PARALLEL MEMORY ARCHITECTURE

Mahdi Nazm Bojnordi

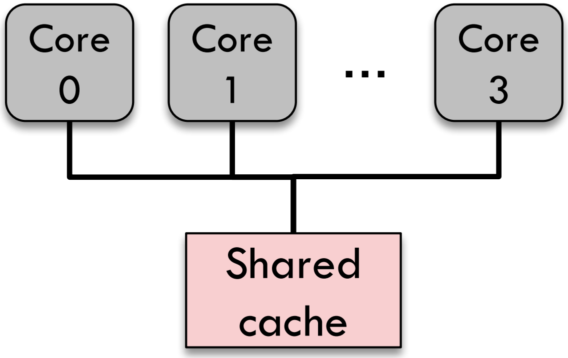
Assistant Professor

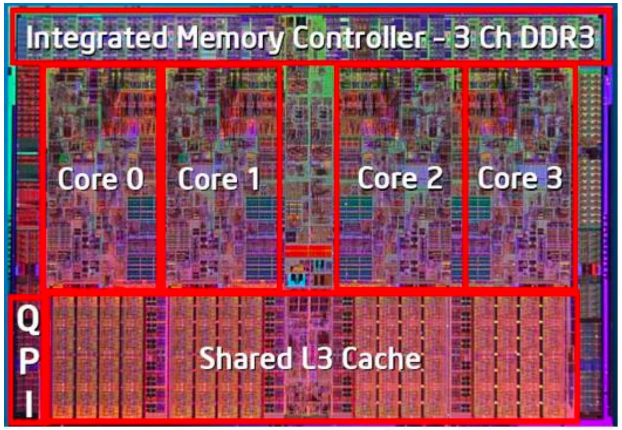
School of Computing

University of Utah

Chip Multiprocessors

- Can be viewed as a simple SMP on single chip
- CPUs are now called cores
  - One thread per core
- Shared higher level caches
  - Typically the last level
  - Lower latency
  - Improved bandwidth
- Not necessarily homogenous cores!





Intel Nehalem (Core i7)

Efficiency of Chip Multiprocessing

- Ideally, n cores provide nx performance
- Example: design an ideal dual-processor
  - Goal: provide the same performance as uniprocessor

	Uniprocessor	Dual-processor
Frequency	1	?
Voltage	1	?
Execution Time	1	1
Dynamic Power	1	?
Dynamic Energy	1	?
Energy Efficiency	1	?

Efficiency of Chip Multiprocessing

- Ideally, n cores provide nx performance
- Example: design an ideal dual-processor
  - Goal: provide the same performance as uniprocessor

$$f \propto V \ \& \ P \propto V^3 \rightarrow V_{dual} = 0.5V_{uni} \rightarrow P_{dual} = 2 \times 0.125 P_{uni}$$

	Uniprocessor	Dual-processor
Frequency	1	0.5
Voltage	1	0.5
Execution Time	1	1
Dynamic Power	1	2x0.125
Dynamic Energy	1	2x0.125
Energy Efficiency	1	4

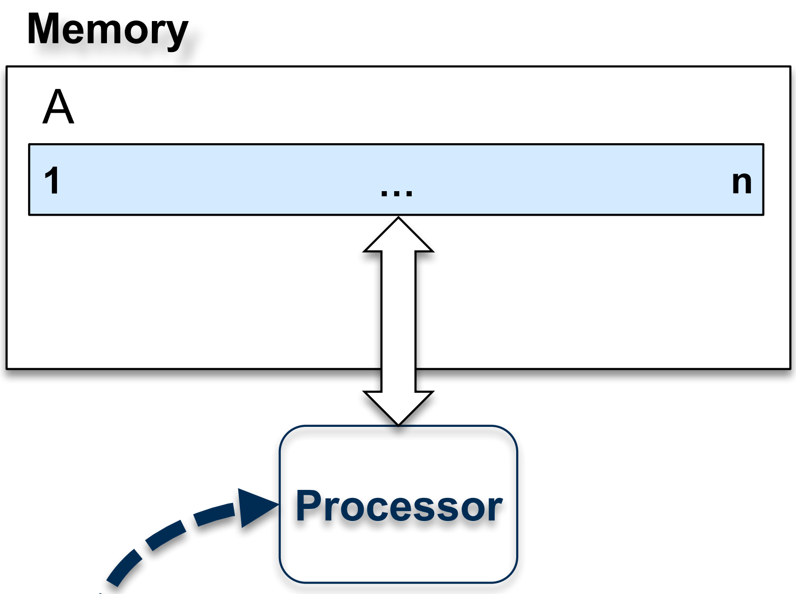
Challenges

Example Code I

- A sequential application runs as a single thread

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      A[i] = A[i] * A[i] + 5;
   }
}
```



Single Thread

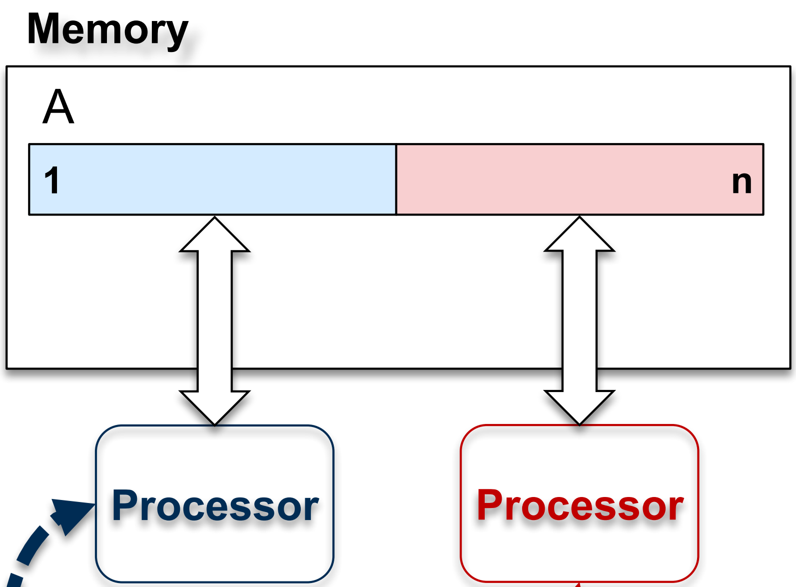
```
main() {
   ...
   kern (1, n);
   ...
}
```

Example Code I

- Two threads operating on separate partitions

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      A[i] = A[i] * A[i] + 5;
   }
}
```



Thread 0

```
main() {
   ...
   kern (1, n/2);
   ...
}
```

Thread 1

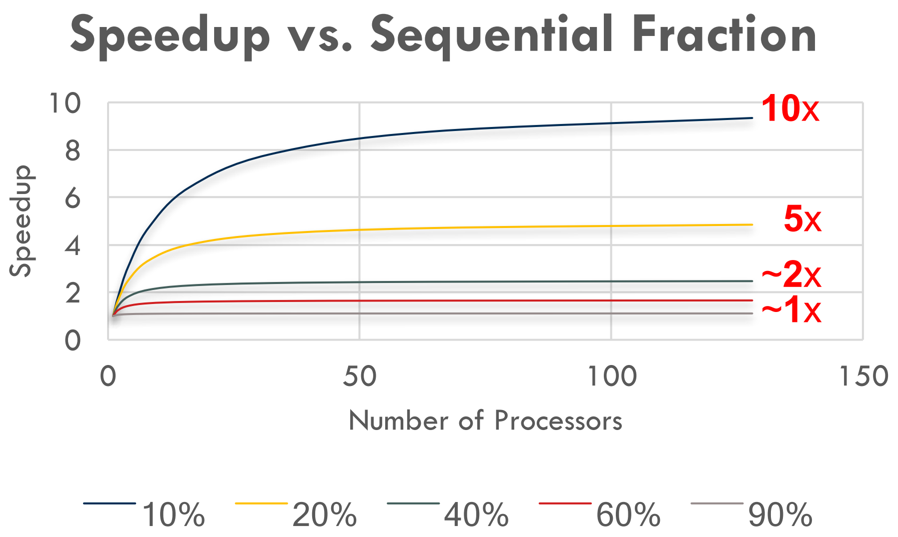
```
kern (n/2+1, n);
```

Performance of Parallel Processing

- Recall: Amdahl's law for theoretical speedup
  - Overall speedup is limited to the fraction of the program that can be executed in parallel

$$speedup = \frac{1}{f + \frac{1-f}{n}}$$

f: sequential fraction



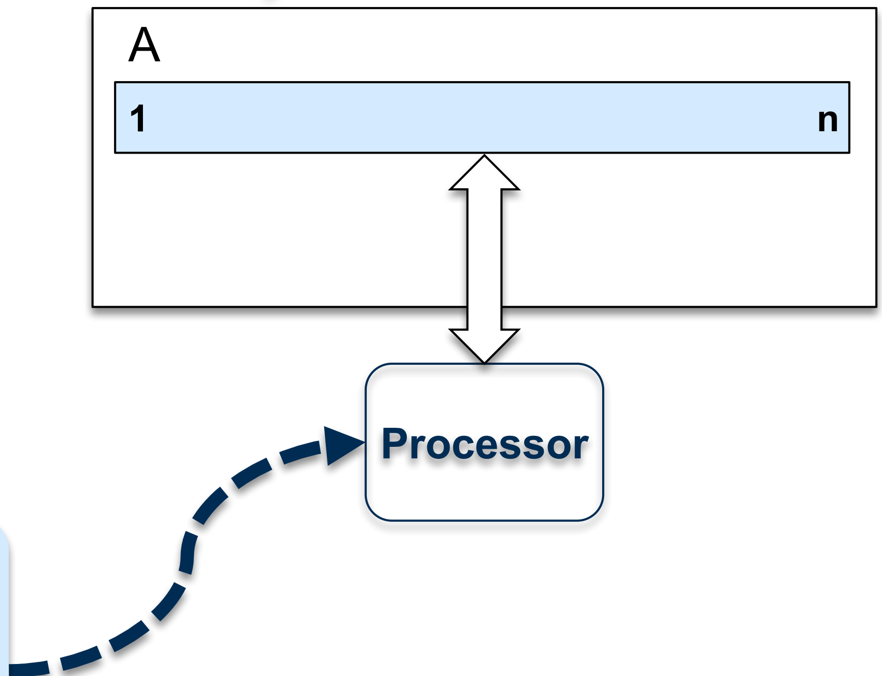
Example Code II

- A single location is updated every time

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      sum = sum + A[i];
   }
}
```

Memory



Thread 0

```
main() {
   ...
   kern (1, n);
   ...
}
```

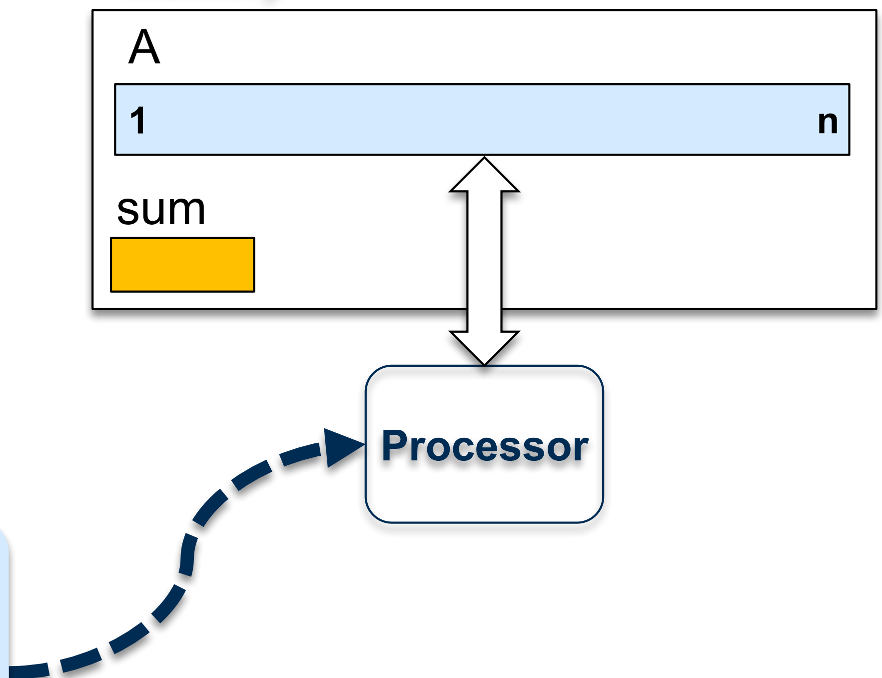
Example Code II

- A single location is updated every time

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      sum = sum + A[i];
   }
}
```

Memory



Thread 0

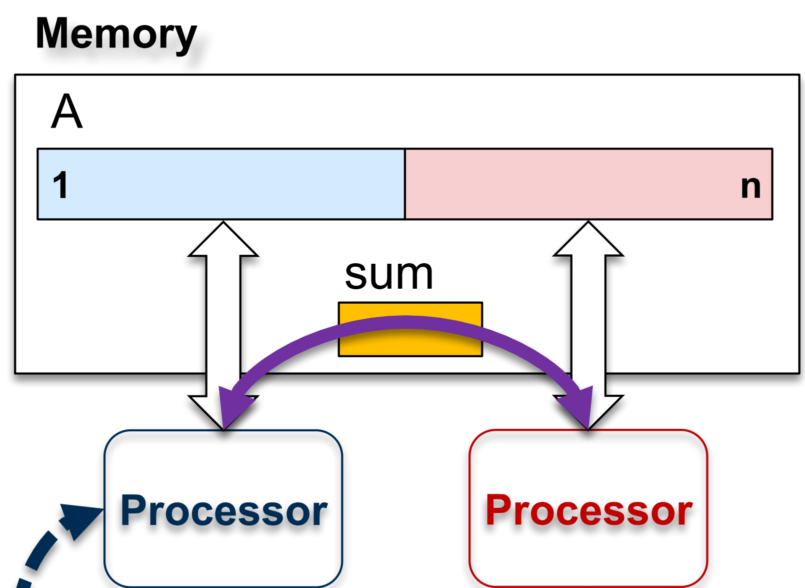
```
main() {
   ...
   kern (1, n);
   ...
}
```

Example Code II

- Two threads operating on separate partitions

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      sum = sum + A[i];
   }
}
```



Thread 0

```
main() {
   ...
   kern (1, n/2);
   ...
}
```

Thread 1

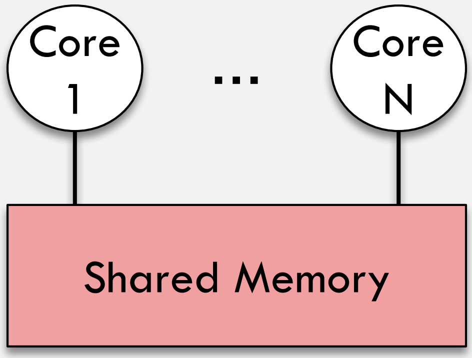
```
kern (n/2+1, n);
```

Communication in Multiprocessors

- How multiple processor cores communicate?

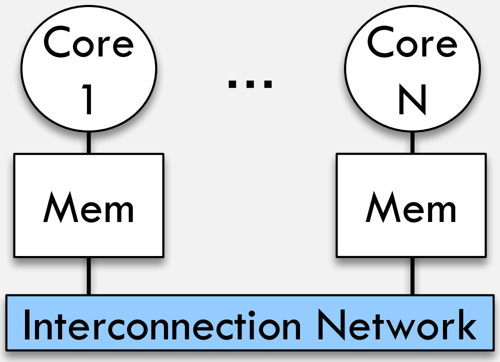
Shared Memory

- Multiple threads employ shared memory
- Easy for programmers (loads and stores)



Message Passing

- Explicit communication through interconnection network
- Simple hardware

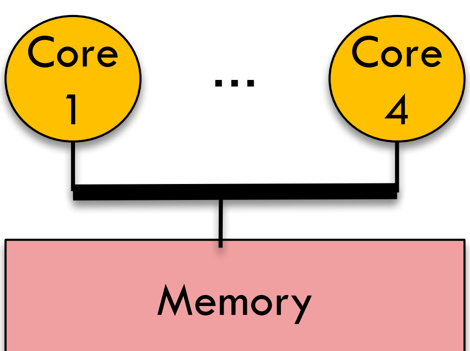


Shared Memory Architectures

Uniform Memory Access

- Equal latency for all processors
- Simple software control

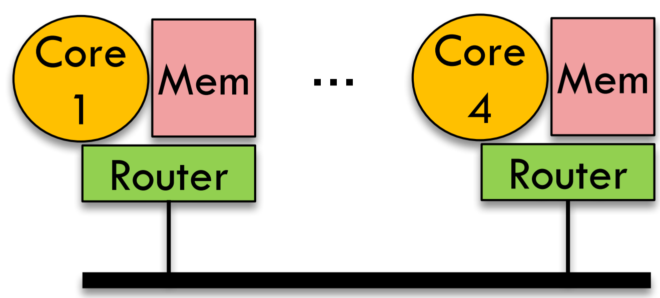
Example UMA



Non-Uniform Memory Access

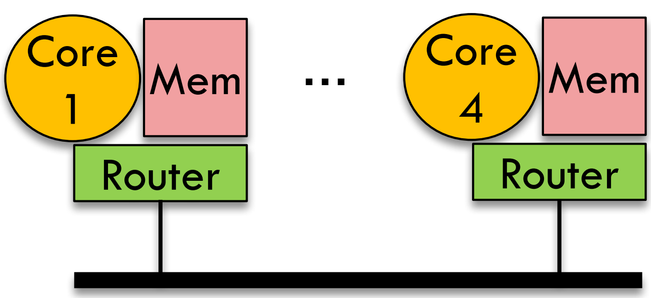
- Access latency is proportional to proximity
  - Fast local accesses

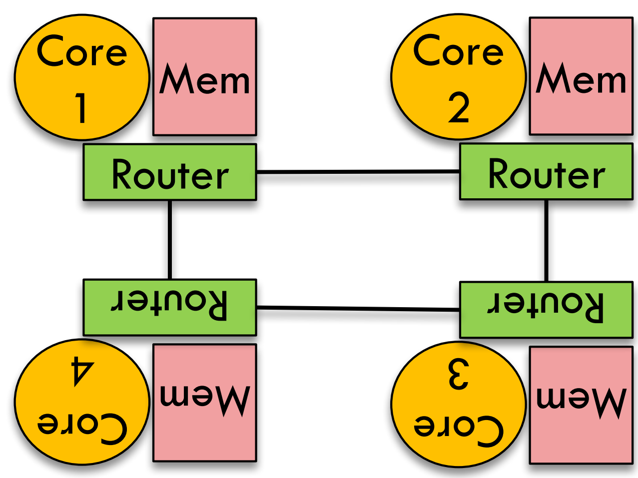
Example NUMA



Network Topologies

Shared Network	Point to Point Network
Low latency	High latency
Low bandwidth	High bandwidth
Simple control	Complex control
e.g., bus	e.g., mesh, ring



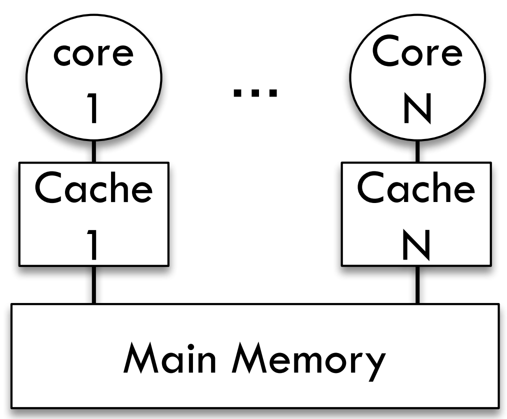


Challenges in Shared Memories

- Correctness of an application is influenced by
  - Memory consistency
    - All memory instructions appear to execute in the program order
    - Known to the programmer
  - Cache coherence
    - All the processors see the same data for a particular memory address as they should have if there were no caches in the system
    - Invisible to the programmer

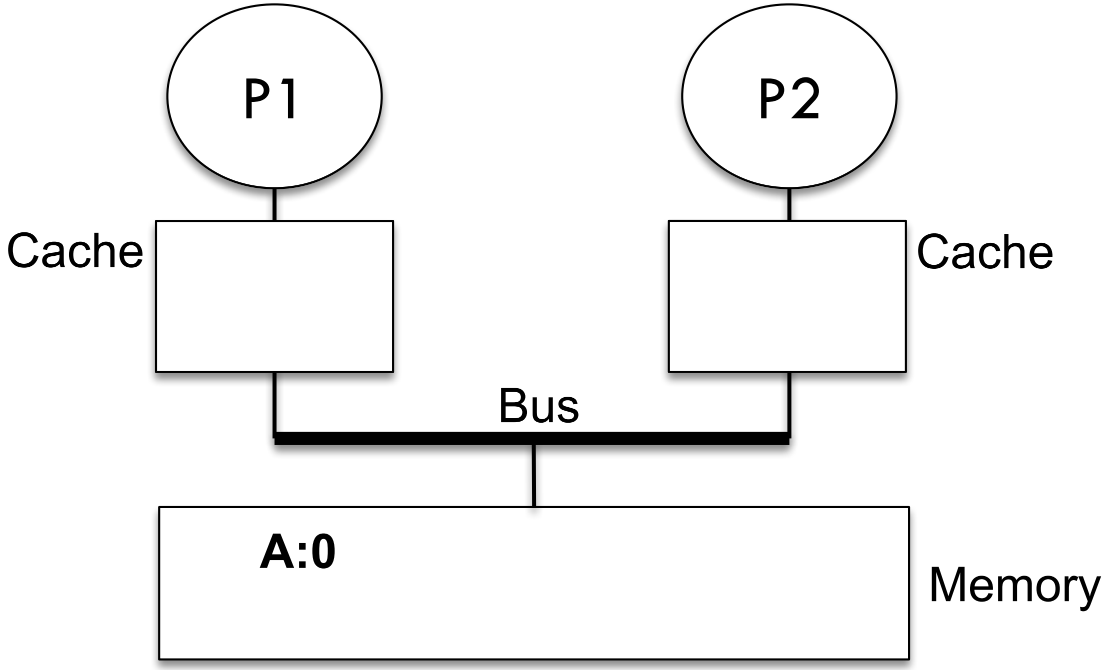
Cache Coherence Problem

- Multiple copies of each cache block
  - In main memory and caches
- Multiple copies can get inconsistent when writes happen
  - Solution: propagate writes from one core to others



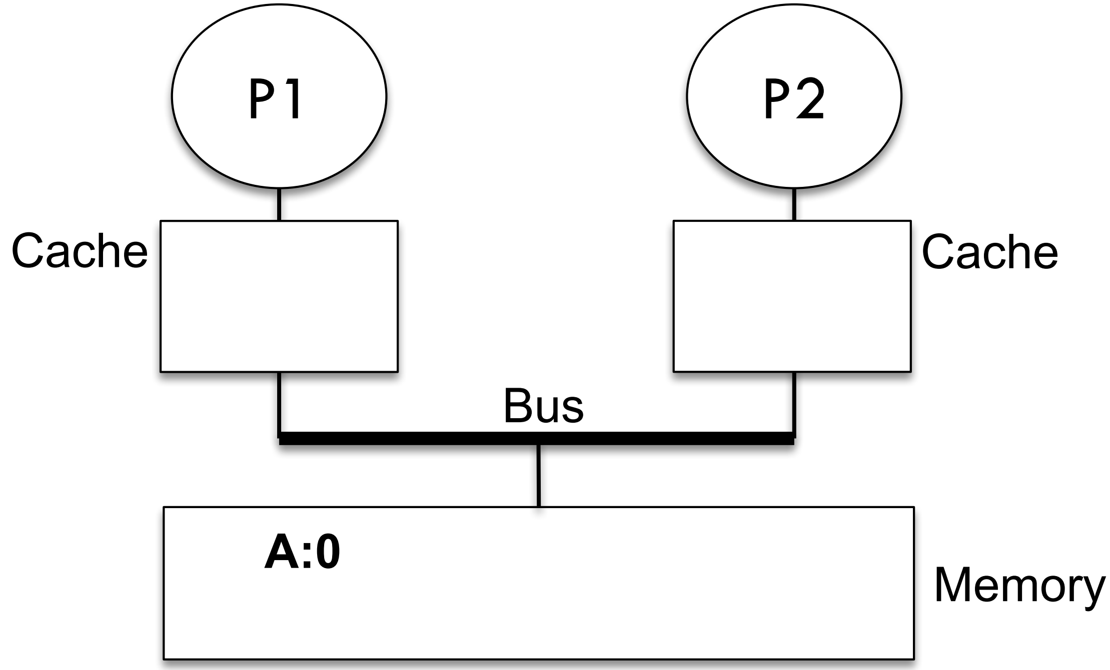
Scenario 1: Loading From Memory

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



Scenario 2: Loading From Cache

- 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



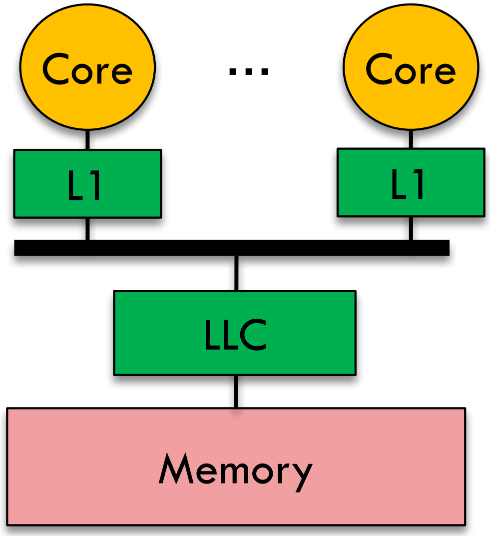
Cache Coherence

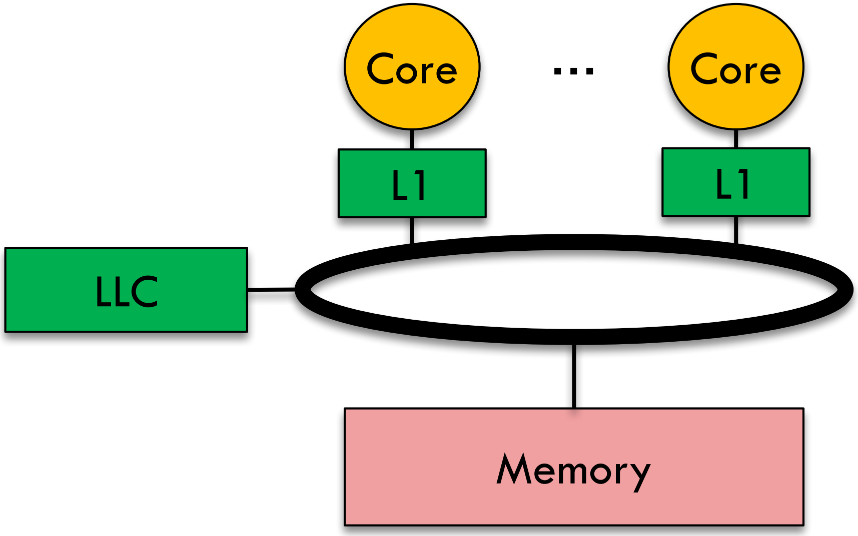
- The key operation is update/invalidate sent to all or a subset of the cores
  - Software based management
    - Flush: write all of the dirty blocks to memory
    - Invalidate: make all of the cache blocks invalid
  - Hardware based management
    - Update or invalidate other copies on every write
    - Send data to everyone, or only the ones who have a copy

- Invalidation based protocol is better. Why?

Snoopy Protocol

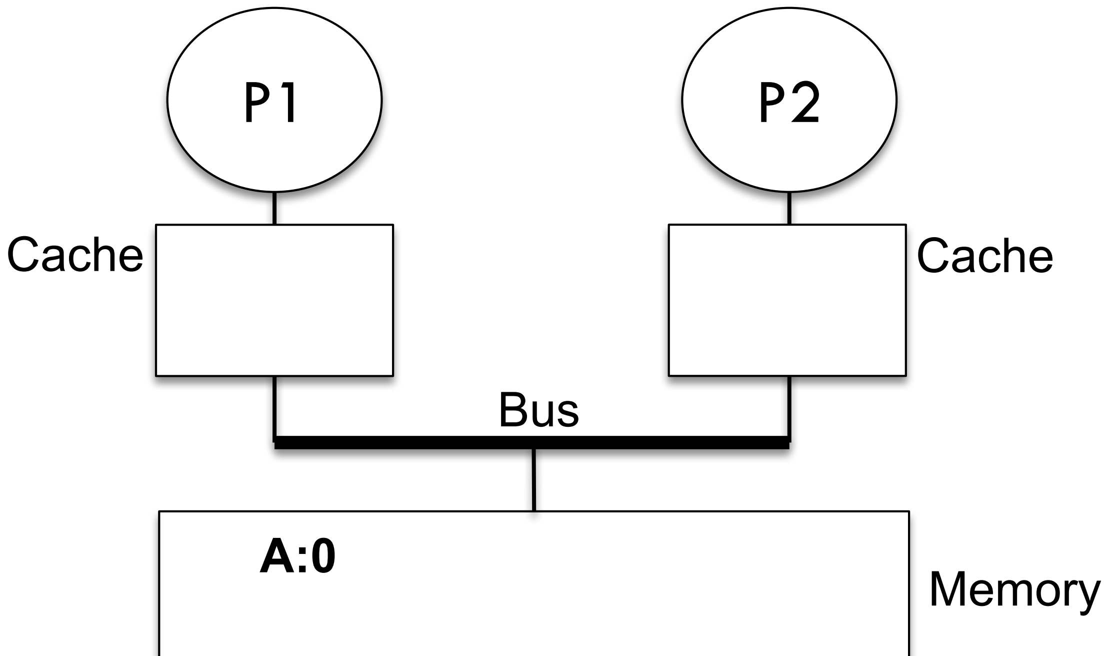
- Relying on a broadcast infrastructure among caches
  - For example shared bus
- Every cache monitors (snoop) the traffic on the shared media to keep the states of the cache block up to date





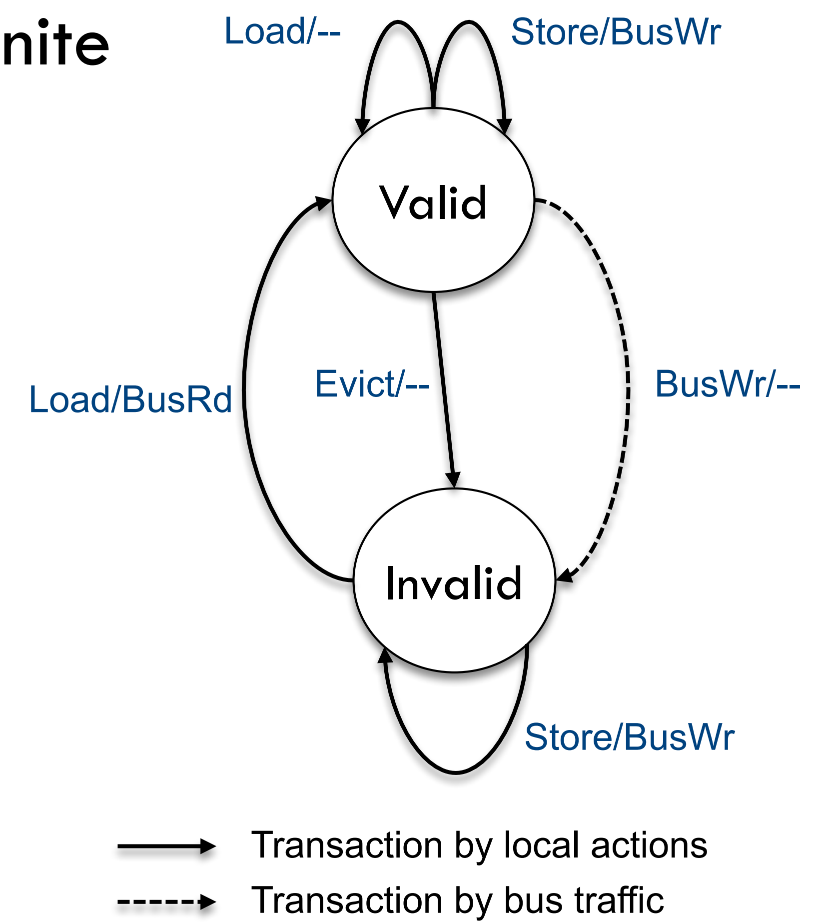
Simple Snooping Protocol

- Relies on write-through, write no-allocate cache
- Multiple readers are allowed
  - Writes invalidate replicas
- Employs a simple state machine for each cache unit



Simple Snooping State Machine

- Every node updates its one-bit valid flag using a simple finite state machine (FSM)
- Processor actions
  - Load, Store, Evict
- Bus traffic
  - BusRd, BusWr



Snooping with Writeback Policy

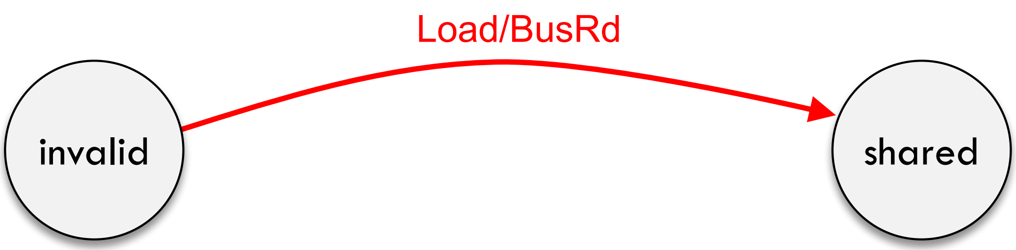
- Problem: writes are not propagated to memory until eviction
  - Cache data maybe different from main memory

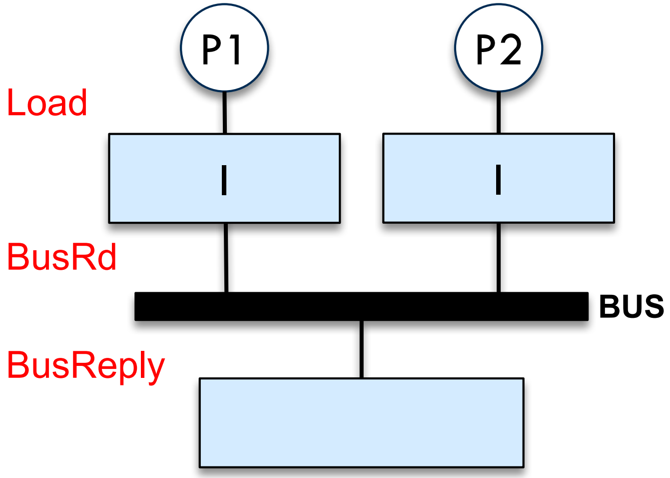
- Solution: identify the owner of the most recently updated replica
  - Every data may have only one owner at any time
  - Only the owner can update the replica
  - Multiple readers can share the data
    - No one can write without gaining ownership first

Modified-Shared-Invalid Protocol

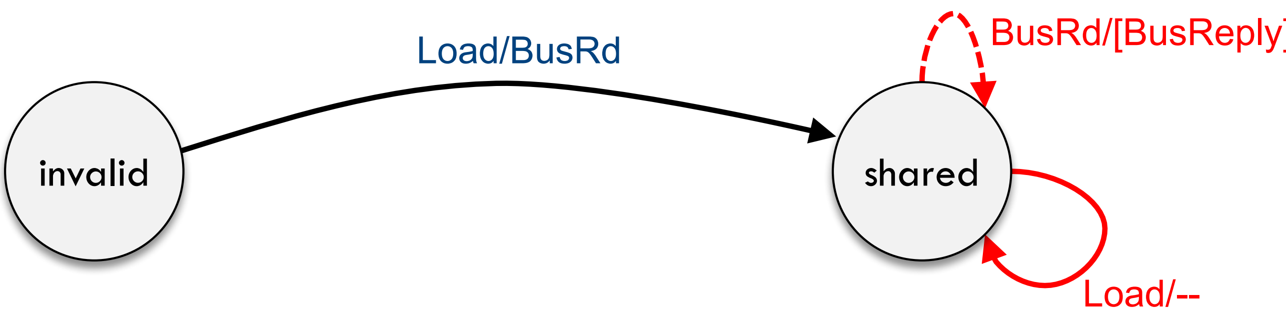
- Every cache block transitions among three states
  - Invalid: no replica in the cache
  - Shared: a read-only copy in the cache
    - Multiple units may have the same copy
  - Modified: a writable copy of the data in the cache
    - The replica has been updated
    - The cache has the only valid copy of the data block
- Processor actions
  - Load, store, evict
- Bus messages
  - BusRd, BusRdX, BusInv, BusWB, BusReply

MSI Example

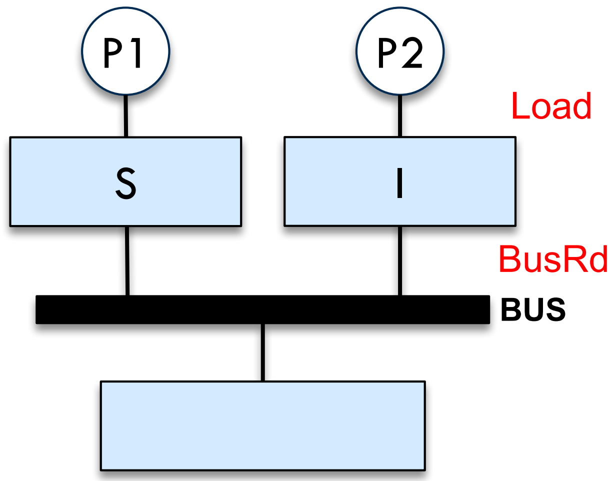




MSI Example

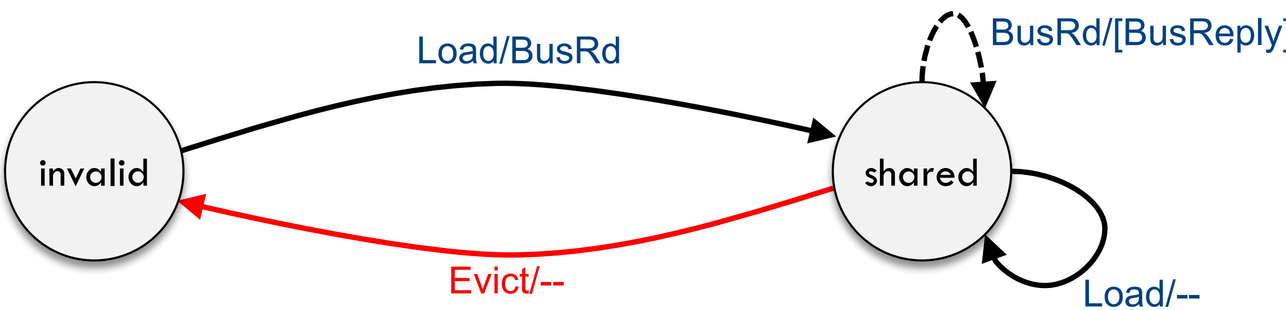


Load/BusRd
BusRd/[BusReply]
invalid
shared
Load/--

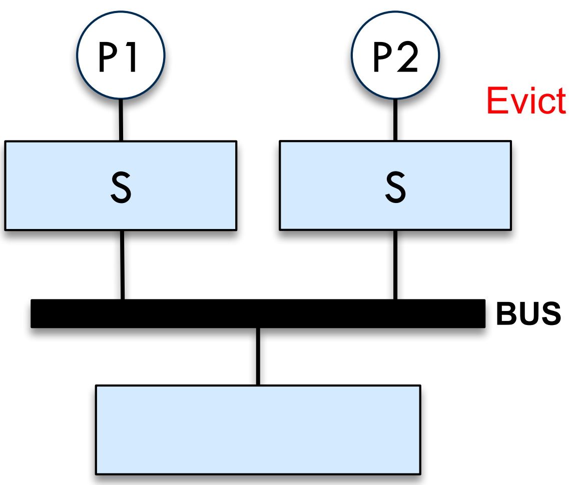


P1
P2
Load
S
I
BusRd
BUS

MSI Example

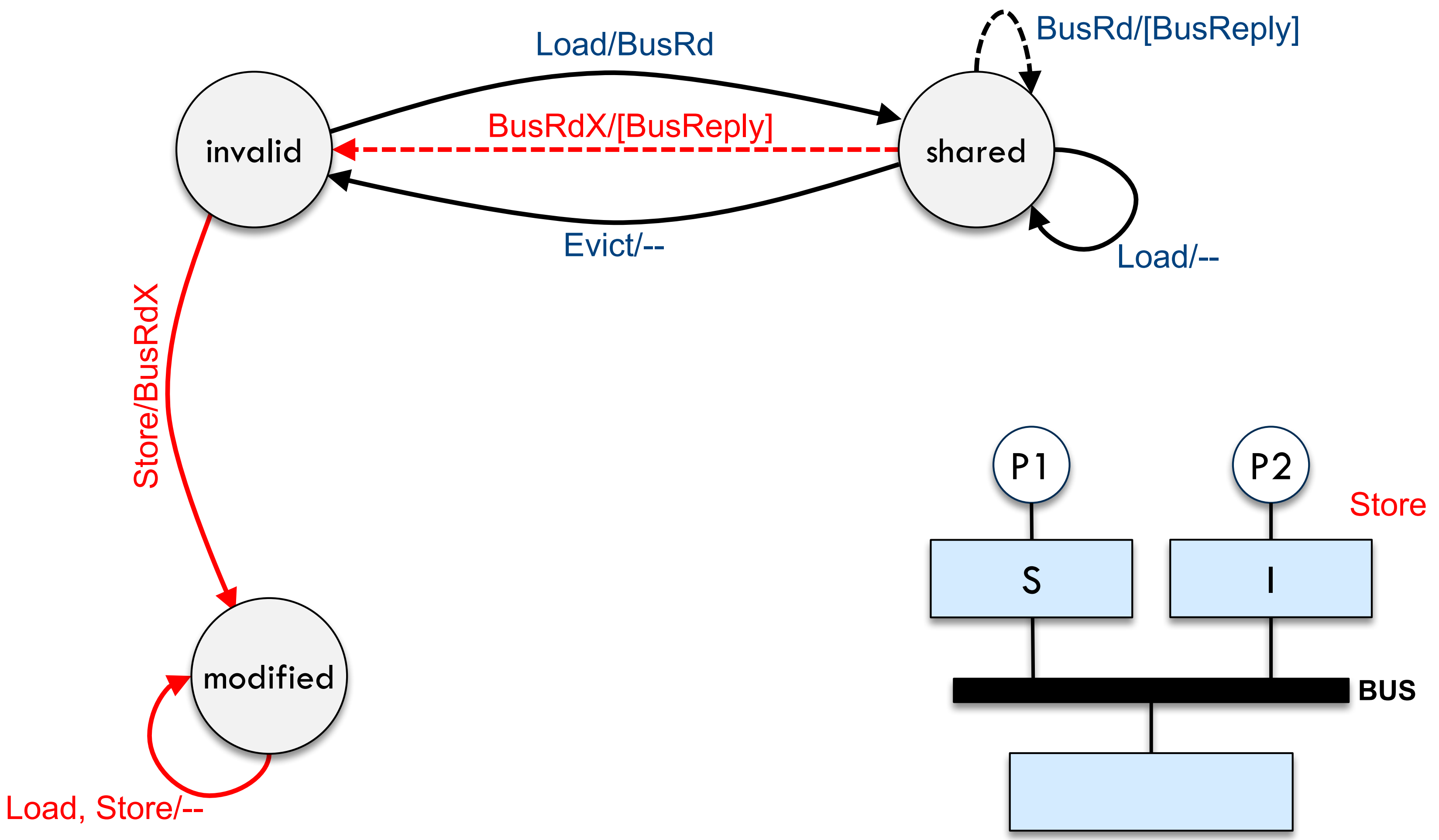


Load/BusRd
BusRd/[BusReply]
invalid
shared
Evict/--
Load/--

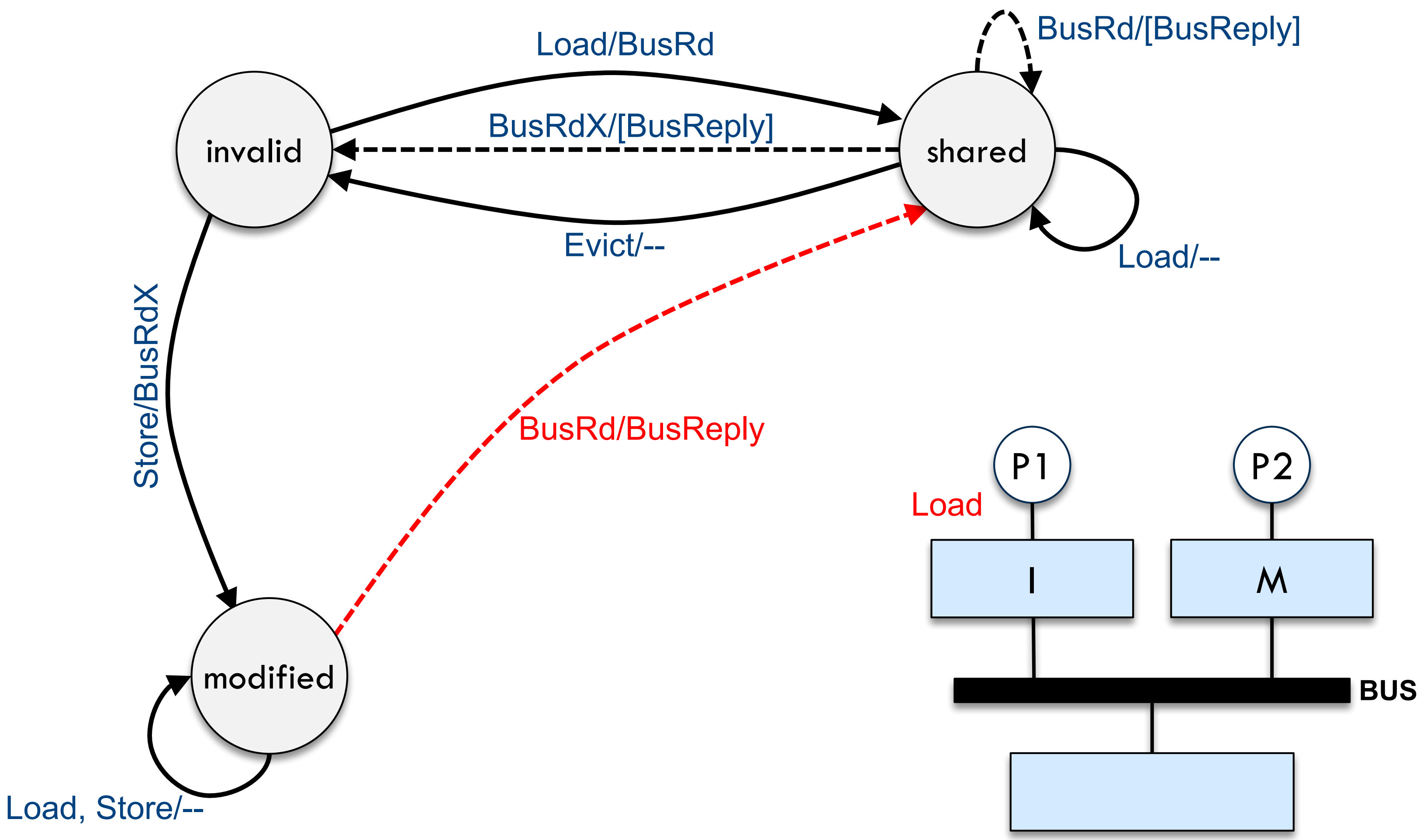


P1
P2
Evict
S
S
BUS

MSI Example

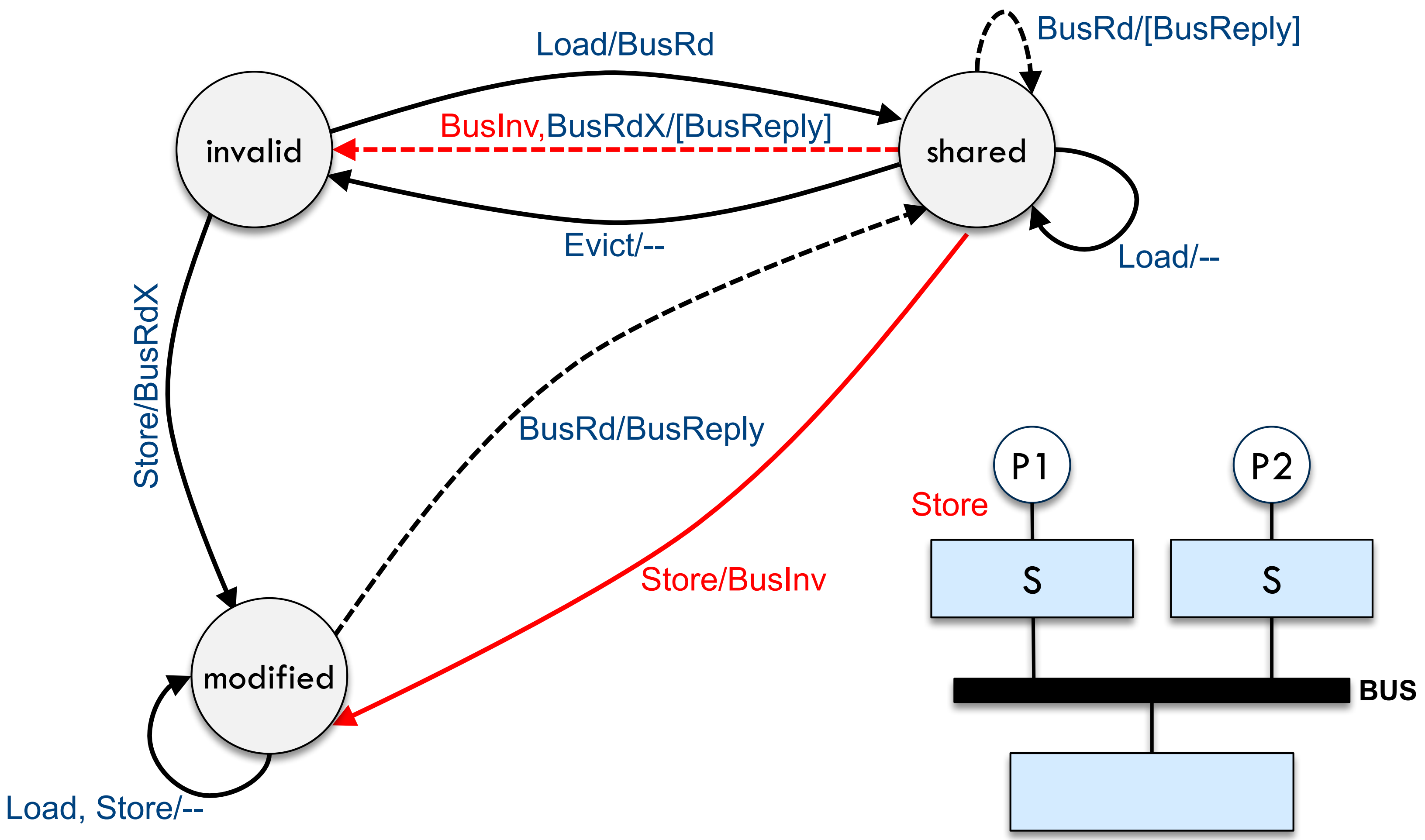

Load/BusRd
BusRd/[BusReply]
BusRdX/[BusReply]
invalid
shared
Evict/--
Load/--
Store/BusRdX
modified
Load, Store/--
P1
P2
Store
S
I
BUS

MSI Example



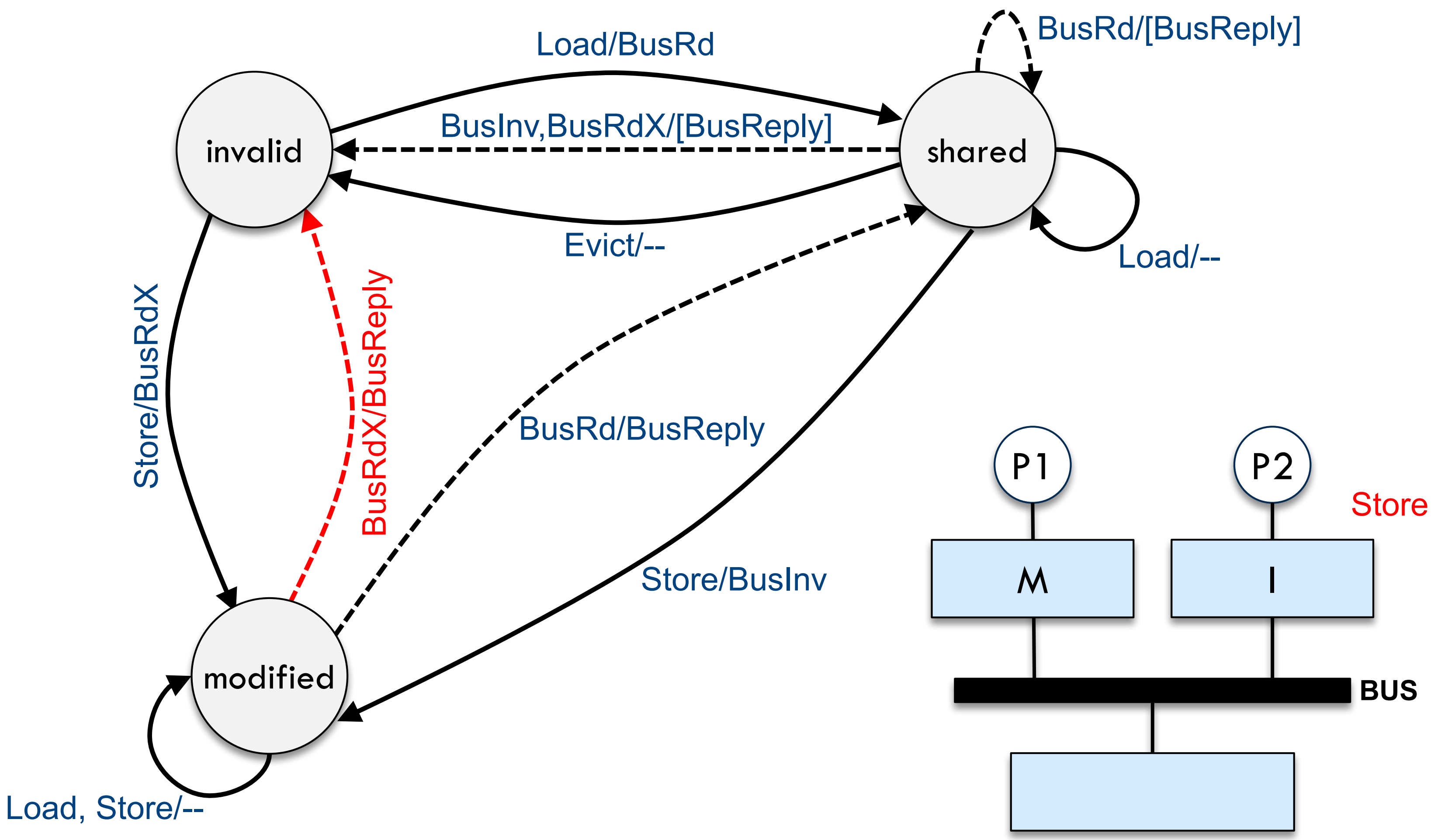
Load/BusRd
BusRd/[BusReply]
BusRdX/[BusReply]
invalid
shared
Evict/--
Load/--
Store/BusRdX
BusRd/BusReply
modified
Load, Store/--
P1
P2
Load
I
M
BUS

MSI Example



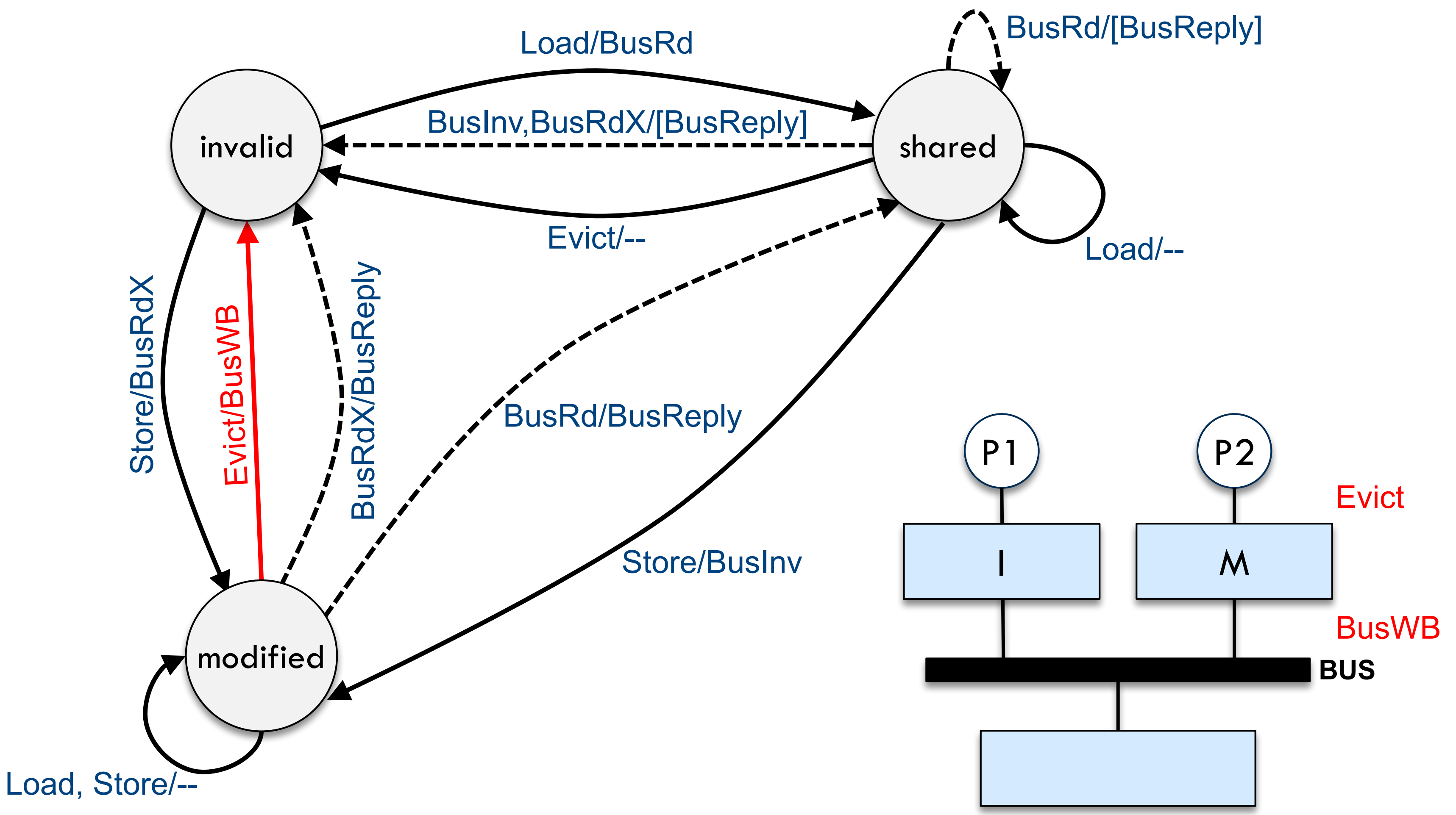
Load/BusRd
BusRd/[BusReply]
BusInv,BusRdX/[BusReply]
invalid
shared
Evict/--
Load/--
Store/BusRdX
BusRd/BusReply
Store/BusInv
modified
Load, Store/--
P1
P2
Store
S
S
BUS

MSI Example



Load/BusRd
BusRd/[BusReply]
BusInv,BusRdX/[BusReply]
invalid
shared
Evict/--
Load/--
Store/BusRdX
BusRdX/BusReply
BusRd/BusReply
Store/BusInv
modified
Load, Store/--
P1
P2
Store
M
I
BUS

MSI Example



invalid
shared
modified
Load/BusRd
BusInv,BusRdX/[BusReply]
Evict/--
BusRd/[BusReply]
Load/--
Store/BusRdX
Evict/BusWB
BusRdX/BusReply
BusRd/BusReply
Store/BusInv
Load, Store/--
P1
P2
I
M
Evict
BusWB
BUS