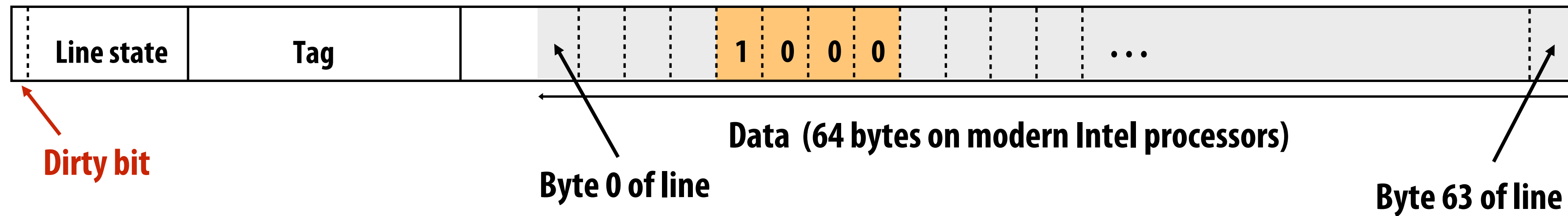**Lecture 8:**

Snooping-Based Cache Coherence

**Parallel Computing
Stanford CS149, Winter 2019**

Cache design review

Let's say your code executes `int x = 1;`

(Assume for simplicity x corresponds to the address 0x12345604 in memory... it's not stored in a register)

One cache line:

Line state	Tag			1	0	0	0	...	

Dirty bit

Data (64 bytes on modern Intel processors)

Byte 0 of line

Byte 63 of line

- **Do you know the difference between a write back and a write-through cache?**

- **What about an allocate vs. write-no-allocate cache?**

Behavior of write-allocate, write-back cache on a write miss (uniprocessor case)

Example: processor executes `int x = 1;`

1. Processor performs write to address that "misses" in cache

2. Cache selects location to place line in cache, if there is a dirty line currently in this location, the dirty line is written out to memory

3. Cache loads line from memory ("allocates line in cache")

4. Whole cache line is fetched and 32 bits are updated

5. Cache line is marked as dirty

Line state	Tag	Data (64 bytes on modern Intel processors)

Dirty bit

Review: Shared address space model (abstraction)

- **Threads Reading/writing to shared variables**
  - **Inter-thread communication is implicit in memory operations**
  - **Thread 1 stores to X**
  - **Later, thread 2 reads X (and observes update of value by thread 1)**
  - **Manipulating synchronization primitives**

    - **e.g., ensuring mutual exclusion via use of locks**

- **This is a natural extension of sequential programming**

A shared memory multi-processor

- **Processors read and write to shared variables**

  - **More precisely: processors issue load and store instructions**

- **A reasonable expectation of memory is:**

  - **Reading a value at address X should return the <u>last value</u> written to address X *by any processor***
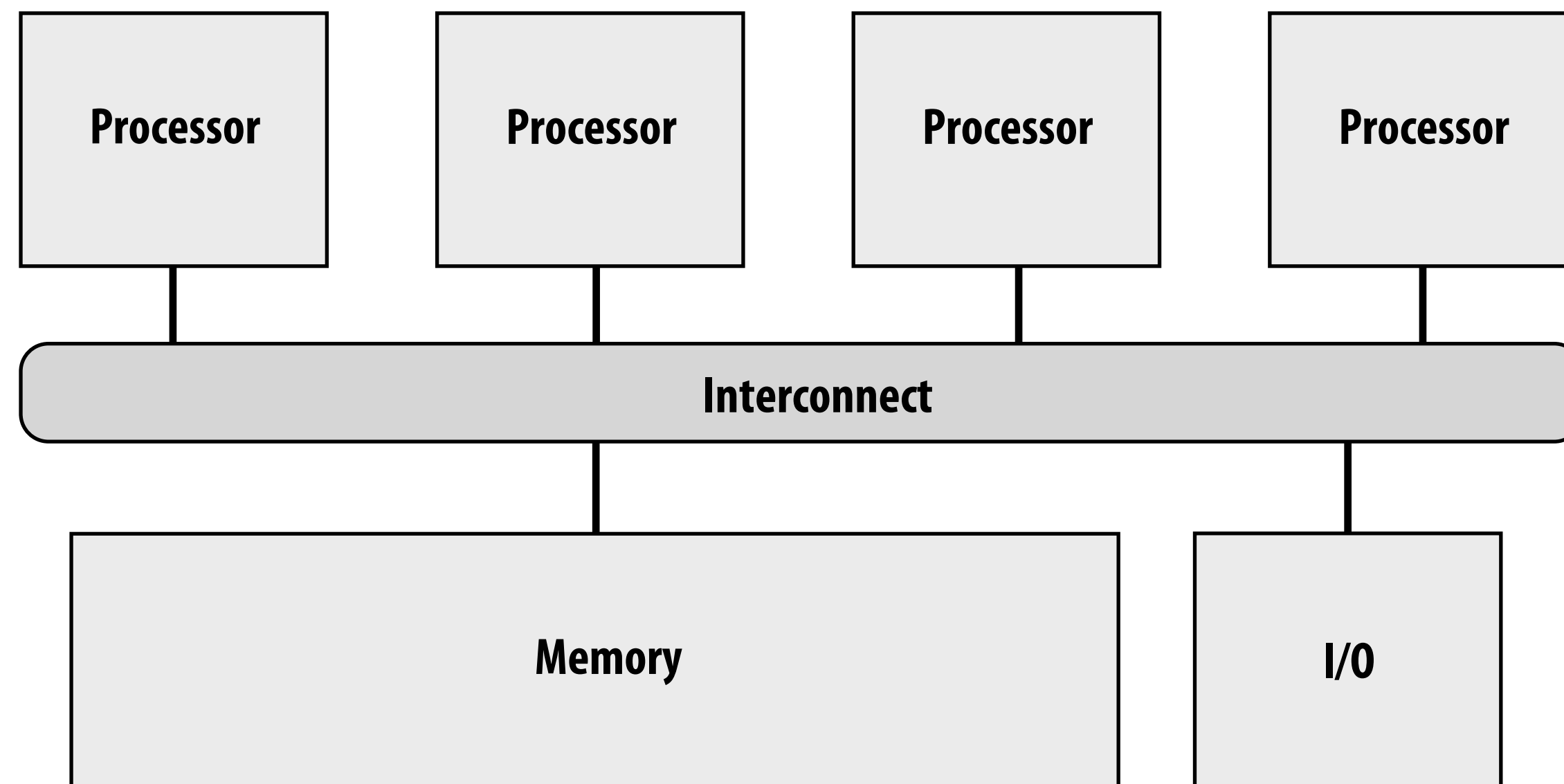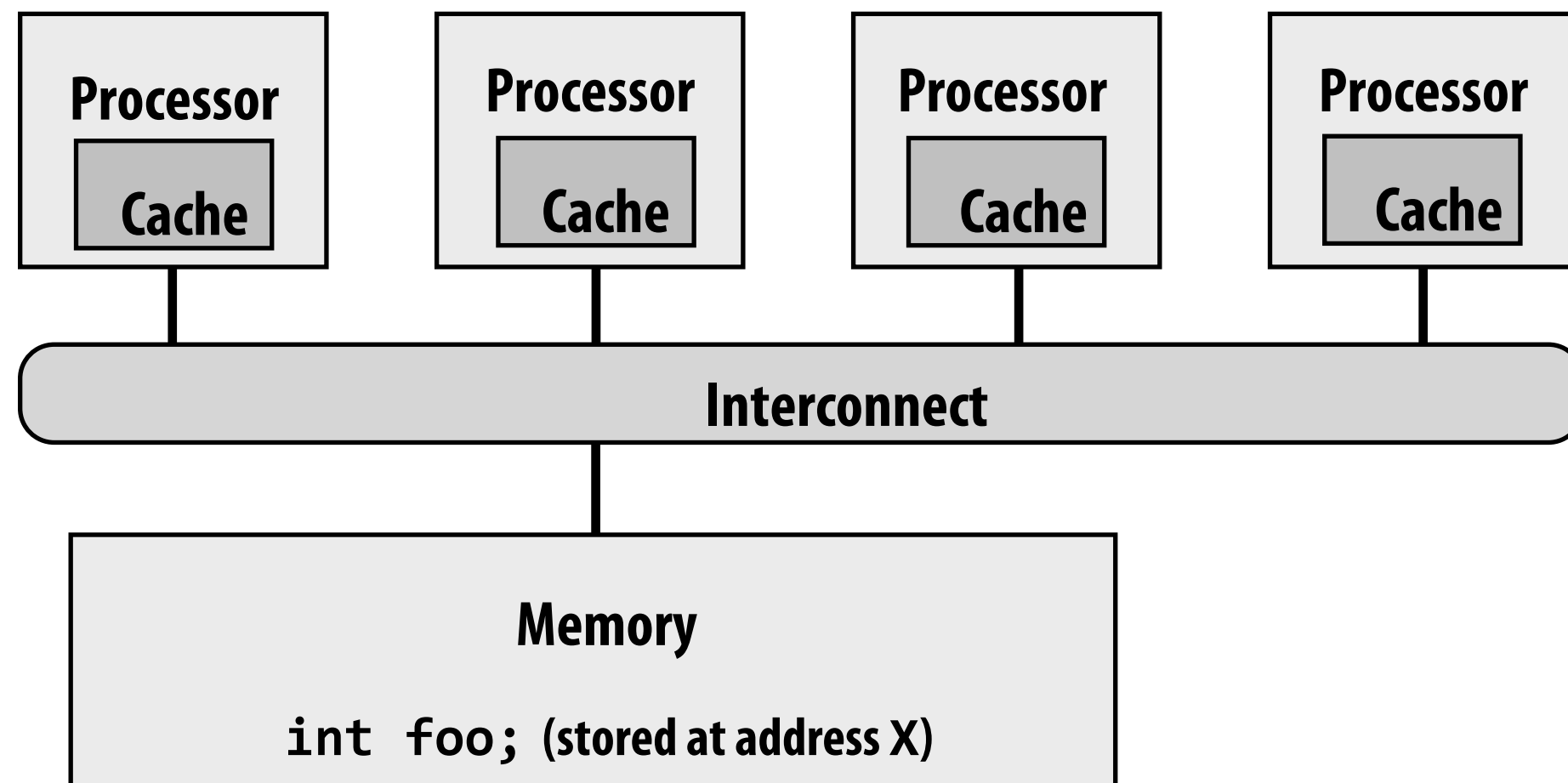

(A simple view of four processors and their shared address space)

The cache coherence problem

Modern processors replicate contents of memory in local caches

Problem: processors can observe different values for the same memory location

```
┌─────────────┐  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐
│  Processor  │  │  Processor  │  │  Processor  │  │  Processor  │
│ ┌─────────┐ │  │ ┌─────────┐ │  │ ┌─────────┐ │  │ ┌─────────┐ │
│ │  Cache  │ │  │ │  Cache  │ │  │ │  Cache  │ │  │ │  Cache  │ │
│ └─────────┘ │  │ └─────────┘ │  │ └─────────┘ │  │ └─────────┘ │
└─────────────┘  └─────────────┘  └─────────────┘  └─────────────┘
```

Interconnect

Memory

`int foo;` **(stored at address X)**

The chart at right shows the value of variable `foo` (stored at address X) in main memory and in each processor's cache
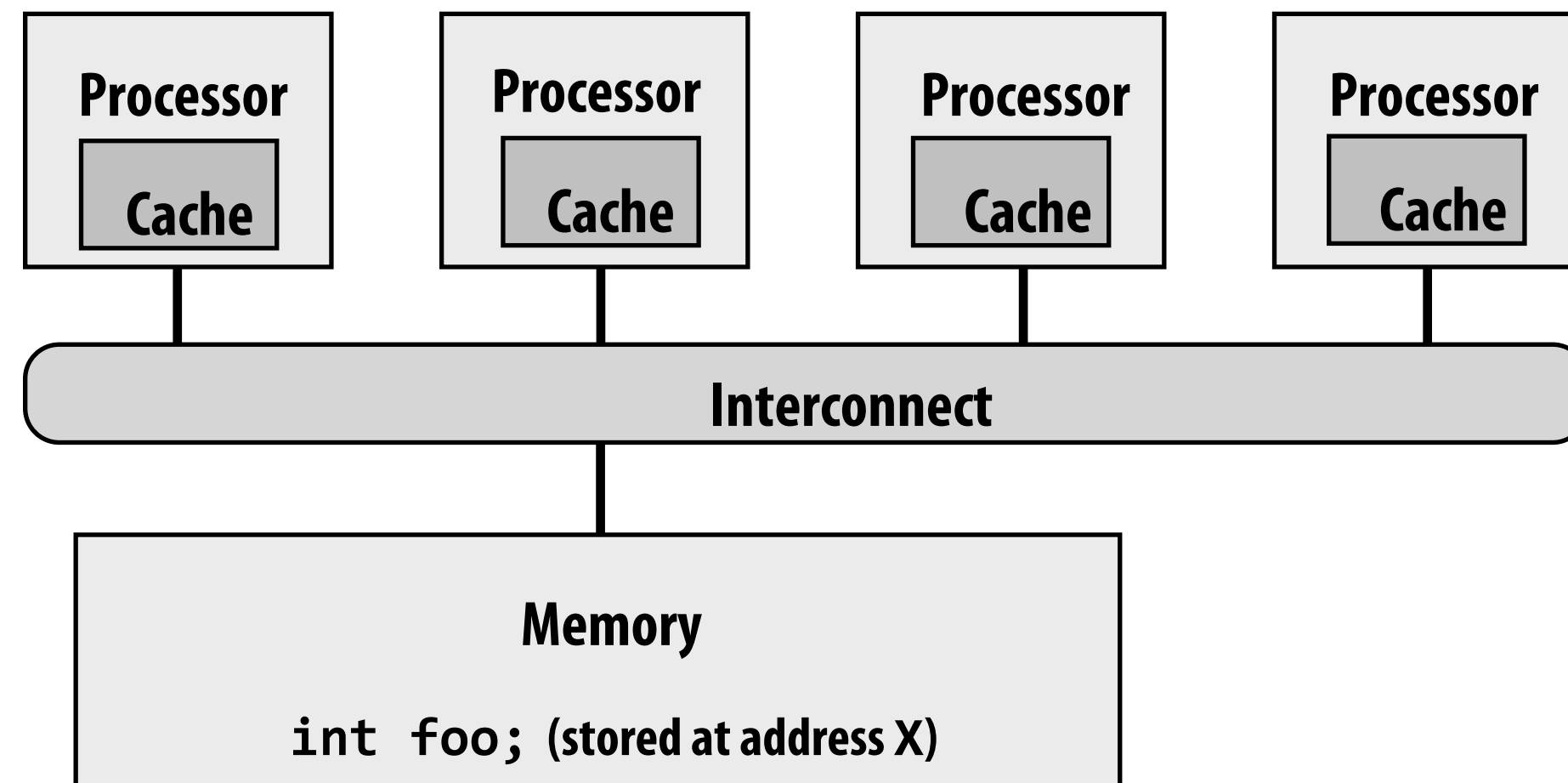
Assume the initial value stored at address X is 0

Assume write-back cache behavior

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y		0	2		1

(assume this load causes eviction of X)

The cache coherence problem

Modern processors replicate contents of memory in local caches

Problem: processors can observe different values for the same memory location

Is this a mutual exclusion problem?

Can you fix the problem by adding locks to your program?

NO!
This is a problem created by replicating the data stored at address X in local caches

Memory

`int foo; (stored at address X)`

The chart at right shows the value of variable `foo` (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0
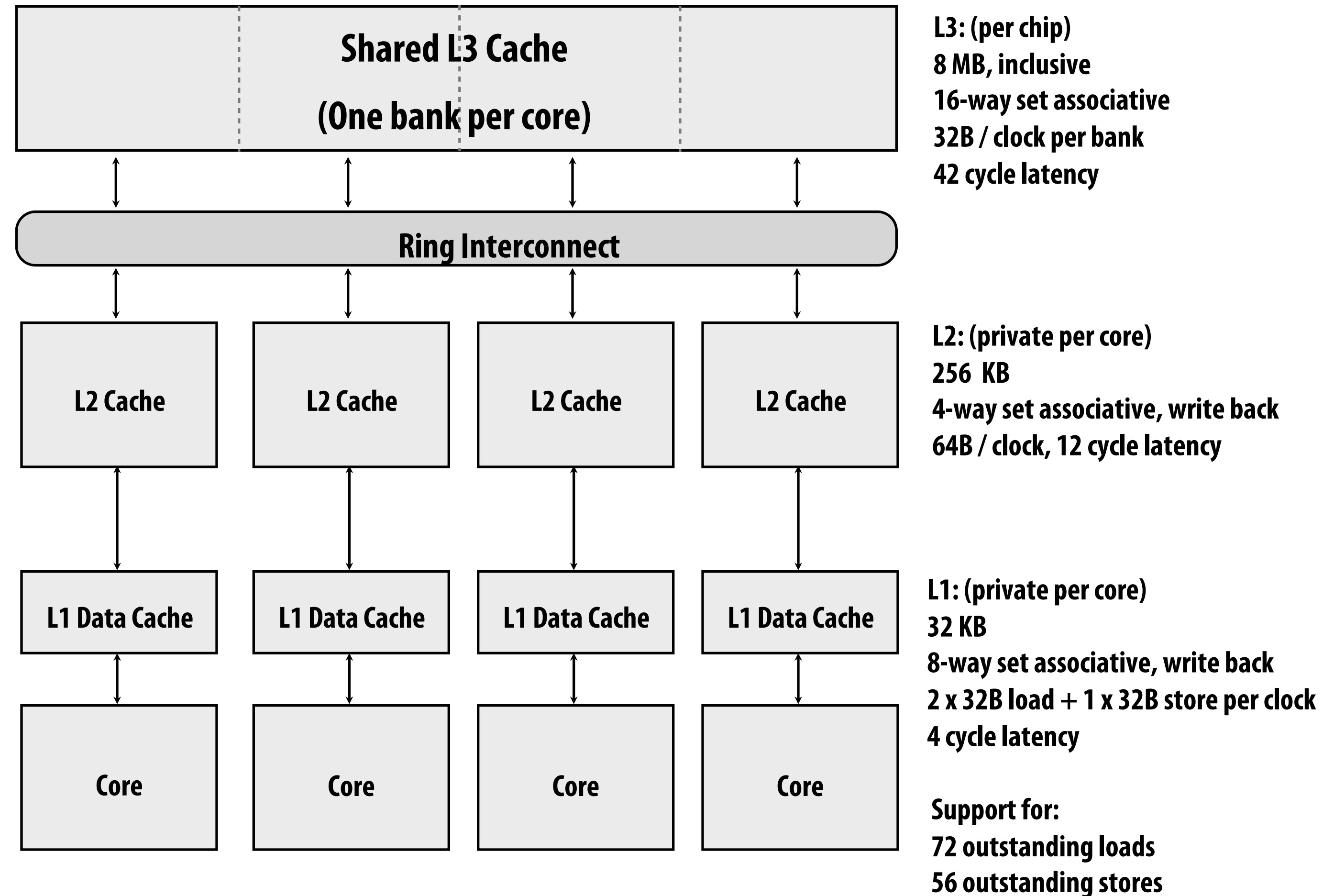
Assume write-back cache behavior

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y		0	2		1

`(assume this load causes eviction of X)`

The memory coherence problem

- **Intuitive behavior for memory system: reading value at address X should return the last value written to address X *by any processor*.**

- **Memory coherence problem exists because there is both global storage (main memory) and per-processor local storage (processor caches) implementing the <u>abstraction</u> of a single shared address space.**

Cache hierarchy of Intel Skylake CPU (2015)

64 byte cache line size

Shared L3 Cache

(One bank per core)

Ring Interconnect

L2 Cache   L2 Cache   L2 Cache   L2 Cache

L1 Data Cache   L1 Data Cache   L1 Data Cache   L1 Data Cache

Core   Core   Core   Core

L3: (per chip)
8 MB, inclusive
16-way set associative
32B / clock per bank
42 cycle latency

L2: (private per core)
256 KB
4-way set associative, write back
64B / clock, 12 cycle latency

L1: (private per core)
32 KB
8-way set associative, write back
2 x 32B load + 1 x 32B store per clock
4 cycle latency

Support for:
72 outstanding loads
56 outstanding stores

Source: Intel 64 and IA-32 Architectures Optimization Reference Manual (June 2016)

Intuitive expectation of shared memory

- **Intuitive behavior for memory system: reading value at address X should return the last value written to address X *by any processor.***

- **On a uniprocessor, providing this behavior is fairly simple, since writes typically come from one source: the processor**
  - **Exception: device I/O via direct memory access (DMA)**

Problems with the intuition

- **Intuitive behavior: reading value at address X should return the last value written to address X *by any processor*.**

- **What does "last" mean?**
  - **What if two processors write at the same time?**

  - **What if a write by P1 is followed by a read from P2 so close in time that it is impossible to communicate the occurrence of the write to P2 in time?**

- **In a sequential program, "last" is determined by program order (not time)**
  - **Holds true within one thread of a parallel program**
  - **But we need to come up with a meaningful way to describe order across threads in a parallel program**

Definition: coherence

A memory system is coherent if:

The results of a parallel program's execution are such that for <u>**each memory location**</u>, there is a hypothetical serial order of all program operations (executed by all processors) to the location that is consistent with the results of execution, and:

1. Memory operations issued by any one processor occur in the order issued by the processor

2. The value returned by a read is the value written by the last write to the location… as given by the serial order

- P0 write: 5
- P1 read (5)
- P2 read (5)
- P0 read (5)
- P1 write: 25
- P0 read (25)
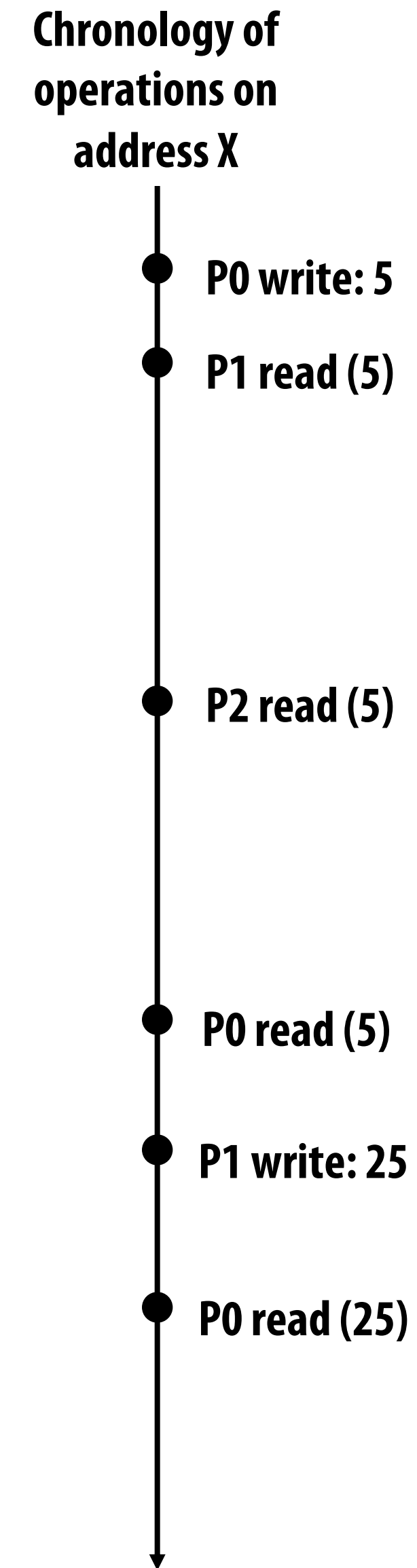
Cache Coherence Invariants (hardware)

1. **Single-Writer, Multiple-Read (SWMR) Invariant**

   For any memory location x, at any given time (epoch), there exists only a single processor(core) that may write to x (and can also read it) or some number of cores that may only read x

2. **Data-Value Invariant (write serialization)**

   The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch


read-only
Cores 2 and 5

read-write
Core 3

read-write
Core 1

read-only
Cores 1, 2, 3

time
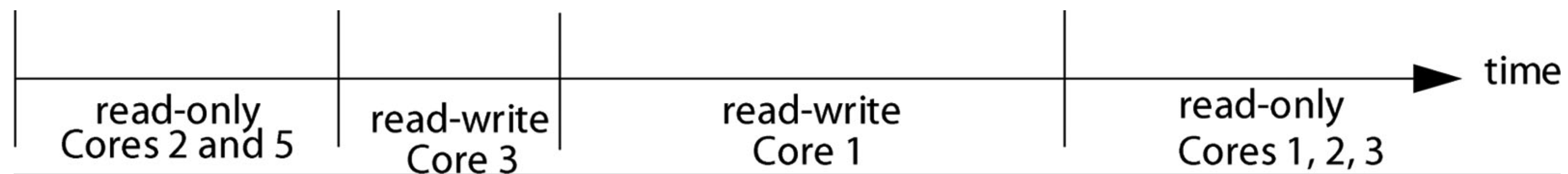
Implementing coherence

- **Software-based solutions (coarse grain: VM page)**
  - OS uses page-fault mechanism to propagate writes
  - Can be used to implement memory coherence over clusters of workstations
  - We won't discuss these solutions

- **Hardware-based solutions (fine grain: cache line)**
  - "Snooping"-based coherence implementations (today)
 - Directory-based coherence implementations (next week)

Shared caches: coherence made easy

- **One single cache shared by all processors**
  - **Eliminates problem of replicating state in multiple caches**
- **Obvious scalability problems (since the point of a cache is to be local and fast)**
  - **Interference / contention due to many clients (destructive)**
- **But shared caches can have benefits:**
  - **Facilitates fine-grained sharing (overlapping working sets)**
  - **Loads/stores by one processor might pre-fetch lines for another processor (constructive)**
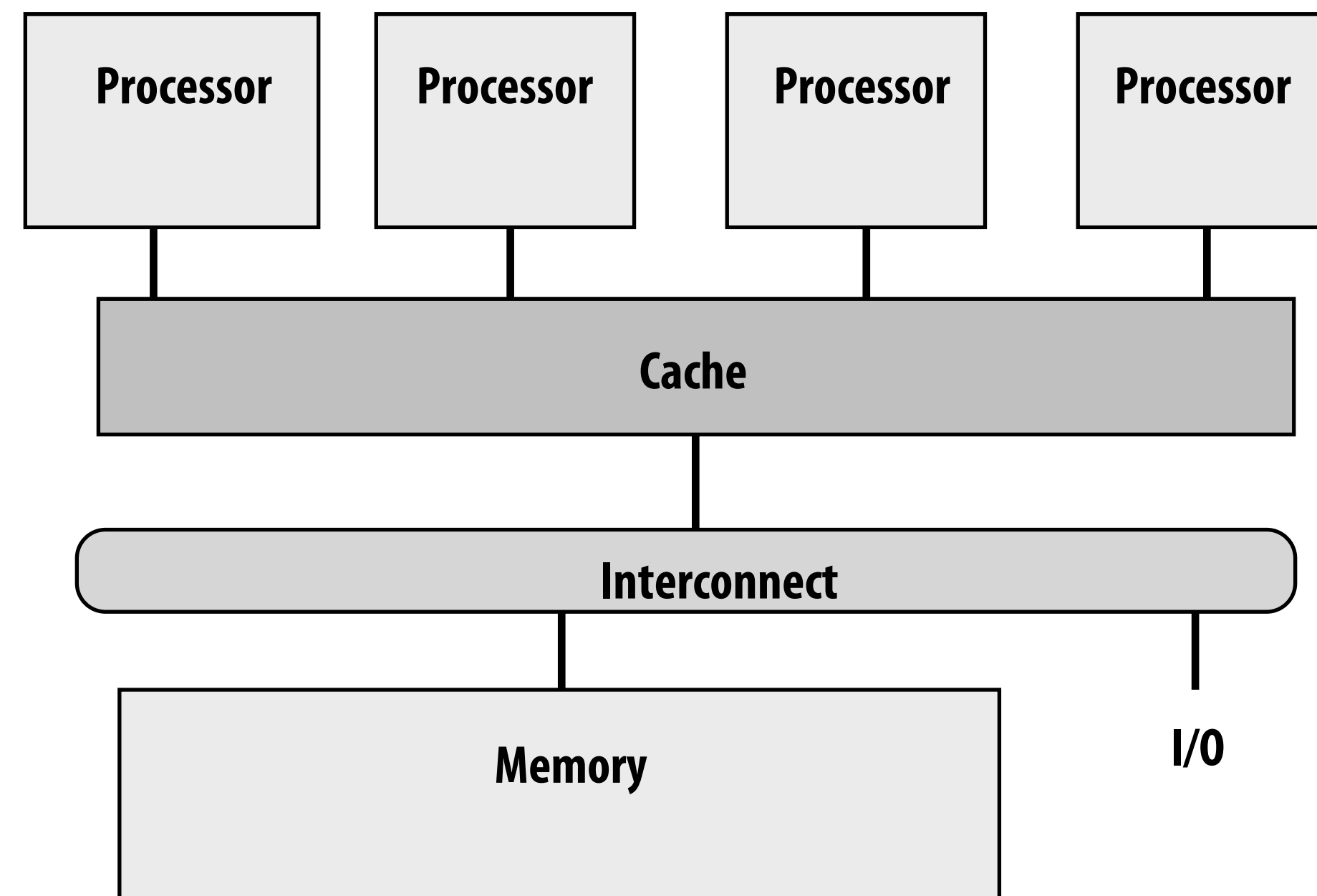
SUN Niagara 2 (UltraSPARC T2)

Note area of crossbar (CCX):
about same area as one core on chip
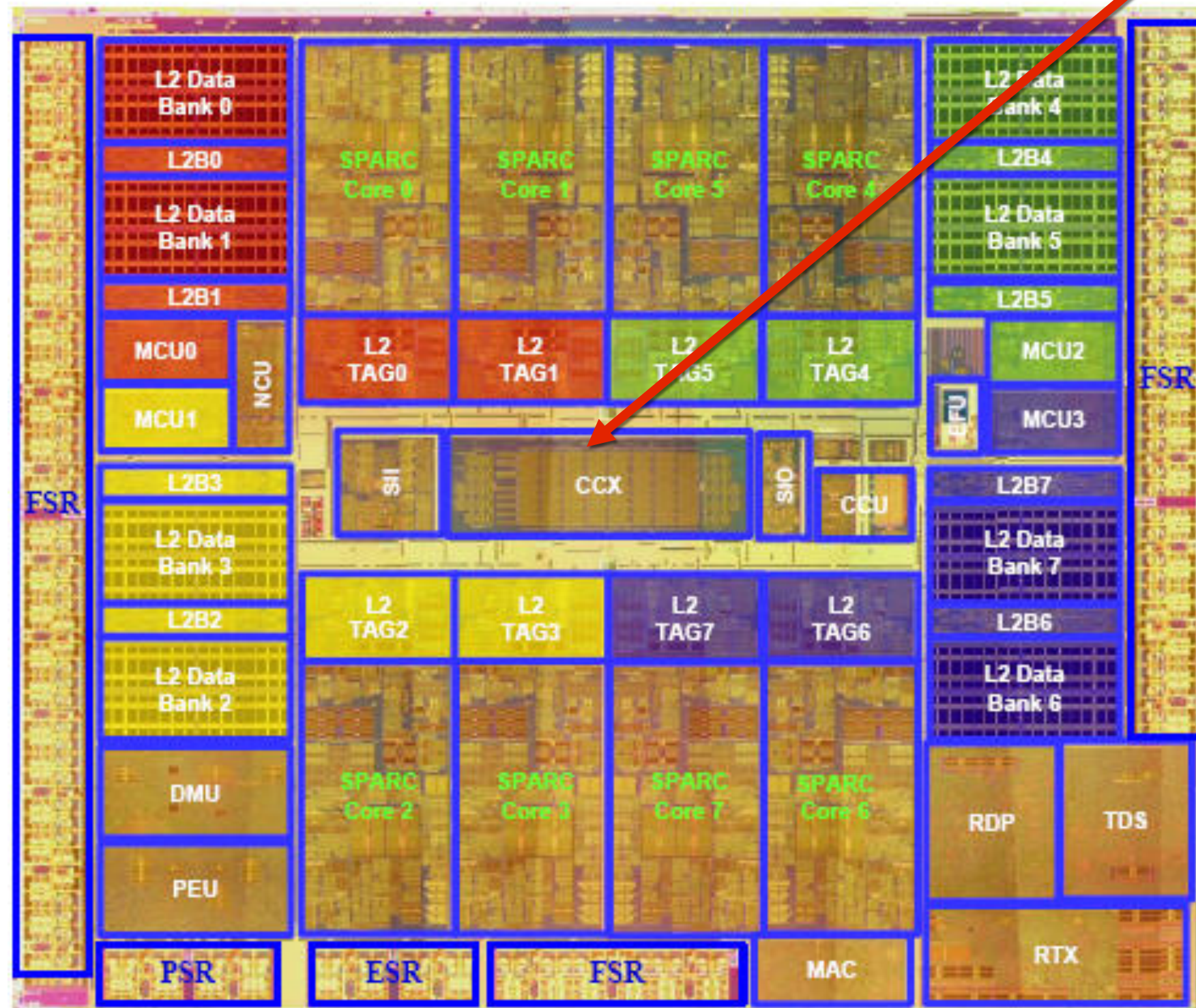
Eight cores
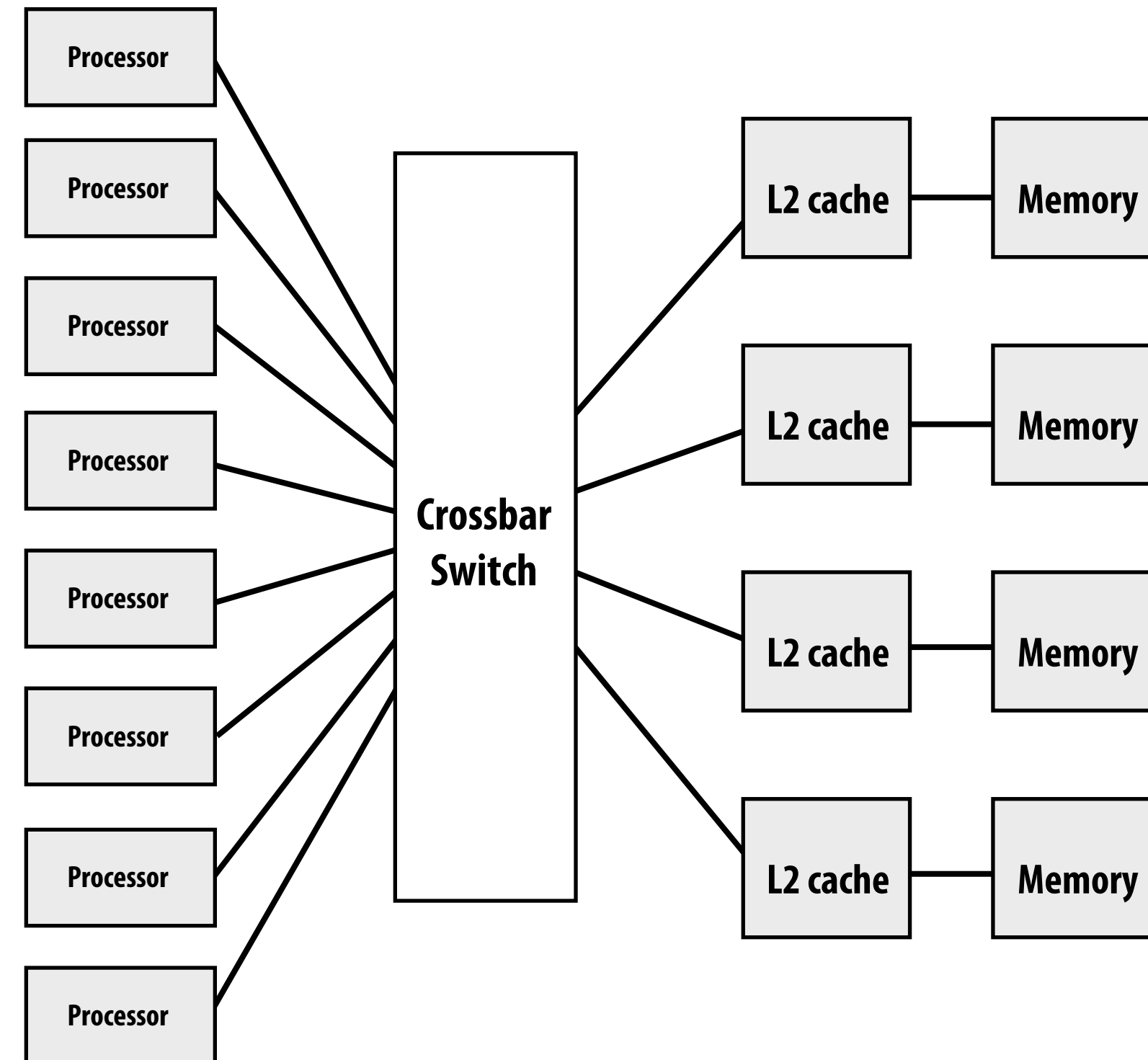
Snooping cache-coherence schemes

- **Main idea: all coherence-related activity is broadcast to all processors in the system (more specifically: to the processor's cache controllers)**

- **Cache controllers monitor ("they snoop") memory operations, and follow cache coherence protocol to maintain memory coherence**

Notice: now cache controller must respond to actions from "both ends":

1. LD/ST requests from its local processor

2. Coherence-related activity broadcast over the chip's interconnect

Processor	Processor	• • •	Processor
Cache	Cache		Cache

Interconnect

Memory
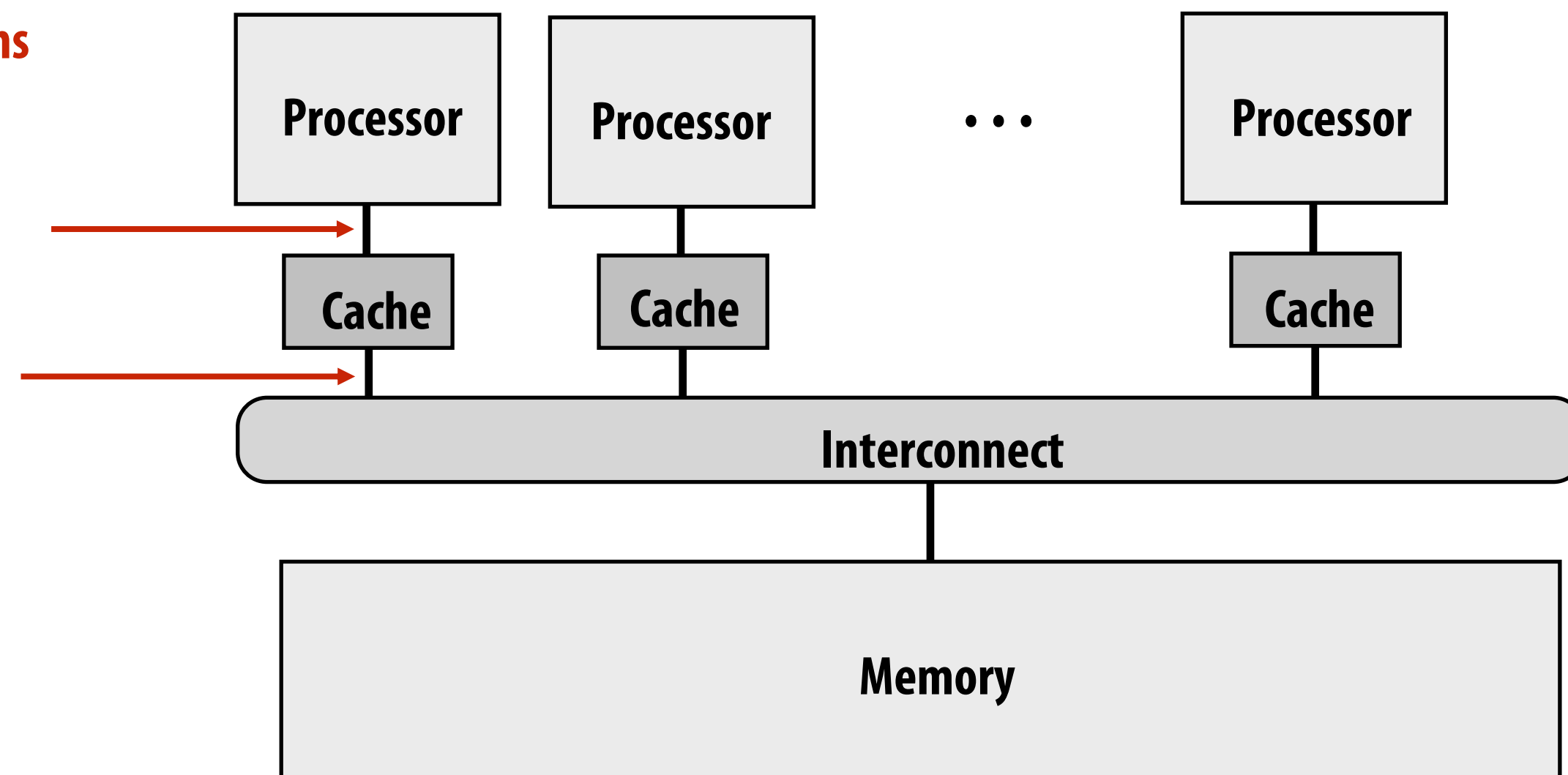
A clarifying note

- **The logic we are about to describe is performed by each processor's cache controller in response to:**

  – Loads and stores by the local processor

  – Messages from other caches

- **If all cache controllers operate according to this described protocol, then coherence will be maintained**

  – The caches "cooperate" to ensure coherence is maintained

Cache coherence with write-back caches

Processor P0 · · · Processor P1

Write X

Read X

X Cache

Cache

Bus

Memory
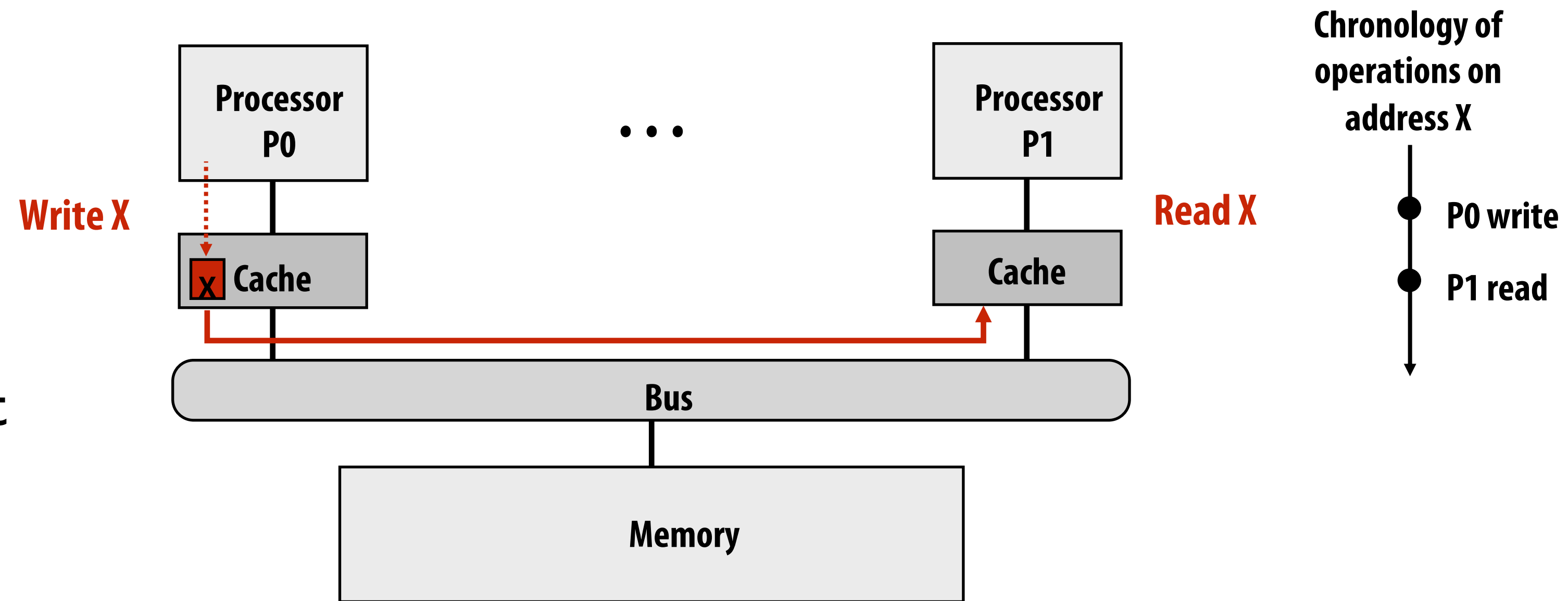
What are two important properties of a bus?

- **Dirty state of cache line now indicates exclusive ownership**
  - Exclusive: cache is only cache with a valid copy of line (it can safely be written to)
  - Owner: cache is responsible for propagating information to other processors when they attempt to load it from memory (otherwise a load from another processor will get stale data from memory)

Invalidation-based write-back protocol

Key ideas:

- **A line in the "exclusive" state can be modified without notifying the other caches**

- **Processor can only write to lines in the exclusive state**
  - Need a way to tell other caches that processor wants exclusive access to the line
  - We accomplish this by sending all the other caches messages

- **When cache controller sees a request for exclusive access to a line it contains**
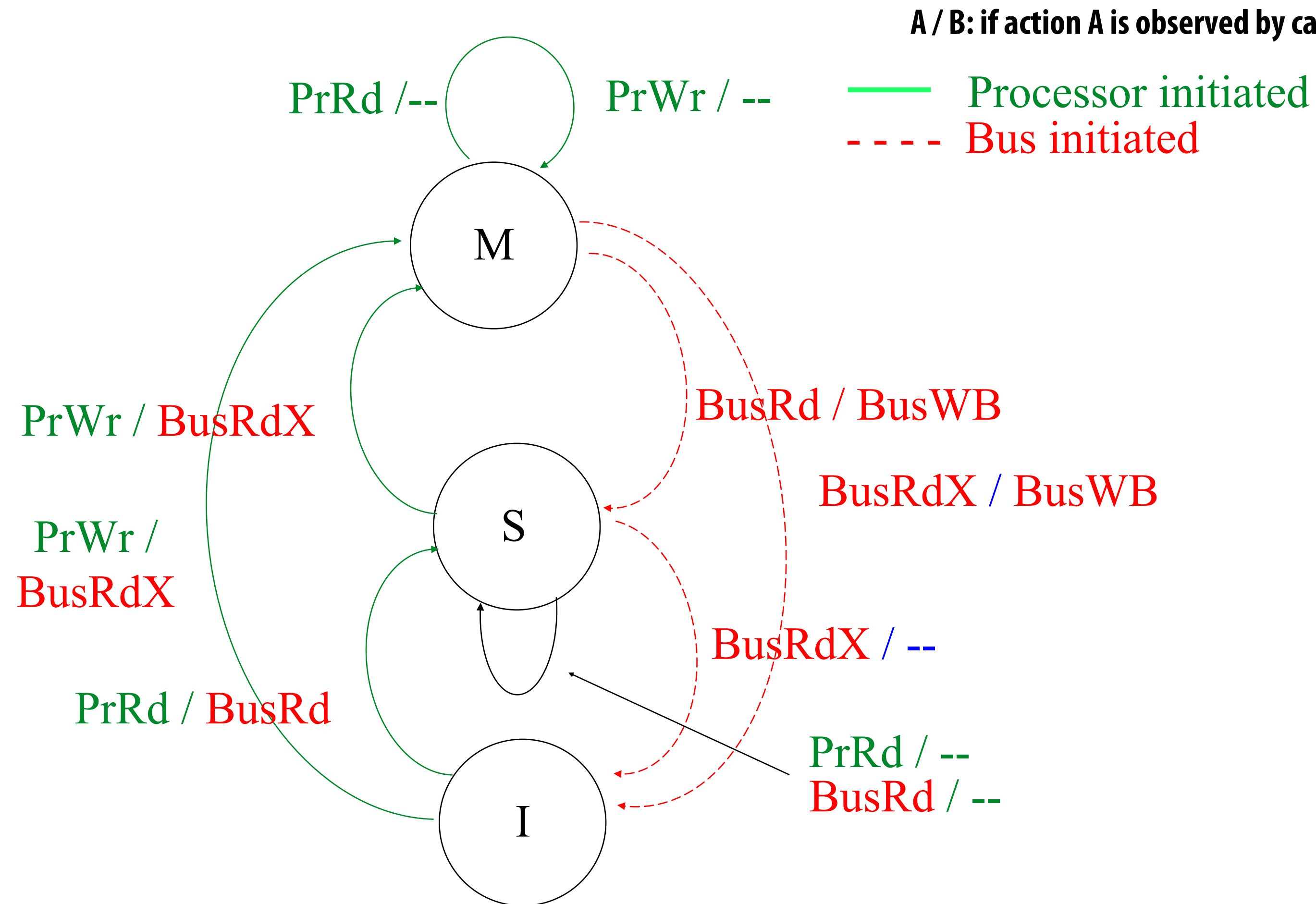 - It must invalidate the line in its cache

Recall cache line state bits

Line state	Tag	Data (64 bytes on modern Intel processors)

Dirty bit
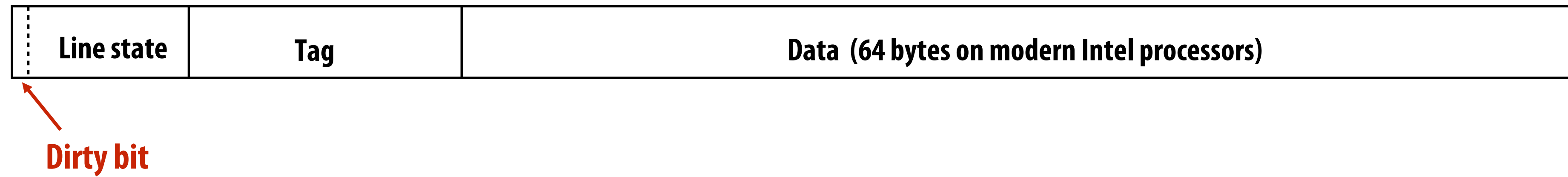
MSI write-back invalidation protocol

- **Key tasks of protocol**
  - Ensuring processor obtains exclusive access for a write
  - Locating most recent copy of cache line's data on cache miss

- **Three cache line states**
  - Invalid (I): same as meaning of invalid in uniprocessor cache
  - Shared (S): line valid in one or more caches
  - Modified (M): line valid in exactly one cache (a.k.a. "dirty" or "exclusive" state)

- **Two processor operations (triggered by local CPU)**
  - PrRd (read)
  - PrWr (write)

- **Three coherence-related bus transactions (from remote caches)**
  - BusRd: obtain copy of line with no intent to modify
  - BusRdX: obtain copy of line with intent to modify
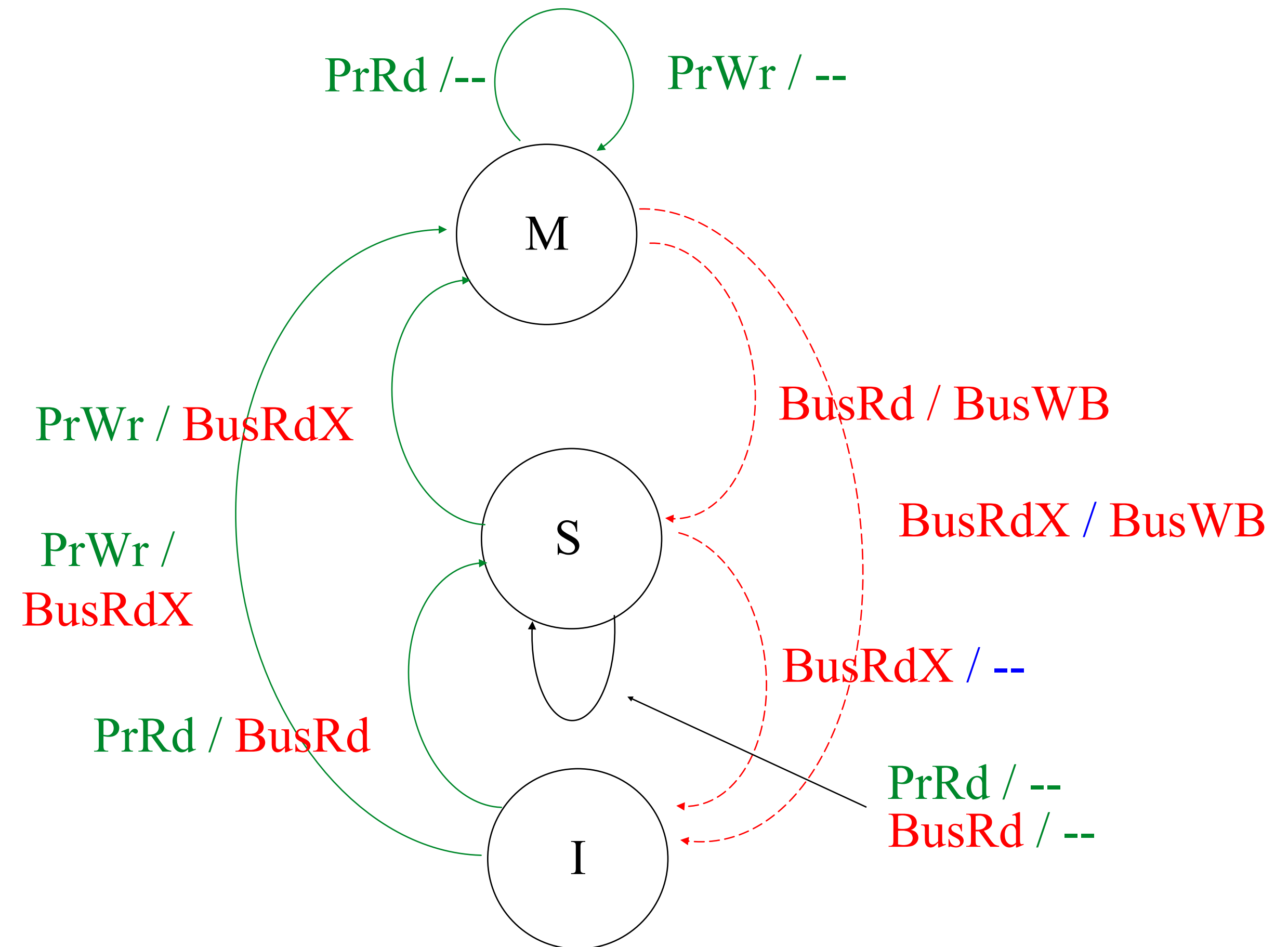 - BusWB: write dirty line out to memory

Cache Coherence Protocol: MSI State Diagram

A / B: if action A is observed by cache controller, action B is taken

— Processor initiated

- - - - Bus initiated

PrRd /--   PrWr / --

M

PrWr / BusRdX

BusRd / BusWB

BusRdX / BusWB

S

PrWr /
BusRdX

BusRdX / --

PrRd / BusRd

PrRd / --
BusRd / --

I

Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback

MSI Invalidate Protocol

- **Read obtains block in "shared"**
  - **even if only cached copy**

- **Obtain exclusive ownership before writing**
  - **BusRdX causes others to invalidate**
  - **If M in another cache, will cause writeback**
  - **BusRdX even if hit in S**
    - **promote to M (upgrade)**

PrRd /--   PrWr / --

M

PrWr / BusRdX

BusRd / BusWB

PrWr /
BusRdX

S

BusRdX / BusWB

BusRdX / --

PrRd / BusRd

I

PrRd / --
BusRd / --

* Remember, all caches are carrying out this logic independently to maintain coherence

A Cache Coherence Example

Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read x	S	--	--	BusRd	Memory
2. P3 read x	S	--	S	BusRd	Memory
3. P3 write x	I	--	M	BusRdX	Memory
4. P1 read x	S	--	S	BusRd	P3's cache
5. P2 read x	S	S	S	BusRd	Memory
6. P2 write x	I	M	I	BusRdX	Memory

- **Single writer, multiple reader protocol**
- **Why do you need Modified to Shared?**
- **Communication increases memory latency**

Summary: MSI

- **A line in the M state can be modified without notifying other caches**
  - No other caches have the line resident, so other processors cannot read these values

    (without generating a memory read transaction)

- **Processor can only write to lines in the M state**

  - If processor performs a write to a line that is not exclusive in cache, cache controller must first broadcast a <u>read-exclusive</u> transaction to move the line into that state
  - Read-exclusive tells other caches about impending write

  ("you can't read any more, because I'm going to write")

  - Read-exclusive transaction is required even if line is valid (but not exclusive... it's in the S state) in processor's local cache (why?)

  - Dirty state implies exclusive

- **When cache controller snoops a "read exclusive" for a line it contains**

  - Must invalidate the line in its cache
  - Because if it didn't, then multiple caches will have the line

   (and so it wouldn't be exclusive in the other cache!)

Does MSI Satisfy Cache Coherence

1. **Single-Writer, Multiple-Read (SWMR) Invariant**
   - Only one cache can be in M-state all others get invalidation message
   - Multiple caches can be in read-only S-state

2. **Data-Value Invariant (write serialization)**
   - On BusRd and BusRdx data is provided by cache with line in M-state
   - Bus serializes all transactions

| read-only
Cores 2 and 5 | read-write
Core 3 | read-write
Core 1 | read-only
Cores 1, 2, 3 | time |
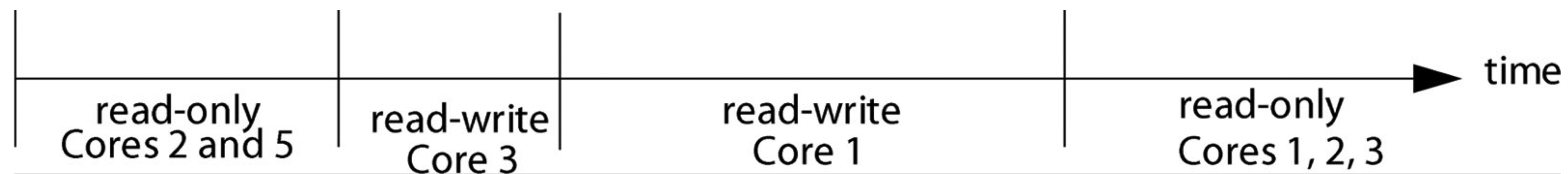
MESI invalidation protocol

MESI, not Messi!

- **MSI requires two interconnect transactions for the common case of reading an address, then writing to it**
  - Transaction 1: BusRd to move from I to S state

  - Transaction 2: BusRdX to move from S to M state

- **This inefficiency exists even if application has no sharing at all**

- **Solution: add additional state E ("exclusive clean")**
  - Line has not been modified, but only this cache has a copy of the line
  - Decouples exclusivity from line ownership (line not dirty, so copy in memory is valid copy of data)
  - Upgrade from E to M does not require an bus transaction

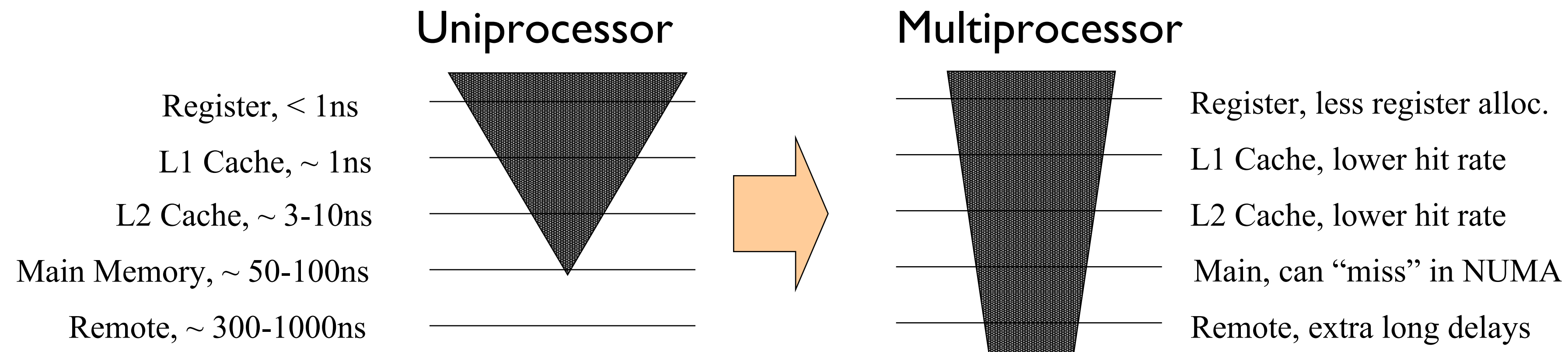# MESI state transition diagram
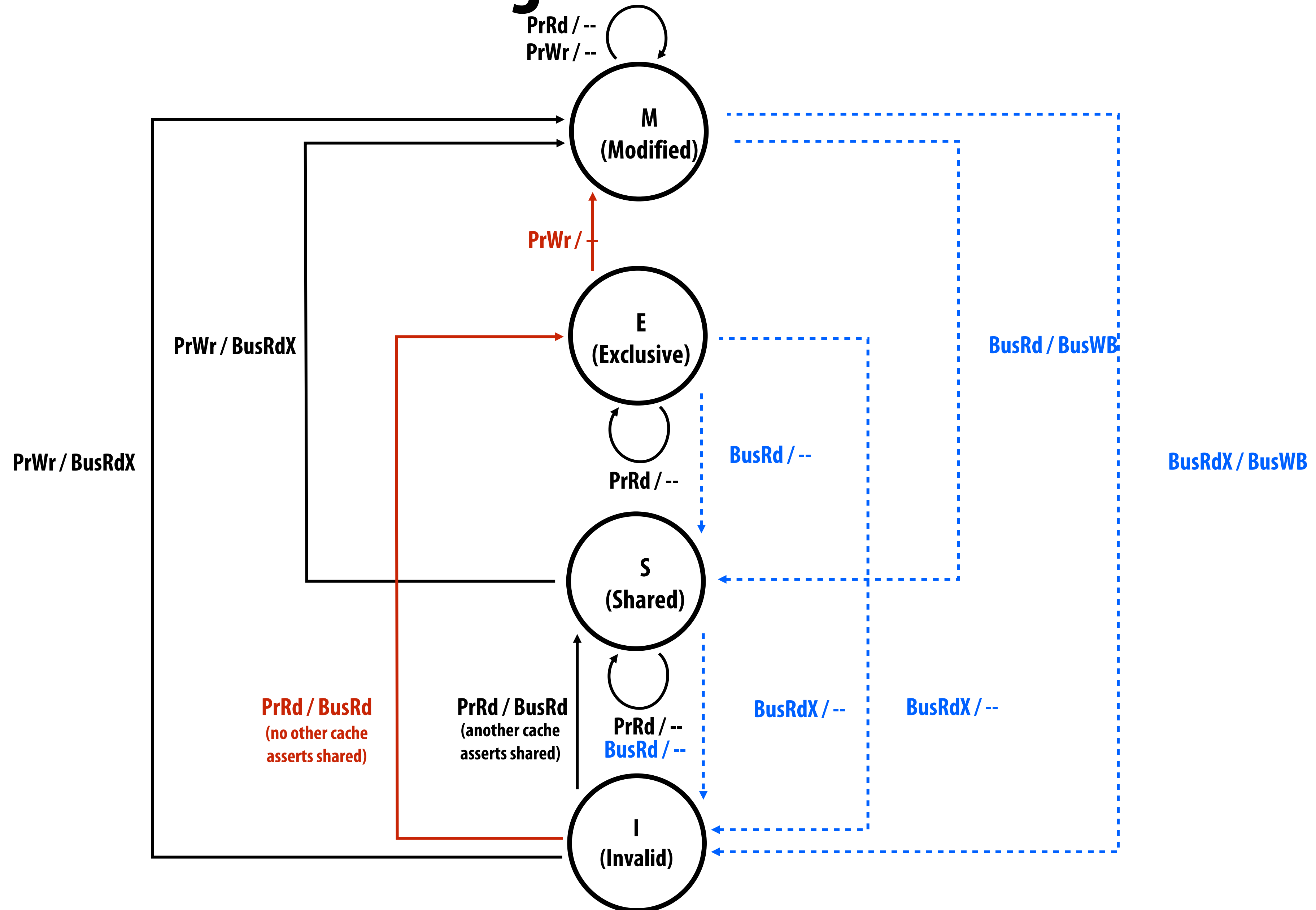
Two Hard Things

There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

Implications of cache coherence
to the programmer

Communication Overhead

- **Communication time is key parallel overhead**
  - **Appears as increased memory latency in multiprocessor**
    - **Extra main memory accesses in UMA systems**
      - **Must determine lowering of cache miss rate vs. uniprocessor**
    - **Some accesses have higher latency in NUMA systems**
      - **Only a fraction of a % of these can be significant!**

Uniprocessor		Multiprocessor
Register, < 1ns		Register, less register alloc.
L1 Cache, ~ 1ns		L1 Cache, lower hit rate
L2 Cache, ~ 3-10ns		L2 Cache, lower hit rate
Main Memory, ~ 50-100ns		Main, can "miss" in NUMA
Remote, ~ 300-1000ns		Remote, extra long delays

Unintended communication via false sharing

What is the potential performance problem with this code?

```
// allocate per-thread variable for local per-thread accumulation
int myPerThreadCounter[NUM_THREADS];
```

Why might this code be more performant?

```
// allocate per thread variable for local accumulation
struct PerThreadState {
    int myPerThreadCounter;
    char padding[CACHE_LINE_SIZE - sizeof(int)];
};
PerThreadState myPerThreadCounter[NUM_THREADS];
```

Demo: false sharing

```
void* worker(void* arg) {

    volatile int* counter = (int*)arg;

    for (int i=0; i<MANY_ITERATIONS; i++)
        (*counter)++;

    return NULL;
}
```

threads update a per-thread counter many times

```
void test1(int num_threads) {

    pthread_t threads[MAX_THREADS];
    int       counter[MAX_THREADS];

    for (int i=0; i<num_threads; i++)
        pthread_create(&threads[i], NULL,
                            &worker, &counter[i]);

     for (int i=0; i<num_threads; i++)
        pthread_join(threads[i], NULL);
}
```

```
struct padded_t {
    int  counter;
    char padding[CACHE_LINE_SIZE - sizeof(int)];
};

void test2(int num_threads) {

    pthread_t threads[MAX_THREADS];
    padded_t  counter[MAX_THREADS];


    for (int i=0; i<num_threads; i++)
        pthread_create(&threads[i], NULL,
                            &worker, &(counter[i].counter));

    for (int i=0; i<num_threads; i++)
        pthread_join(threads[i], NULL);
}
```

Execution time with num_threads=8
on 4-core system: 14.2 sec

Execution time with num_threads=8
on 4-core system: 4.7 sec

False sharing

- **Condition where two processors write to different addresses, but addresses map to the same cache line**

- **Cache line "ping-pongs" between caches of writing processors, generating significant amounts of communication due to the coherence protocol**

- **No inherent communication, this is entirely** <u>**artifactual communication**</u>

- **False sharing can be a factor in when programming for cache-coherent architectures**

Cache line

P1　　P2
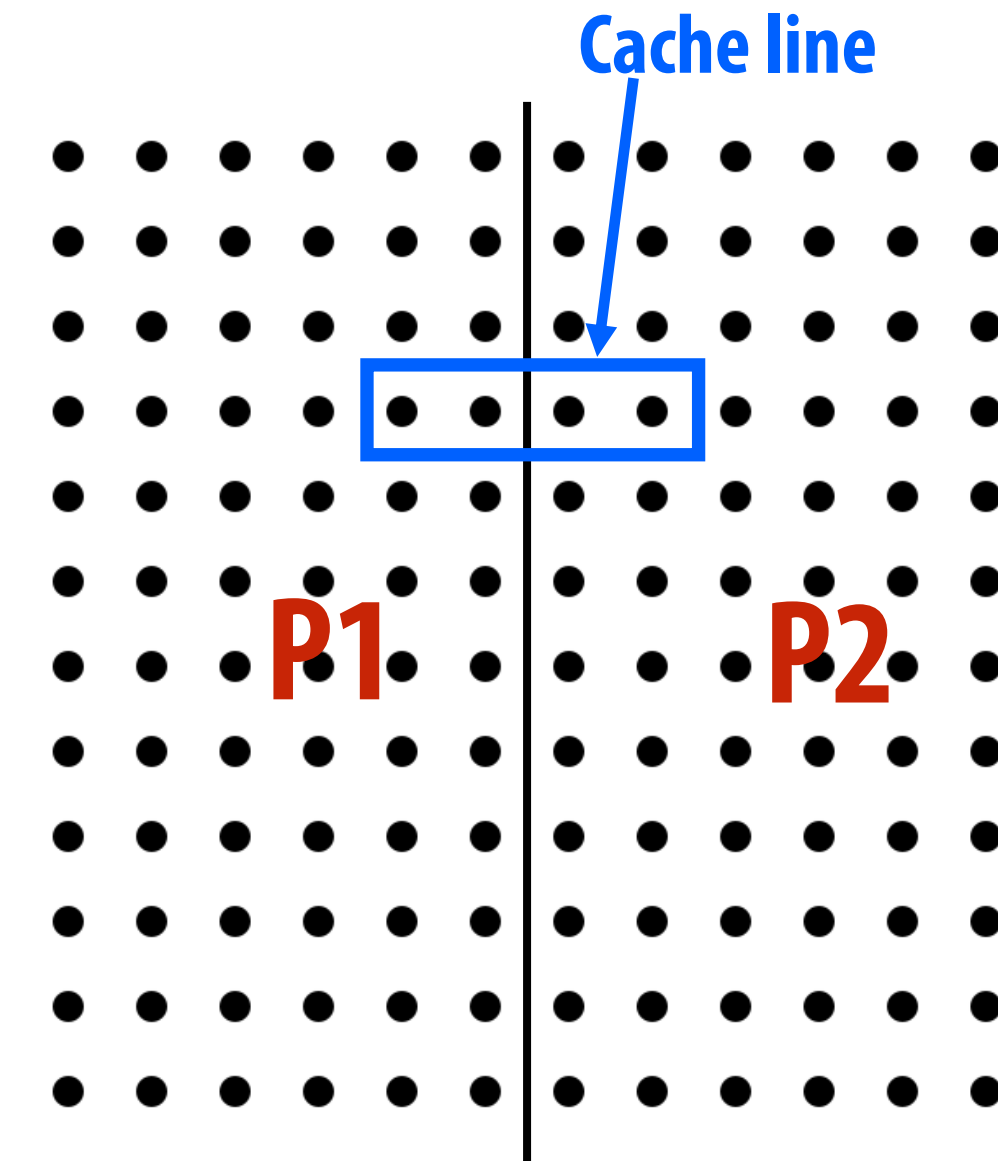
Summary: snooping-based coherence

- **The cache coherence problem exists because the <u>abstraction</u> of a single shared address space is not <u>implemented</u> by a single storage unit**
  - Storage is distributed among main memory and local processor caches
  - Data is replicated in local caches for performance

- **Main idea of snooping-based cache coherence: whenever a cache operation occurs that could affect coherence, the cache controller broadcasts a notification to all other cache controllers in the system**
  - Challenge for HW architects: minimizing overhead of coherence implementation
  - Challenge for SW developers: be wary of artifactual communication due to coherence protocol (e.g., false sharing)

- **Scalability of snooping implementations is limited by ability to broadcast coherence messages to all caches!**
  - In a future lecture: scaling cache coherence via directory-based approaches

What is OpenMP?

- **OpenMP is a pragma based API that provides a simple extension to C/C++ and FORTRAN**

- **It is designed for shared memory programming**

- **OpenMP is a very simple interface to threads based programming**

  - **Compiler directives**

  - **Environment variables**

  - **Run time routines**

Data Parallelism with OpenMP

For-loop with independent iterations

```
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

For-loop parallelized using
an OpenMP pragma

```
#pragma omp parallel for  \
        shared(n, a, b, c)\
        private(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out
```

```
gcc source.c -fopenmp
```

Privatizing Variables

- **Critical to performance!**

- **OpenMP pragmas:**

  - Designed to make parallelizing sequential code easier

  - Makes copies of "private" variables *automatically*

    - And performs some automatic initialization, too

  - Must specify shared/private per-variable in parallel region

    - private: Uninitialized private data

      - Private variables are undefined on entry and exit of the parallel region

    - shared: All-shared data

    - threadprivate: "static" private for use across several parallel regions

Firstprivate/Lastprivate Clauses

- **`firstprivate (list)`**
  - All variables in the list are initialized with the value the original object had before entering the parallel region

- **`lastprivate(list)`**
  - The thread that executes the last iteration or section in sequential order updates the value of the objects in the list

Example Private Variables

```
main()
{
  A = 10;


#pragma omp parallel
{
  #pragma omp for private(i) firstprivate(A) lastprivate(B)...
  for (i=0; i<n; i++)
  {
      ....
      B = A + i;          /*-- A undefined, unless declared
                                  firstprivate --*/


      ....
  }

  C = B;                  /*-- B undefined, unless declared
                                  lastprivate --*/



} /*-- End of OpenMP parallel region --*/
}
```

for directive Example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for

      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;

    #pragma omp for

      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
                              (implied barrier)
```
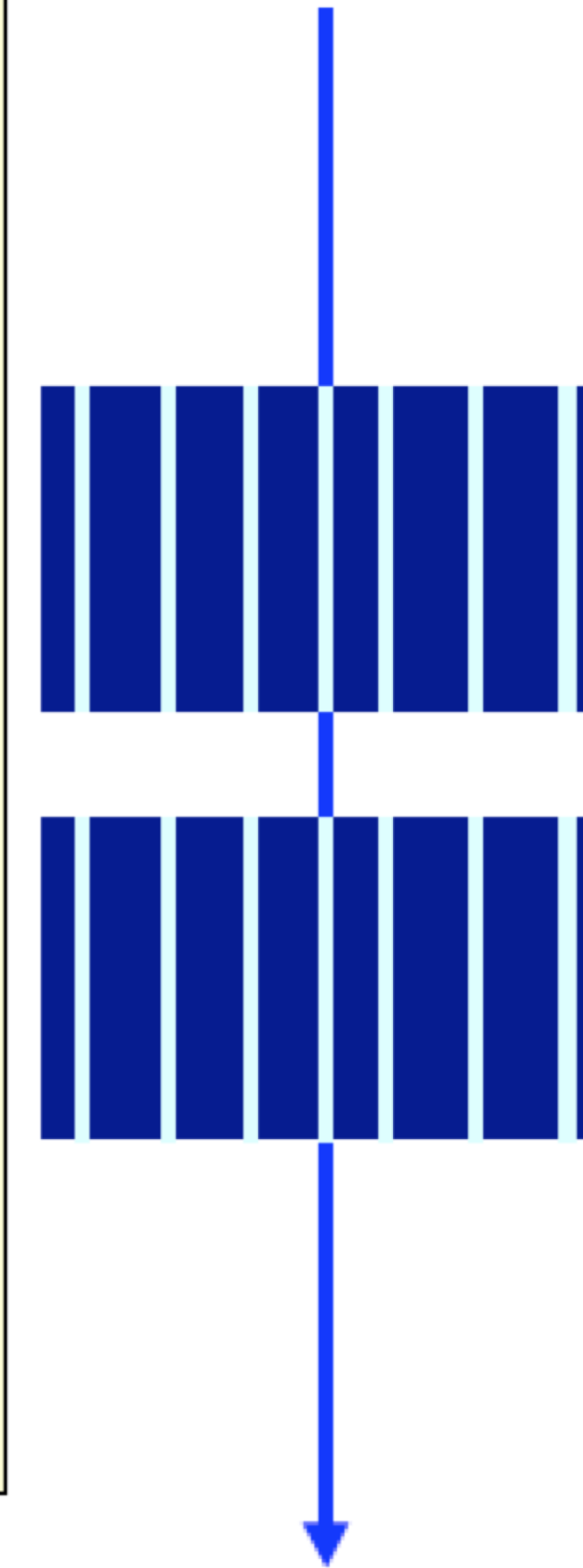
Nested Loop Parallelism

```
#pragma omp parallel for
 for(int y=0; y<25; ++y)
 {
    #pragma omp parallel for
    for(int x=0; x<80; ++x)
       tick(x,y);
 }
```

Multiple Part Parallel Regions

- **You can also have a "multi-part" parallel region**
  - Allows easy alternation of serial & parallel parts
  - Doesn't require re-specifying # of threads, etc.

```
#pragma omp parallel . . .
{
 #pragma omp for
 . . . Loop here . . .
 #pragma omp single
 . . . Serial portion here . . .
 #pragma omp sections
 . . . Sections here . . .
}
```
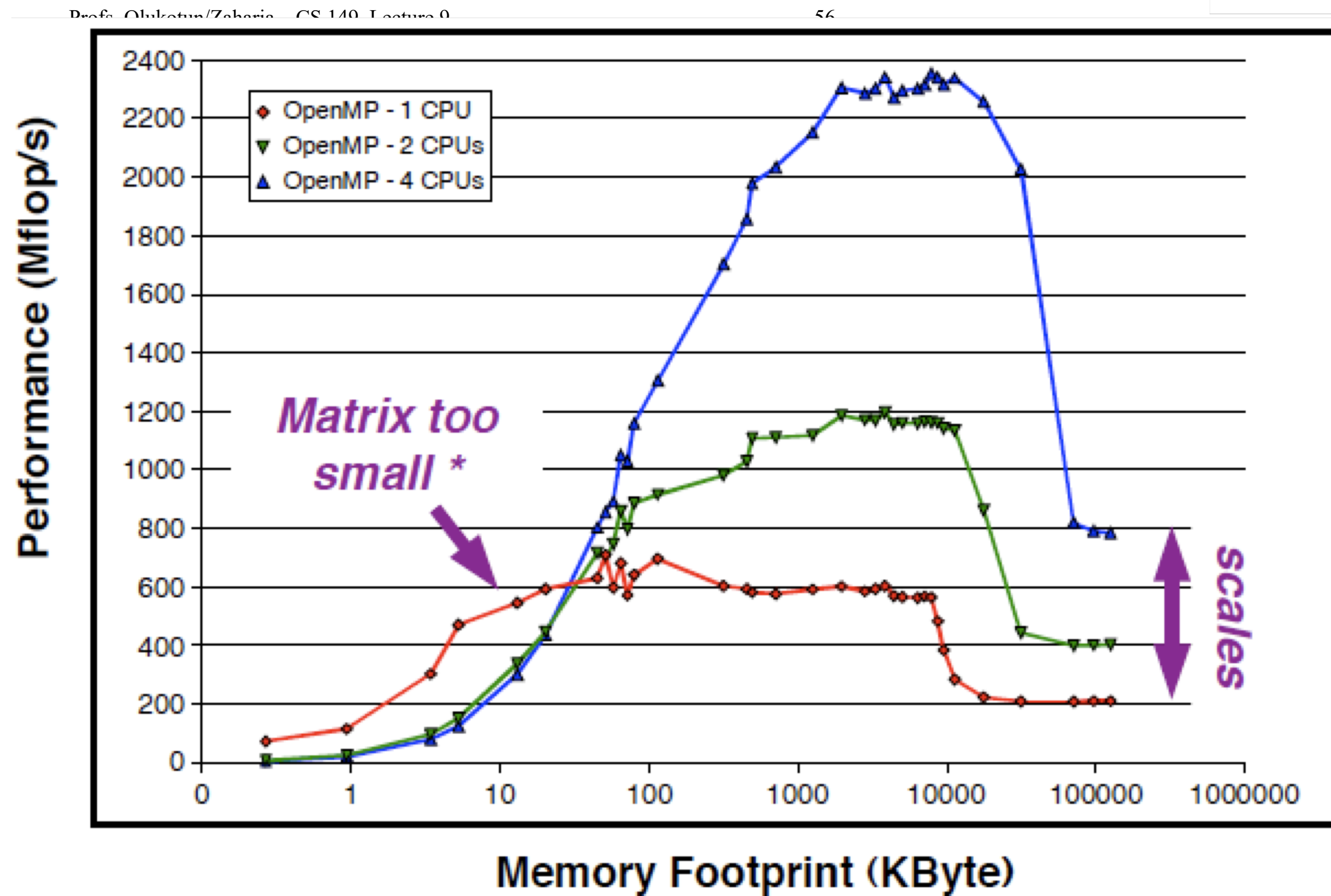
OMP Directives Overheads
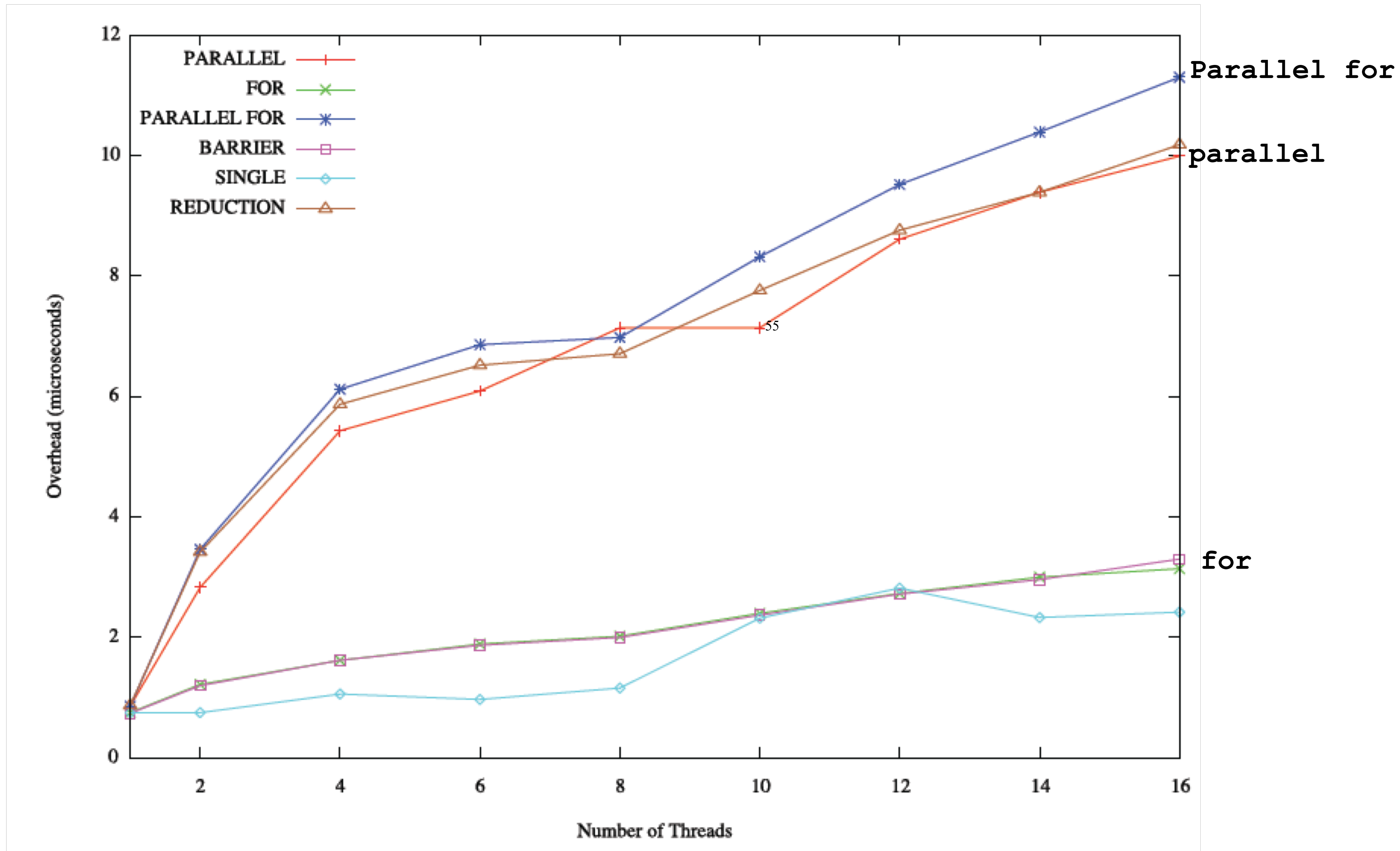
"if" Clause

- **if (scalar expression)**
  - Only execute in parallel if expression evaluates to true
  - Otherwise, execute serially

```
#pragma omp parallel if (n > threshold) \
        shared(n,x,y) private(i)
{
  #pragma omp for
    for (i=0; i<n; i++)
        x[i] += y[i];
} /*-- End of parallel region --*/
```

Performance without if clause

Reductions in OpenMP

- **May add reduction clause to parallel for pragma**

- **Specify reduction operation and reduction variable**

- **OpenMP takes care of storing partial results in private variables and combining partial results after the loop**

- **The reduction clause has this syntax:**
  ```
  reduction (<op> :<variable>)
  ```

- **Operators**
  - **`+`   Sum**
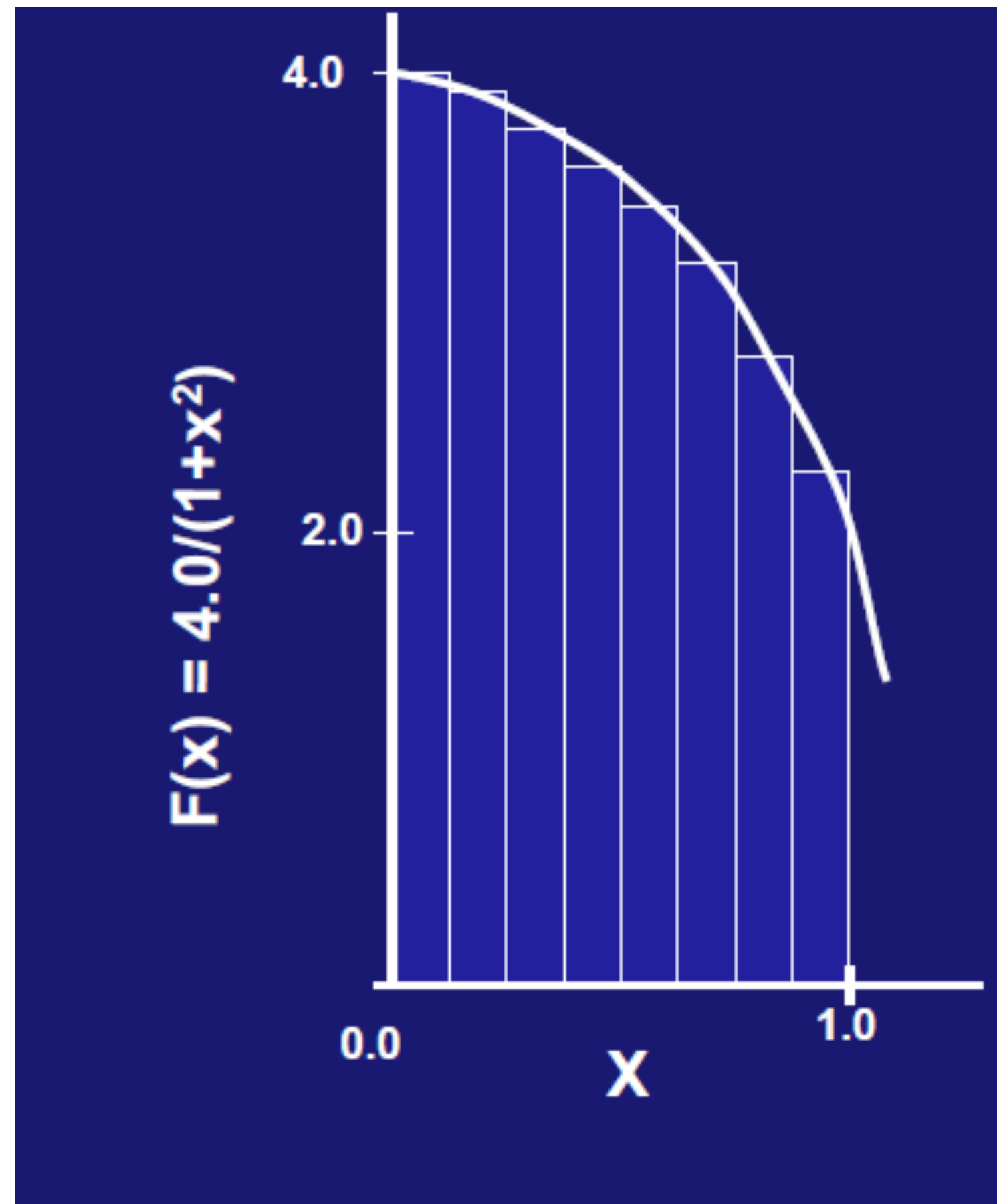 - **`*`   Product**
  - **`&, |, ^`   Bitwise and, or , exclusive or**
  - **`&&, ||`   Logical and, or**

Example: Numerical Integration

- **We know mathematically that**

$$\pi = \int_0^1 \frac{4.0}{(1 + x^2)} \, dx$$

$$\sum_{i=0}^{N} F(x_i) \Delta x \approx \pi$$

- **We can approximate the integral as a sum of rectangles:**

Sequential Pi Computation

```
static long num_steps = 100000;
double step;

void main () {
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Loop Parallelized Pi Computation

```c
#include <omp.h>
static long num_steps = 1000000; double step;
#define NUM_THREADS 8

void main (){
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(x) reduction(+:sum)
   for (i=0;i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

- Notice that we haven't changed any lines of code, only added 4 lines

- Compare to MPI