Stony Brook University

Beyond ILP

In Search of More Parallelism

Nima Honarmand

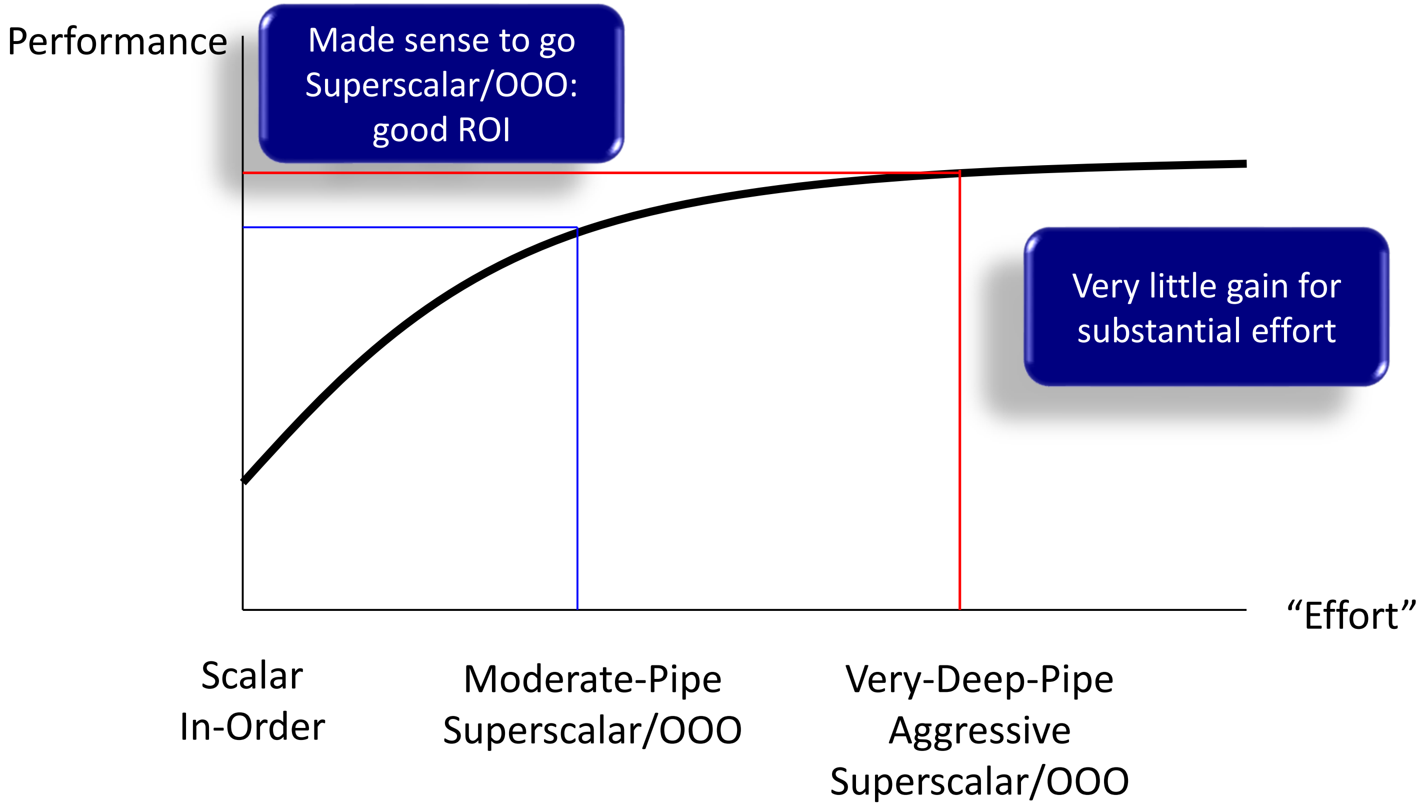
ILP Is NOT Enough (1)

- 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
    - # ROB, RS and LSQ entries, functional units
  - True data dependences
    - Coming from algorithm and compiler
  - Branch prediction accuracy
  - Imperfect memory disambiguation

ILP Is NOT Enough (2)

- To get more performance, we can keep pushing IPC and/or frequency
  - Design complexity (time to market)
  - Cooling (cost)
  - Power delivery (cost)
  - …
- But it is too costly for the marginal improvements gained

Higher Complexity not Worth Effort



Implicit Parallelism

- Problem: HW is in charge of finding parallelism
  - Implicit 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 (smaller, faster transistors)
  - Higher IPC (different micro-arch)
  - But this was not sustainable…

Explicit Parallelism

- Alternative: Explicit Parallelism
  - HW user (programmer, compiler or OS) responsible for finding and expressing parallelism
  - HW does not need to allocate resources 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
    - pthreads
  - Parallel languages and libraries
    - OpenMP, Cilk, TBB, etc…

Architectures to Exploit TLP

- ***Multiprocessors (MP):*** Different threads run on different processors
  - Multiple processor chips
  - Chip Multiprocessors (CMP), a.k.a. Multicore processors
  - A combination of the above
- ***Hardware Multithreading (MT)***: Multiple threads share the same processor pipeline
  - Coarse-grained MT (CGMT)
  - Fine-grained MT (FMT)
  - Simultaneous MT (SMT)

Classes of Multiprocessors

- **Shared Memory Multiprocessors**
  - Single OS manages all the processors
  - OS sees multiple CPUs
    - Runs one process (or thread) on each CPU
  - Code running on different cores communicate by reading/writing a shared physical address space
  - Most common form of multiprocessors today

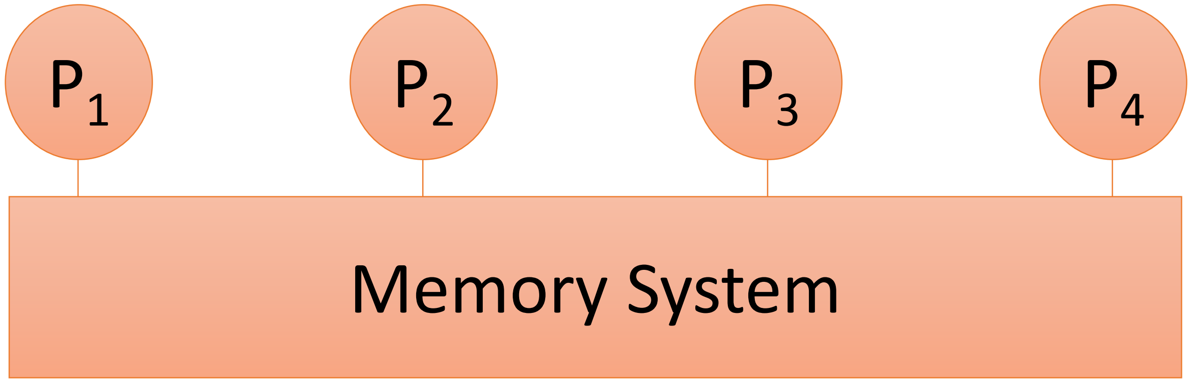
- **Message-Passing Multiprocessors (Multicomputers)**
  - Composed of multiple nodes (computers)
  - Nodes do not have access to each other's memory (no shared memory)
  - Nodes communicate by passing explicit messages
  - <u>Supercomputers</u> are the most common examples

We focus on Multiprocessors

Shared-Memory Multiprocessors

Logical View

- Multiple threads use **shared memory** (physical address space) to communicate
  - "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

Overall: most popular form of parallel programming (for now)

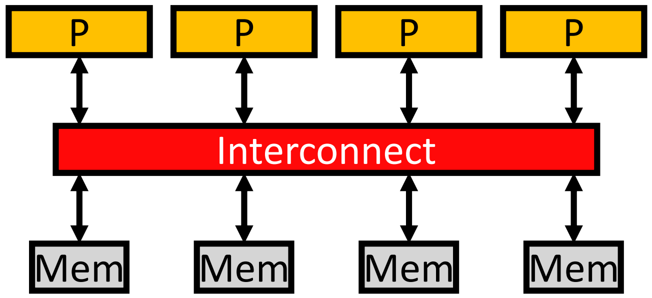
Physical Architecture (1)

- Almost always there is a hierarchical structure
  - Within socket, within a machine, across machines
  - Contrary to the flat view of shared-memory <u>programming</u> models
- At each level of hierarchy, there are
  - multiple ***processing elements*** (cores, sockets, boxes, …)
  - multiple ***memory elements*** (caches, DRAM modules, …)
  - an ***interconnection network*** connecting them together

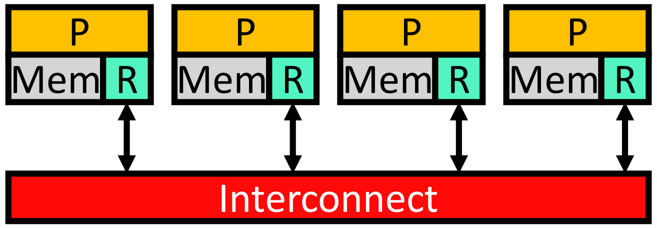
Physical Architecture (2)

- At each level, two general configs. w.r.t. proc-mem connection
- ***Uniform Memory Access (UMA)***: equal latency to memory from all processors
  - Simpler software, doesn't matter where you put data
  - Lower peak performance
- ***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:

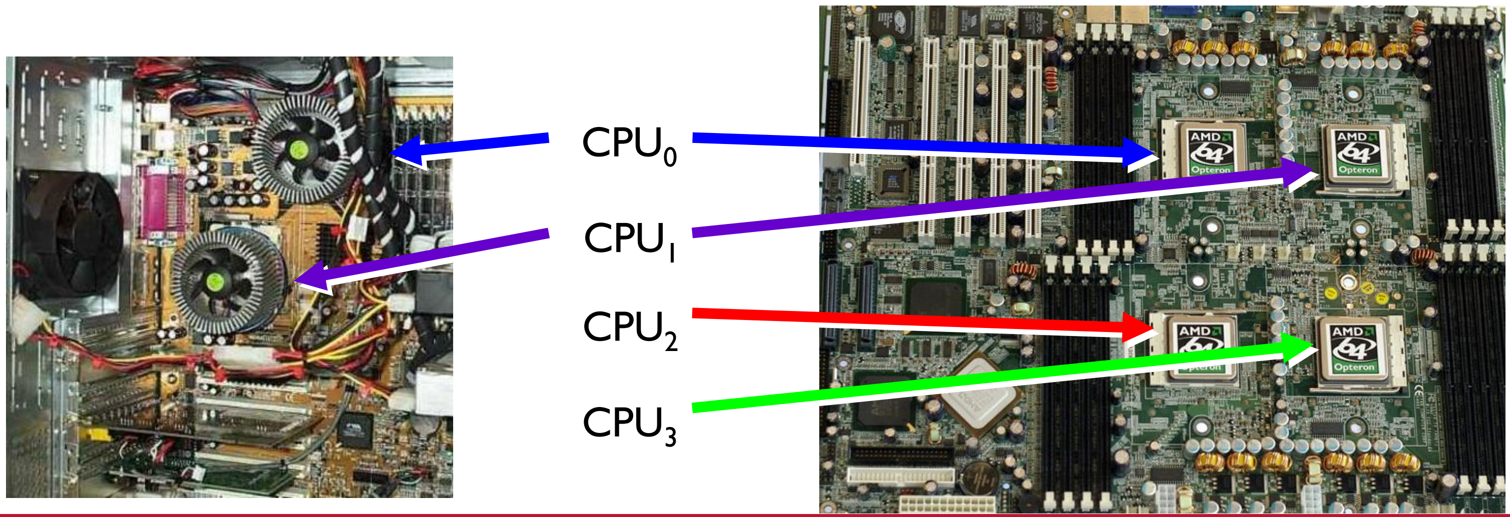


NUMA:



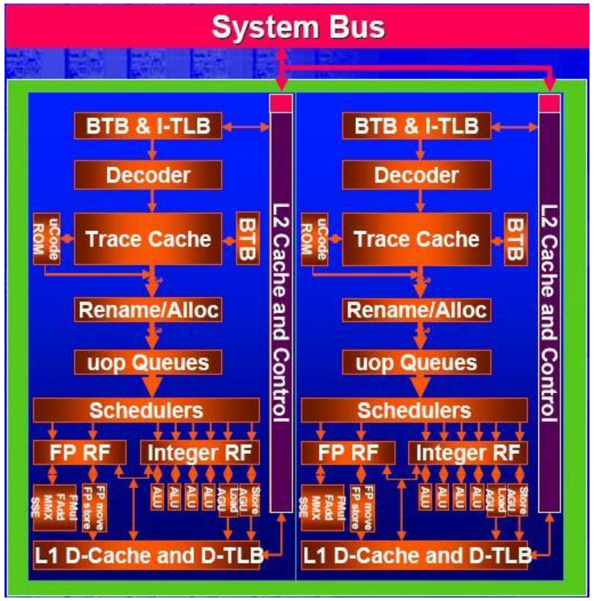
Example: Symmetric Multiprocessors (SMP)

- Earlier form of multiprocessors
  - One CPU per socket
  - Symmetric = All CPUs are the same and have “equal” access to memory
  - All CPUs are treated as similar by the OS (no master/slave, no bigger or smaller CPUs, …)

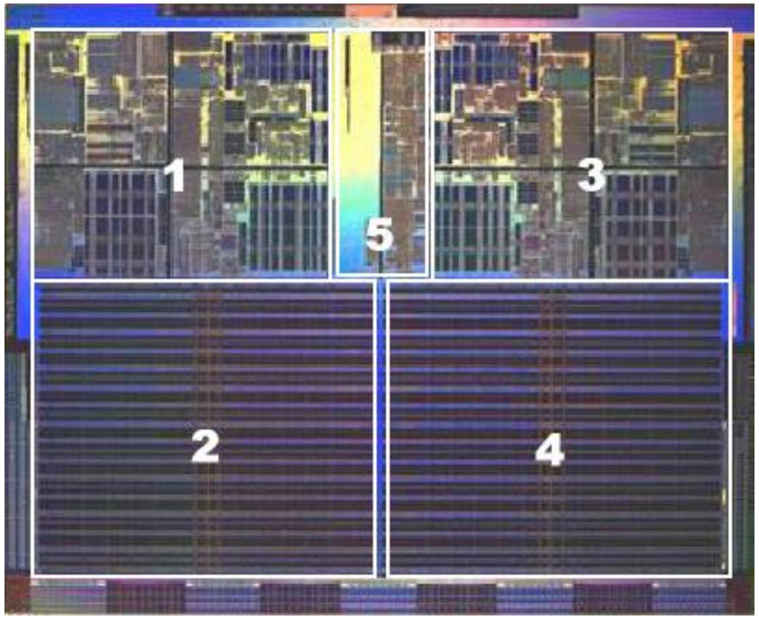


Example: Chip-Multiprocessors (CMP)

- 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

Multi-socket machines Have multiple CMPs on the same motherboard

Benefits of CMP (1)

- 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
    - On-die communication 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

Benefits of CMP (2)

- Can use parallelism to reduce power needs
- Let's say I have a fixed power budget
- To use parallelism, let's use two processors instead of one
  - 2x CPUs $\rightarrow$ ½ power for each
  - Maybe a little better than ½ if resources can be shared
- Back-of-the-Envelope calculation:
  - 3.8 GHz CPU at 100W
  - Dual-core: 50W per Core
  - $P \propto V^3$: $V_{orig}{}^3/V_{CMP}{}^3 = 100W/50W \rightarrow V_{CMP} = 0.8\ V_{orig}$
  - $f \propto V$: $f_{CMP} = 0.8 \times f_{orig} = 3.0GHz$
  - Peak performance: **1.6x**
  - **Same power budget, same micro-arch, better performance**

Can use parallelism to improve performance/power ratio!

Example: Shared-Memory Supercomputers

- One shared memory machine, composed of multiple racks, each containing multiple nodes, each containing multiple CMPs
- All the memory is shared across all the processors in the whole machine
  - Using a special interconnect

Columbia Suptercomputer at NASA (Cluster of multi-rack 512-processor shared-memory machines)

Source: Wikipedia

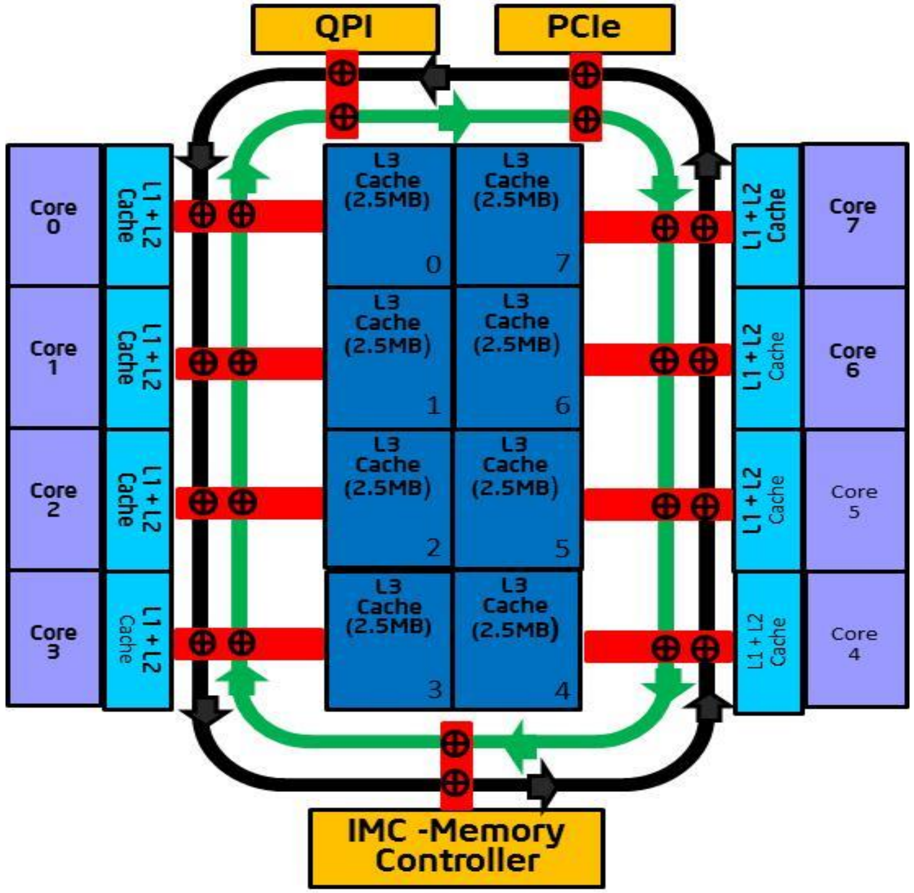
Interconnection Networks

- **Shared networks** (a.k.a. bus)
  - All elements connected to a shared physical medium
  - Low latency, low bandwidth
  - Doesn’t scale to large number of elements
  - Simpler protocols (e.g., cache coherence)
- **Point-to-point networks**
  - Each link is only connected to two end points
  - Many examples: fully connected, ring , crossbar, (fat) tree, mesh, torus, hypercube, …
  - High latency (many “*hops*”), higher bandwidth per element
    - Scales to 1000s of elements
  - Complex protocols (e.g., cache coherence)

Can have different types at different levels of hierarchy!

Example: On-Chip Interconnect

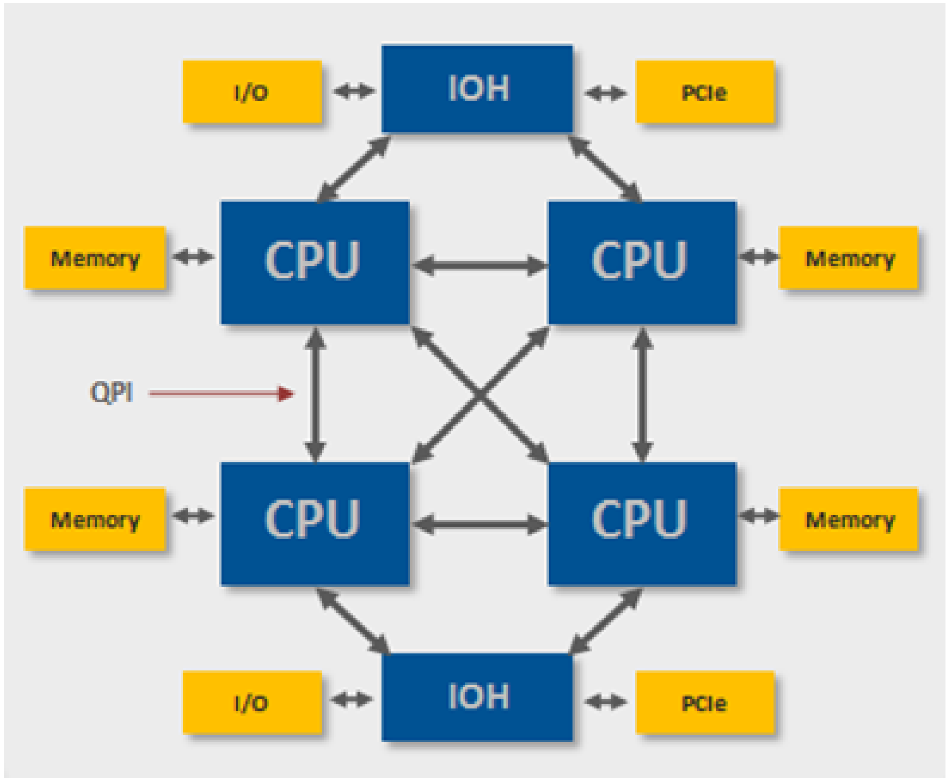
- Intel Xeon® E5-2600 family
  - Multi-ring interconnect
  - Connecting 8 cores and 8 L3 banks
  - On-Chip interconnect



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

Example: Board-Level Interconnect

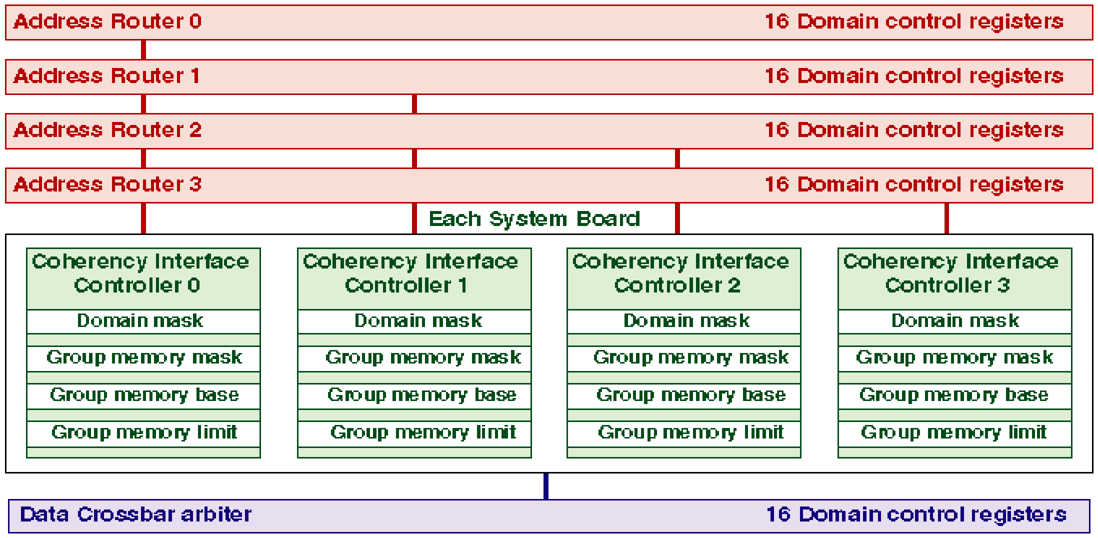
- Intel Quick Path Interconnect (QPI)
  - Off-chip interconnect
  - Fully connected
  - Connecting processor sockets to each other and IO hubs
  - Memory directly connected to processor sockets using a DDR bus



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

Example: Board-Level Interconnect

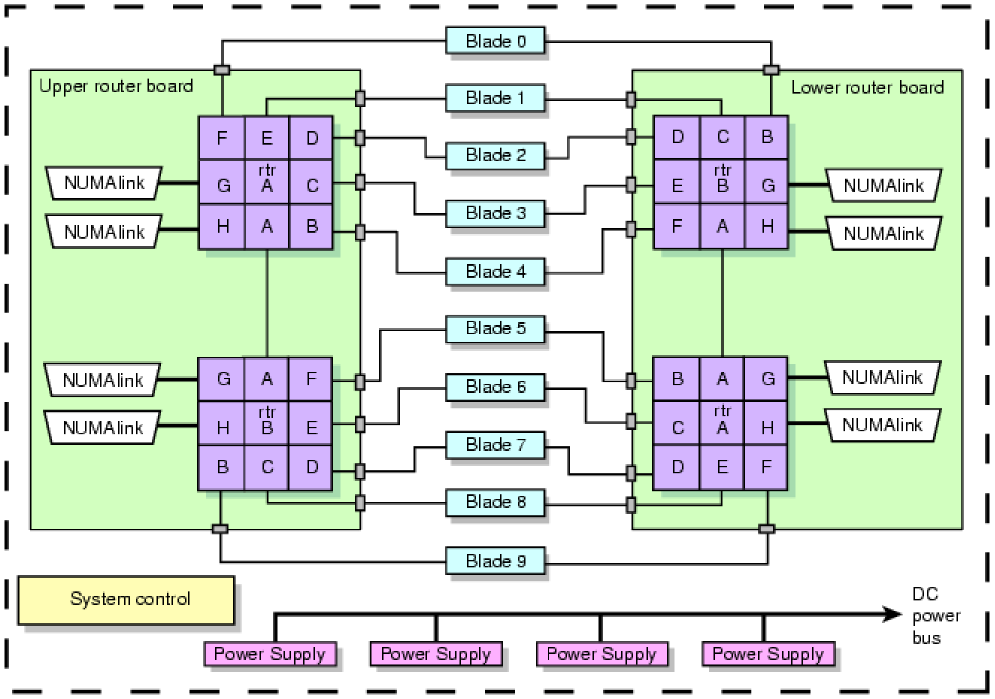
- Sun Starfire Interconnect
  - Separate Address and Data networks
  - Partitioned bus for address
    - Bus to simplify coherence protocol
    - Partitioned to improve bandwidth
  - Crossbar for data (to improve bandwidth)



Source: http://www.filibeto.org/~aduritz/truetrue/e10000/starfire-interconnect.html

Example: Rack-Level Interconnect

- Altix machines are shared-memory supercomputpters
- SGI Altix 4700
  - Crossbar switches
  - Connecting nodes (server blades) to each other



Source: http://techpubs.sgi.com/library/manuals/4000/007-4823-001/sgi_html/ch03.html

Issues for Shared Memory Systems

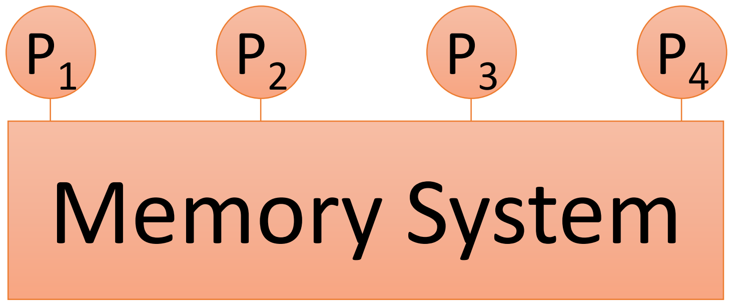
Four major issues:

1) Interconnection network design
   - A course on its own
   - not covered in here; see course schedule for some readings
2) Cache coherence
3) Memory consistency model
4) Synchronization support
   - Locks, barriers, etc.

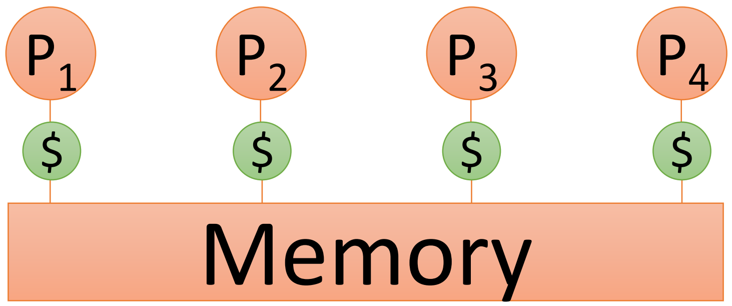
Cache Coherence

Cache Coherence: The Problem (1)

- 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

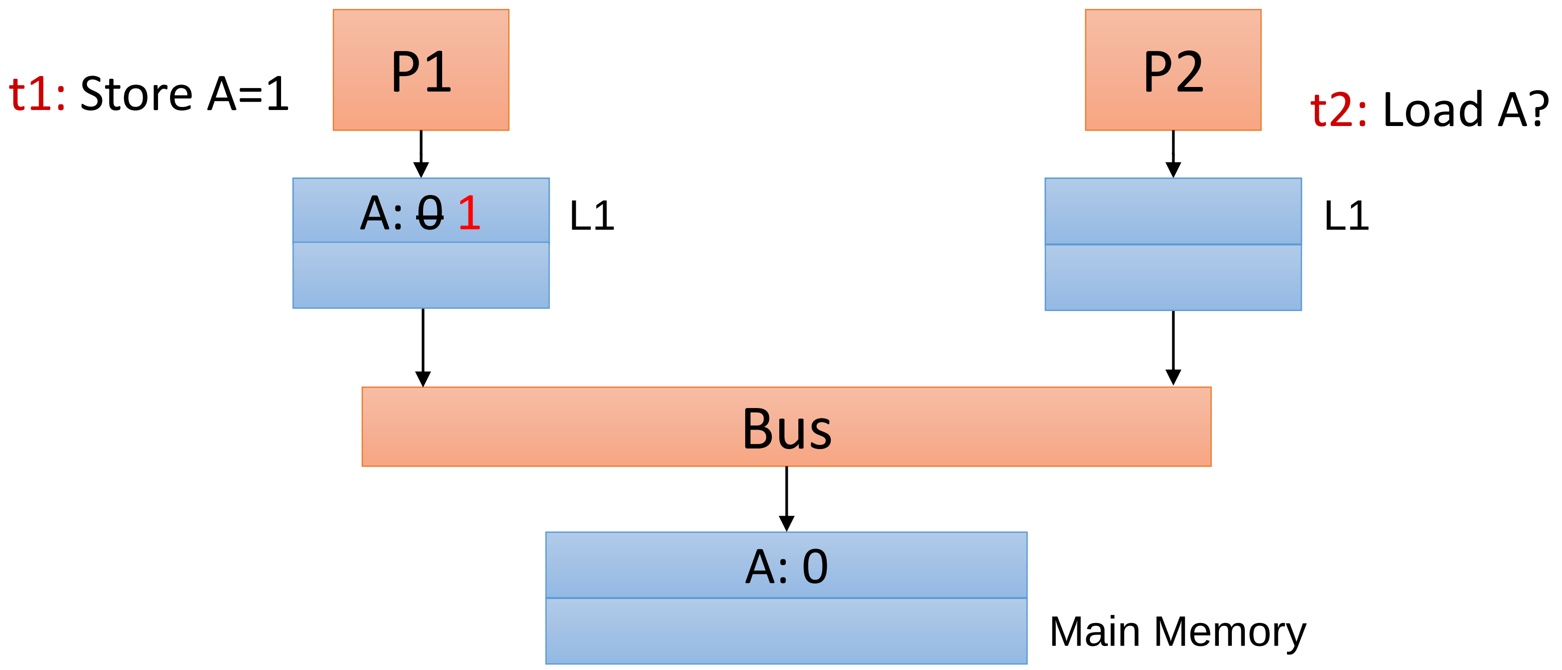


More Realistic View

Should propagate one processor's write to others

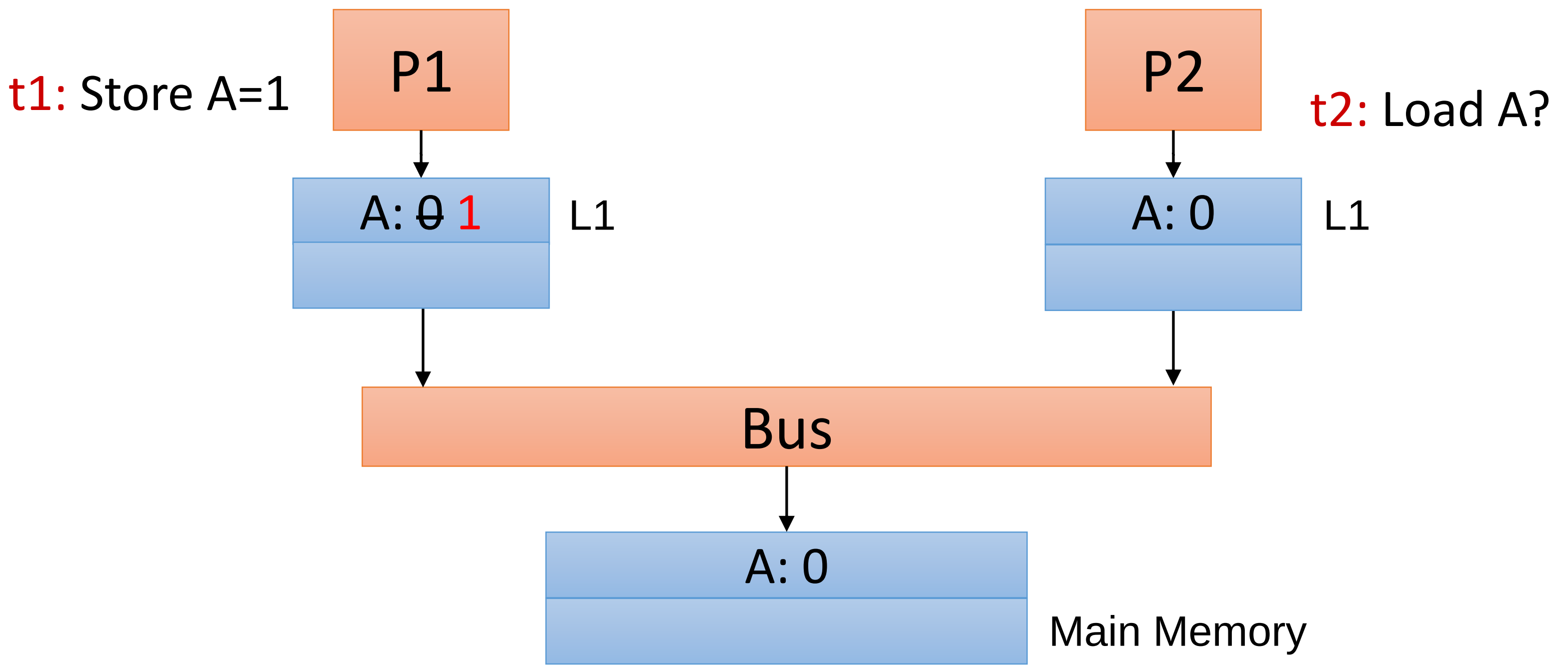
Cache Coherence: The Problem (2)

- 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)

- 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 locations 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

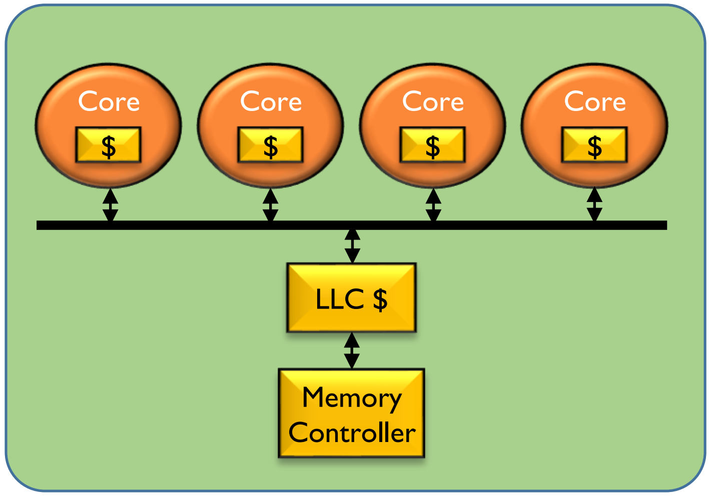
- Hardware solutions are far more common
  - System ensures everyone always sees the latest value

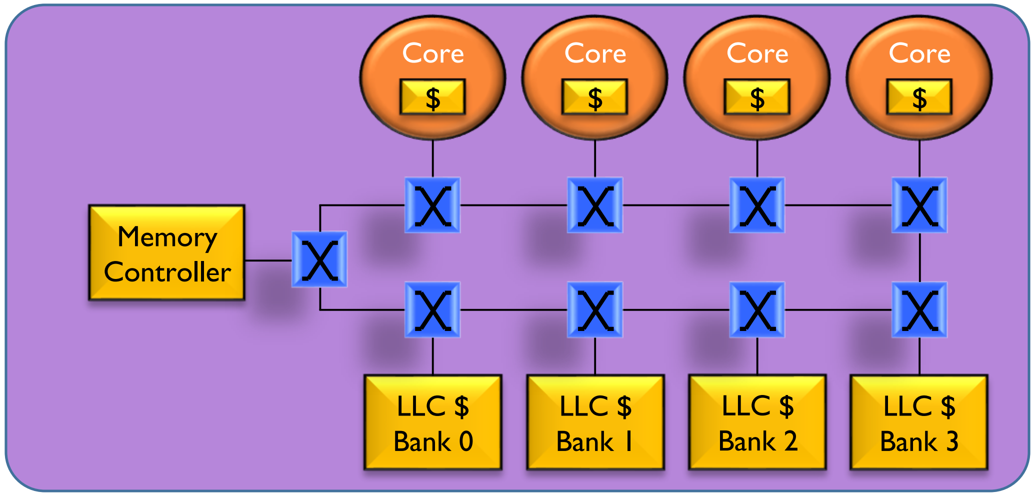
Two important aspects

- ***Update*** vs. ***Invalidate***: on a write
  - update other copies, or
  - invalidate other copies
  - Invalidation protocols are far more common (our focus)
- ***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***)

Snoopy Protocols

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

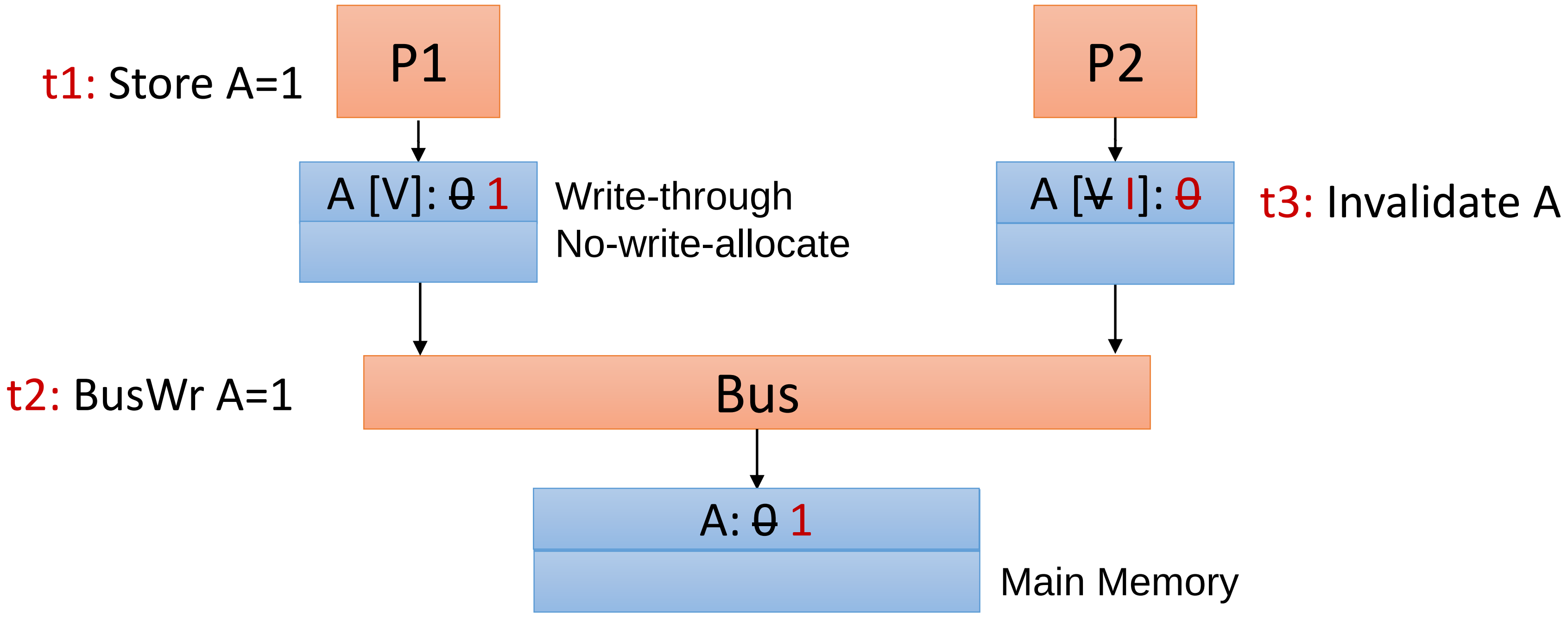




- All caches must monitor (aka "*snoop*") all traffic
  - And keep track of cache line *states* based on the observed traffic

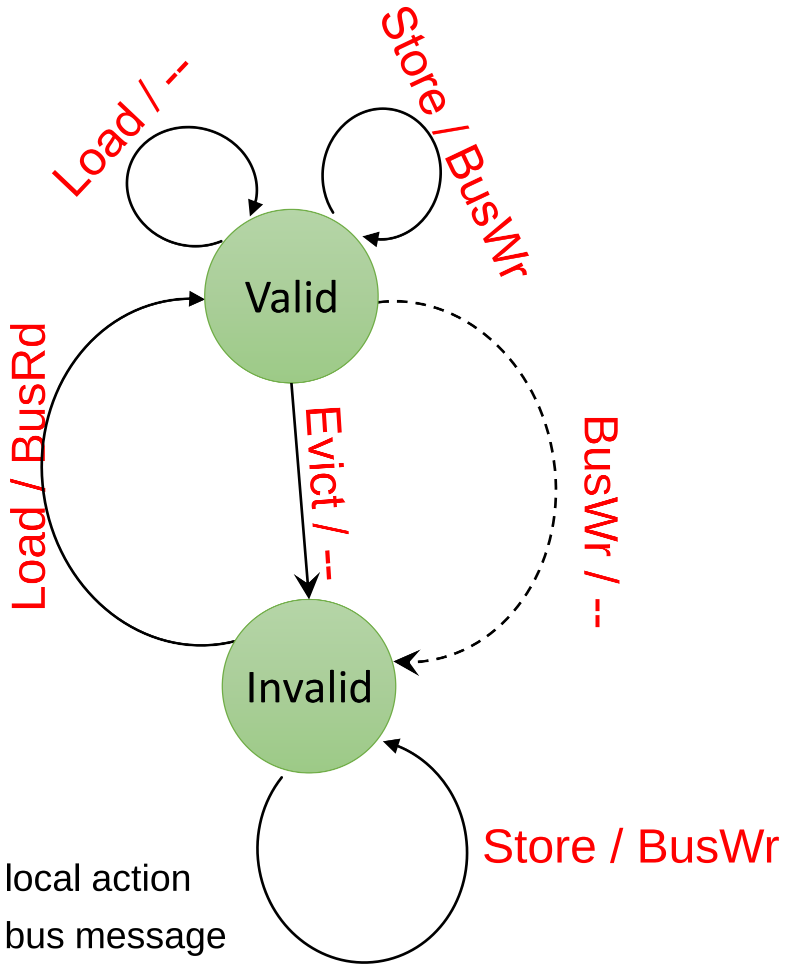
Example 1: Snoopy w/ Write-through $

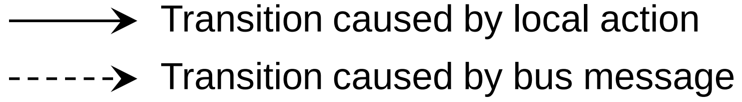
- Assume write-through, no-write-allocate cache
- Allows multiple readers, but writes through to bus
- Simple state machine for each cache frame



Valid/Invalid Snooping Protocol

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




Transition caused by local action
Transition caused by bus message

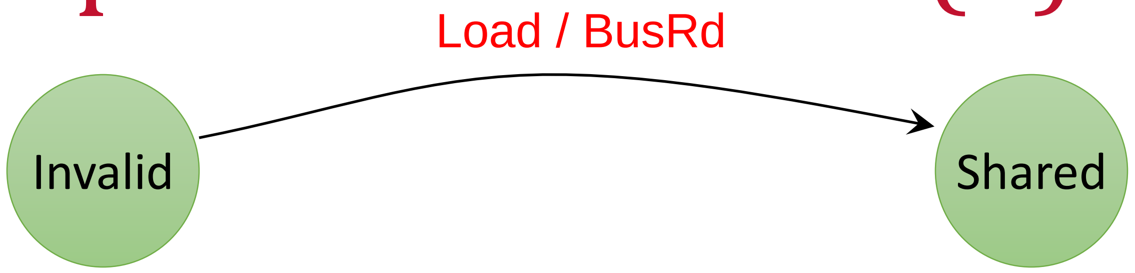
Example 2: Supporting Write-Back $

- 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

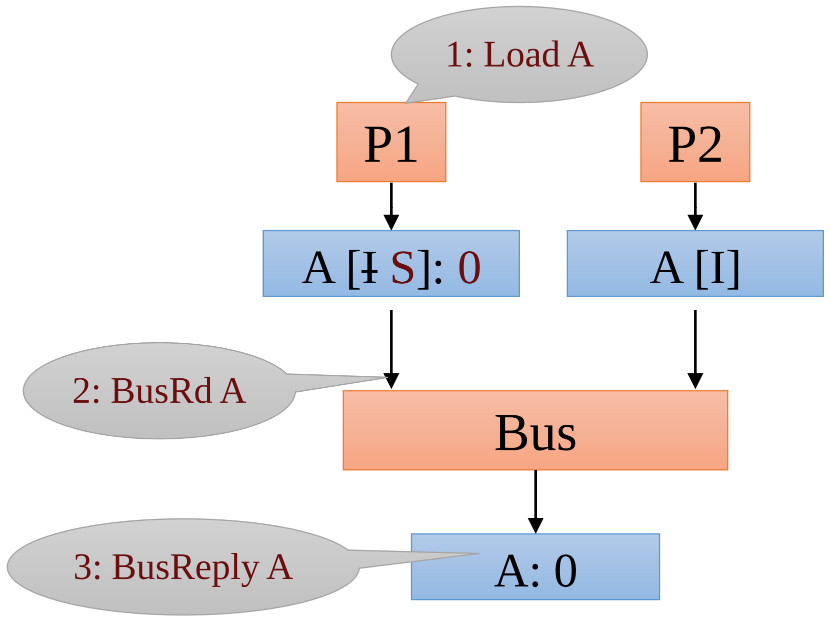
Modified/Shared/Invalid (MSI) States

- Each cache, tracks 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)

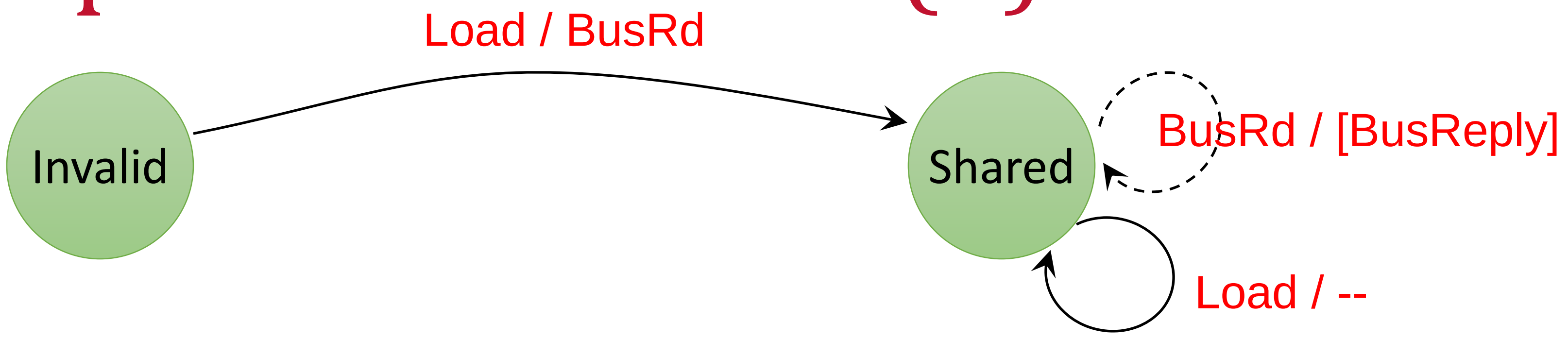
Simple MSI Protocol (1)



$\longrightarrow$ Transition caused by local action

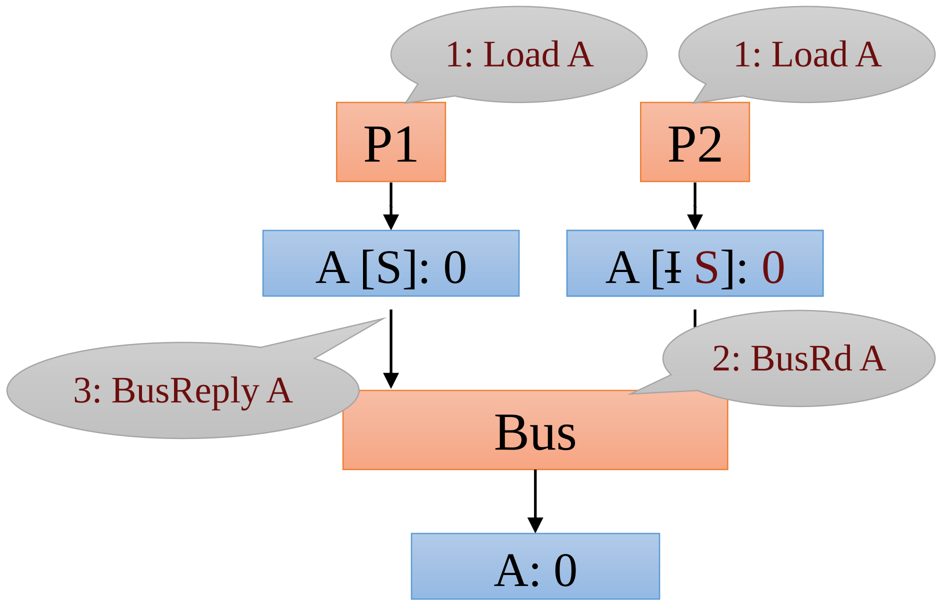


Simple MSI Protocol (2)

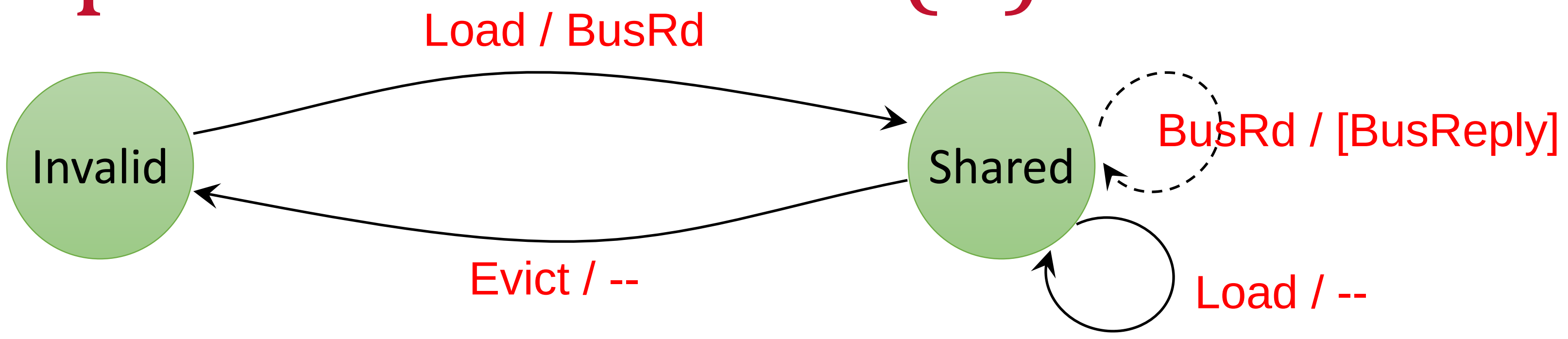


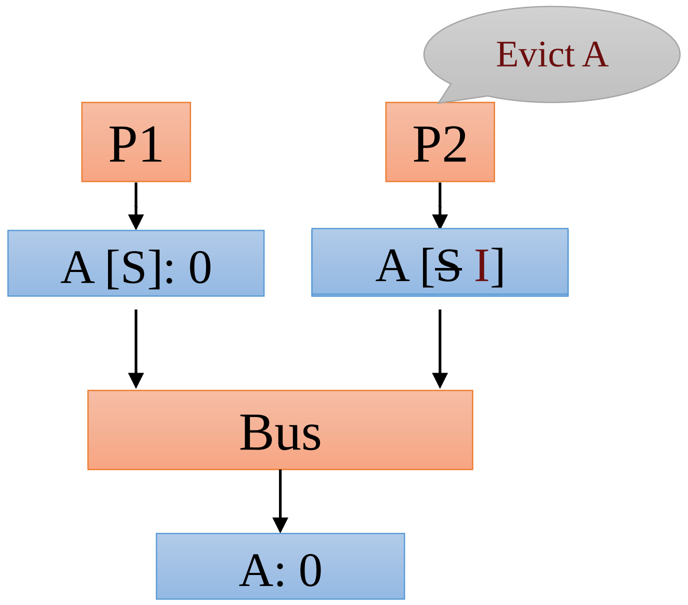
→ Transition caused by local action

⇢ Transition caused by bus message

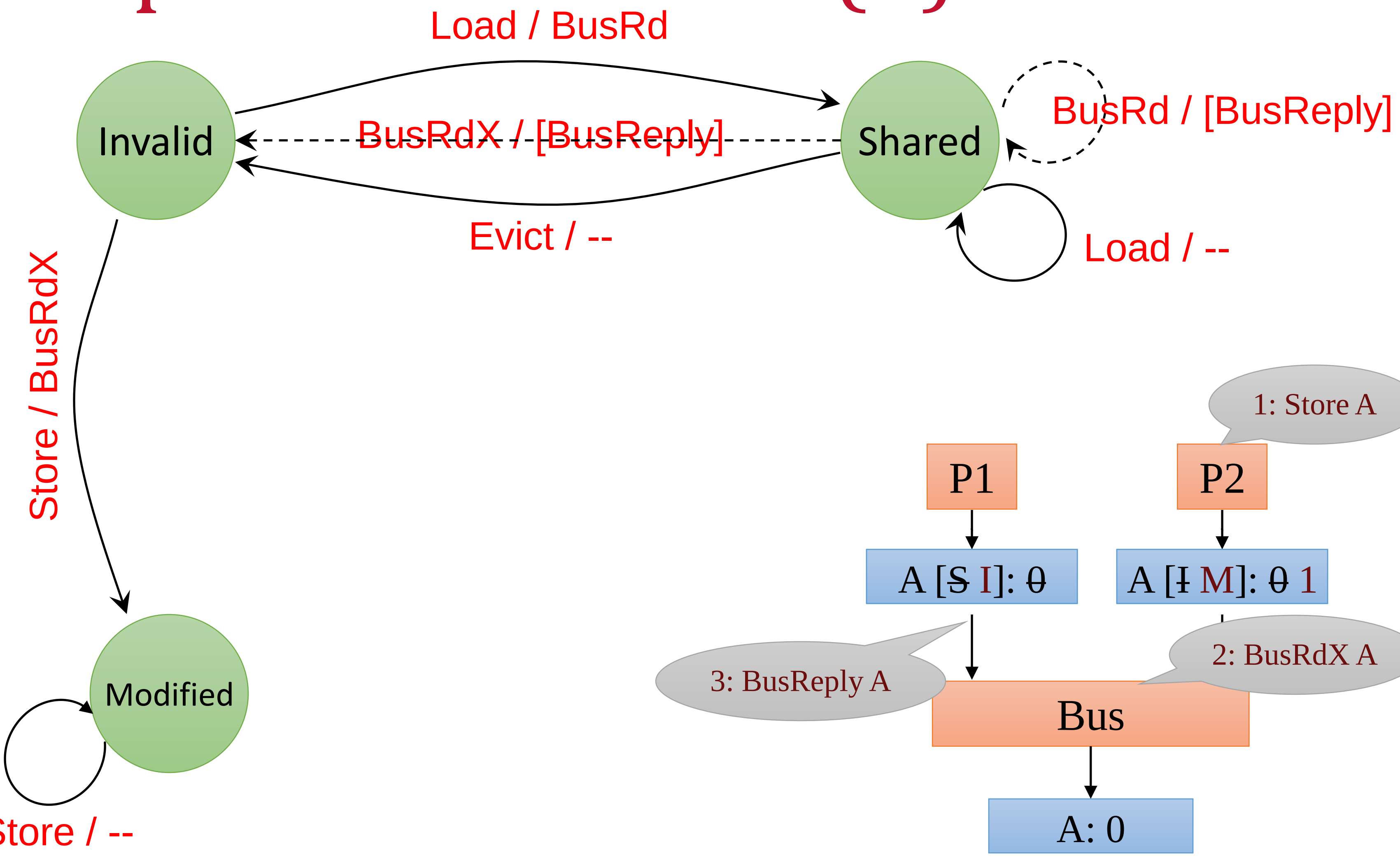


Simple MSI Protocol (3)

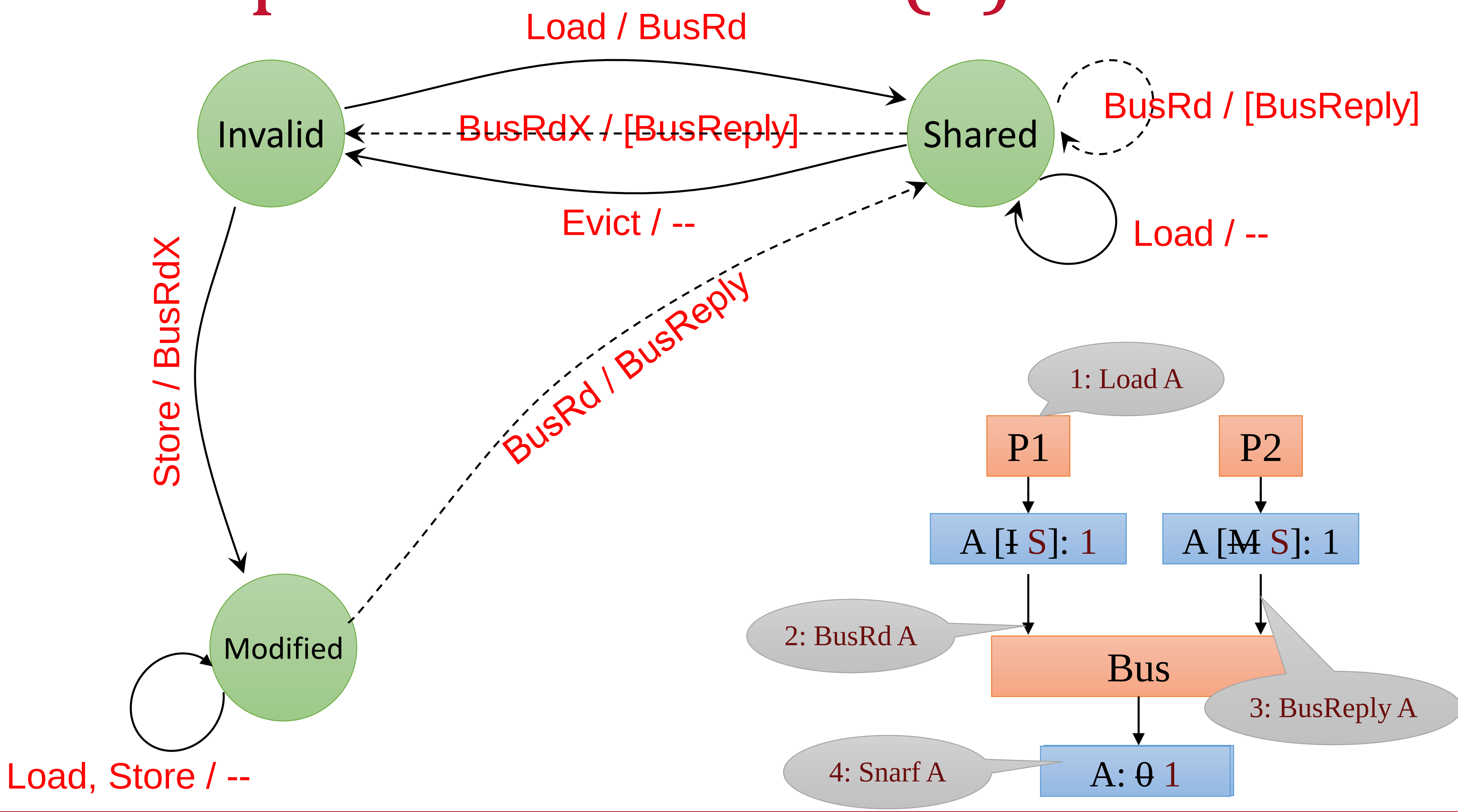




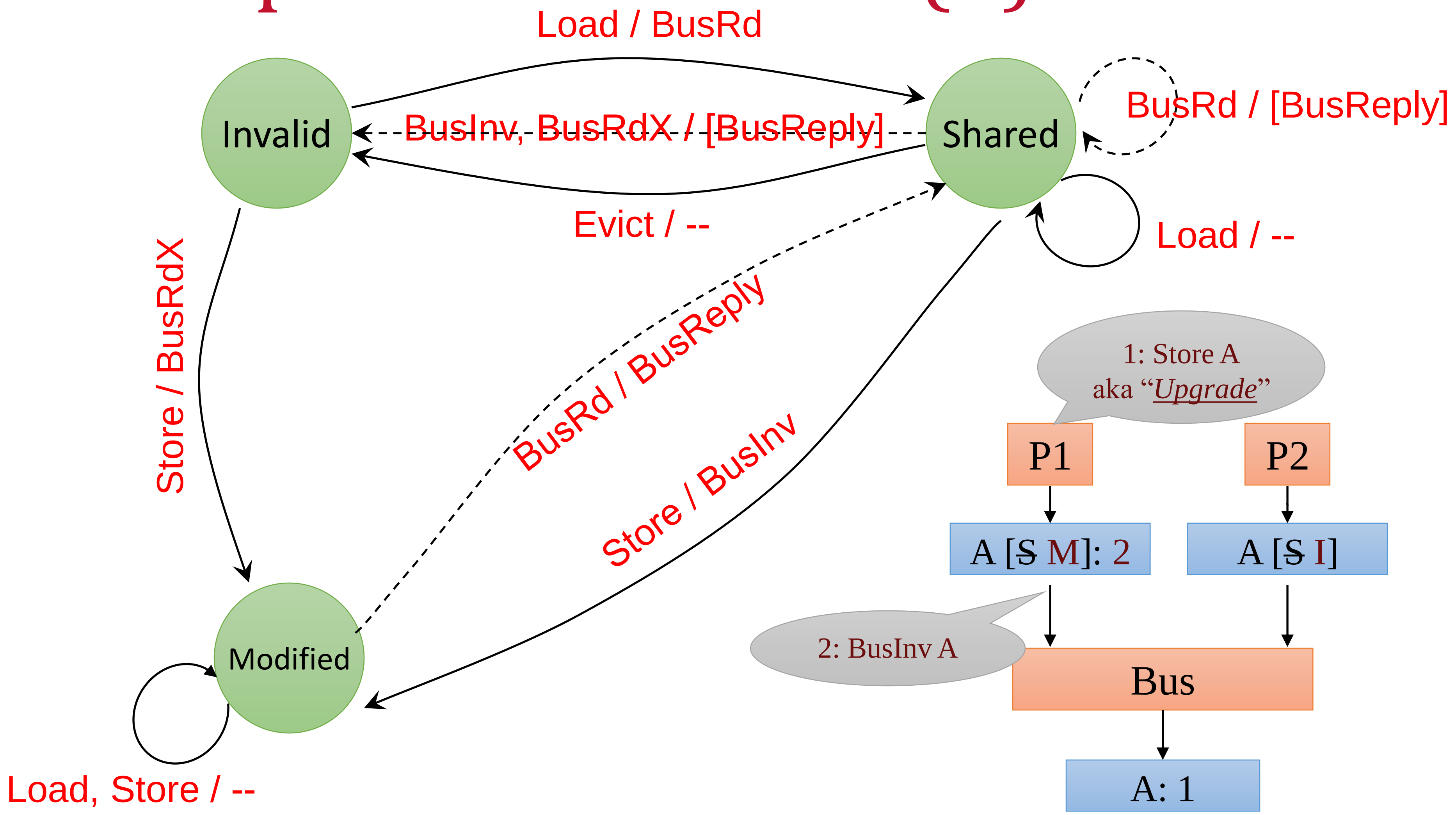
Simple MSI Protocol (4)



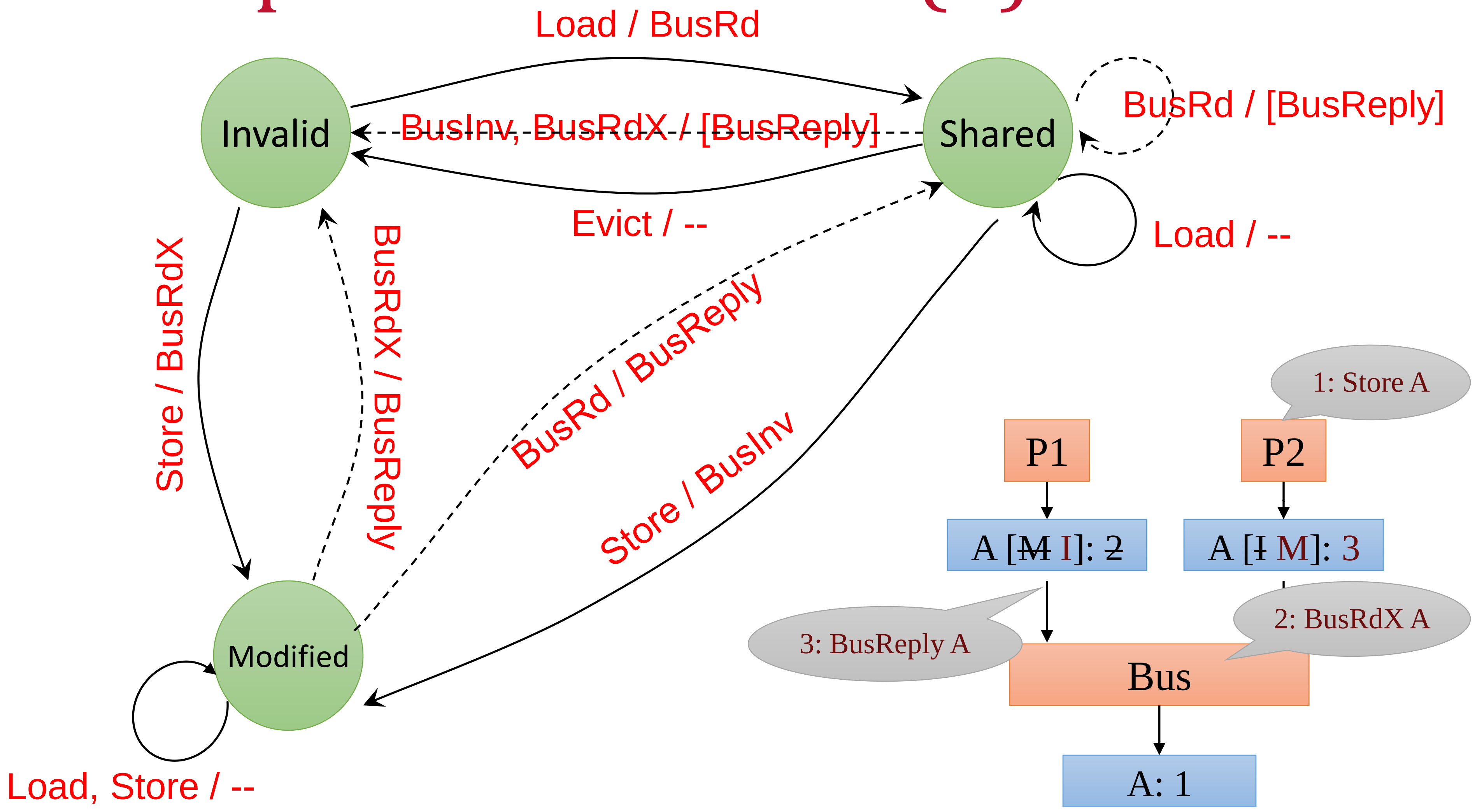
Simple MSI Protocol (5)



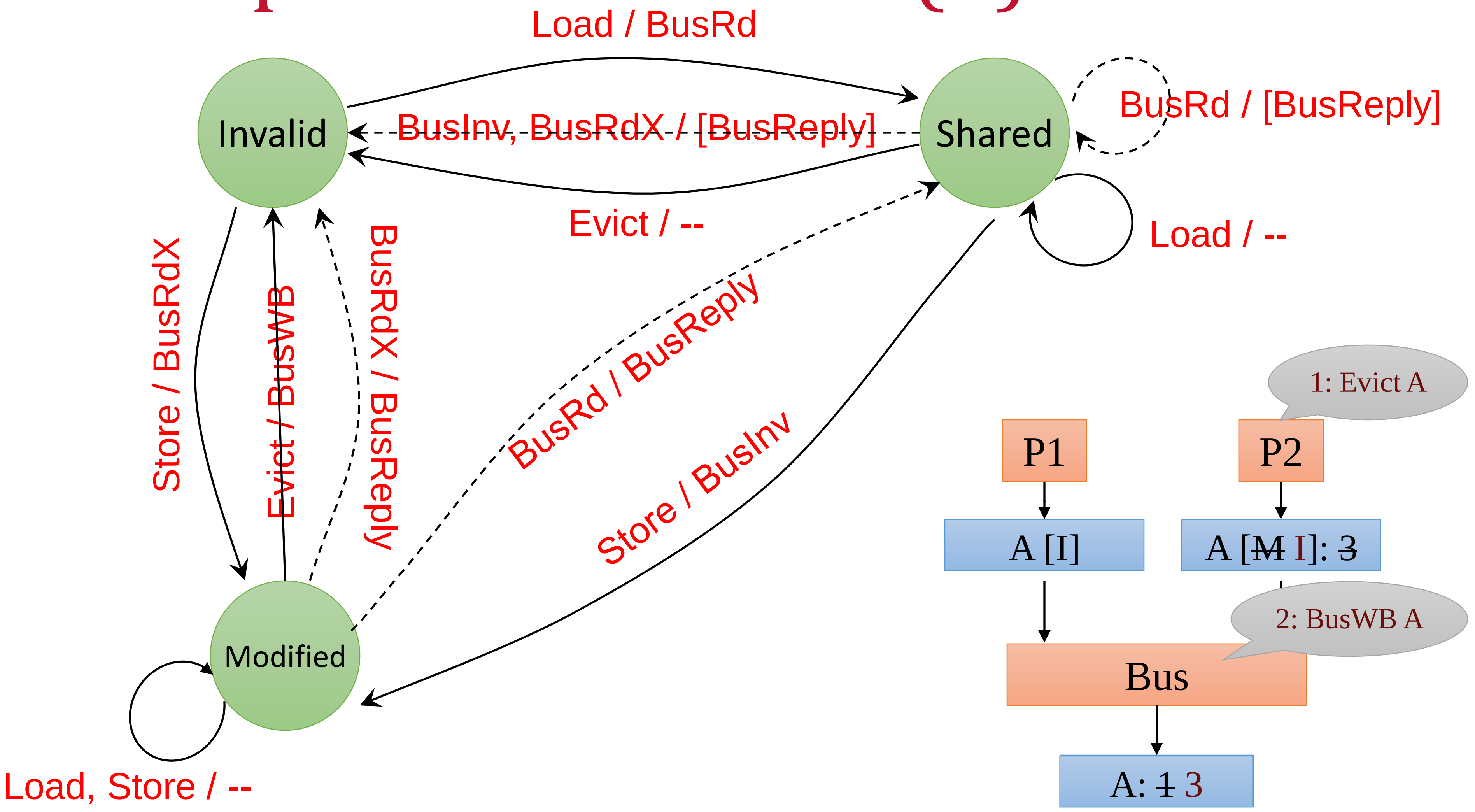
Simple MSI Protocol (6)



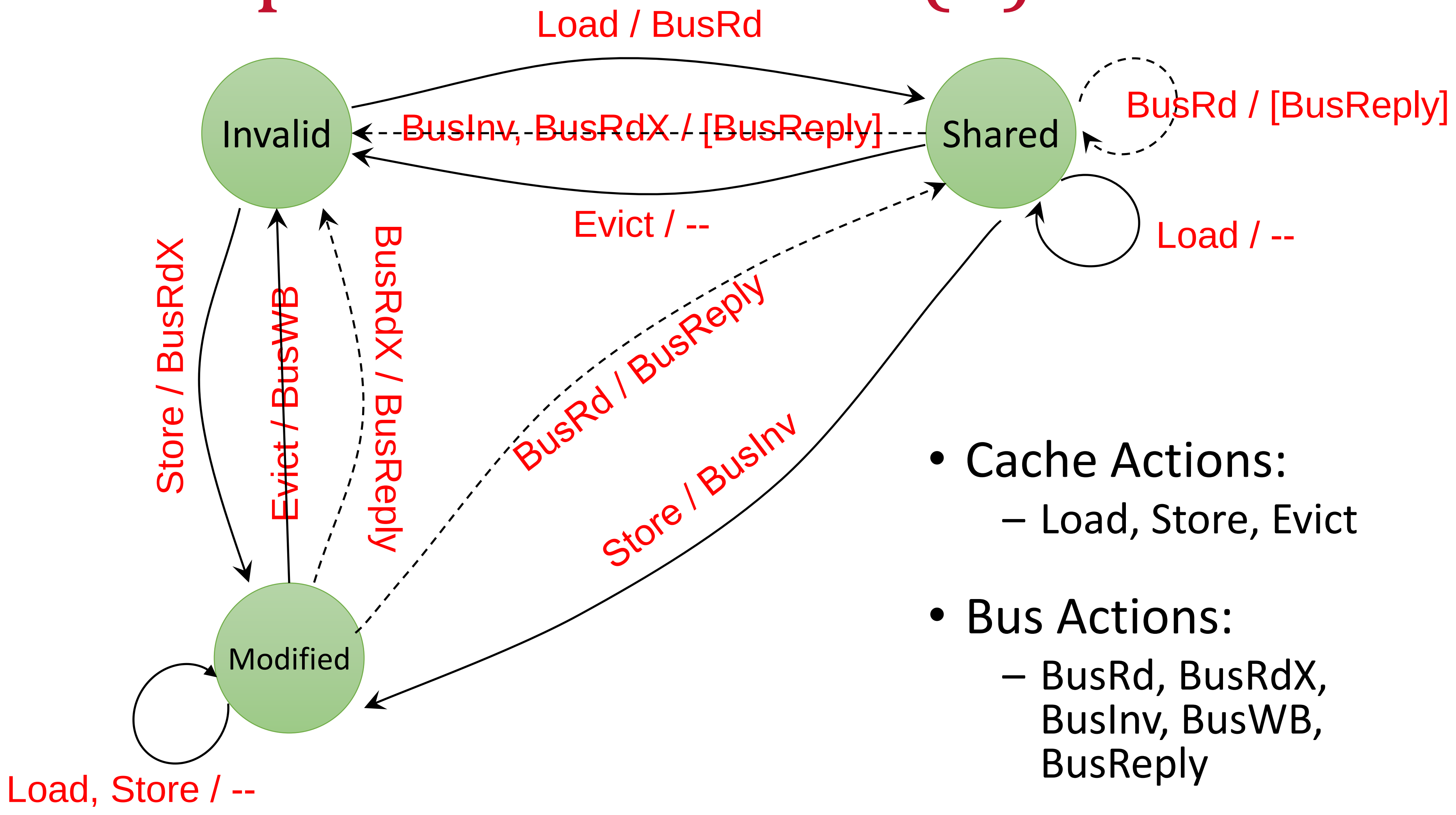
Simple MSI Protocol (7)



Simple MSI Protocol (8)



Simple MSI Protocol (9)



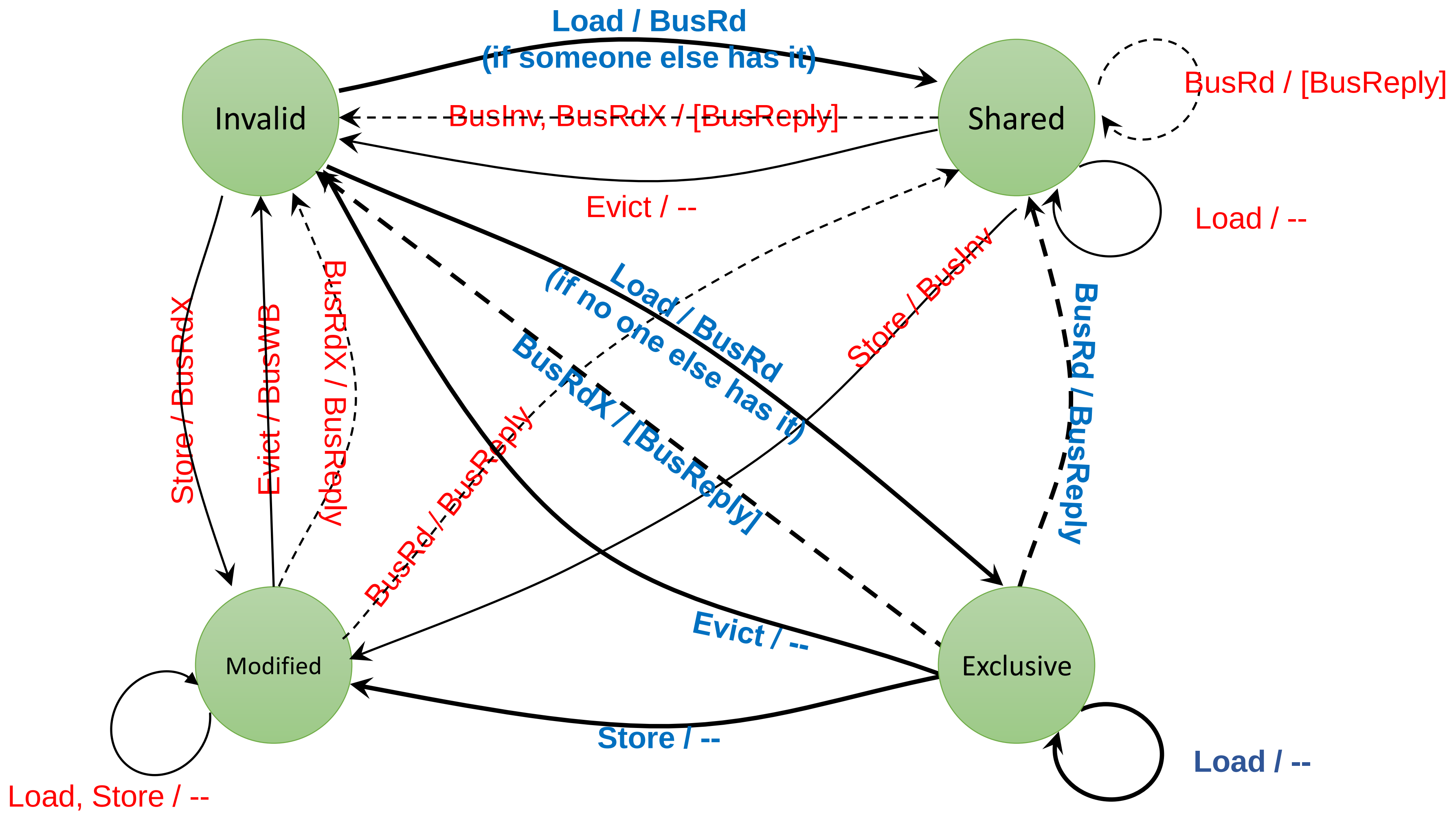
- Cache Actions:
  - Load, Store, Evict
- Bus Actions:
  - BusRd, BusRdX, BusInv, BusWB, BusReply

Usable coherence protocol

MESI Protocol (1)

- States: Invalid, Exclusive, Shared, Modified
  - Called ***MESI*** ☺
  - Variations widely used in real processors
- Two features :
  - The cache knows if it is the only copy (Exclusive state)
  - 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

MESI Protocol (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 interconnect (ring or bus) with a scalable one (e.g., mesh)

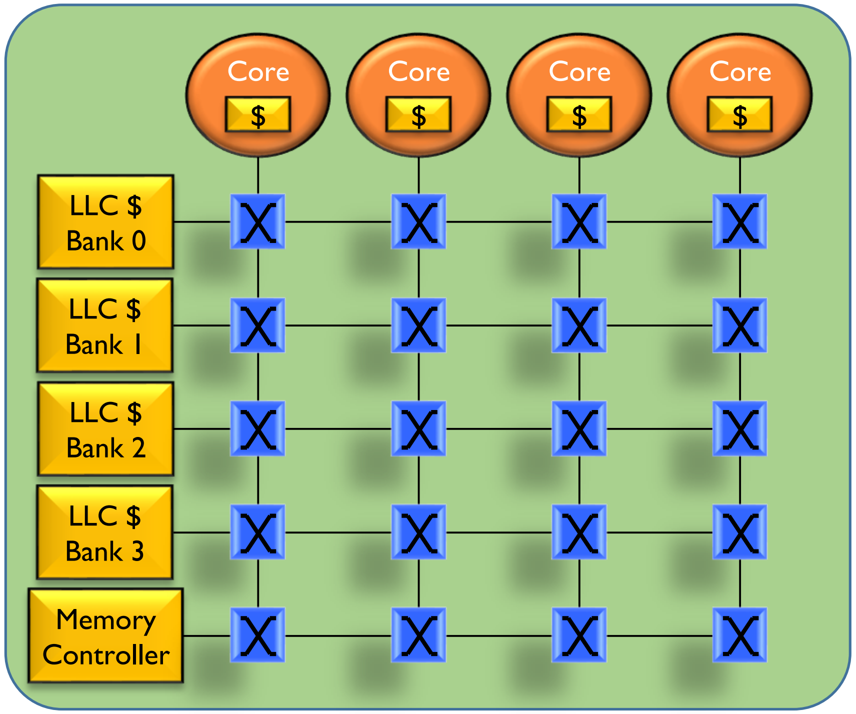
2) Cache snooping bandwidth
   – Problem: All caches must monitor all bus traffic; most snoops result in no action
   – Solution: Replace non-scalable broadcast protocol (spam everyone) with scalable <u>directory</u> protocol (notify cores that care)
     • The “directory” keeps track of “sharers”

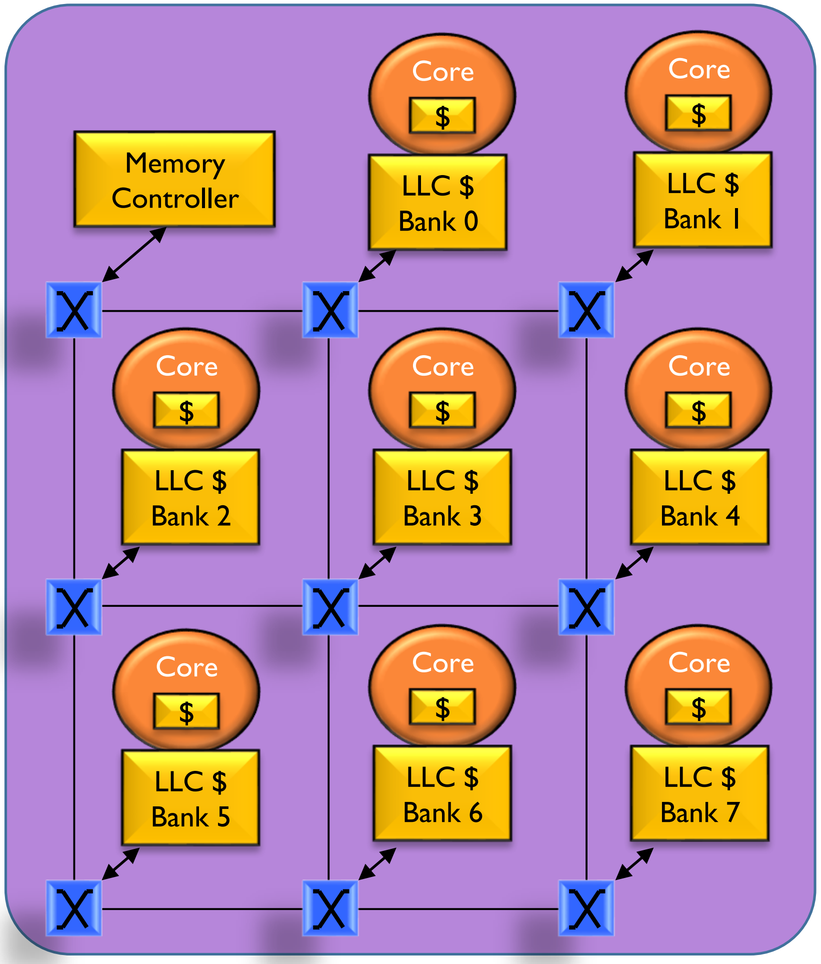
Directory Coherence Protocols

- Each physical cache line has a ***home***
- Extend memory controller (or LLC bank) to track caching information for cache lines for which it is home
  - 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 requests to home directory
- Home directory forwards messages only to cores that "care" (i.e., cores that might have a copy of the line)

Directory Coherence Protocols

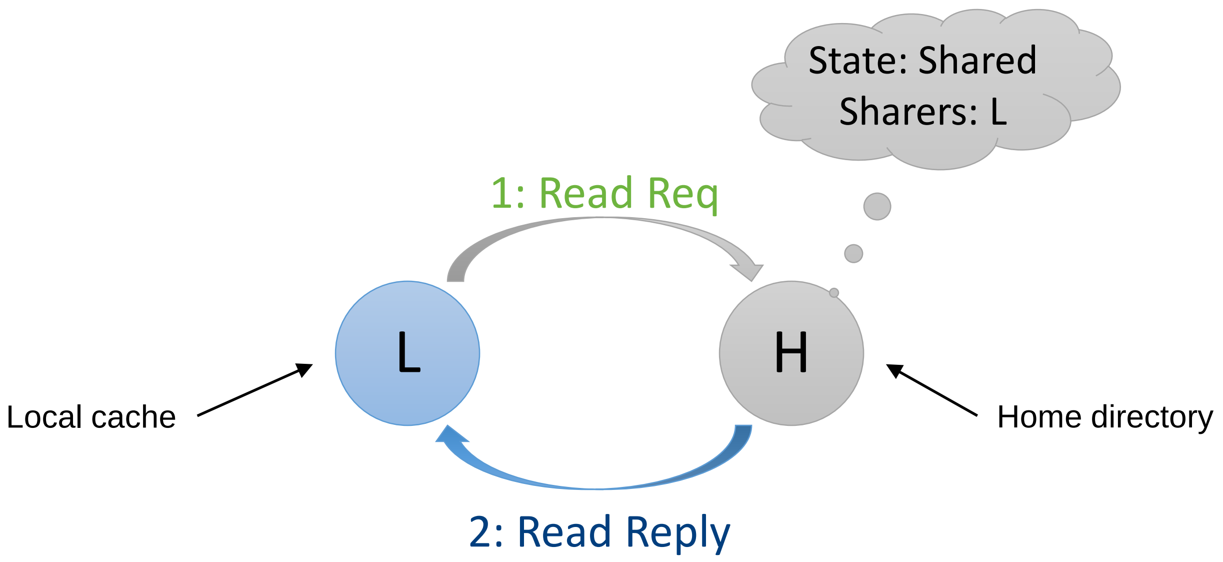
- Typically use point-to-point scalable networks
  - Such as Crossbar or Mesh





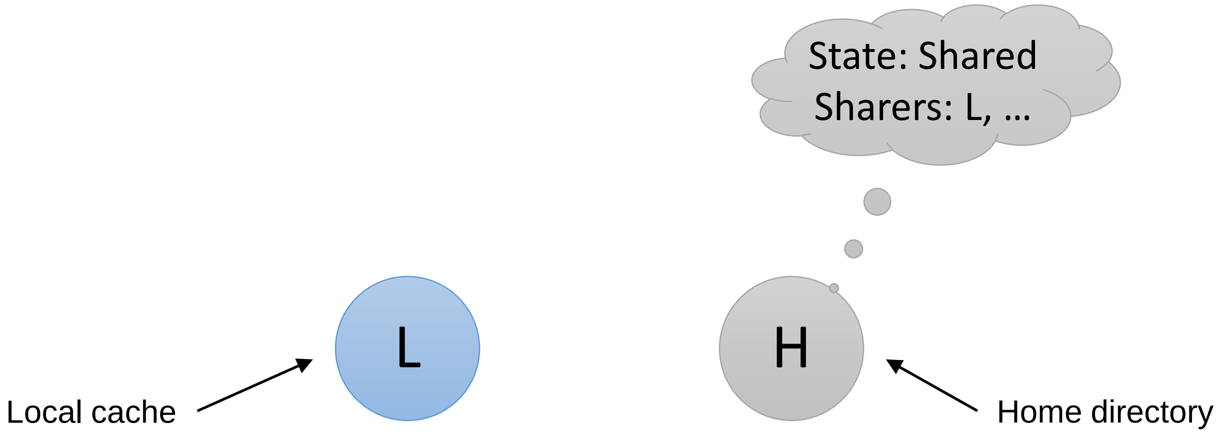
Example: 2-hop Read Transaction (1)

- L (local cache) has a cache miss on a load instruction
  - And directory indicates no or clean sharing



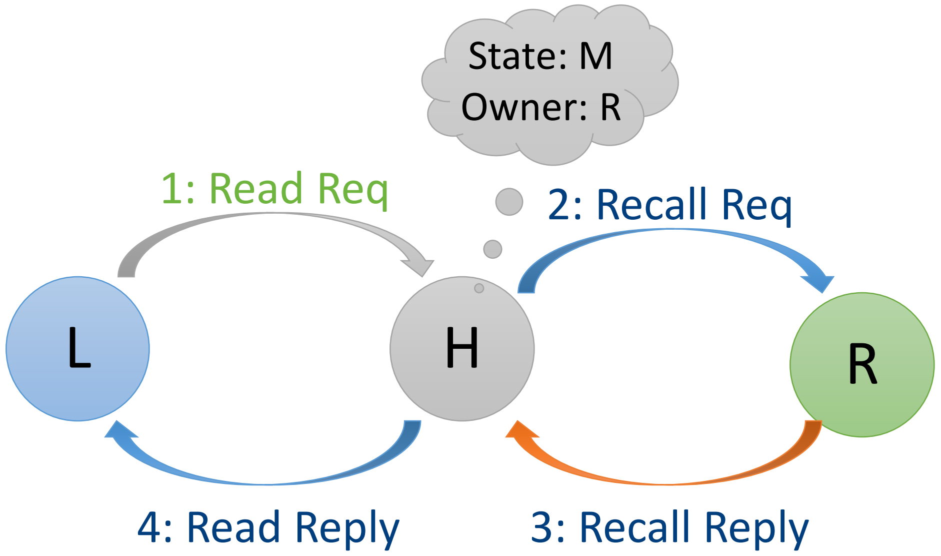
Example: 2-hop Read Transaction (2)

- L (local cache) has a cache miss on a load instruction
  - And directory indicates no or clean sharing



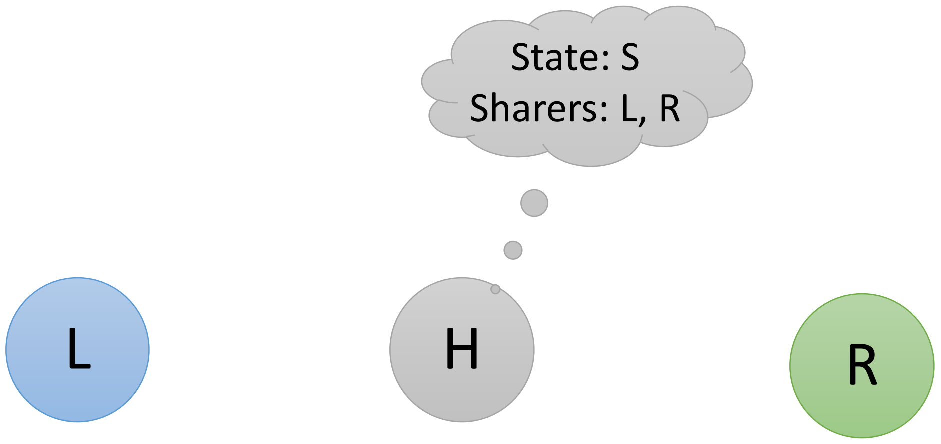
Example: 4-hop Read Transaction (1)

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



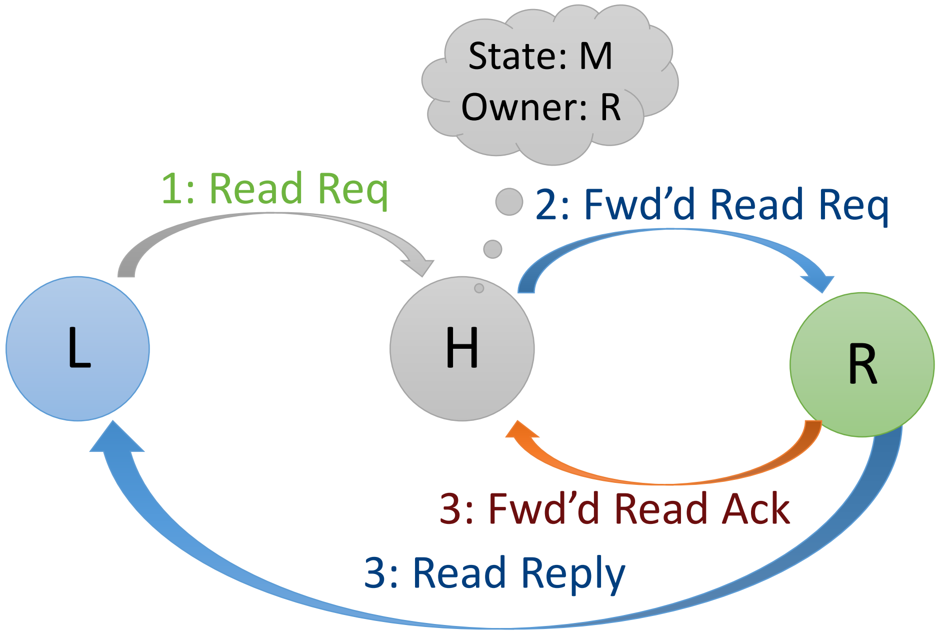
Example: 4-hop Read Transaction (2)

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



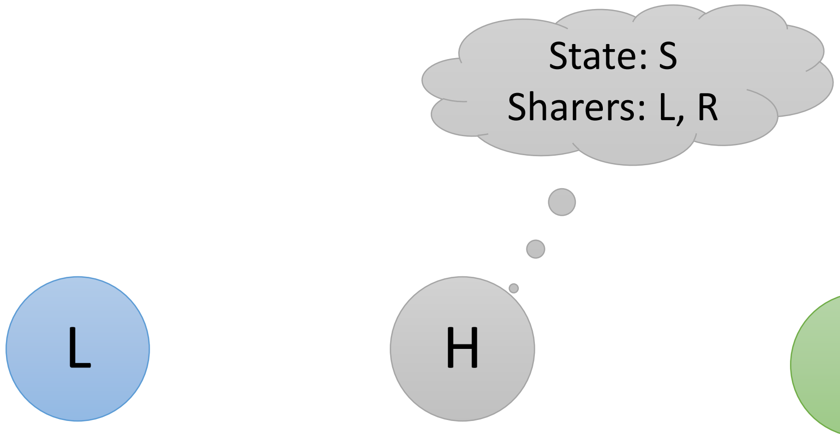
Example: 3-hop Read Transaction (1)

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



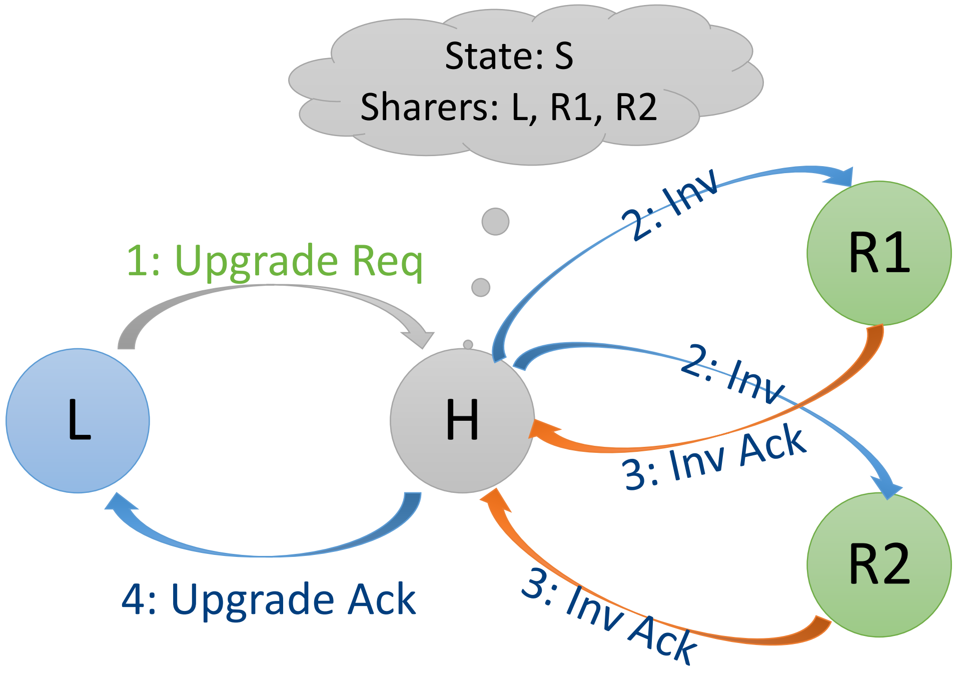
Example: 3-hop Read Transaction (2)

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



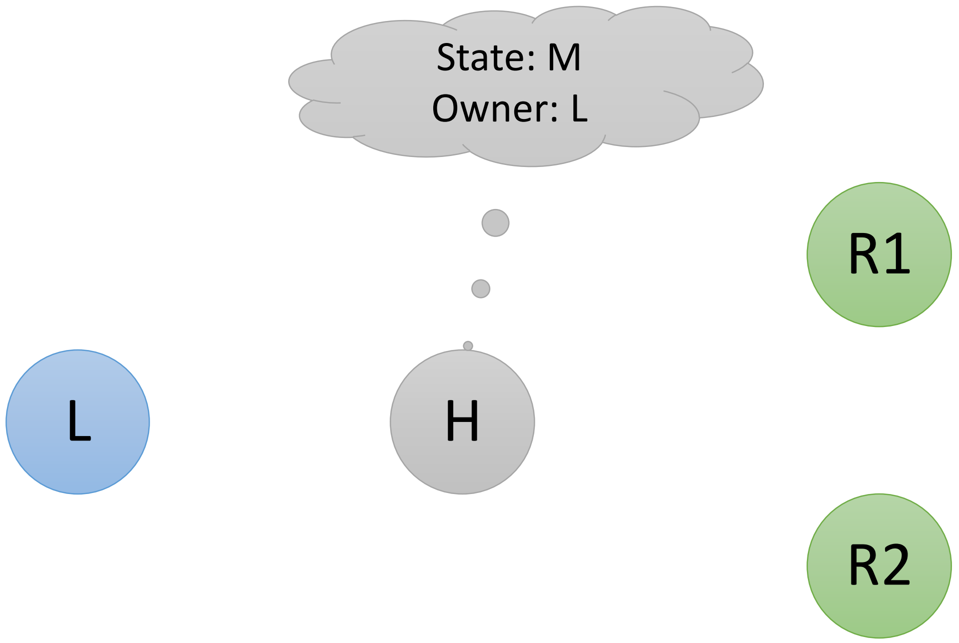
Example: 4-hop Write Transaction (1)

- L, R1 and R2 all have shared copies; L wants to do a store
  - L should be upgraded to M (as far as the directory is concered)
  - R1 and R2 should be invalidated



Example: 4-hop Write Transaction (2)

- L, R1 and R2 all have shared copies; L wants to do a store
  - L should be upgraded to M (as far as the directory is concered)
  - R1 and R2 should be invalidated



Coherence Protocols in Practice

- Cache coherence protocols are much more complicated than presented here, because of…

1) Race conditions
   - What happens if multiple processors try to read/write the same memory location simultaneously?
2) Multi-level cache hierarchies
   - How to maintain coherence among multiple levels?
3) Complex interconnection networks and routing protocols
   - Must avoid live-lock and dead-lock issues
4) 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	**Processor 2**
Store A ← 1	Store B ← 1
Load r_1 ← B	Load r_2 ← 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	Processor 2	Processor 3	Processor 4
Store A $\leftarrow$ 1	Store B $\leftarrow$ 1	Load $r_1 \leftarrow$ A	Load $r_3 \leftarrow$ B
		Load $r_2 \leftarrow$ 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	**Processor 2**	**Processor 3**

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

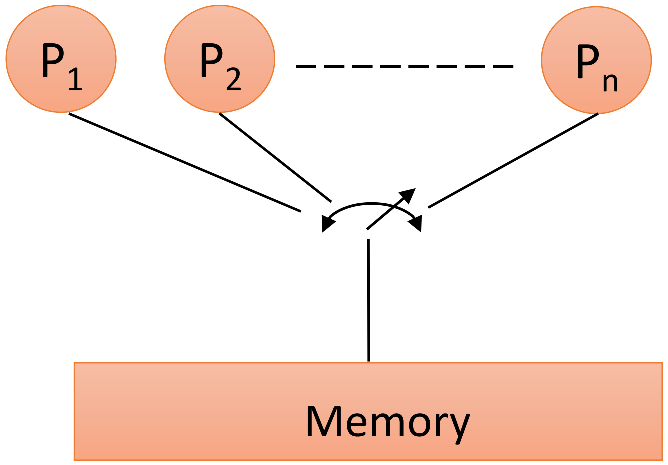
- Or just ***Memory Model***
- Given a program and its input, determines whether a particular execution/outcome is <u>valid</u> w.r.t. its memory operations
  - If yes, then execution is consistent w/ memory model
- An execution might be inconsistent w/ one model and consistent w/ another one
- You (the parallel programmer) rely on the memory model to reason about correctness of your program

Example Model: 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 of SC dictate:
    - No concurrency among memory access
    - Strict ordering of memory accesses at each processors
    - Essentially precludes most out-of-order CPU benefits

  → Conflicts with common latency-hiding techniques

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 (P2) to load B (A) before storing A (B)

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 Memory Models (1)

- SC is unnecessarily restrictive
  - Most parallel programs won't notice out-of-order accesses
  - No real processor today implements SC
- Instead, they use **Relaxed Memory Models**
  - "Relax" some ordering requirements imposed by SC
- Examples:
  - Total Store Ordering (TSO) only relaxes W → R
    - Memory model of x86 and SPARC: a read can bypass earlier writes
  - IBM Power and ARM relax almost all orderings (RW → RW)

Relaxed Memory Models (2)

- In relaxed-memory systems, programmer can use ***fence*** instructions to enforce ordering between otherwise unordered instructions

Dekker Example with fences:

Processor 1	Processor 2
Lock_A:	Lock_B:
A = 1;	B = 1;
mfence;	*mfence;*
if (B != 0) …	if (A != 0) …

- Each ISA has different types of fence instructions with different semantics
- E.g., `mfence` in x86 forces all memory instructions before the fence to complete before executing any memory instruction after the fence

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 memory ops)
  - Partial Redundancy Elimination (add memory ops)
- If interested, take a look at Java and C/C++11 memory models

Hardware Multithreading (MT)

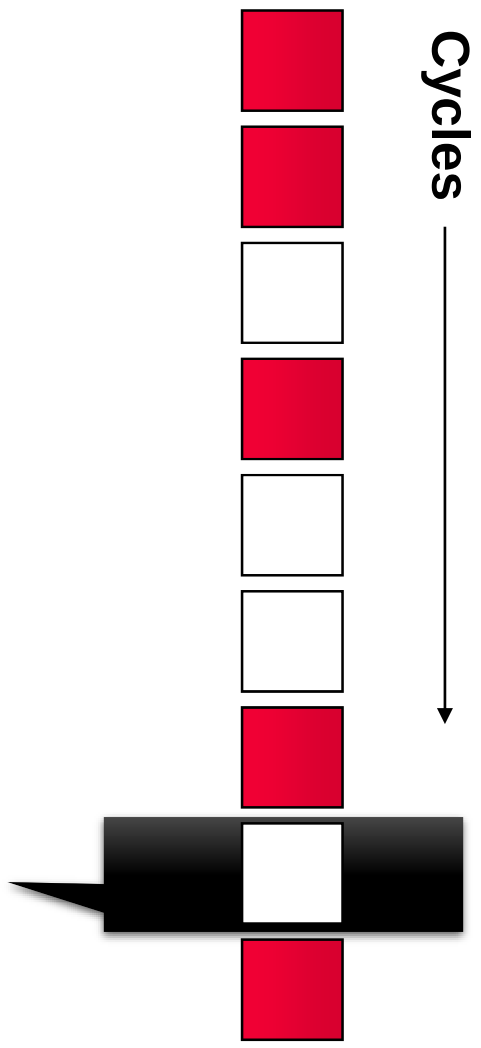
Hardware Multi-Threading

- Uni-processor: 4-6 wide, lucky if you get 1-2 IPC
  - Poor utilization of transistors
- CMP: multiple cores, but need independent threads
  - Poor utilization as well
  - Especially, 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 (and/or map table)
  - Each core appears as multiple logical CPUs to OS

Scalar Pipeline

🟥 Busy Functional Unit (or issue slot)

□ Idle Functional Unit (or issue slot)



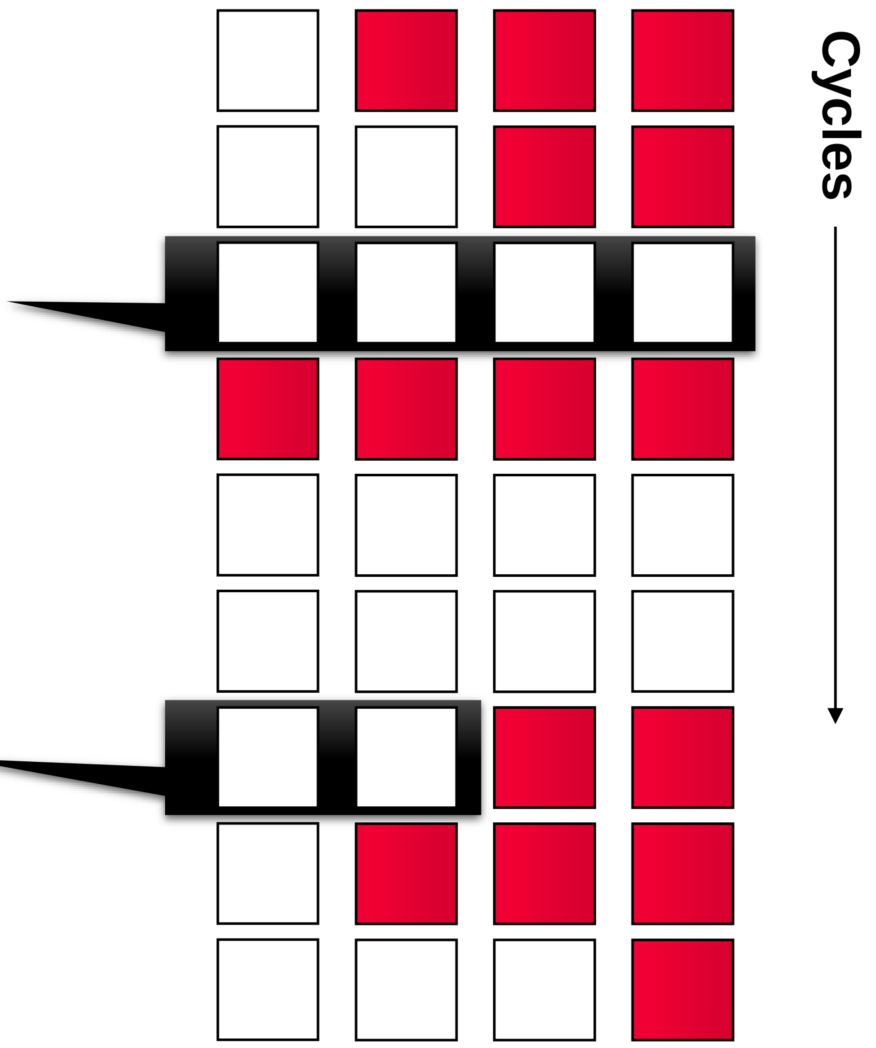
Waste: cycle in which next instruction is not issued to execute

Dependencies limit functional unit utilization

Superscalar Pipeline

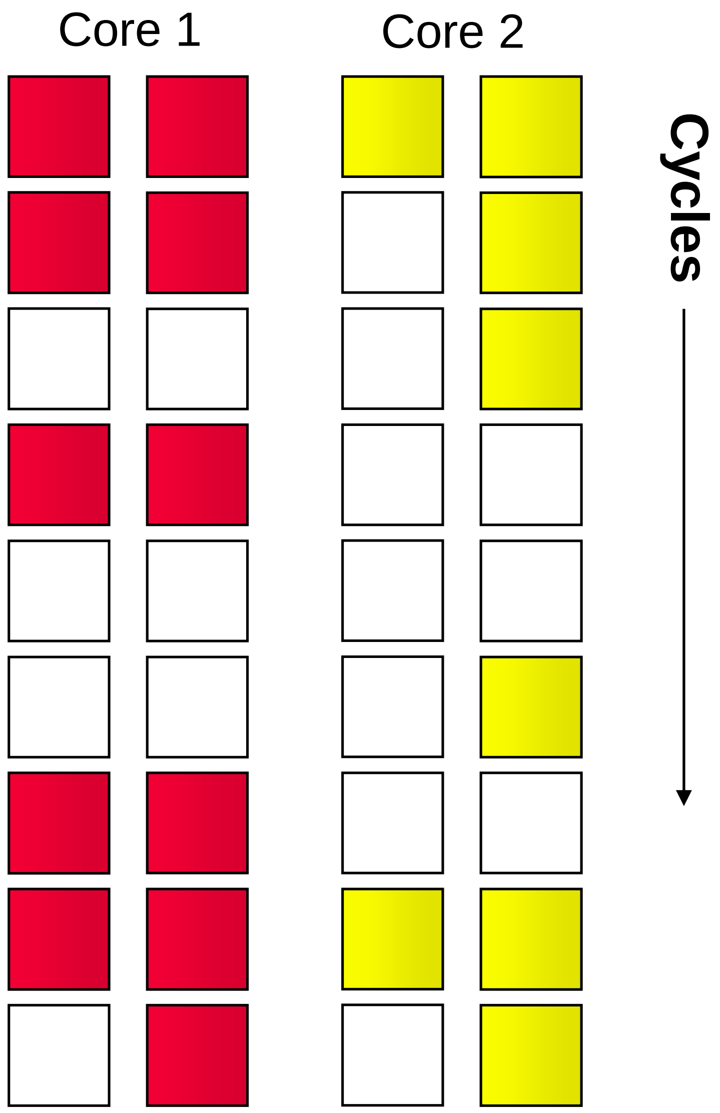
Vertical waste: cycle in which no instruction is issued
(no instruction ready to execute)

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



Higher performance than scalar, but lower utilization

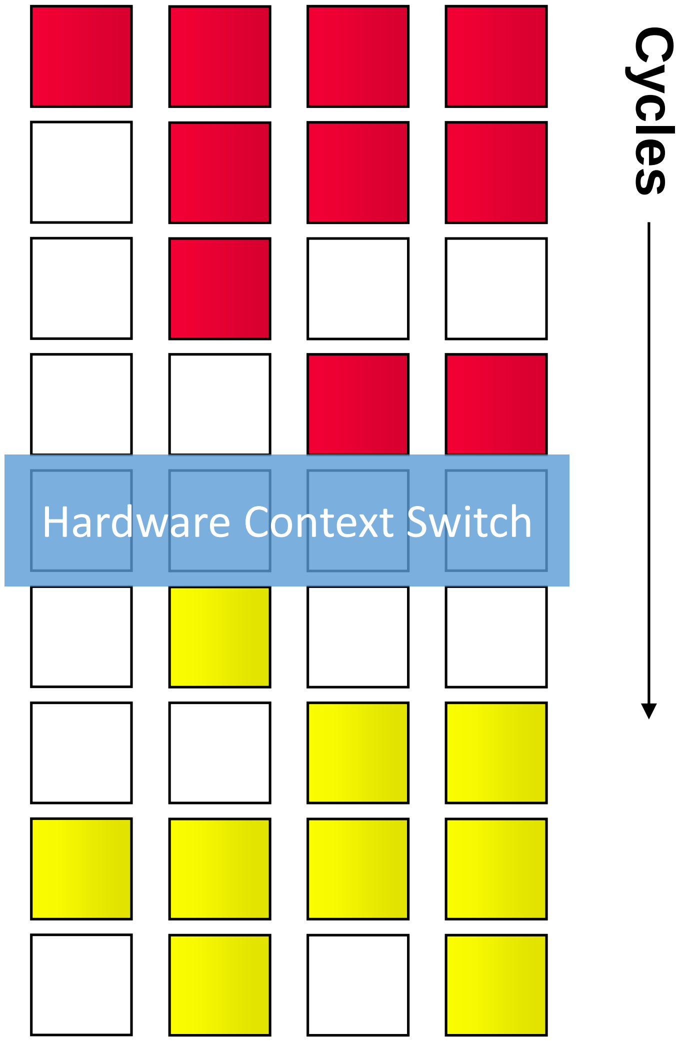
Chip-Multiprocessing (CMP)



Limited utilization when running one thread

Coarse-Grained Multithreading (1)

- Hardware switches to another thread when current thread stalls on a long latency op
  - E.g., L2 miss
- The OS should have already scheduled both threads on the CPU
  - But only one thread in the pipeline at any time

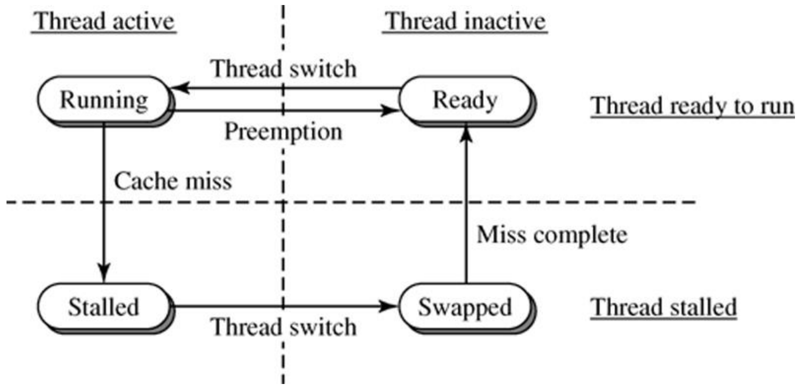


Only good for long latency ops (i.e., cache misses)

Coarse-Grained Multithreading (2)

- Needs HW "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)

✓✗ Sacrifices a little single thread performance

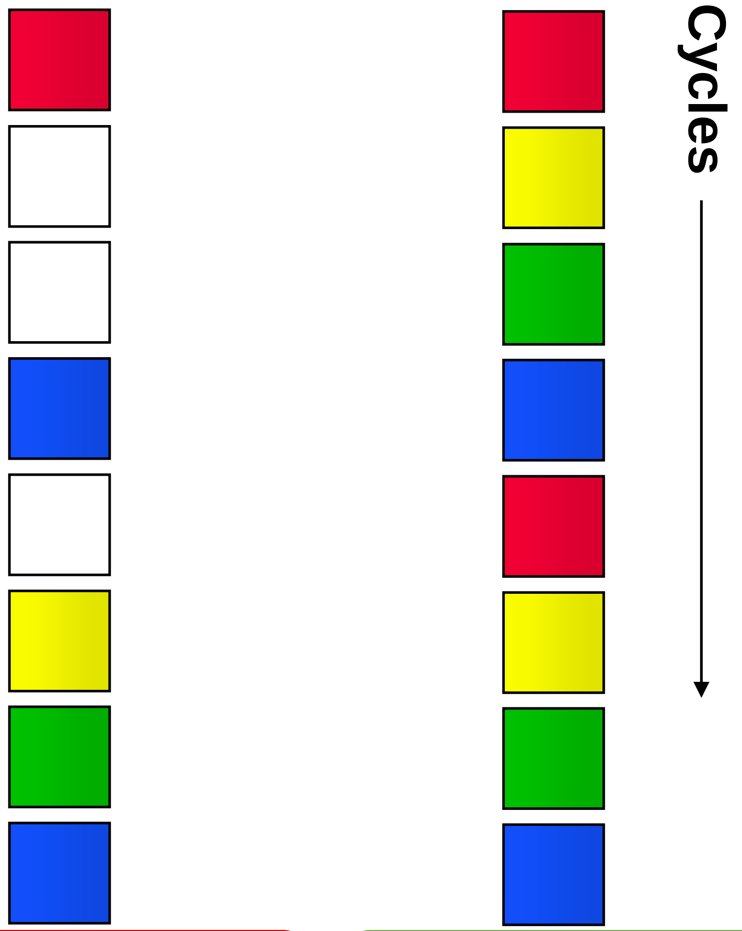
✗ Tolerates only long latencies (e.g., L2 misses)

  ✗ 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 switch cost otherwise

Fine-Grained Multithreading (1)

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



Unsaturated workload
→ Lots of stalls

Saturated workload
→ Lots of threads

Intra-thread dependencies still limit performance

Fine-Grained Multithreading (2)

- ✗ Sacrifices significant single-thread performance
- ✗ Does not eliminate horizontal waste

- ✓ Tolerates everything
  - ✓ L2 misses
  - ✓ Mis-predicted branches
  - ✓ etc...
- → Eliminates most vertical waste

- Good for throughput-bound workload, bad for latency-bound

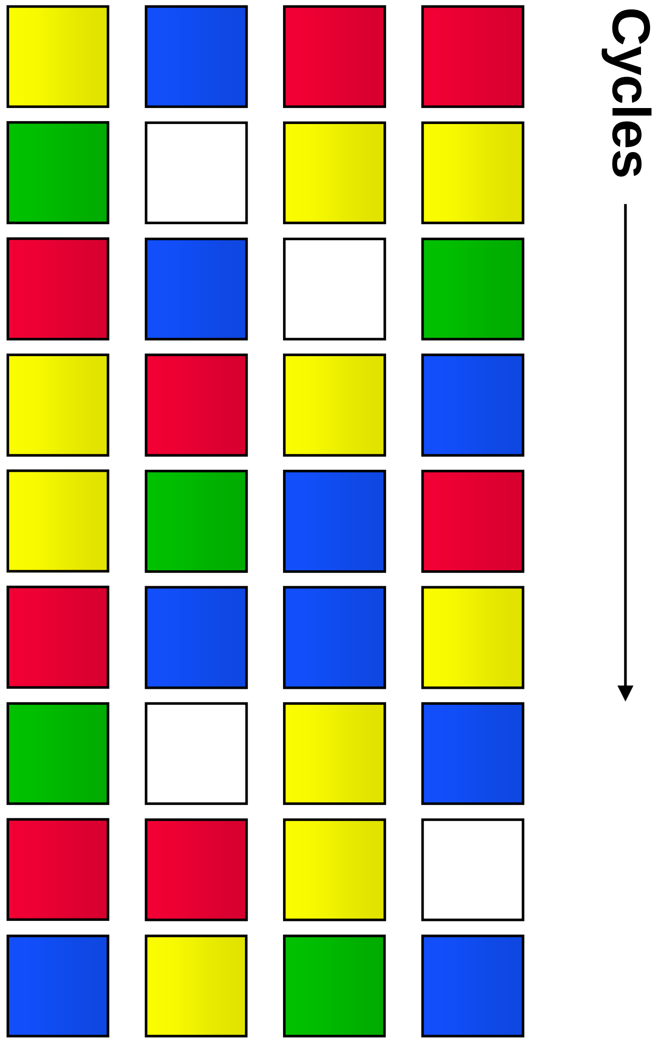
- Thread scheduling policy
  - Switch threads often (e.g., every cycle)
  - Use round-robin policy, skip threads with long-latency pending ops

- Pipeline partitioning
  - Dynamic, no flushing
  - Length of pipeline doesn’t matter

Example: Cray Threadstorm4 (128 threads per proc)

Simultaneous Multithreading (1)

- Fine- or 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)

✗ Sacrifices some single thread performance

✓ Tolerates all latencies

✓ Targets both vertical and horizontal waste

✓ A natural extension of superscalar OOO pipelines
  – Front-end handles 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 one cycle
- Pipeline partitioning
  – Dynamic
- Examples
  – Pentium 4 (hyper-threading): 5-way issue, 2 threads
  – Alpha 21464: 8-way issue, 4 threads (didn’t see light of day)

MT Issues

- Cache interference
  - Concern for all MT variants
  - Multi-threaded programs 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
    - Out-of-order execution tolerates L1 misses
- Large physical register file
  - #phys-regs = (#threads * #arch-regs) + #in-flight 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)

- Systems can have SMP, CMP, and Hardware MT at the same time
- Example x86 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)

- 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 better scheduled on different sockets/cores
- Cooperative tasks (e.g., pthreads) might be better scheduled on same core/socket
- How can the OS know?
  - Usually can't! It tries to use SMT as last resort
  - But user can set thread affinities to help