusRd



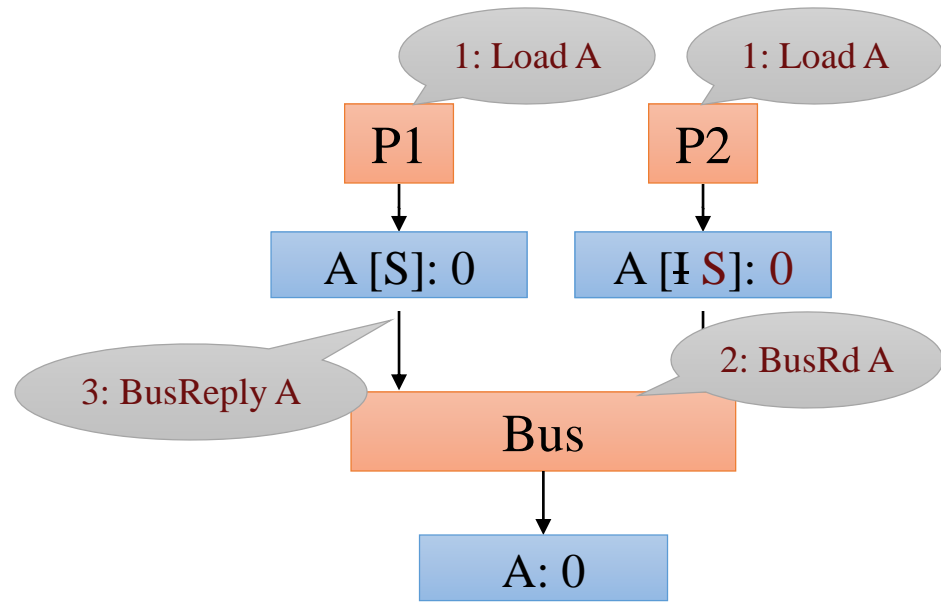
→ Transition caused by local action



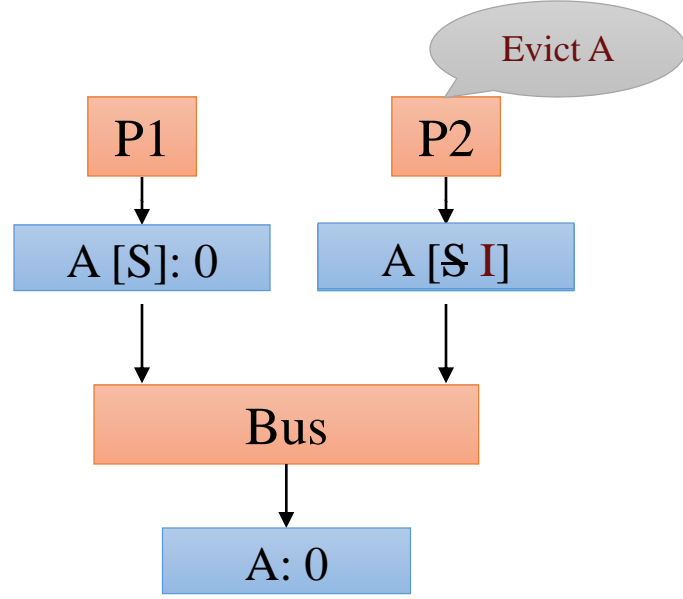
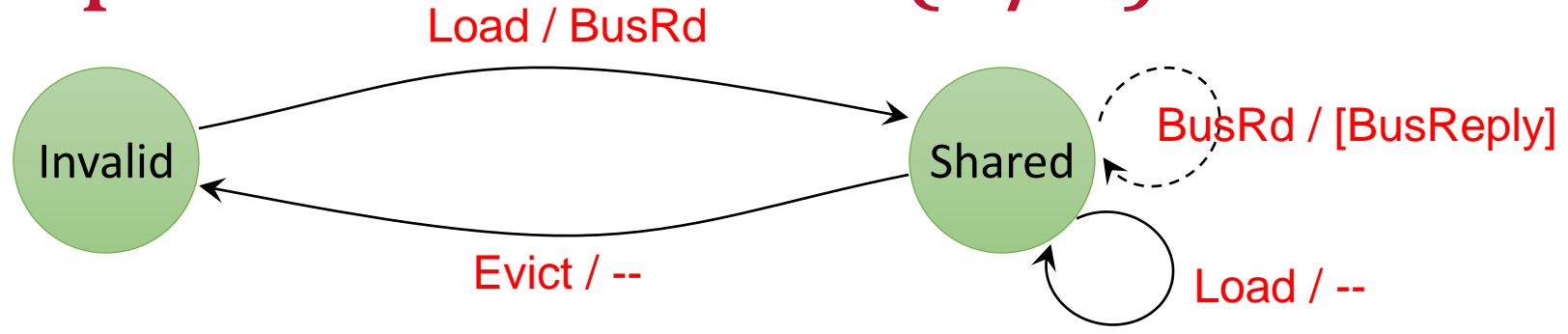
Simple MSI Protocol (2/9)



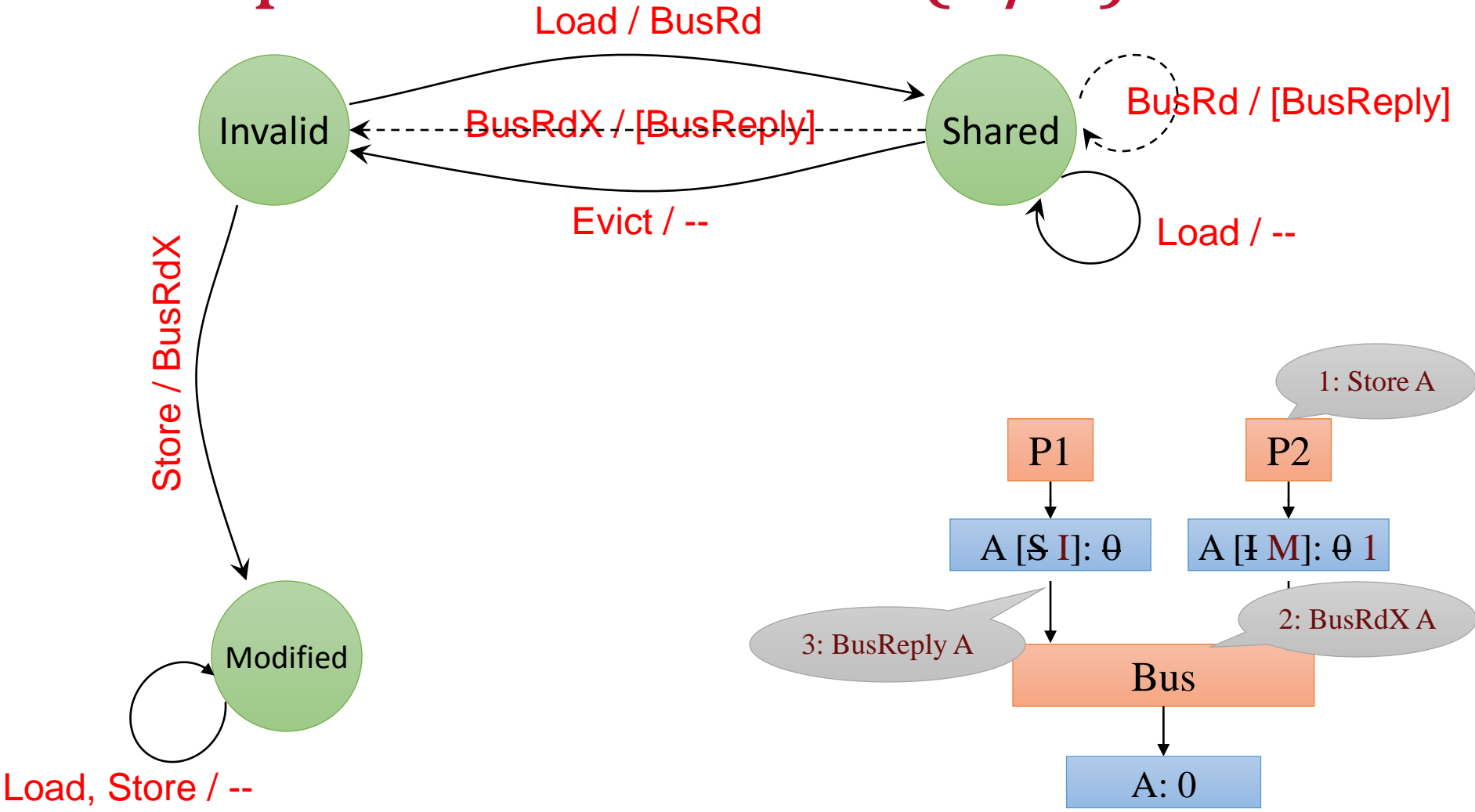
—————> Transition caused by local action
 - - - - -> Transition caused by bus message



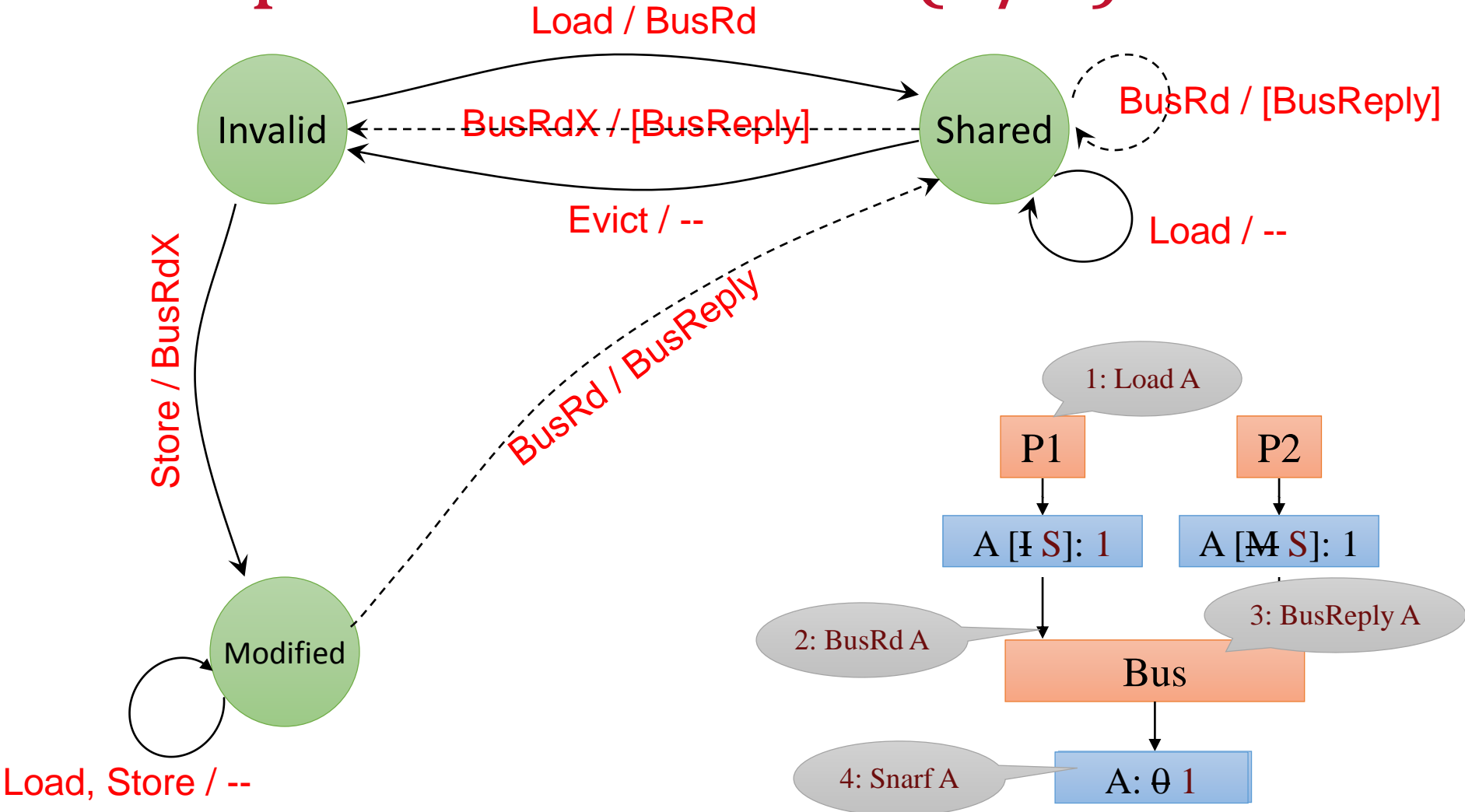
Simple MSI Protocol (3/9)



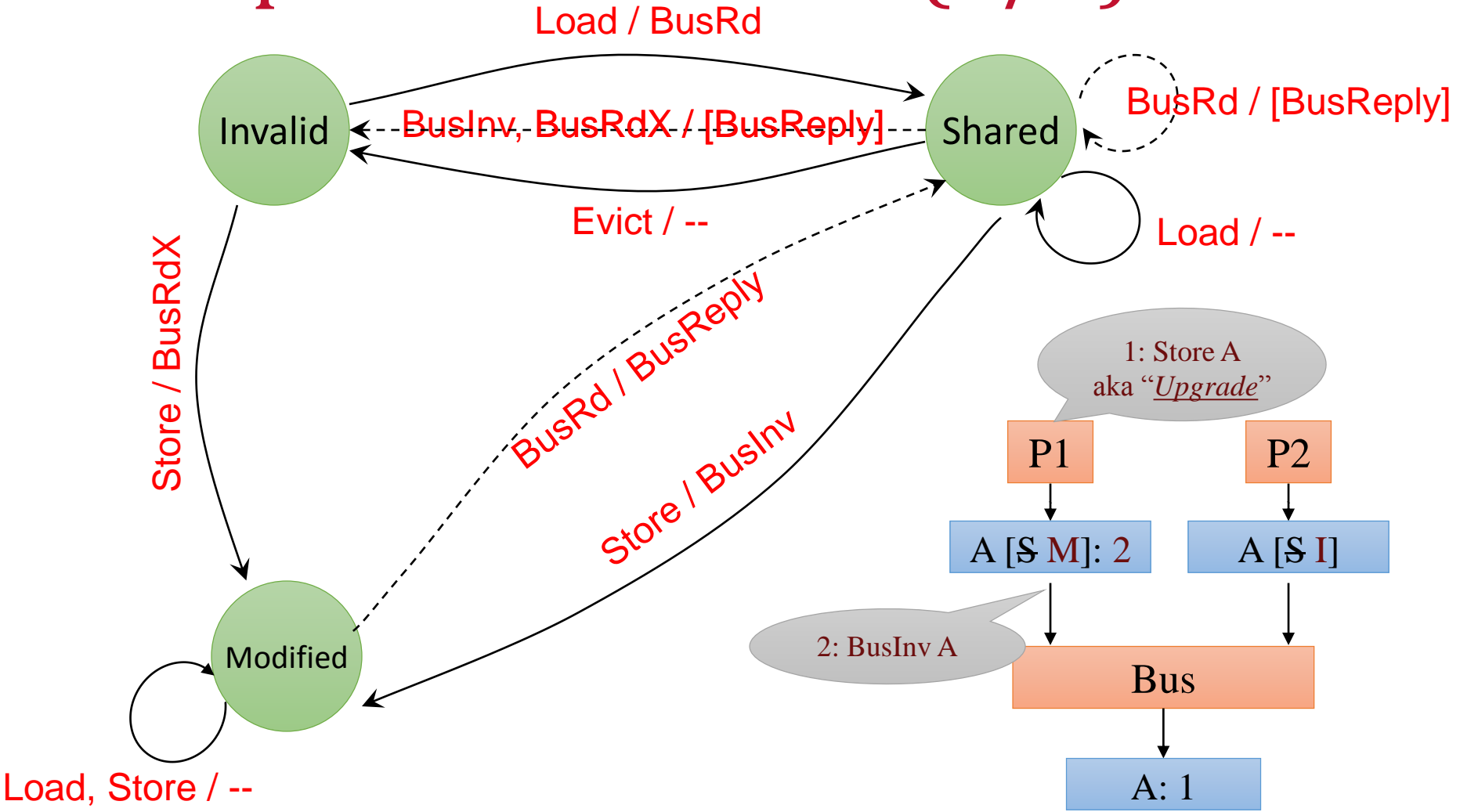
Simple MSI Protocol (4/9)



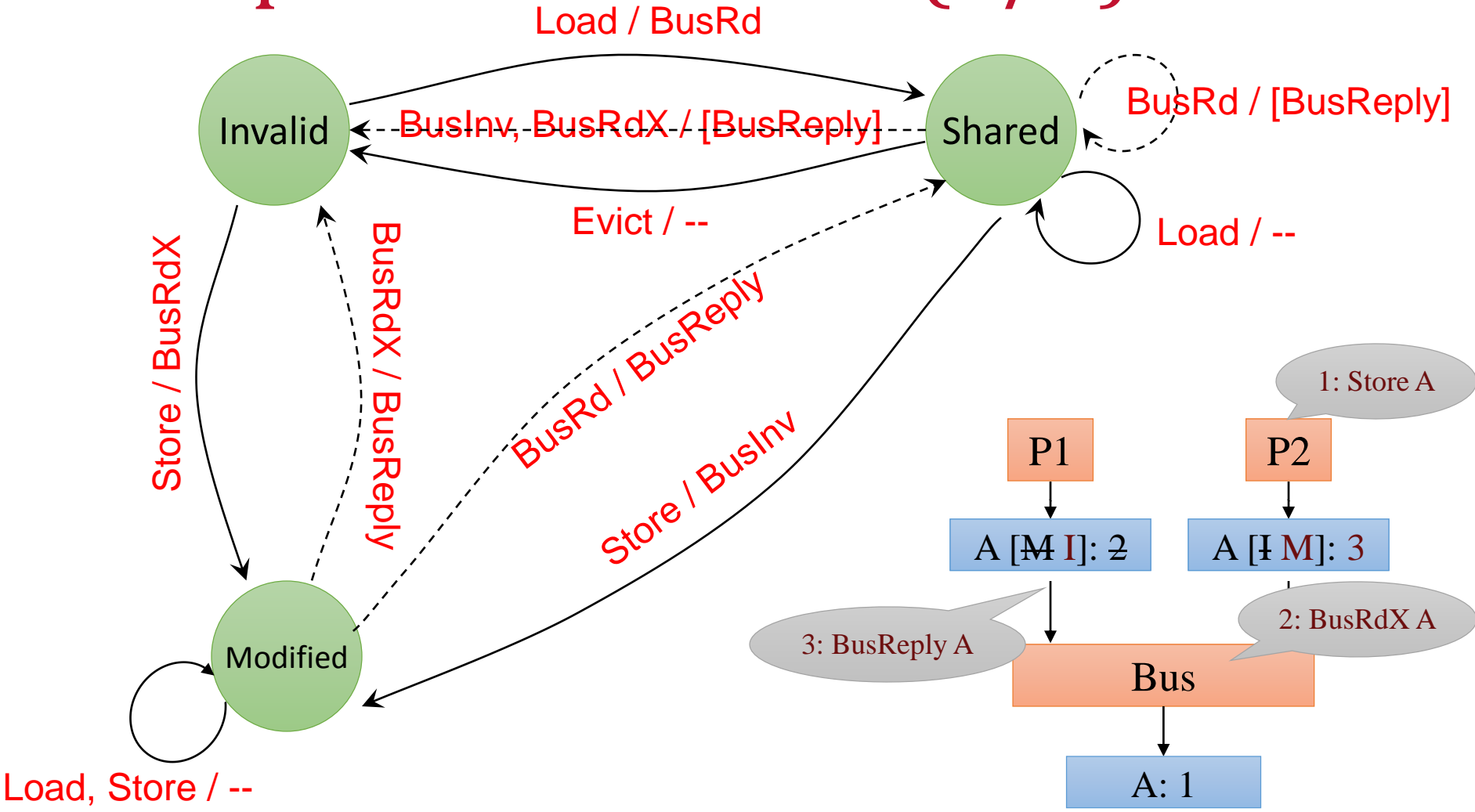
Simple MSI Protocol (5/9)



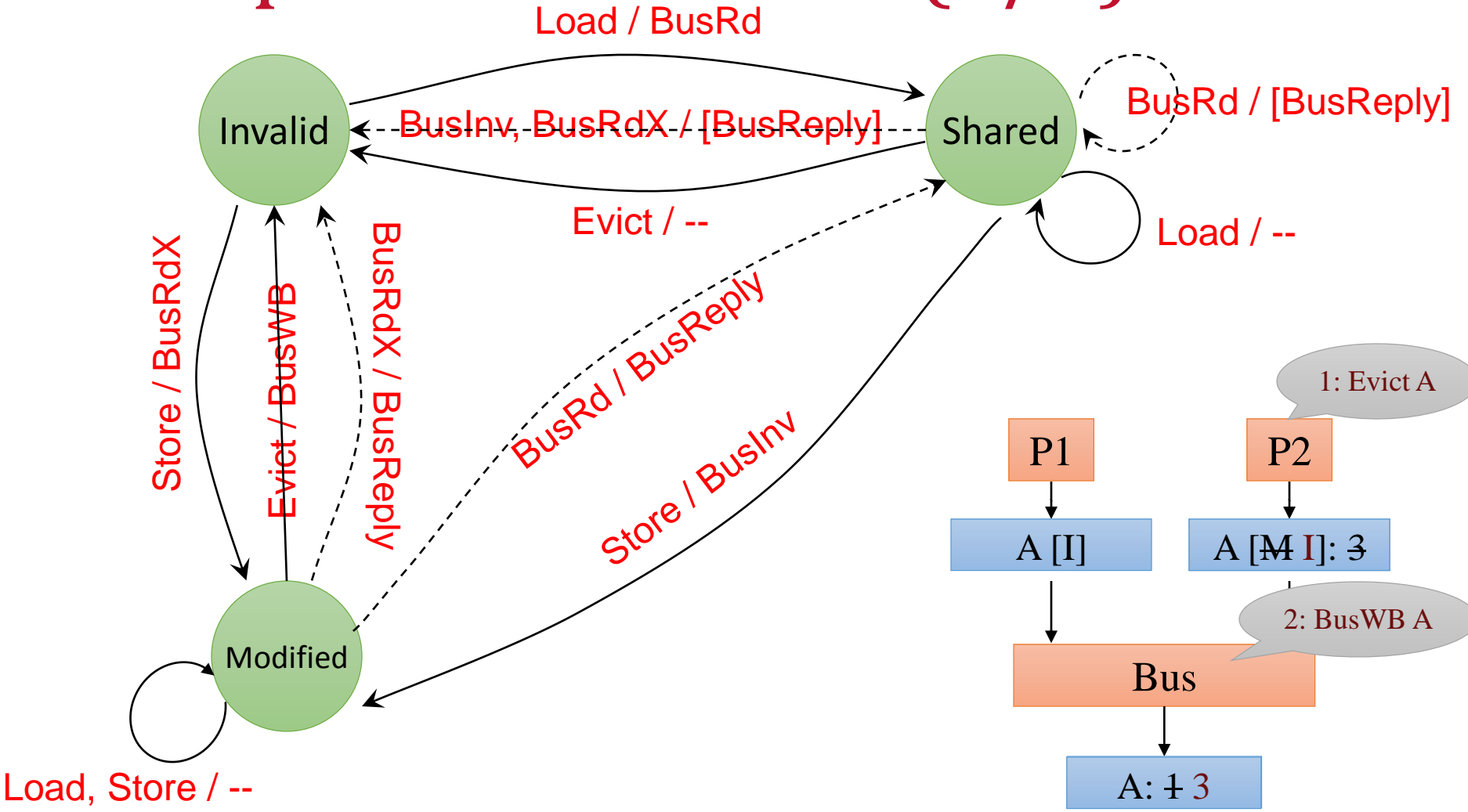
Simple MSI Protocol (6/9)



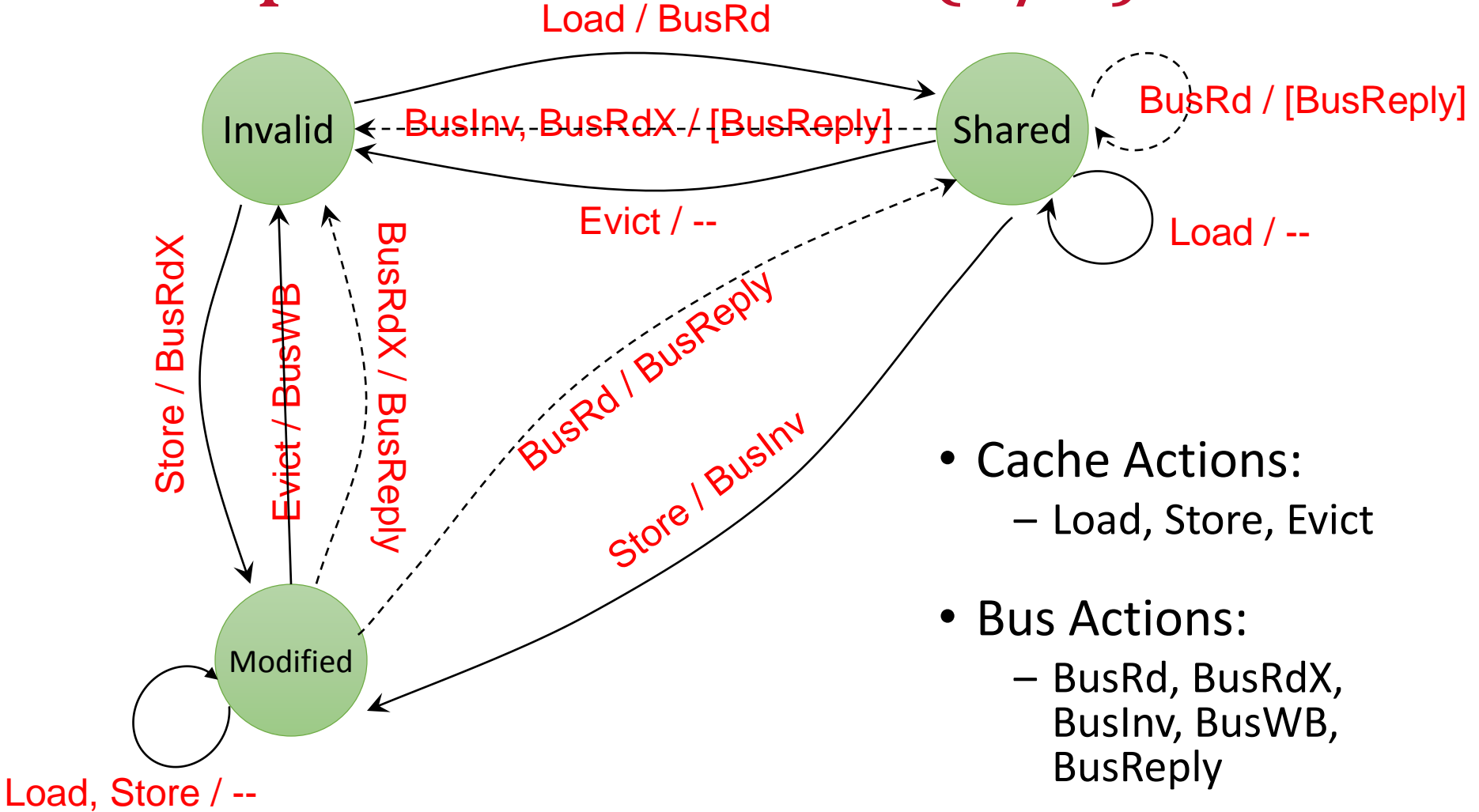
Simple MSI Protocol (7/9)



Simple MSI Protocol (8/9)



Simple MSI Protocol (9/9)



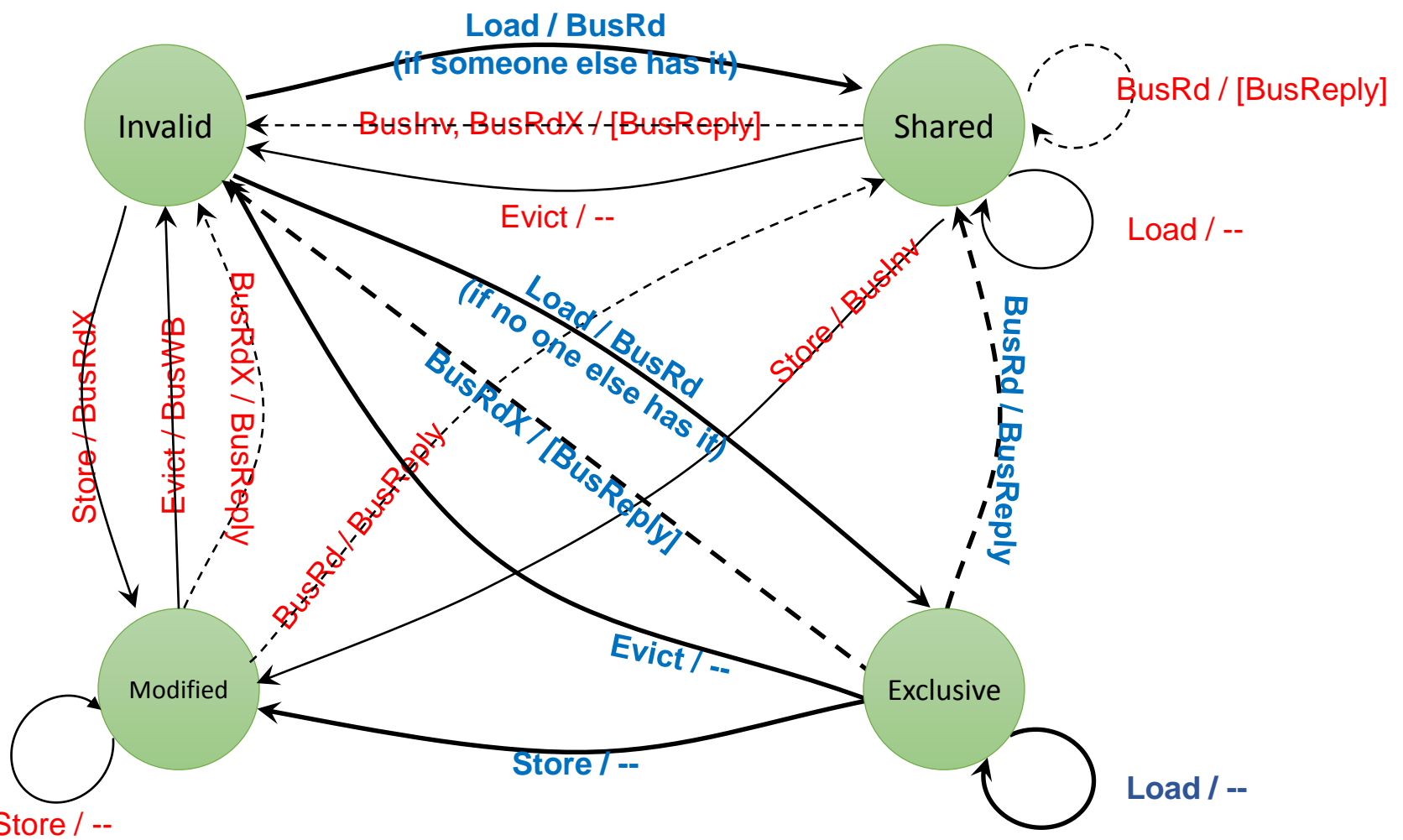
- Cache Actions:
 - Load, Store, Evict
- Bus Actions:
 - BusRd, BusRdX, BusInv, BusWB, BusReply

Usable coherence protocol

MESI or Illinois Protocol (1/2)

- States: Invalid, Exclusive, Shared, Modified
 - Called **MESI** 😊
 - Widely used in real processors
- Two features :
 - The cache knows if it has an Exclusive (E) copy
 - If some cache has a copy in E state, cache-cache transfer is used
- Advantages:
 - In E state no invalidation traffic on write-hits
 - Cuts down on **upgrade** traffic for lines that are first read and then written
 - Closely approximates traffic on a uniprocessor for sequential programs
 - Cache-cache transfer can cut down latency in some machine
- Disadvantages:
 - complexity of mechanism that determines exclusiveness
 - memory needs to wait before sharing status is determined

Illinois Protocol (2/2)



Problems w/ Snoopy Coherence

1. Interconnect bandwidth

- **Problem**: Bus and Ring are not scalable interconnects
 - Limited bandwidth
 - Cannot support more than a dozen or so processors
- **Solution**: Replace non-scalable bandwidth substrate (bus) with a scalable-bandwidth one (e.g., mesh)

2. Processor snooping bandwidth

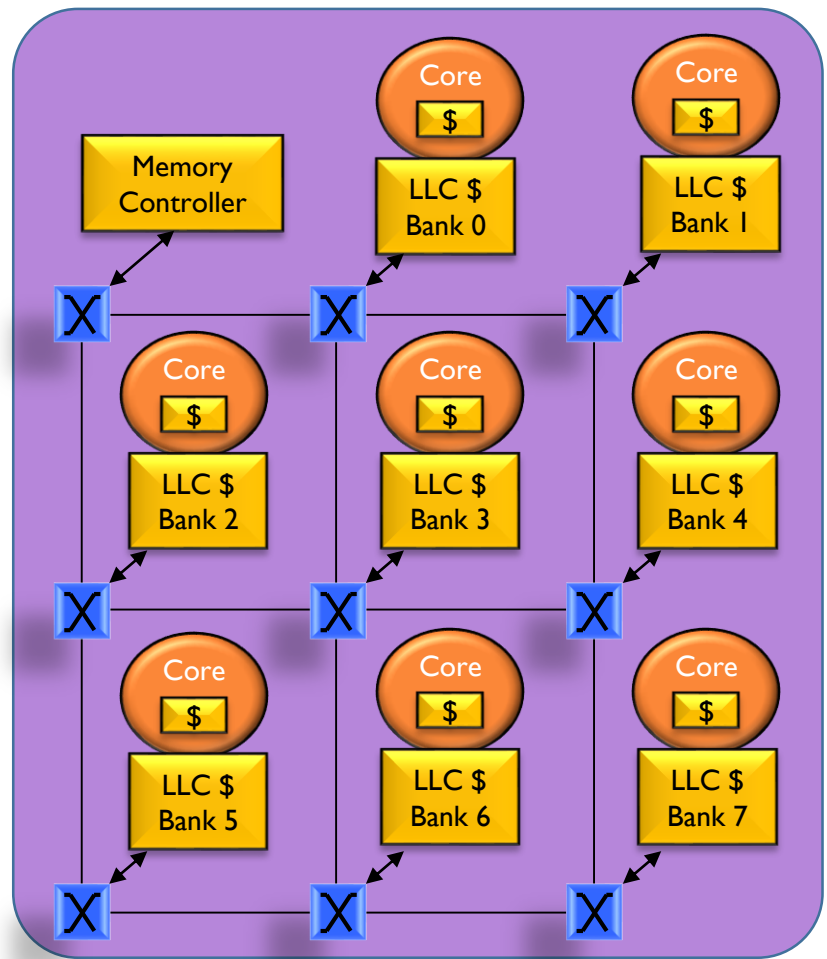
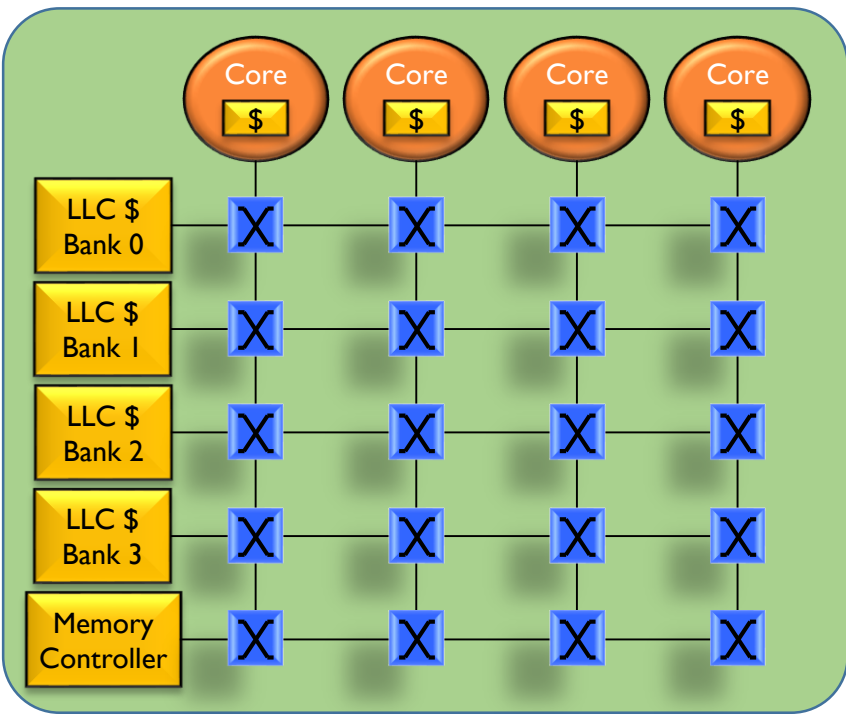
- **Problem**: All processors must monitor all bus traffic; most snoops result in no action
- **Solution**: Replace non-scalable broadcast protocol (spam everyone) with scalable directory protocol (notify cores that care)
 - The “directory” keeps track of “sharers”

Directory Coherence Protocols

- Extend memory (or LLC) to track caching information
 - Information kept in a hardware structure called **Directory**
- For each physical cache line, a home directory tracks:
 - **Owner**: core that has a dirty copy (i.e., M state)
 - **Sharers**: cores that have clean copies (i.e., S state)
- Cores send coherence events (requests) to home directory
 - Home directory only sends events to cores that “care”
 - i.e., cores that might have a copy of the line

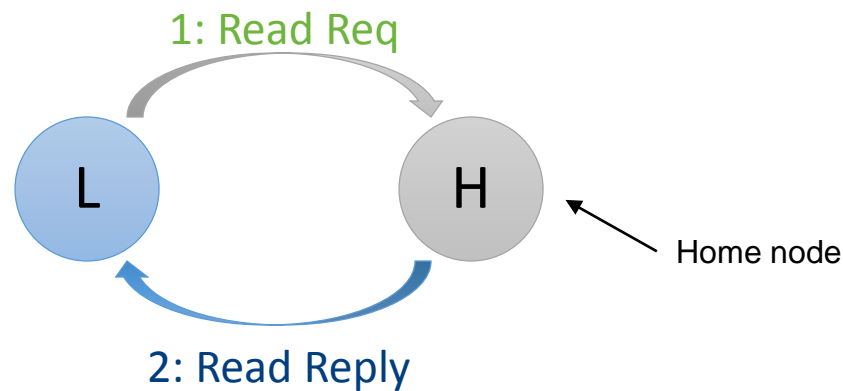
Directory Coherence Protocols

- Typically use point-to-point networks
 - Such as **Crossbar** or **Mesh**



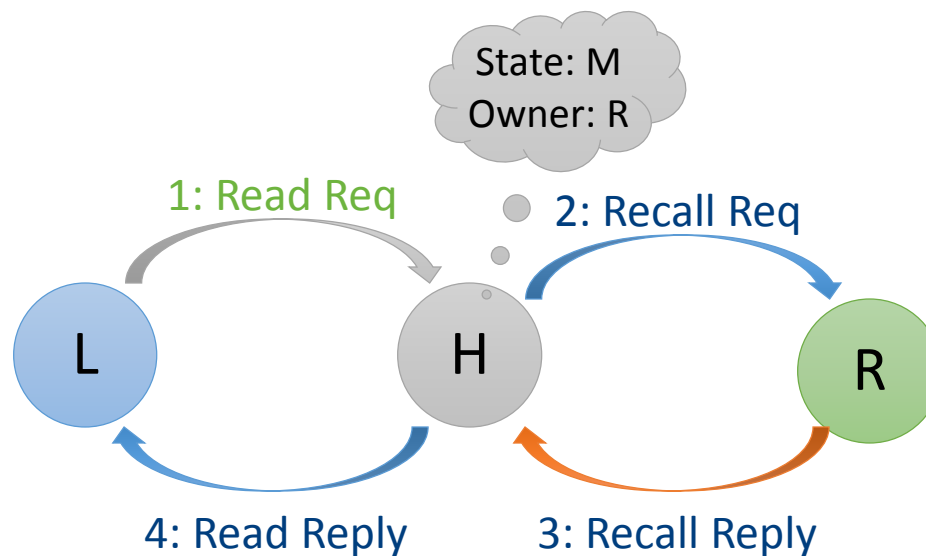
Read Transaction

- L has a cache miss on a load instruction



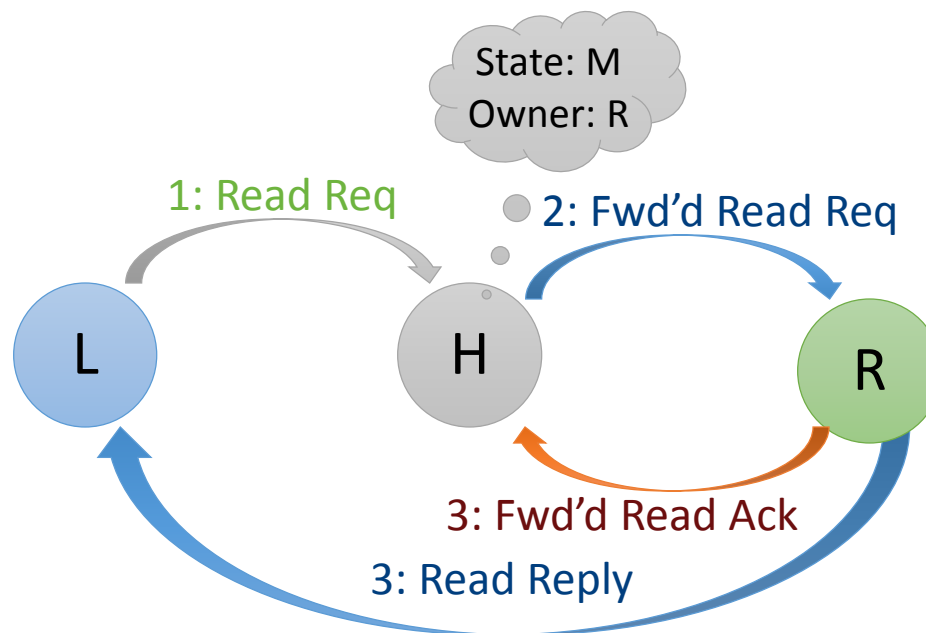
4-hop Read Transaction

- L has a cache miss on a load instruction
 - Block was previously in modified state at R



3-hop Read Transaction

- L has a cache miss on a load instruction
 - Block was previously in modified state at R



Coherence Protocols in Practice

- Cache coherence protocols are **very much** more complicated than presented here, because of...
- Race conditions
 - What happens if multiple processors try to read/write the same memory location simultaneously?
- Multi-level cache hierarchies
 - How to maintain coherence among multiple levels?
- Complex inter-connection networks and routing protocols
 - Must avoid live-lock and dead-lock issues
- Complex directory structures

Memory Consistency Models

Problem: Example 1

$\{A, B\}$ are memory locations; $\{r_1, r_2\}$ are registers.
Initially, $A = B = 0$

Processor 1

Store $A \leftarrow 1$
Load $r_1 \leftarrow B$

Processor 2

Store $B \leftarrow 1$
Load $r_2 \leftarrow A$

- Assume coherent caches
- Is this a possible outcome: $\{r_1=0, r_2=0\}$?
- Does cache coherence say anything?
 - Nope, different memory locations

Problem: Example 2

{A, B} are memory locations; { r_1, r_2, r_3, r_4 } are registers.
Initially, $A = B = 0$

Processor 1

Store A \leftarrow 1

Processor 2

Store B \leftarrow 1

Processor 3

Load $r_1 \leftarrow$ A

Load $r_2 \leftarrow$ B

Processor 4

Load $r_3 \leftarrow$ B

Load $r_4 \leftarrow$ A

- Assume coherent caches
- Is this a possible outcome: $\{r_1=1, r_2=0, r_3=1, r_4=0\}$?
- Does cache coherence say anything?

Problem: Example 3

{A, B} are memory locations; { r_1, r_2, r_3 } are registers.
Initially, $A = B = 0$

Processor 1

Store $A \leftarrow 1$

Processor 2

Load $r_1 \leftarrow A$
if ($r_1 == 1$)
 Store $B \leftarrow 1$

Processor 3

Load $r_2 \leftarrow B$
if ($r_2 == 1$)
 Load $r_3 \leftarrow A$

- Assume coherent caches
- Is this a possible outcome: $\{r_2=1, r_3=0\}$?
- Does cache coherence say anything?

Memory Consistency Model

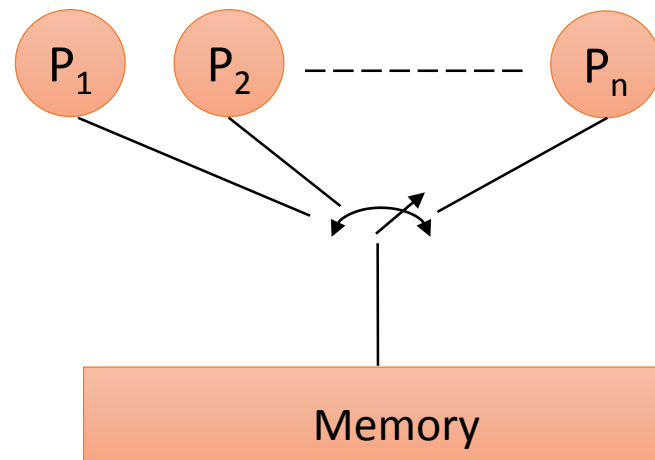
- Or just *Memory Model*
- Given a program and its input, determines whether a particular execution/outcome is valid w.r.t. its memory operations
 - if yes, then execution is **consistent** w/ memory model
 - An execution might be **inconsistent** w/ one model and consistent w/ another one
- Alternatively, memory model determines all possible executions/outcomes of a program given a fixed input
- You rely on the memory model when reasoning about the correctness of your (parallel) programs

Example: Sequential Consistency (SC)

“A multiprocessor 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.”

-Lamport, 1979

Processors issue memory ops in program order



Each op executes atomically (at once), and switch randomly set after each memory op

Problems with SC Memory Model

- Difficult to implement efficiently in hardware
 - Straight-forward implementations:
 - No concurrency among memory access
 - Strict ordering of memory accesses at each processors
 - Essentially precludes out-of-order CPUs
 - Conflicts with latency hiding techniques
- Unnecessarily restrictive
 - Most parallel programs won't notice out-of-order accesses

Dekker's Algorithm Example

- Mutually exclusive access to a critical region
 - Works as advertised under SC
 - Can fail in presence of store queues
 - OoO allows P1 to read B before writing A to memory/cache

Processor 1

Lock_A:

- 3 A = 1;
 - 1 if (B != 0)
 - { A = 0; goto Lock_A; }
- /* critical section*/
A = 0;

Processor 2

Lock_B:

- 4 B = 1;
 - 2 if (A != 0)
 - { B = 0; goto Lock_B; }
- /* critical section*/
B = 0;

Relaxed Models

- No real processor today implements SC
- Instead, they use “*Relaxed Memory Models*”
 - “Relax” some ordering requirements imposed by SC
 - For example:
 - Total Store Ordering (TSO) relaxes **W** → **R** (x86 and SPARC): a **read** can bypass earlier **writes**
 - IBM Power and ARM relax almost all orderings (RW → RW)
- In a relaxed-memory system, can use ***fence*** instructions to enforce ordering between otherwise unordered instructions

	<u>Processor 1</u>	<u>Processor 2</u>
Dekker Example with fences:	Lock_A: A = 1; <i>mfence</i> ; if (B != 0) ...	Lock_B: B = 1; <i>mfence</i> ; if (A != 0) ...

Consistency vs. Coherence

- Coherence only concerns reads/writes to the same memory location; specifically
 - “All stores to any given memory location should be seen in the same order by all processors”
- Memory consistency concerns accesses to all memory locations
 - “A memory model determines, for each load operation L in an execution, the set of store operations whose value might be returned by L ”
- A memory consistency model may or may not require coherence
 - i.e., coherence is a required property of some (an not all) memory models

And More...



- Memory model is not just a hardware concept...
 - Programming languages have memory models as well
- Because compilers/interpreters too can re-order, add or remove read/write operations
 - E.g., Code motion (re-order)
 - Register Allocation and Common Subexpression Elimination (remove)
 - Partial Redundancy Elimination (add)
- If interested, take a look at Java and C/C++11 memory models

Hardware Multithreading (MT)

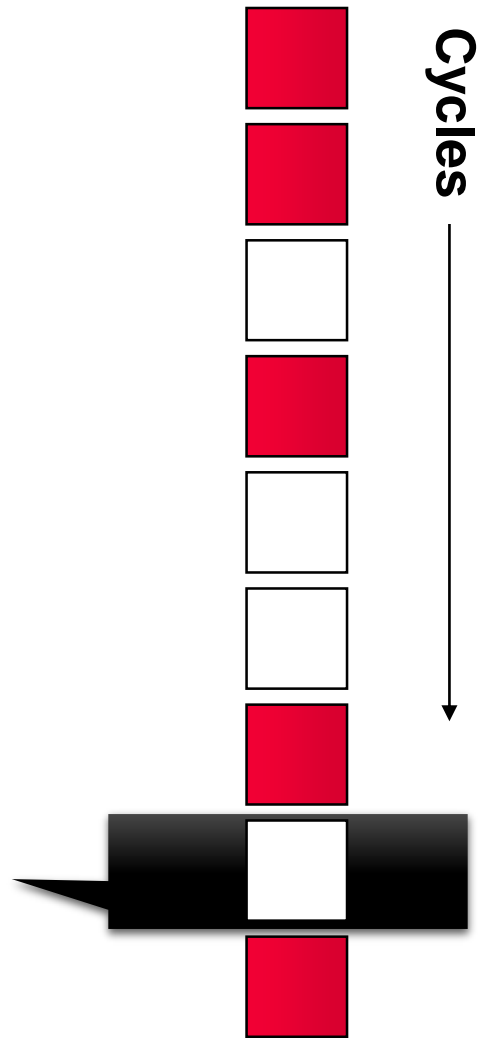
Hardware Multi-Threading

- Uni-Processor: 4-6 wide, lucky if you get 1-2 IPC
 - Poor utilization of transistors
- SMP: 2-4 CPUs, but need independent threads
 - Poor utilization as well (if limited # of threads)
- *{Coarse-Grained, Fine-Grained, Simultaneous}-MT*
 - Use single large uni-processor as a multi-processor
 - Single core provides multiple hardware contexts (threads)
 - Per-thread PC
 - Per-thread ARF (or map table)
 - Each core appears as multiple CPUs
 - OS designers still call these “CPUs”

Scalar Pipeline

-  Busy Functional Unit (or issue slot)
-  Idle Functional Unit (or issue slot)

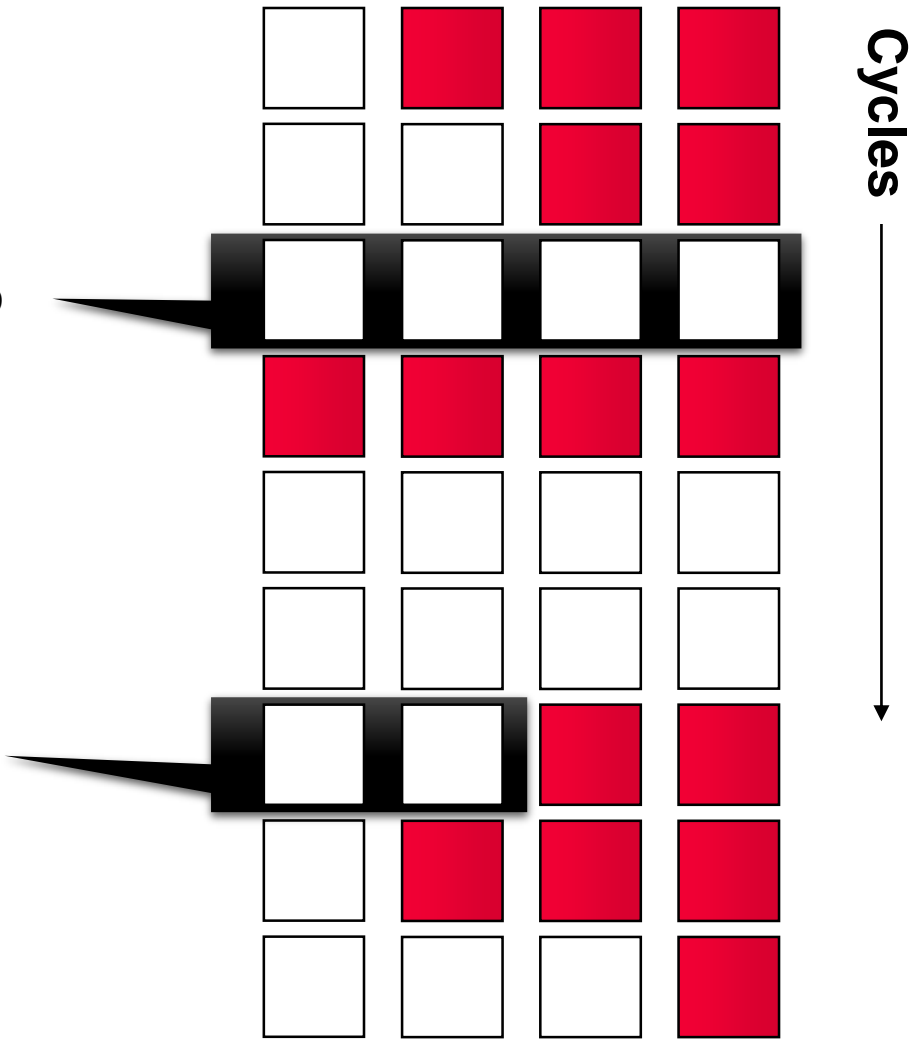
Waste: cycle in which next instruction is not issued to execute



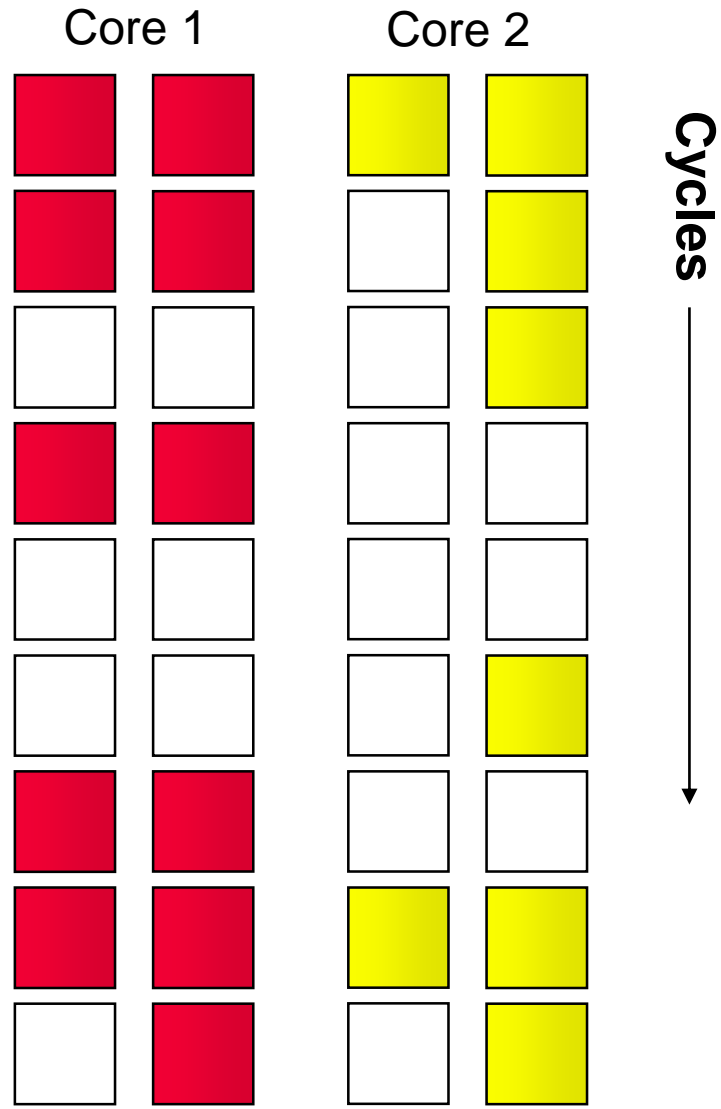
Superscalar Pipeline

Vertical waste: cycle in which no instruction is issued
(no instruction ready to execute)

Horizontal waste: some of the issue slots in a cycle wasted
(not enough instructions ready to execute)



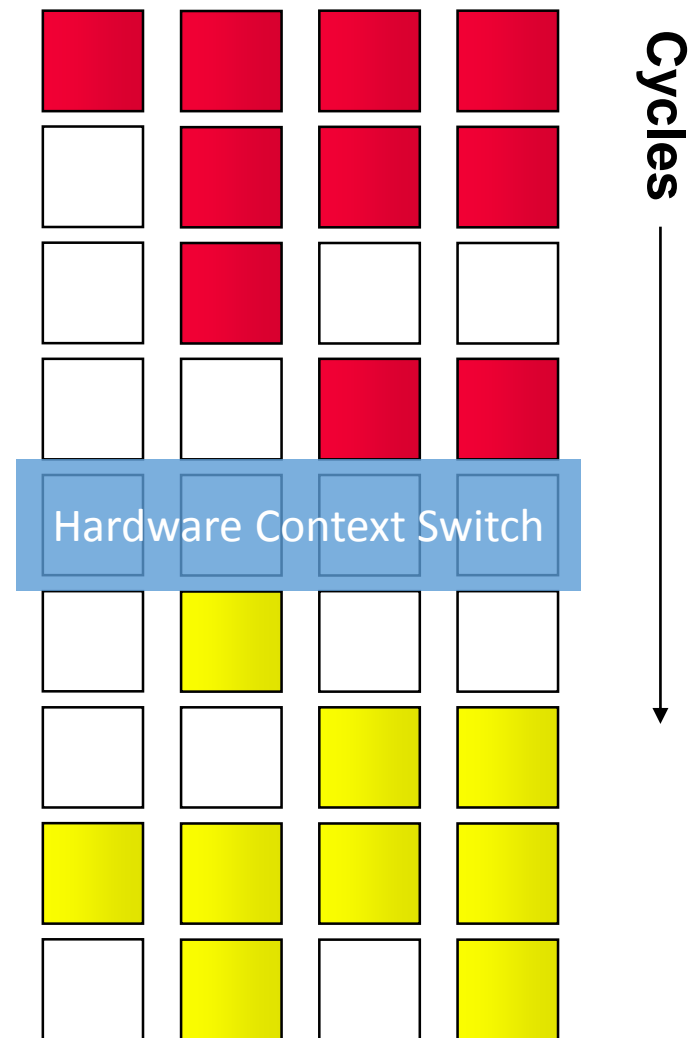
Chip-Multiprocessing (CMP)



Limited utilization when running one thread

Coarse-Grained Multithreading (1/3)

- Hardware switches to another thread when current thread stalls on a long latency op
 - E.g., L2 miss
- Only one thread in the pipeline at any time

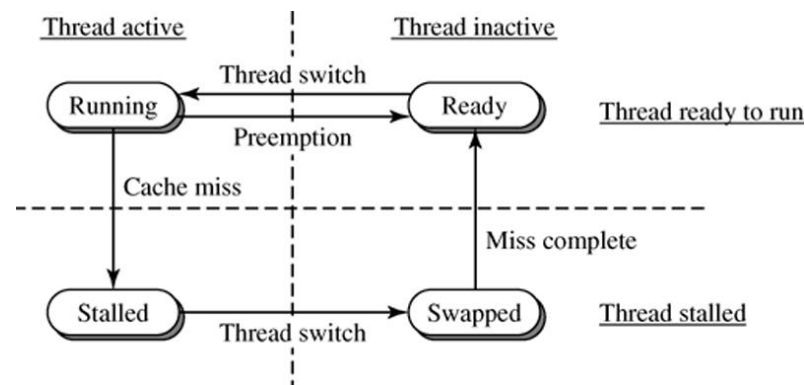


Only good for long latency ops (i.e., cache misses)

Coarse-Grained Multithreading (2/3)

- Needs “preemption” and “priority” mechanisms to ensure fairness and high utilization
 - Different from OS preemption and priority
 - E.g., HW “preempts” long running threads with no L2 miss
 - High “priority” means thread should not be preempted
 - E.g., when in a critical section
 - Priority changes communicated using special instructions

**Thread State
Transition Diagram in a
CGMT Processor**

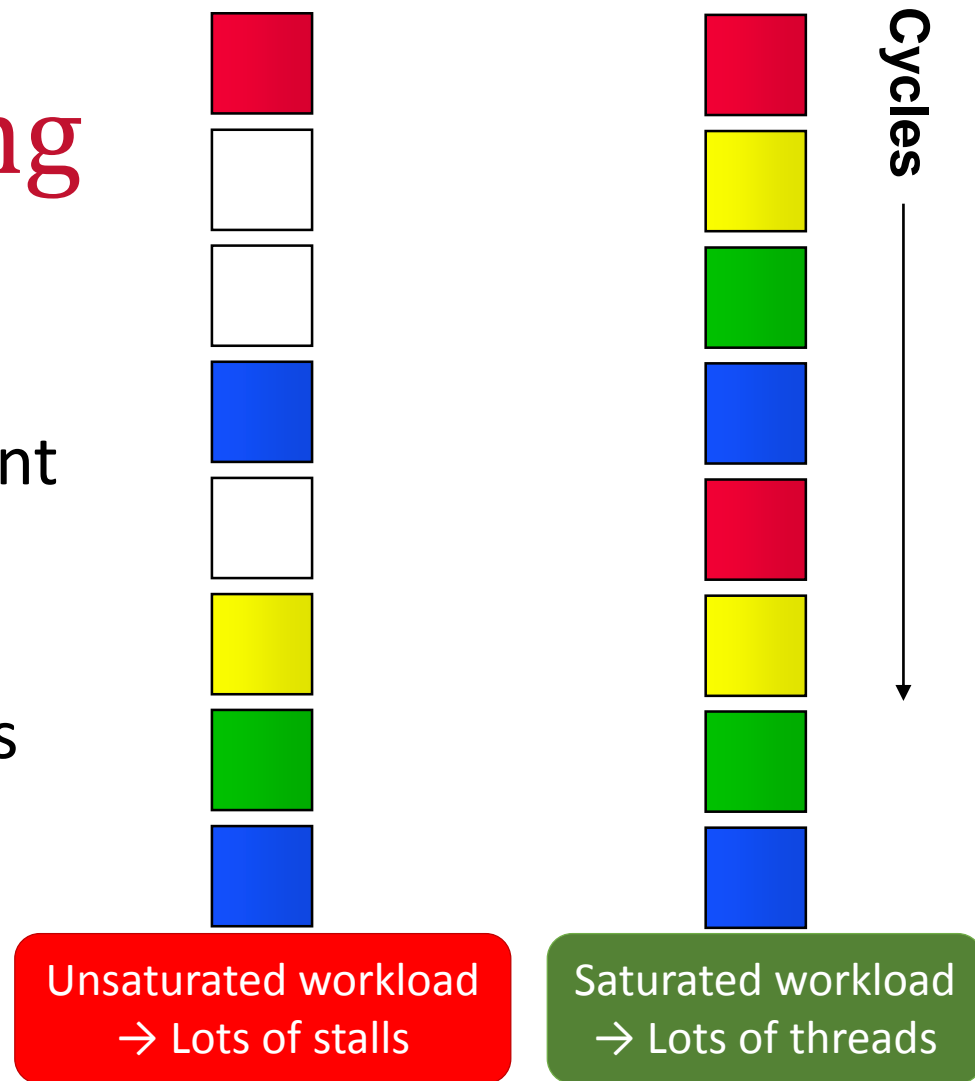


Coarse-Grained Multithreading (3/3)

- ✓ ✗ Sacrifices a little single thread performance
- ✗ Tolerates only long latencies (e.g., L2 misses)
 - ✗ Only eliminating some of the vertical waste
- Thread scheduling policy
 - Designate a “preferred” thread (e.g., thread A)
 - Switch to thread B on thread A L2 miss
 - Switch back to A when A L2 miss returns
- Pipeline partitioning
 - None, flush on switch
 - Need short in-order pipeline for good performance
 - High context switch cost otherwise

Fine-Grained Multithreading (1/2)

- Every cycle, a different thread fetches and issues instructions
- (Many) more threads
- Multiple threads in pipeline at once



Intra-thread dependencies still limit performance

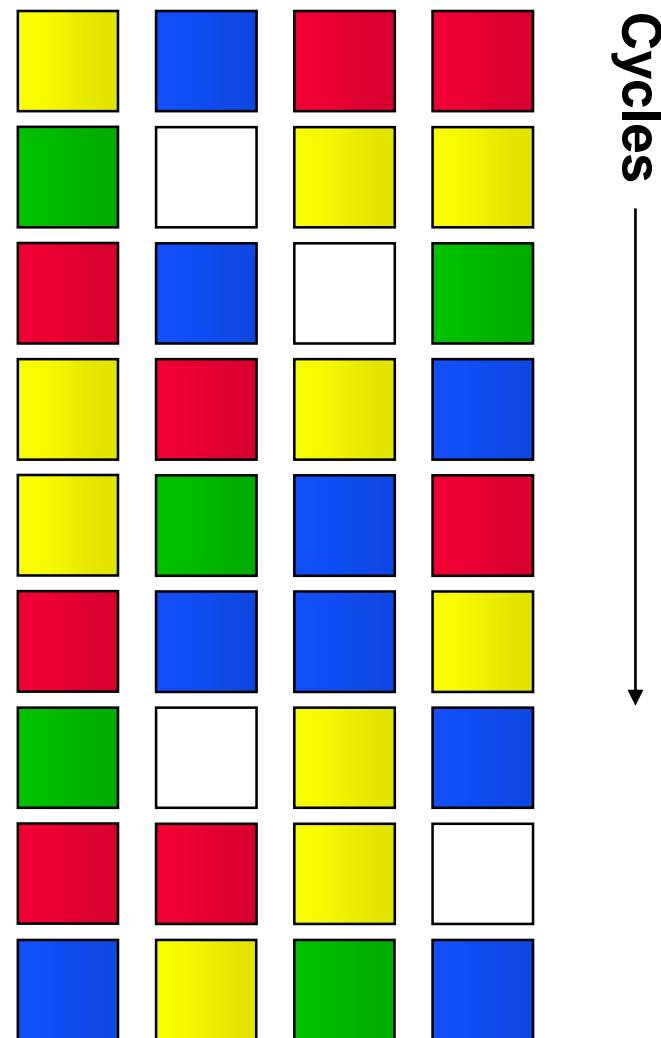
Fine-Grained Multithreading (2/2)

- ✗ Sacrifices significant single-thread performance
- ✗ Does not eliminate horizontal waste
- ✓ Tolerates everything
 - ✓ L2 misses
 - ✓ Mispredicted branches
 - ✓ etc...
- Eliminates most vertical waste
- Good for throughput-bound workload, bad for latency-bound
- Thread scheduling policy
 - Switch threads often (e.g., every cycle)
 - Use round-robin policy, skip threads with long-latency pending ops
- Pipeline partitioning
 - Dynamic, no flushing
 - Length of pipeline doesn't matter

Example: Cray Threadstorm (128 threads per proc)

Simultaneous Multithreading (1/2)

- Fine/Coarse-grained MT only eliminates *vertical waste*
- SMT also eliminates *horizontal waste*: Issue any ready-instruction from any thread



Max utilization of functional units

Simultaneous Multithreading (2/2)

- ✗ Sacrifices some single thread performance
- ✓ Tolerates all latencies
- ✓ Attacks both vertical and horizontal waste
- ✓ A natural extension of super-scalar OoO pipelines
 - Front-end handle fetching and dispatching from separate threads
 - The back-end can just mix instructions from all threads
- Fetch scheduling policy
 - Usually round-robin (like Fine-Grained MT)
 - Can fetch from multiple threads in each cycle
- Pipeline partitioning
 - Dynamic
- Examples
 - Pentium 4 (hyper-threading): 5-way issue, 2 threads
 - Alpha 21464: 8-way issue, 4 threads (canceled)

Issues for SMT

- Cache interference
 - Concern for all MT variants
 - Shared memory SPMD threads may help here
 - Same insns. → share I\$
 - Shared data → less D\$ contention
 - MT is (probably) good for “server” workloads
 - SMT might want a larger L2 (which is OK)
 - Out-of-order tolerates L1 misses
- Large physical register file
 - $\#phys\text{-}regs = (\#threads * \#arch\text{-}regs) + \#in\text{-}flight\text{ insns}$
- Some hardware resources should be partitioned or duplicated
 - ROB, LSQ, RAS, Map Table, ...
- Most resources can be shared
 - TLB, Branch Predictor, Functional Units, ...

Latency vs. Throughput

- MT trades (single-thread) latency for throughput
 - Sharing processor degrades latency of individual threads
 - But improves aggregate latency of both threads
 - Improves utilization
- Example
 - Thread A: individual latency=10s, latency with thread B=15s
 - Thread B: individual latency=20s, latency with thread A=25s
 - Sequential latency (first A then B or vice versa): 30s
 - Parallel latency (A and B simultaneously): 25s
 - MT slows each thread by 5s
 - But improves total latency by 5s

Benefits of MT depend on workload

Combining TLP Techniques (1/2)

- System can have SMP, CMP, and Hardware MT at the same time
- Example machine with 48 threads
 - Use 2-socket SMP motherboard with two chips
 - Each chip with an 12-core CMP
 - Where each core is 2-way SMT
- Example machine with 1024 threads: Oracle T5-8
 - 8 sockets
 - 16 cores per socket – dual-issue, out-of-order cores
 - 8 threads per core

Combining TLP Techniques (2/2)

- Makes life difficult for the OS scheduler
 - OS needs to know which CPUs are...
 - Real physical processor (SMP): highest independent performance
 - Cores in same chip: fast core-to-core communication, but shared resources
 - Threads in same core: competing for resources
 - Distinct tasks scheduled on different CPUs
 - Cooperative tasks (e.g., pthreads) scheduled on same core
 - Use SMT as last choice (or don't use for some applications)
 - How can the OS know?