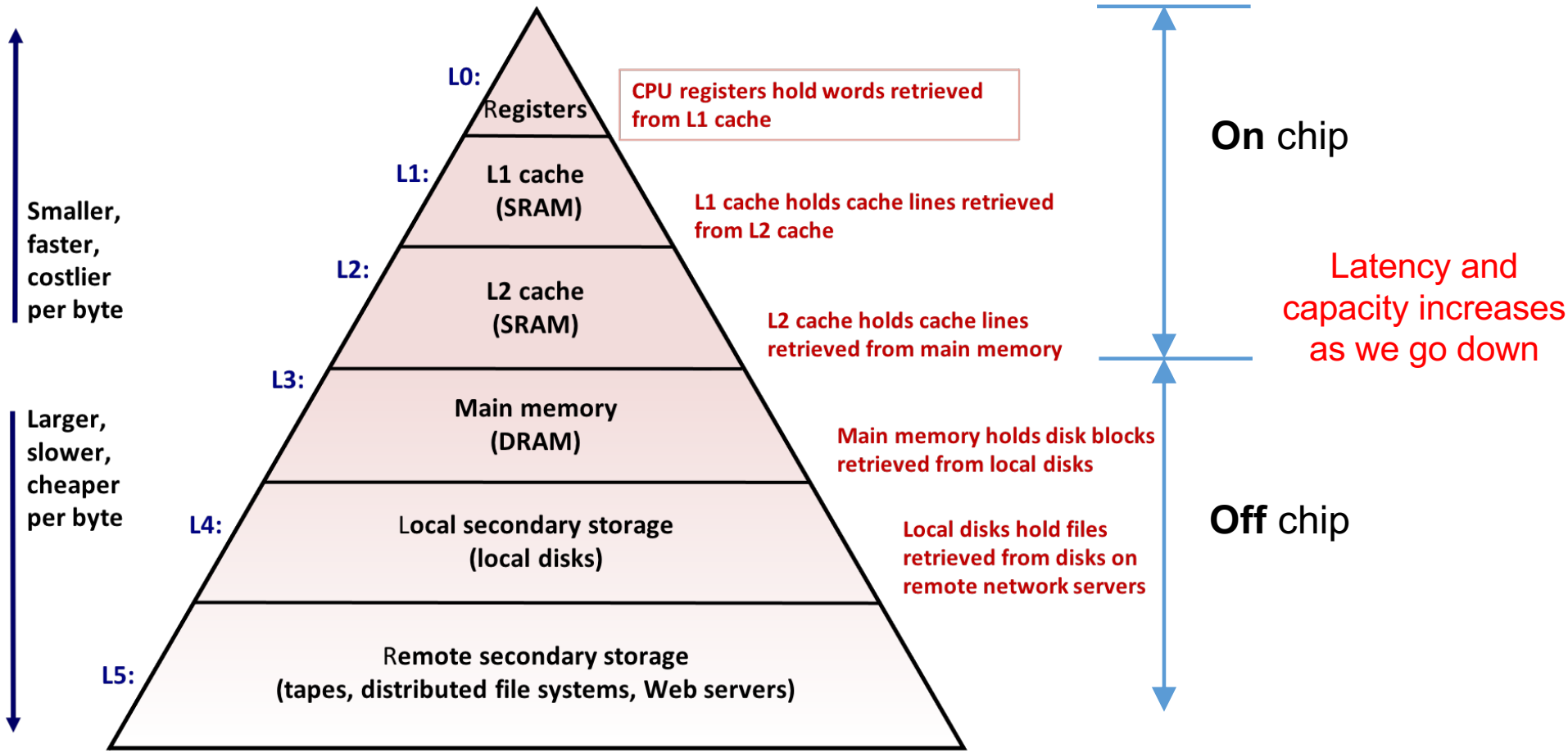
Lecture 19: Cache Coherency

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Class

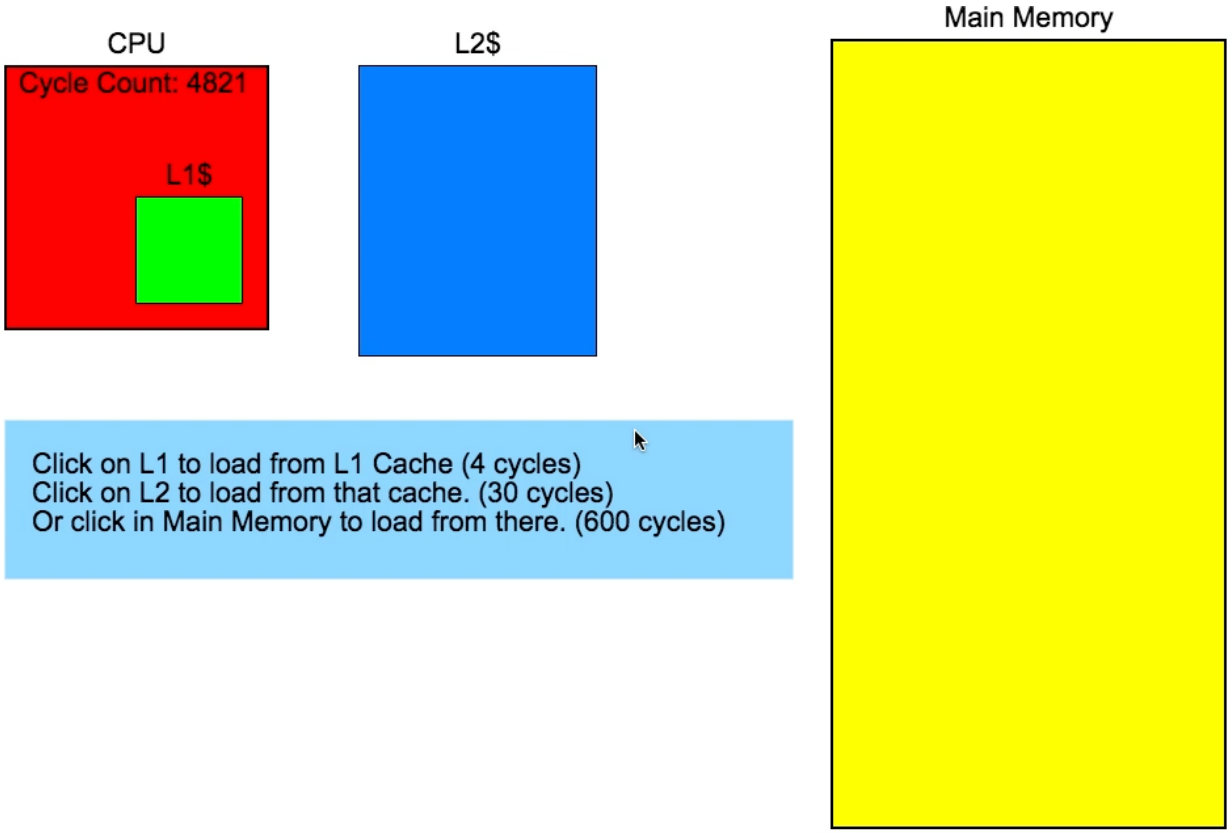
- Cache coherency
- MSI protocol
- MESI protocol

Memory Hierarchy



Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx

How Bad is Latency as we go Down?



- Another analogy
  - Normalizing with L1 latency, and assuming one seconds is equal to 4 cycles
    - L1 = one second
    - L2 = 7.5 seconds
    - Main memory = 2.5 minutes
    - Hard drive = in several days!

Animation source: https://overbyte.com.au/misc/Lesson3/CacheFun.html

Which is Better & Why?

```
float A[n][n]; // initialized
float sum=0;
...
for(int row=0; row<n; row++) {
    for(int col=0; col<n; col++) {
        sum += A[row][col];
    }
}
```

v/s

```
float A[n][n]; // initialized
float sum=0;
...
for(int col=0; col<n; col++) {
    for(int row=0; row<n; row++) {
        sum += A[row][col];
    }
}
```

```
float A[n], B[n], C[n]; // initialized

for(int i=0; i<n; i++) {
    C[i] = A[i] + B[i];
}
```

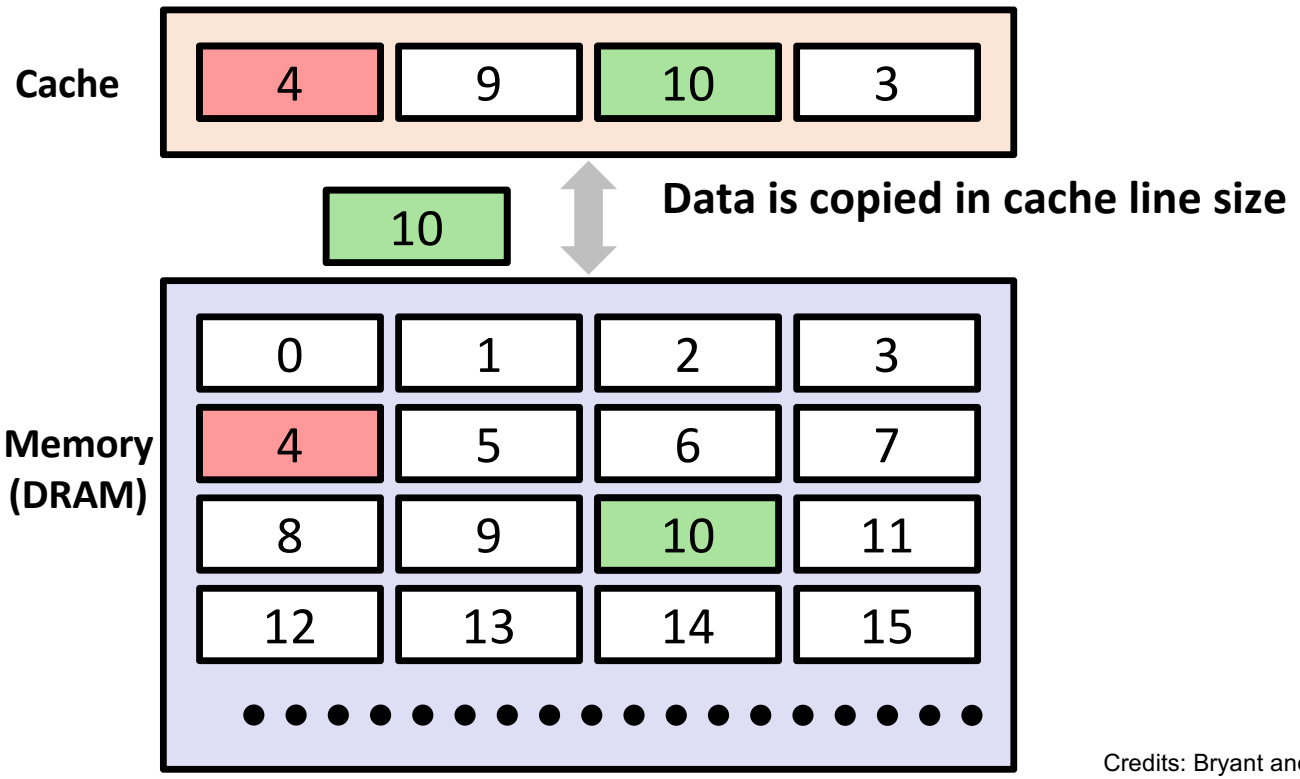
v/s

```
typedef struct Triplet {
    float A;
    float B;
    float C;
} Triplet;

Triplet T[n]; // initialized

for(int i=0; i<n; i++) {
    T[i].C = T[i].A + T[i].B;
}
```

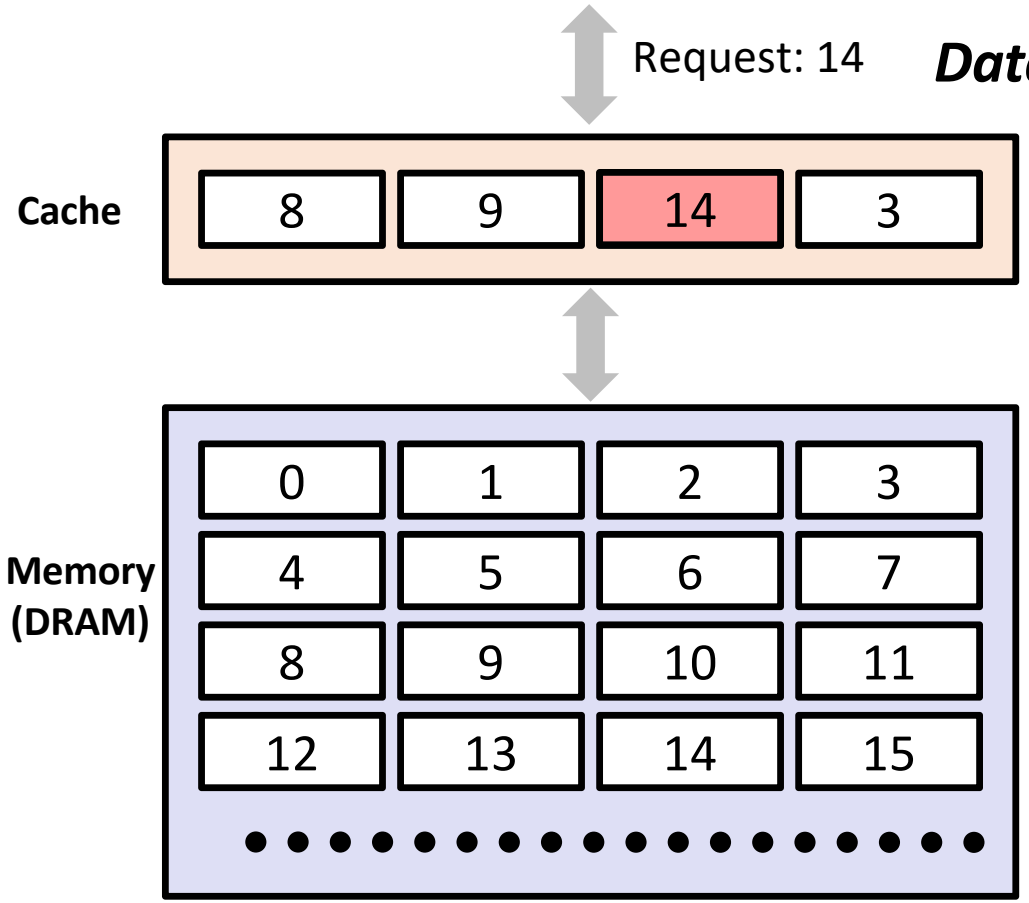
General Cache Concepts



- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device
  - Temporal and spatial locality

Credits: Bryant and O'Hallaron, Lecuture 9, CMU 15-213/18-243

General Cache Concepts



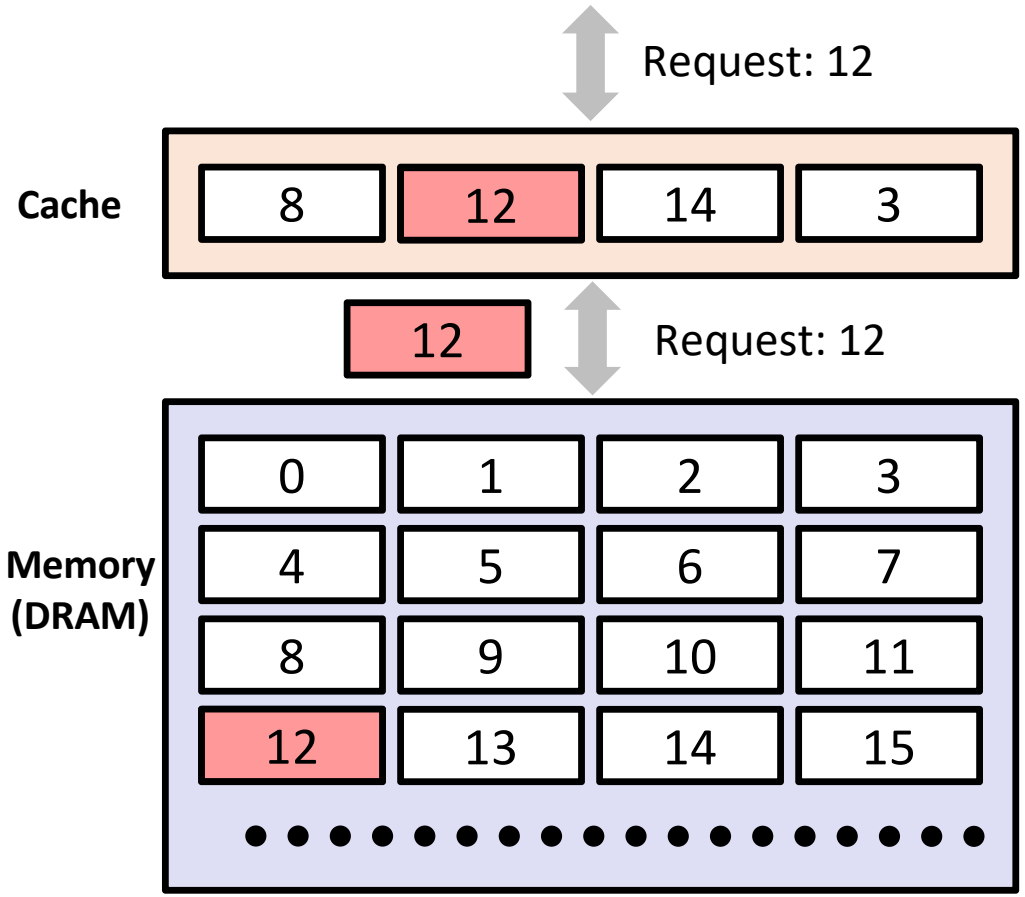
Line b is in cache:

Hit!

- Cache hit
  - Data is already in the cache in some cache line
    - Cache line size is 64 bytes on x86 processors
    - Cache line is the smallest granularity of load/store of any memory block from DRAM

Credits: Bryant and O'Hallaron, Lecuture 9, CMU 15-213/18-243

General Cache Concepts



Data in line b is needed

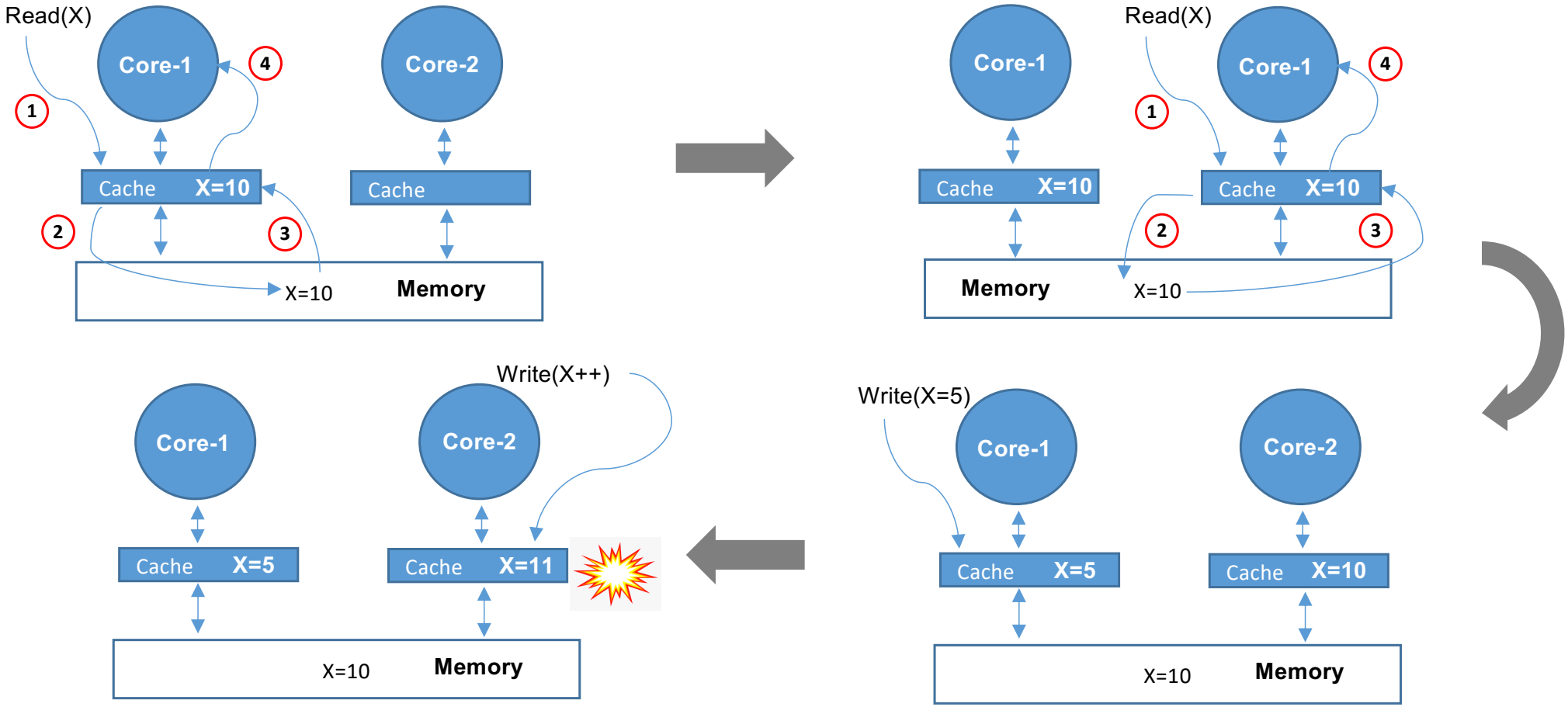
Line b is not in cache:
Miss!

Line b is fetched from *memory*

Line b is stored in cache
By evicting some old line

Credits: Bryant and O'Hallaron, Lecuture 9, CMU 15-213/18-243

The Cache Coherence Problem

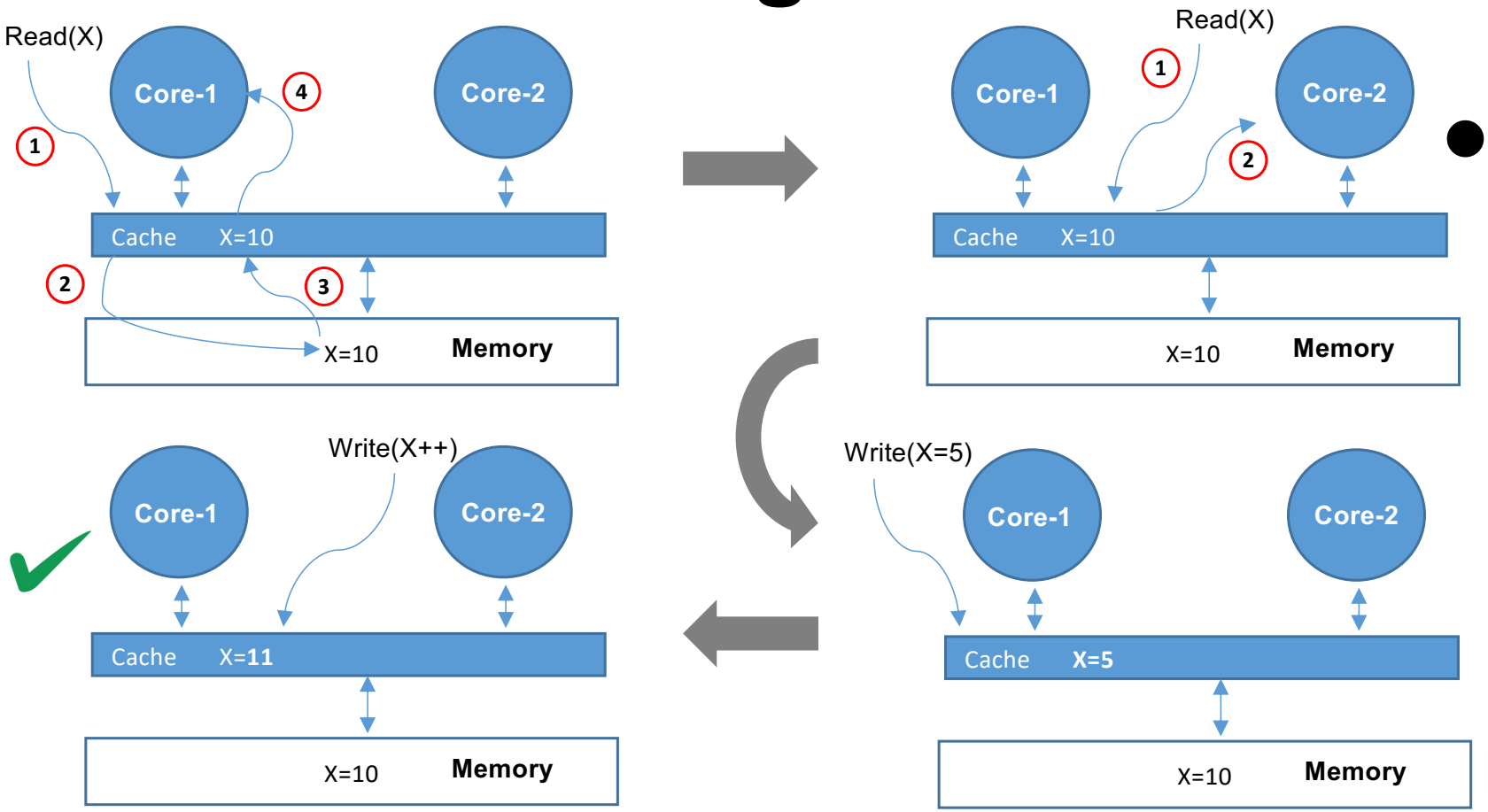


Defining Cache Coherence Differently (As we have studied Memory Consistency)

- **Program order** must be maintained at a single processor
  - A read by processor P to address X that follows a write by P to address X, should return the value of the write by P
    - Assuming no other processor wrote to X in between
- **Write propagation** to other processors
  - A read by processor P1 to address X that follows a write by processor P2 to X returns the written value... if the read and write are "sufficiently separated" in time (store buffers!)
    - Assuming no other writes to X occurs in between
- **Write serialization**
  - Writes to the same address are serialized: two writes to address X by any two processors are observed in the same order by all processors
    - E.g., if values 1 and then 2 are written to address X, no processor observes X having value 2 before value 1

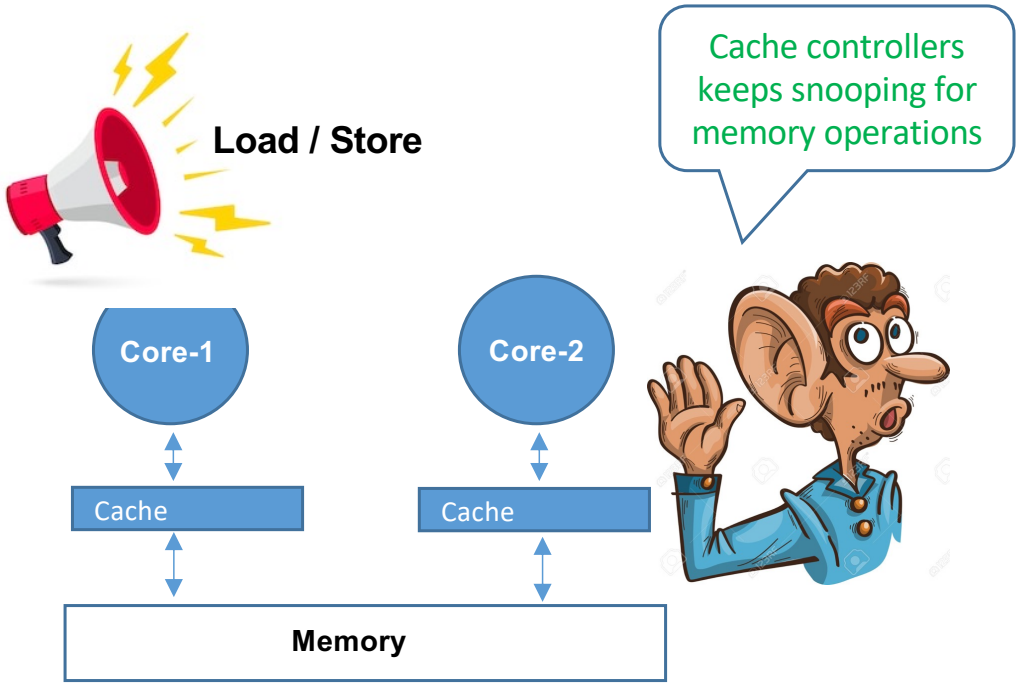
Credits: Fatahalian and Bryant, CMU 15-418/618

Coherence using Shared Cache Only



- While it is easy to implement, it would be very costly and inefficient
  - Why?

Coherence using Private Caches

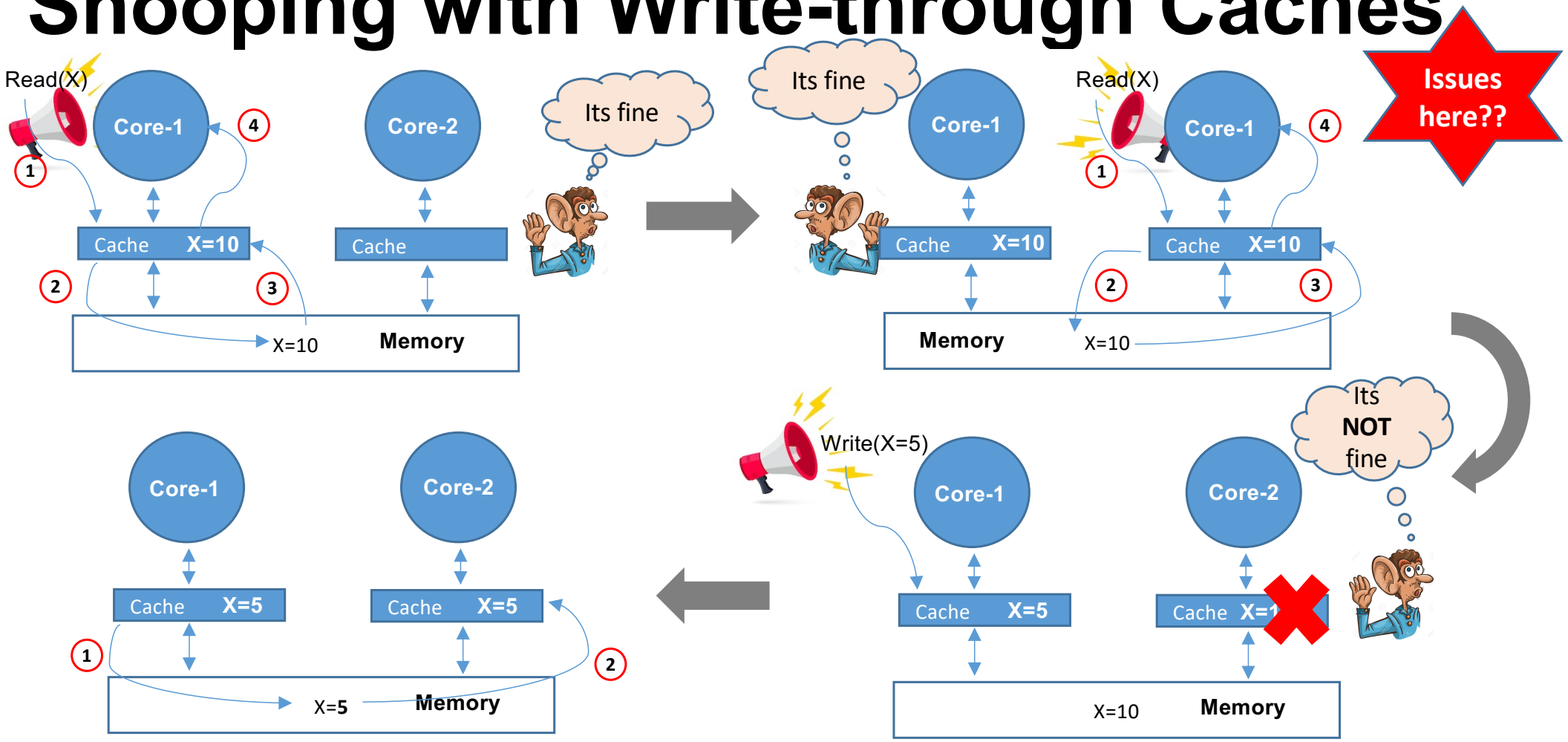


- Snooping based coherency protocol
  - Each core's cache controller broadcast any memory operations it wishes to perform before actually carrying out that operation
  - Rest of the core's cache controllers having that memory operations acts like a good citizen and help others to carry out their intended operation

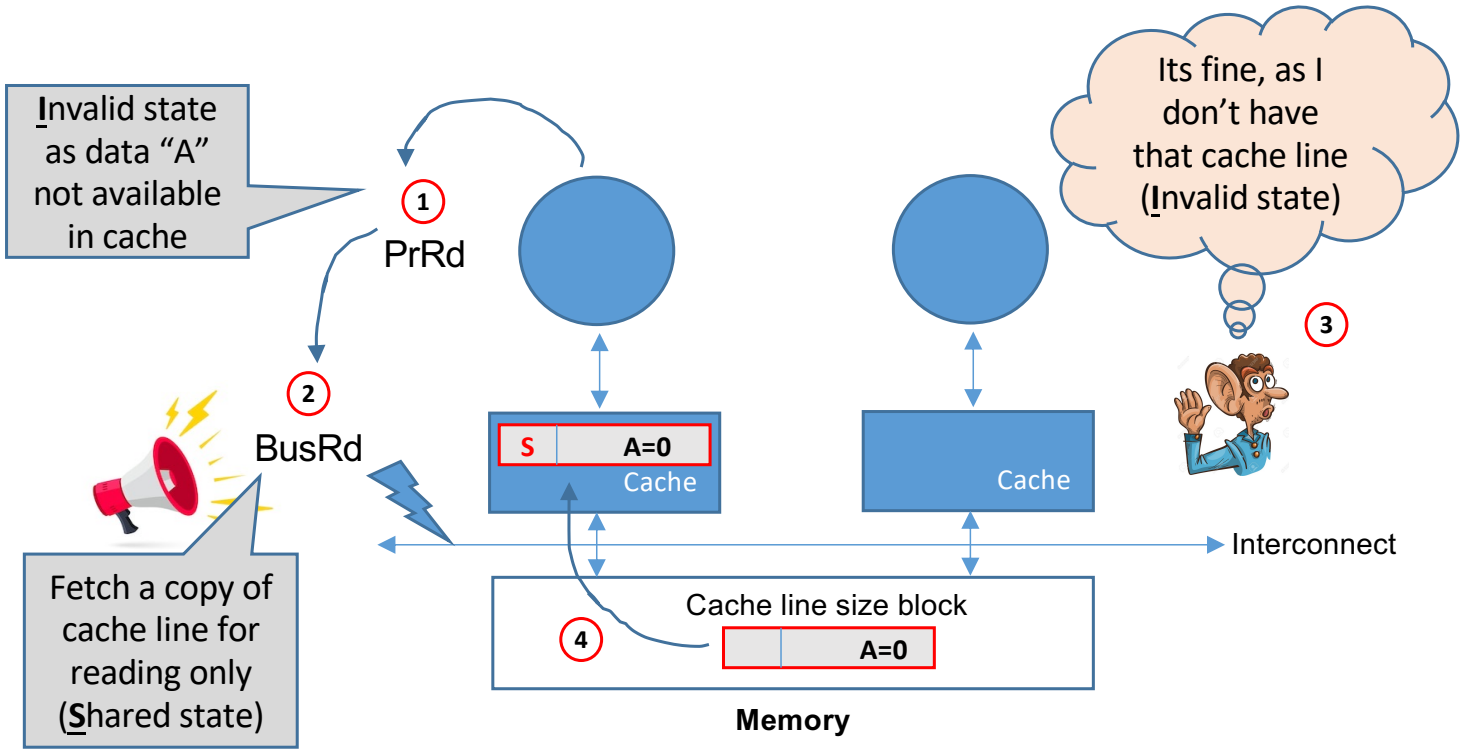
Private v/s Shared Cache Coherency

- Coherency using shared cache
  - Cache just have to look up to the processor and do the load/store instructions issued by the processor
- Coherency using private caches
  - Each cache has its own core to which it look after, but it also pays attention to what is going on in other caches or what is going over the interconnect
    - They are snooping!

Snooping with Write-through Caches

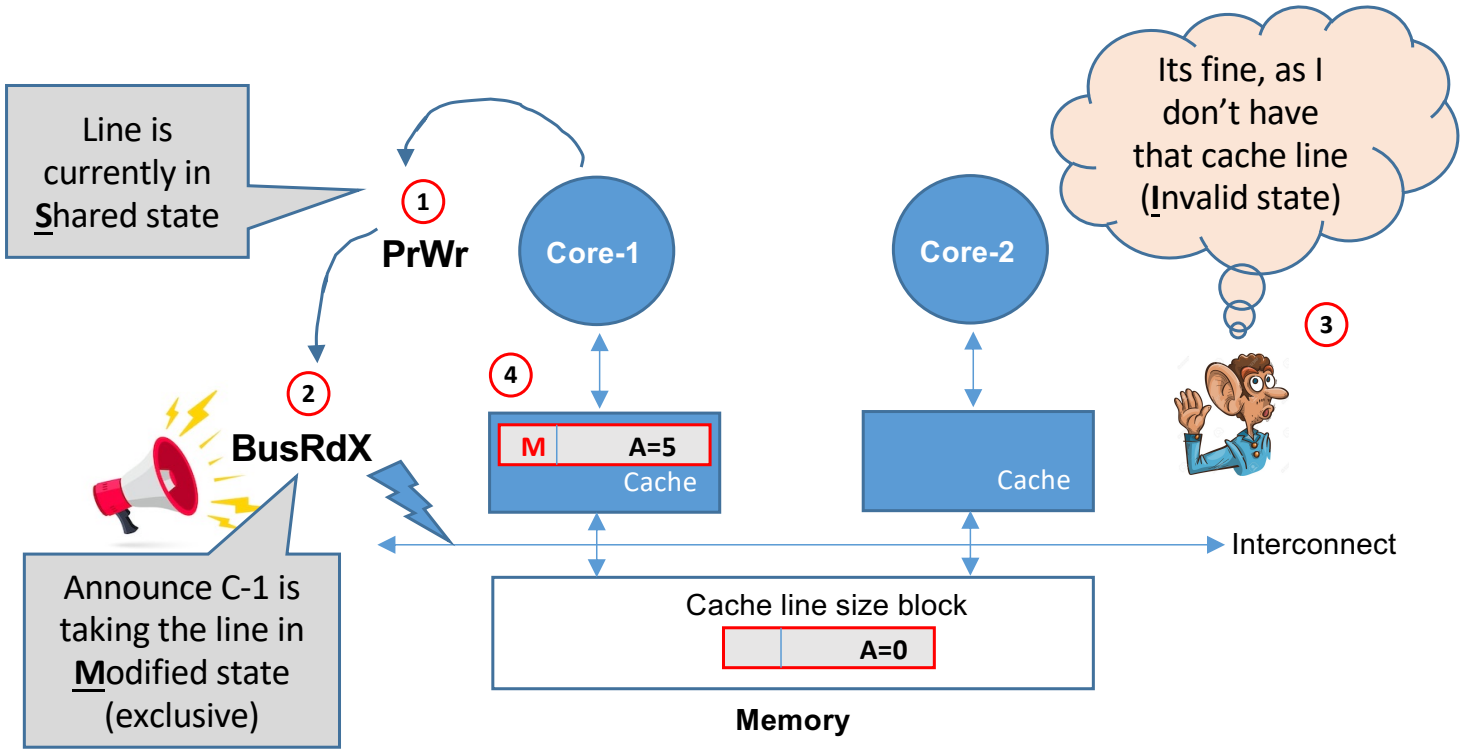


Snooping using Write-back Caches (1/5)



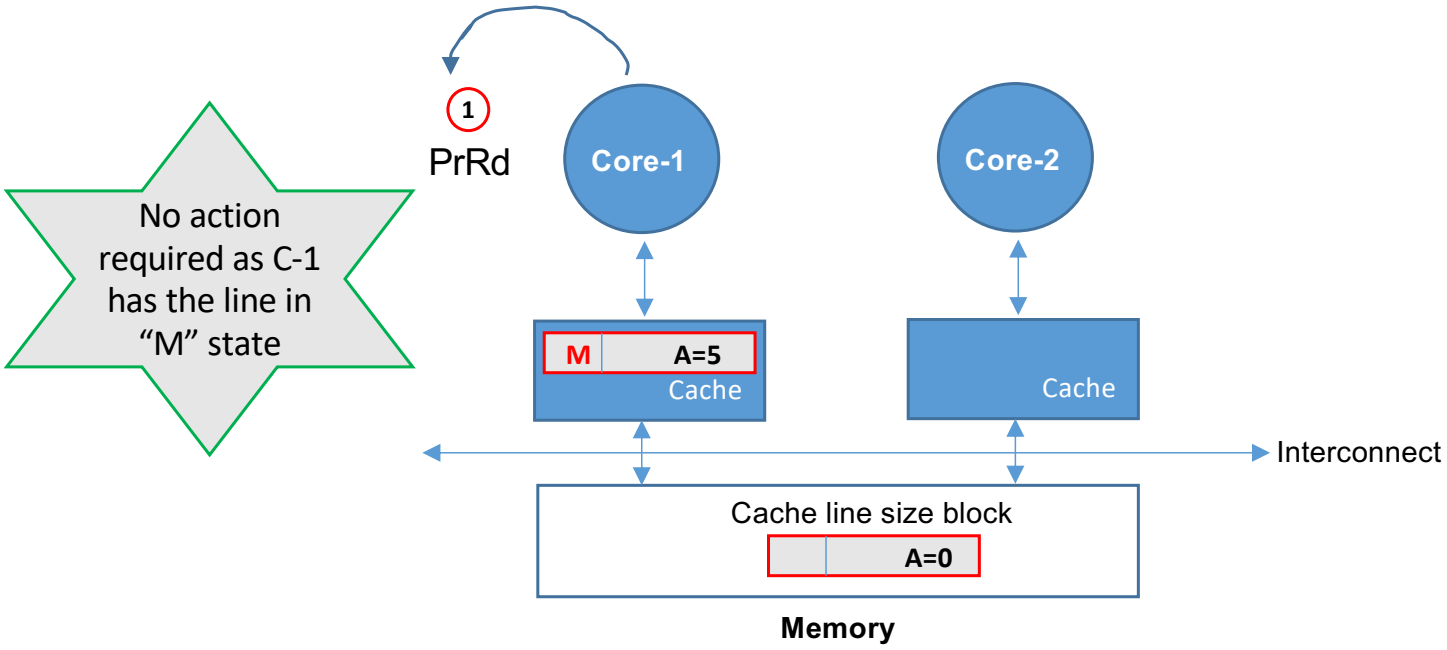
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

Snooping using Write-back Caches (2/5)



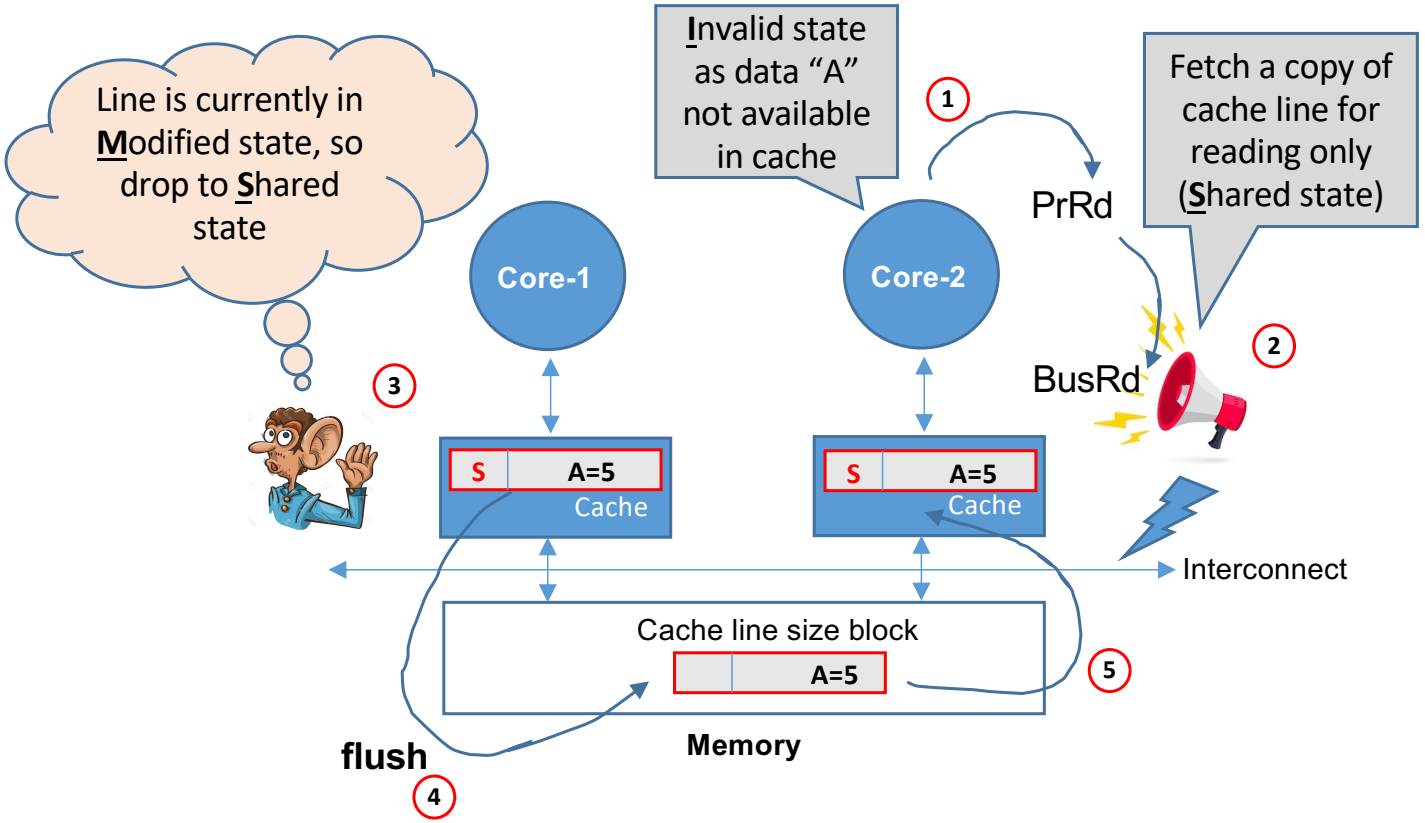
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

Snooping using Write-back Caches (3/5)



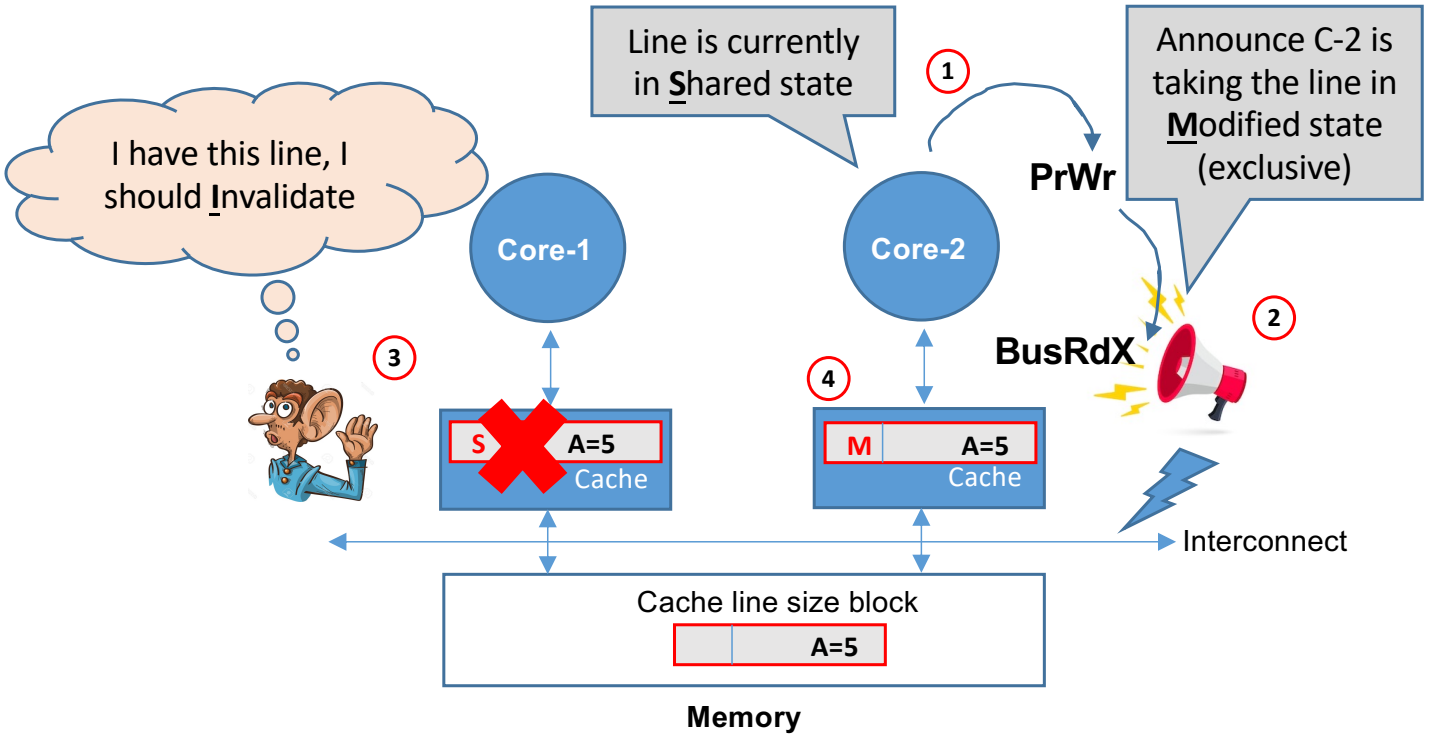
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

Snooping using Write-back Caches (4/5)



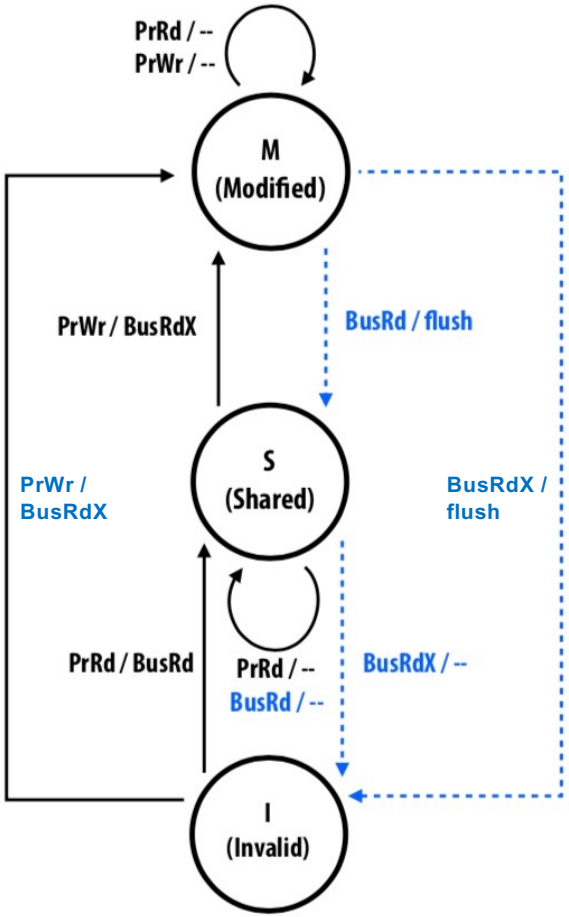
- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

Snooping using Write-back Caches (5/5)



- **MSI** write-back invalidation protocol
  - Invalid
    - Line not available on cache
  - Shared
    - Line in read only mode
  - Modified
    - Line in modified or dirty state

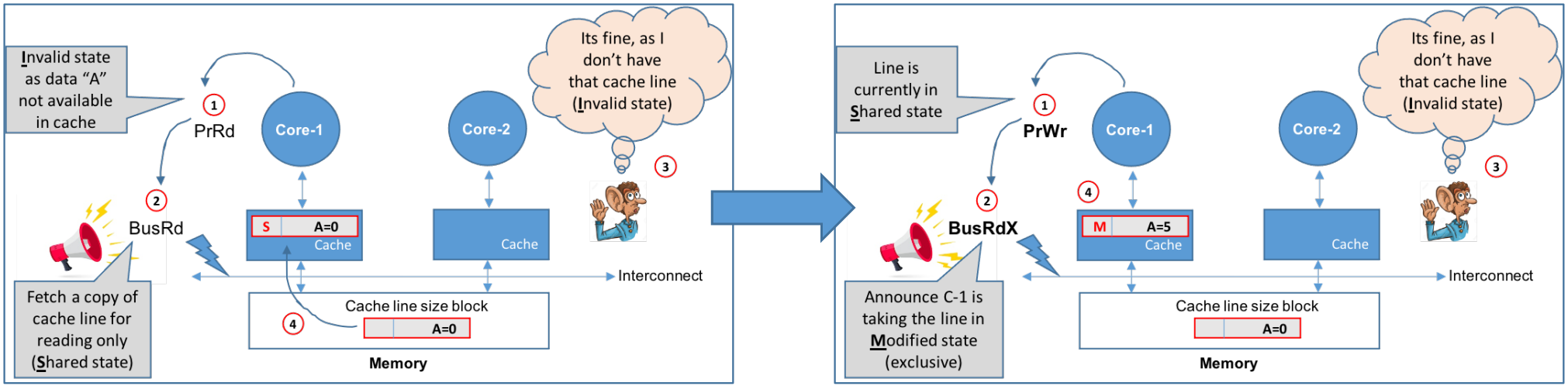
MSI Protocol Summary



- A line in the **M** state can be modified without notifying other caches
- Processor can only write to lines in the **M** state
  - If line is not already exclusive in cache, cache controller must first broadcast a **read-exclusive** transaction to move the line into that state
    - Required even if the line is in **S**hared state
- When other processor's 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 same line

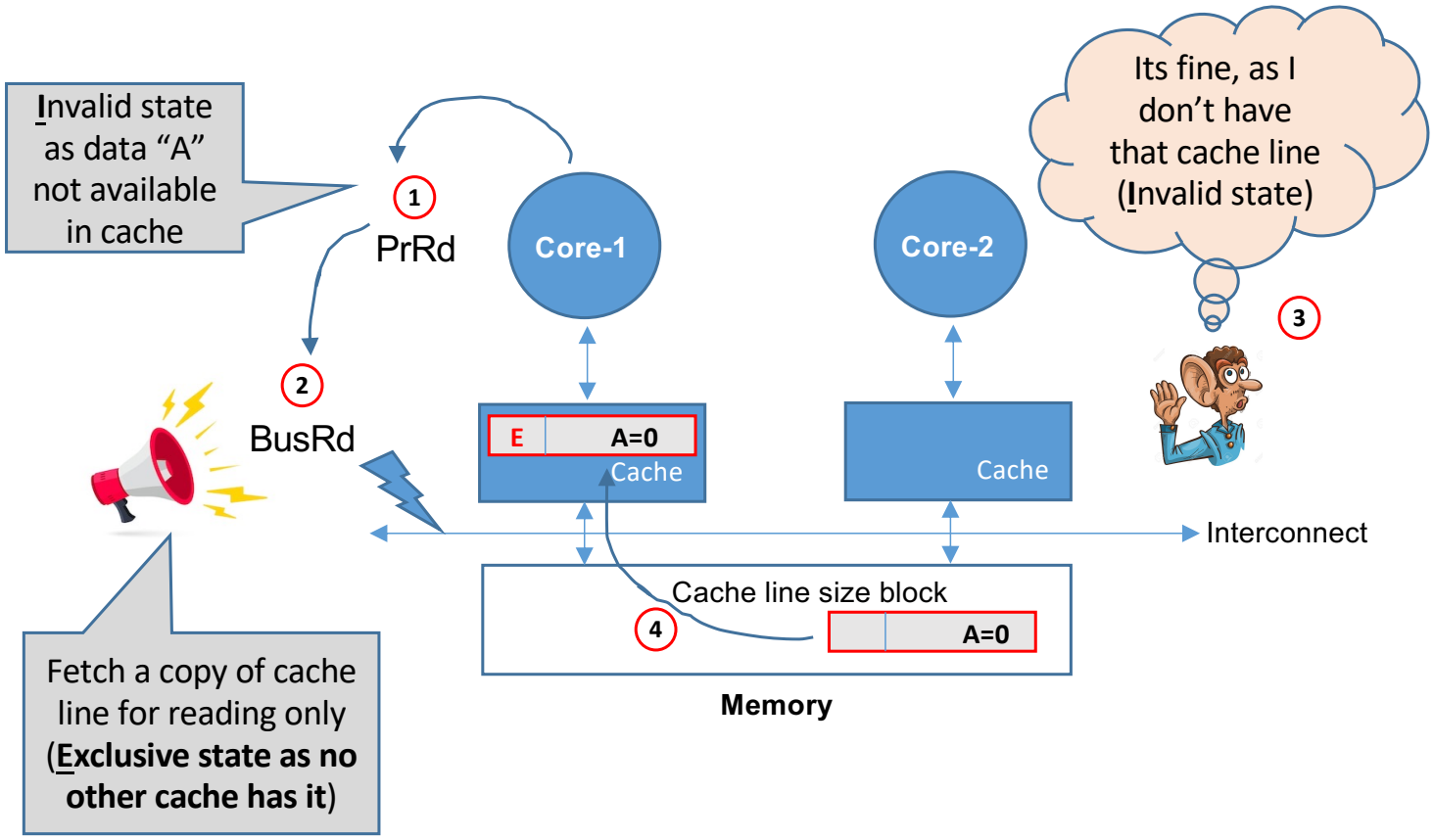
Credits: Fatahalian and Bryant, CMU 15-418/618

What is Wrong with MSI?



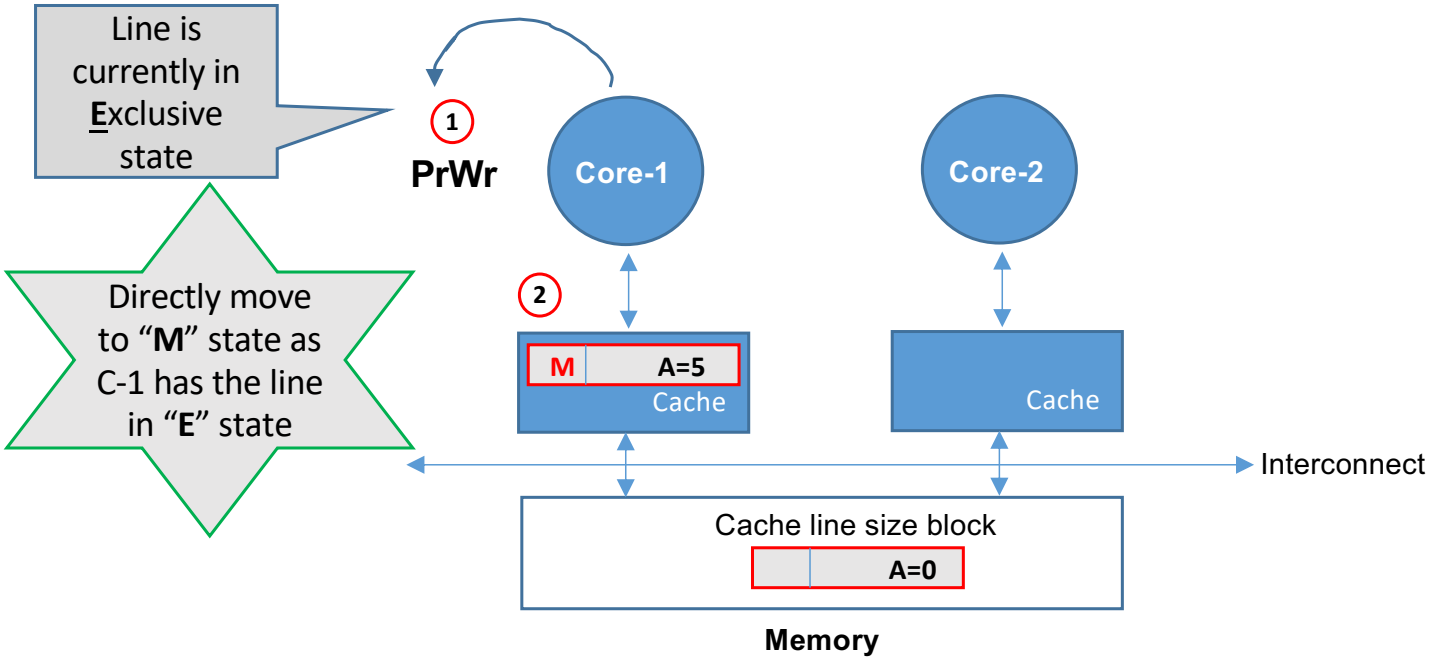
- Core-1 reads a data, and then wishes to modify it
  - The line is only in Core-1's, but it's cache controller still has to perform BusRdX operation for moving the line from "S" state to "M" state
    - Redundant traffic over the interconnect

MESI Protocol (1/3)



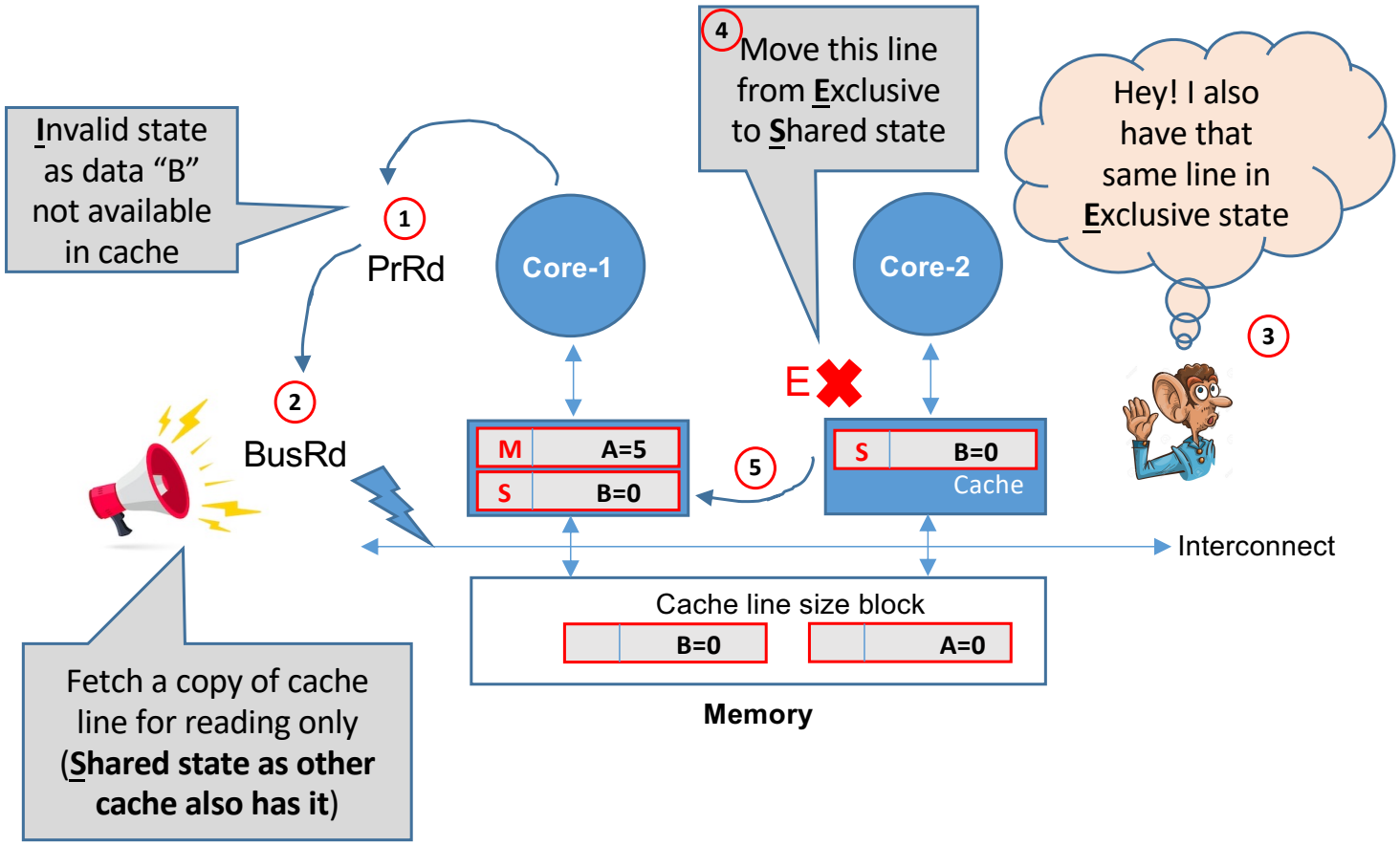
- An additional state **E**
  - Exclusive clean
  - Implies no other cache has a copy of this line

MESI Protocol (2/3)



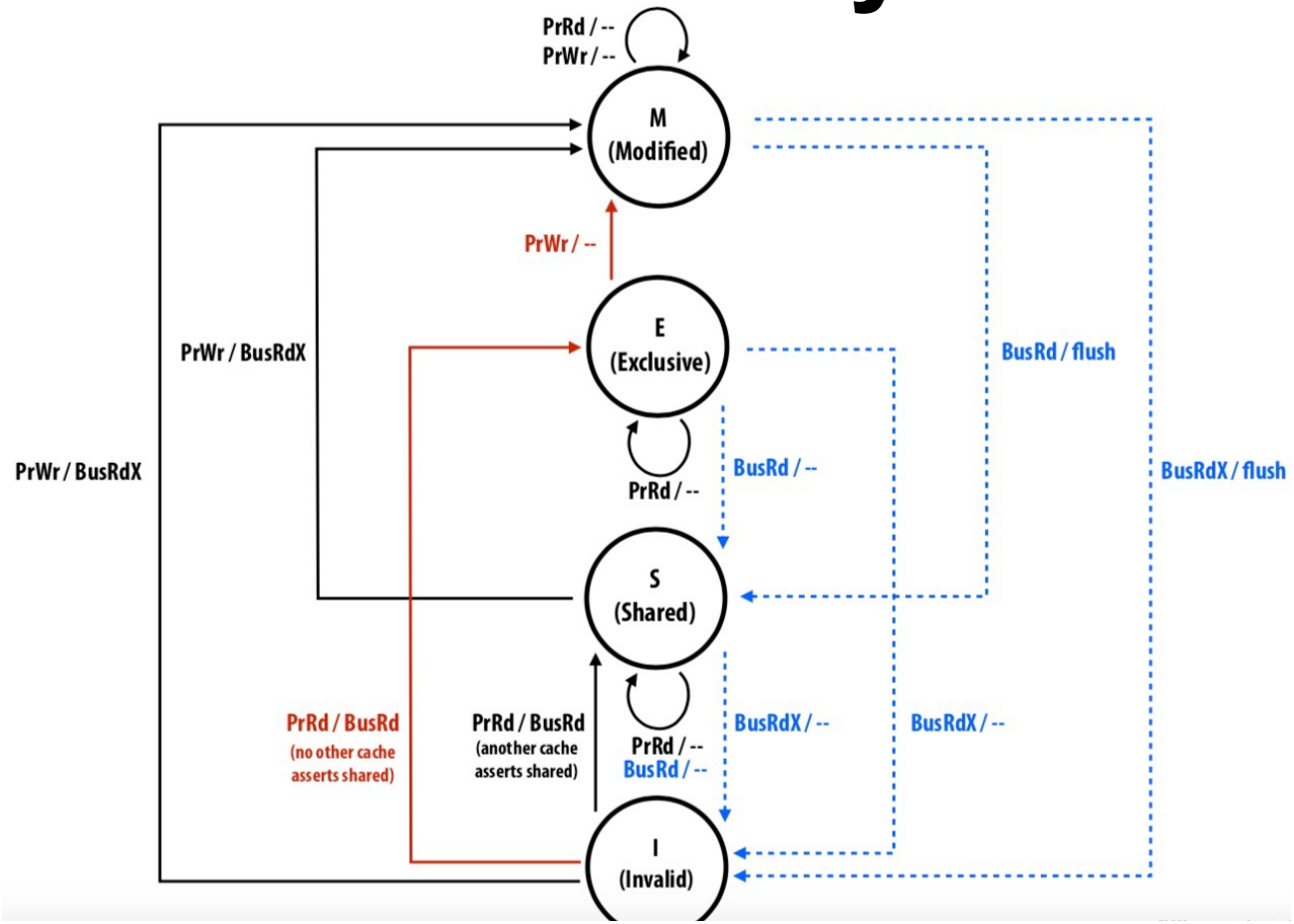
- Moving from **E** to **M** state
  - No action required to be performed on interconnect
  - Present **E** state implies the line is not in any other cache

MESI Protocol (3/3)



- C-1 wants to read a line (B) which C-2 also has (E state)
  - C-1 cannot have it in **E** state, as C-2 also wants to own it for read purpose
  - C-2 drops it from E to S state
  - C-1 has the line in S state

MESI Protocol Summary



Credits: Fatahalian and Bryant, CMU 15-418/618

How Many Cache Misses Below?

```
float A[n][n]; // initialized
float sum=0;
...
for(int row=0; row<n; row++) {
    for(int col=0; col<n; col++) {
        sum += A[row][col];
    }
}
```

v/s

```
float A[n][n]; // initialized
float sum=0;
...
for(int col=0; col<n; col++) {
    for(int row=0; row<n; row++) {
        sum += A[row][col];
    }
}
```

```
float A[n], B[n], C[n]; // initialized

for(int i=0; i<n; i++) {
    C[i] = A[i] + B[i];
}
```

v/s

```
typedef struct Triplet {
    float A;
    float B;
    float C;
} Triplet;

Triplet T[n]; // initialized

for(int i=0; i<n; i++) {
    T[i].C = T[i].A + T[i].B;
}
```

- Assume n is 32, cache line size is 64 bytes, and single precision floating point variable

Next Lecture

- False sharing