Parallel Programming

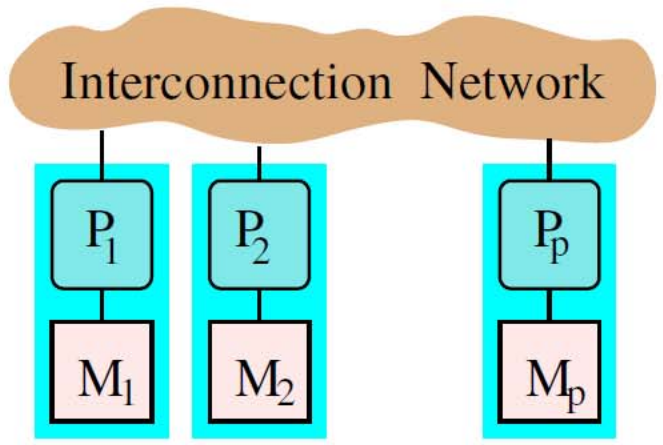
Part 2:
Shared Memory Parallel Architectures

foils by C. Kessler, T. Fahringer

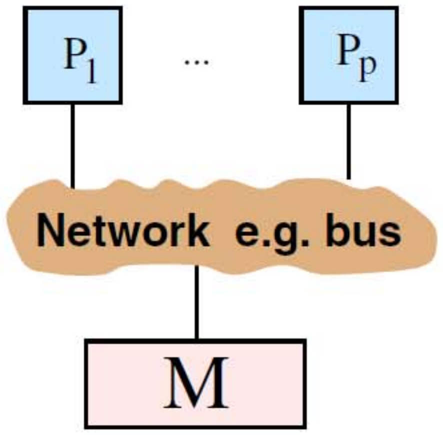
Outline

- Memory hierarchy
- Consistency issues and coherence protocols
- Performance issues, e.g. false sharing
- Optimizations for data locality

Distributed Memory vs. Shared Memory



Distributed memory system

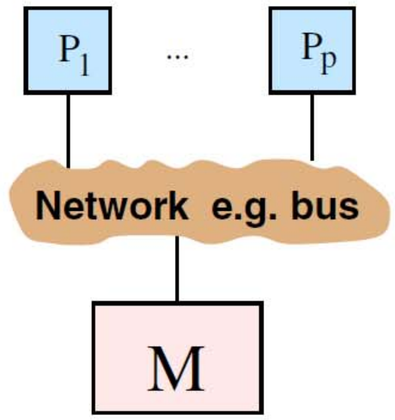


Shared memory system

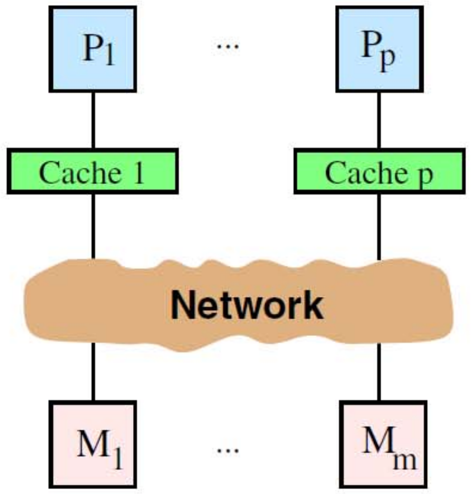
Shared Memory – Variants

- Single shared memory module (UMA) with identical costs for accessing a memory location quickly becomes a performance bottleneck.
- Often implemented with caches to leverage access locality
- Can even be realized on top of physically distributed memory system (NUMA – non-uniform memory access)

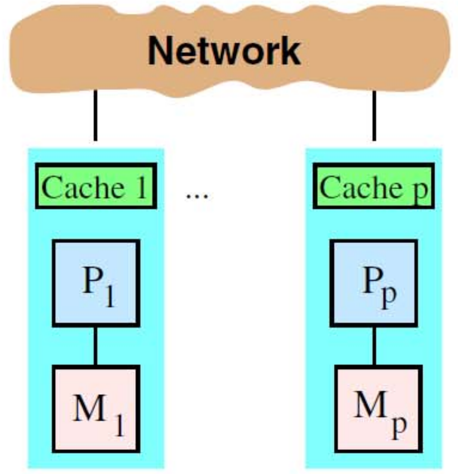
SMP (Symmetric Multiprocessing)



SMP with caches



Distributed shared memory (DSM / NUMA)



Cache

Cache = <u>small</u>, <u>fast</u> memory (SRAM) between processor and main memory, today typically on-chip

- contains copies of main memory words
  - cache hit = accessed word already in cache, get it fast.
  - cache miss = not in cache, load from main memory (slower)
- **Cache line** holds a copy of a **block** of adjacent memory words
  - size: from 16 bytes upwards, can differ for cache levels
  - currently, typically 64 bytes (8 words 64 bit each)

In the following, we mainly refer to **data cache**

- Other cache types in a processor: instruction cache, TLB

Cache (cont.)

Mapping memory blocks → cache lines:

- direct mapped: $\forall j\ \exists! i:\ B_j \mapsto C_i,$
- fully-associative: any memory address may be placed in any cache line
- n-way set-associative
  - caches with n cache lines per set.
  - every memory address maps to a specific set but can be placed in any of the n cache lines of this set.

– modern processors use direct-mapped, 2-way set-associative or 4-way set associative caches.

Cache (cont.)

- Cache-based systems profit from
  - spatial access locality
    - access also other data in same cache line
  - temporal access locality
    - access same words in cache line multiple times

- HW-controlled cache line replacement (which cache line) (for fully and set-associative caches)
  - E.g., LRU – Least Recently Used (usually, approximations) → dynamic adaptivity of cache contents

- Suitable for applications with high (also dynamic) data locality

Classification of Cache Misses

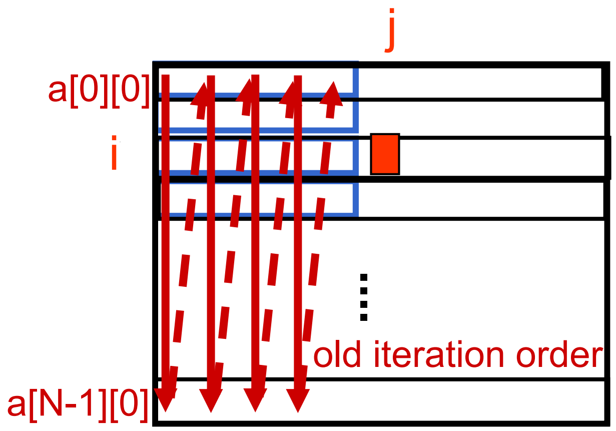
- **Cold miss** – data was never in cache for this process (first access in program execution)
  - Usually unavoidable
- **Capacity miss** – data was evicted earlier when cache capacity was exceeded (for fully associative caches)
  - Might be (partly) avoided by redesigning the programs' algorithm to improve access locality.
- **Associativity miss** – data was evicted earlier because of cache set conflict (for set-associative and direct-mapped caches)

Optimizing Programs for Improved Access Locality

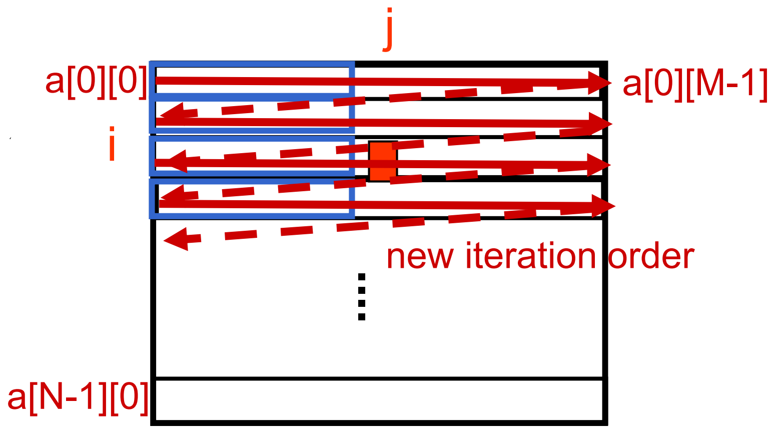
- **Example:**
  Loop Interchange

```
for (j=0;  j<M;  j++)
  for (i=0;  i<N;  i++)
     a[ i ][ j ] = 0.0 ;
```

```
for (i=0;  i<N;  i++)
  for (j=0;  j<M;  j++)
     a[ i ][ j ] = 0.0 ;
```

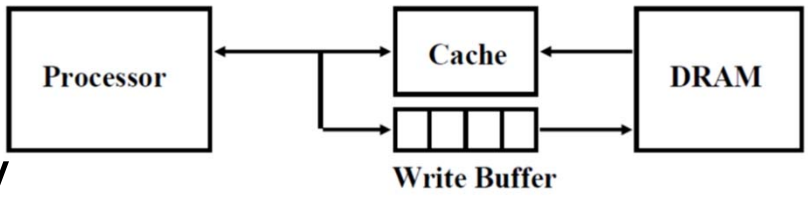


row-wise storage of 2D-arrays in C, Java



– Can improve spatial locality of memory accesses (fewer cache misses).

Caches: Memory Update Strategies



Write-through

+ write both cache and memory

+ consistency (data identical in cache and memory)

– slow, write stall (→ write buffer)

Write-back

+ update only cache

+ write back to memory only when replacing cache line

+ write only if modified, marked by „dirty“ bit for each C_i

– not consistent,

DMA access (I/O, other procs) may access stale values

→ write back on request (flush) or if cache line is evicted from cache.

Memory Hierarchy Example

Memory hierarchy of the Itanium2 processors in SGI Altix 3700 Bx2

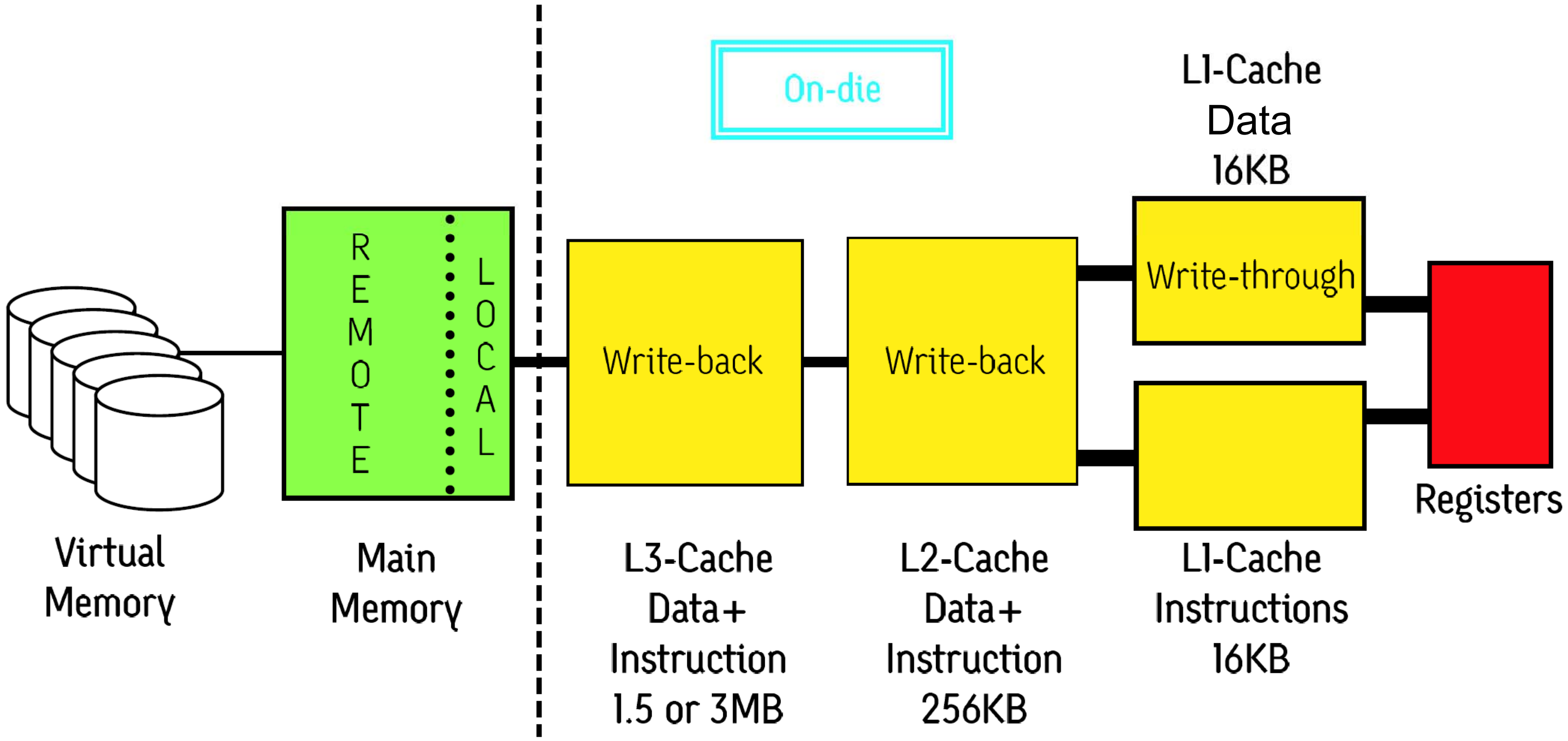


Fig. 2. Itanium 2 memory and cache hierarchy diagram

Source: Michael Woodacre et al.: The SGI Altix^{TM} 3000 Global Shared-Memory Architecture. White Paper, SGI, 2003.

Cache Coherence and Memory Consistency

Caching of (shared) variables leads to consistency problems.

A cache management system is called coherent

> if a read access to a (shared) memory location x reproduces always the value corresponding to the *most recent write access* to x. Cache coherence deals with ordering of writes to a single memory location.
>
> → no access to stale values

A memory system is consistent (at a certain time)

> if all copies of shared variables in the main memory and in the caches are identical. Deals with ordering of reads/writes to all memory locations.

Permanent memory-consistency implies cache-coherence.

Cache Coherence – Formal Definition

What does „most recent write access" to *x* mean?

Formally, 3 conditions must be fulfilled for coherence:

(a) Each processor sees its *own* writes and reads in program order.

*P*1 writes *v* to *x* at time *t*1, P1 reads from *x* at *t*2 > *t*1, no other processor writes to x between *t*1 and *t*2 → read yields *v*

(b) the written value is eventually visible to *all* processors. *P*1 writes *v* to *x* at *t*1, *P*2 reads from *x* at *t*2 > *t*1, no other processor writes to *x* between *t*1 and *t*2, and *t*2 > *t*1 sufficiently large, then *P*2 reads *v*.

(c) All processors see the same total order of all write accesses. *(total store ordering)*

Cache Coherence Protocols

Inconsistencies occur when modifying only the copy of a shared variable in a cache, not in the main memory and all other caches where it is held.

Write-update protocol (word basis)

> At a write access, all other copies in the system must be updated as well. Updating must be finished before the next access.

Write-invalidate protocol (cache line basis)

> Before modifying a copy in a cache, all other copies in the system must be declared as „invalid“.

Most cache-based SMPs use a write-invalidate protocol.

Updating / invalidating straightforward in bus-based systems (bus-snooping), otherwise, a directory mechanism is necessary

Details: Write-Invalidate Protocol

Implementation: multiple-reader-single-writer sharing

At any time, a data item (usually, entire cache line) may either be:

accessed in read-only mode by one or more processors

read and written (exclusive mode) by a single processor

Items in read-only mode can be copied indefinitely to other processors.

Write attempt to read-only mode data *x*:

1. Broadcast invalidation message to all other copies of *x*
2. Await acknowledgements before the write can take place
3. Any processor attempting to access *x* is blocked if a writer exists.
4. Eventually, control is transferred from the writer and other accesses may take place once the update has been sent

→ all accesses to x processed on first-come-first-serve basis.

Achieves sequential consistency.

Write-Invalidate Protocol (cont.)

+ parallelism (multiple readers)

+ updates propagated only when data are read

    + several updates by the same processor can take place before communication is necessary

– Cost of invalidating read-only copies before a write can occur

    + ok if read/write ratio is sufficiently high

    + for small read/write ratio: single reader-single-writer scheme (at most one process gets read-only access at a time)

Write-Update Protocol

Protocol is implemented on the basis of words.

Write *x*:

- done locally + broadcast new value to all who have a copy of *x*

  these update their copies immediately.
- several processors may write the same data item at the same time (multiple-reader-multiple-writer sharing); blocking writes.
- Sequential consistency if broadcasts are totally ordered and blocking

  → all processors agree on the order of updates.

  → the reads between writes are well defined

Read *x*:

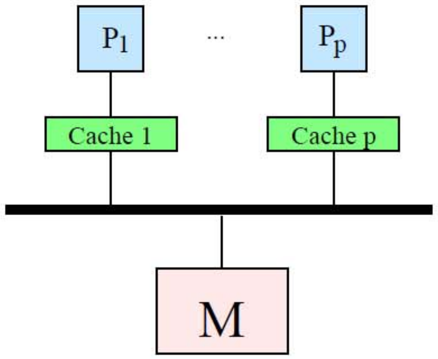
read local copy of *x*, no need for communication.

→ multiple readers

+ Reads are cheap

– totally ordered broadcast protocols quite expensive to implement

Bus-Snooping



For bus-based SMP with caches and write-through strategy.

All relevant memory accesses go via the central bus.

Cache-controller of each processor listens to addresses on the bus:

write access to main memory is recognized
and committed to the own cache.

– bus is performance bottleneck $\rightarrow$ poor scalability

Write-back invalidation protocol (MSI-protocol)

MSI protocol applied for write-back and write-invalidate caches and uses bus snooping technique.

A block held in cache has one of 3 states:

M (modified)

only this cache entry is valid, all other copies + MM location are not.

S (shared)

cached on one or more processors, all copies are valid.

I (invalid)

this cache entry contains invalid values.

MSI-Protocol: State Transitions

State transitions:

triggered by observed bus operations and local processor reads/writes

Bus read (BusRd)

Place read miss on bus

Bus read exclusive (BusRdX)

Place write miss on bus

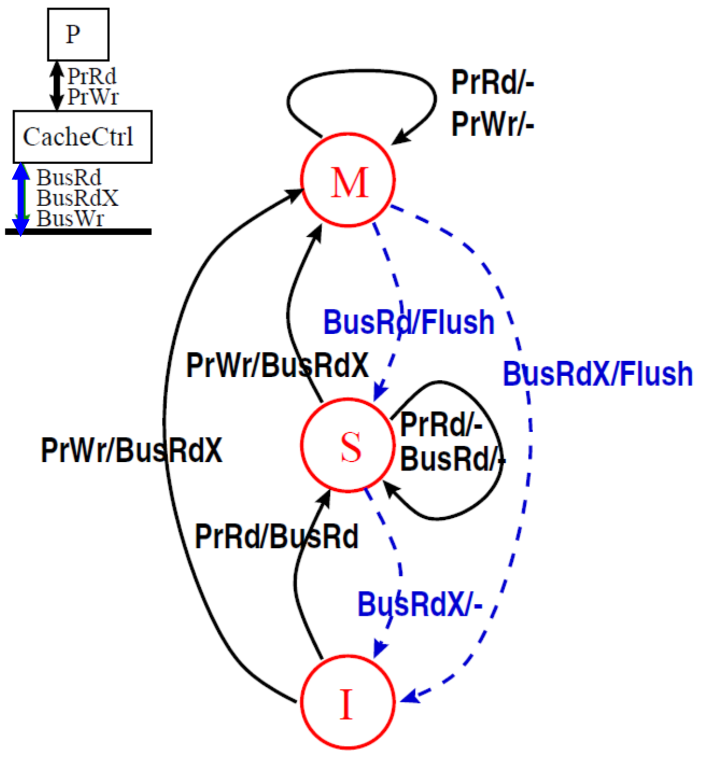
→ must invalidate other copies

Write back (BusWr), due to replacement

→ Flush

Processor reads (PrRd)

Processor writes (PrWr)



Processor operation / Cache controller operation

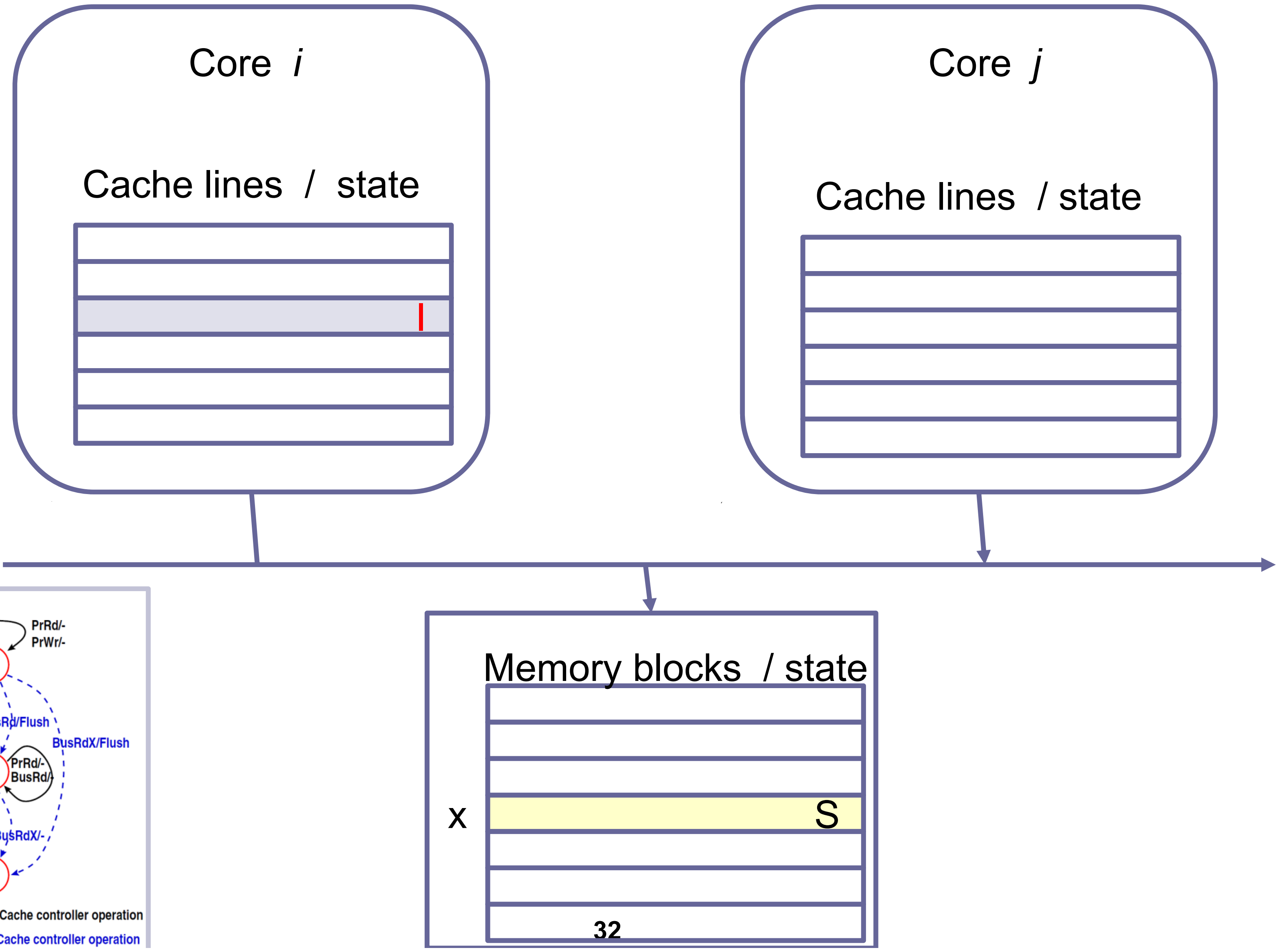
Observed operation / Cache controller operation

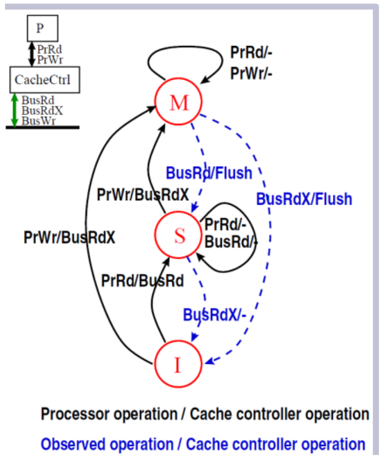
Flush = desired value put on the bus

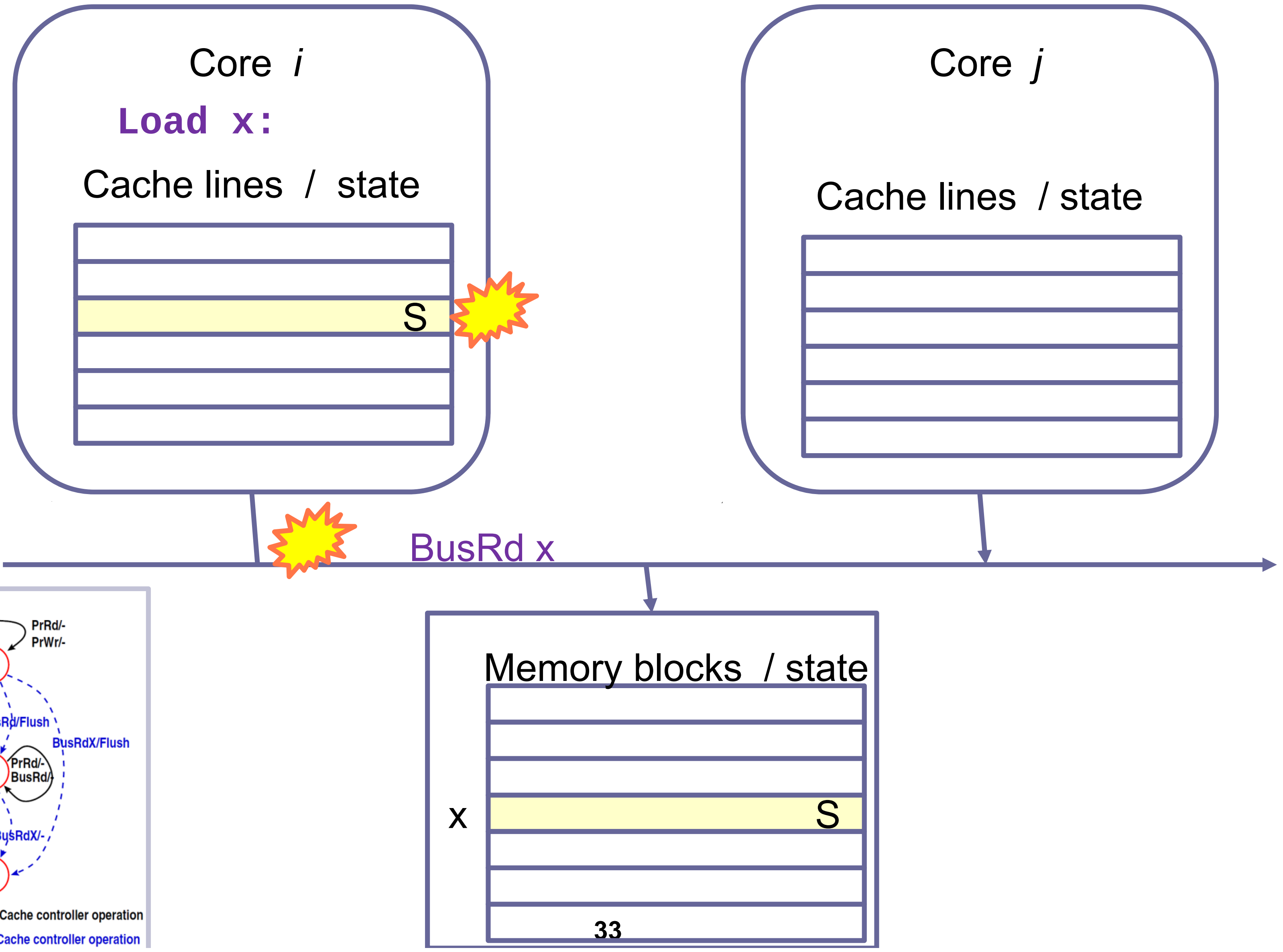
Missing edges - no change of state

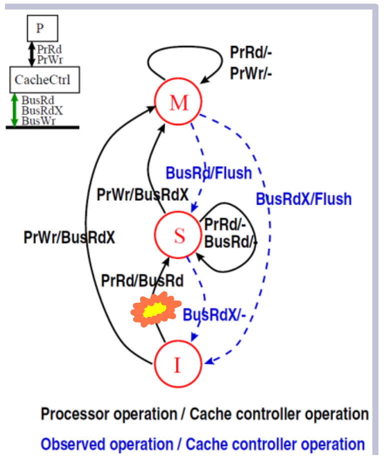
Example

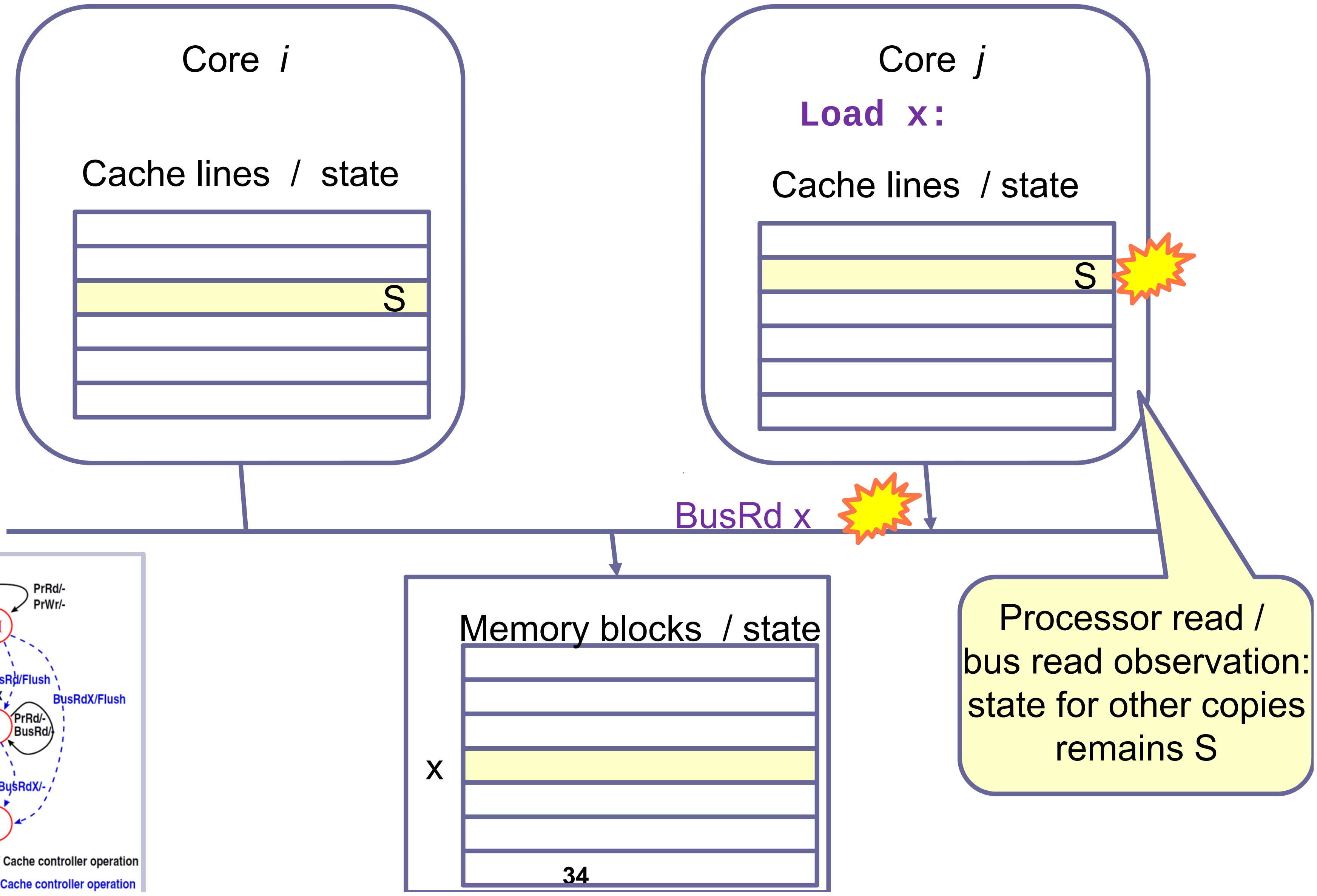
Initial situation…



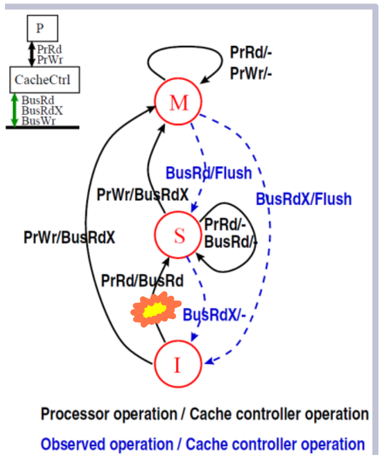



Core i
Load x:
Cache lines / state
S
Core j
Cache lines / state
BusRd x
Memory blocks / state
x
S

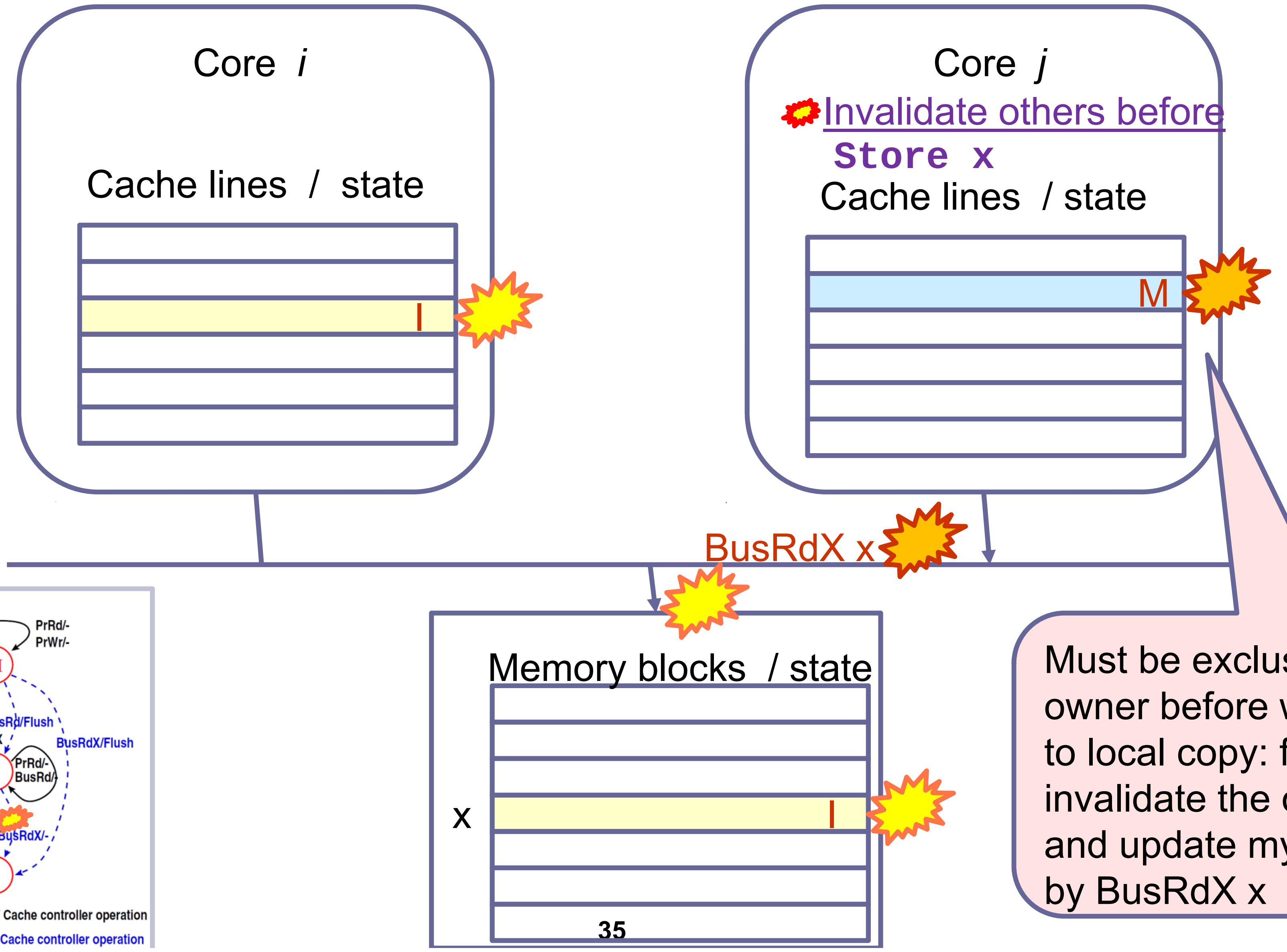

P
PrRd
PrWr
CacheCtrl
BusRd
BusRdX
BusWr
M
PrRd/-
PrWr/-
BusRd/Flush
PrWr/BusRdX
BusRdX/Flush
PrWr/BusRdX
S
PrRd/-
BusRd/-
PrRd/BusRd
BusRdX/-
I
Processor operation / Cache controller operation
Observed operation / Cache controller operation



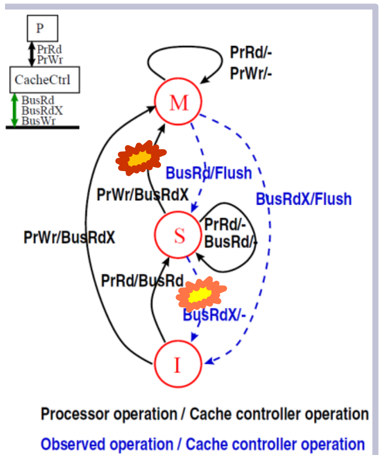
Processor read / bus read observation: state for other copies remains S



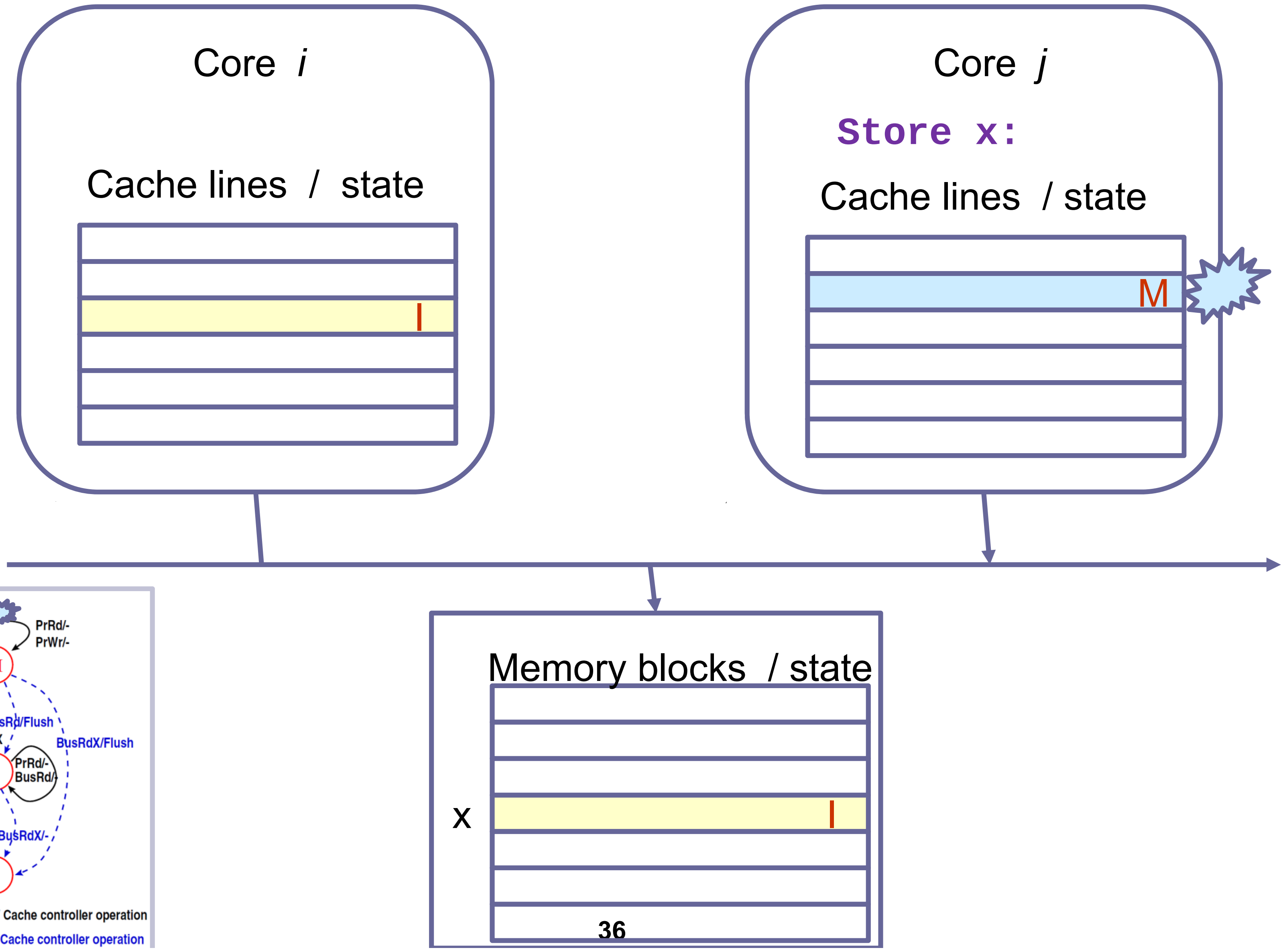
Write requires invalidation

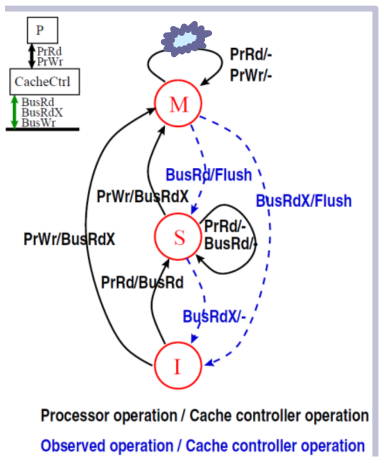


Must be exclusive owner before writing to local copy: first invalidate the others and update my copy, by BusRdX x

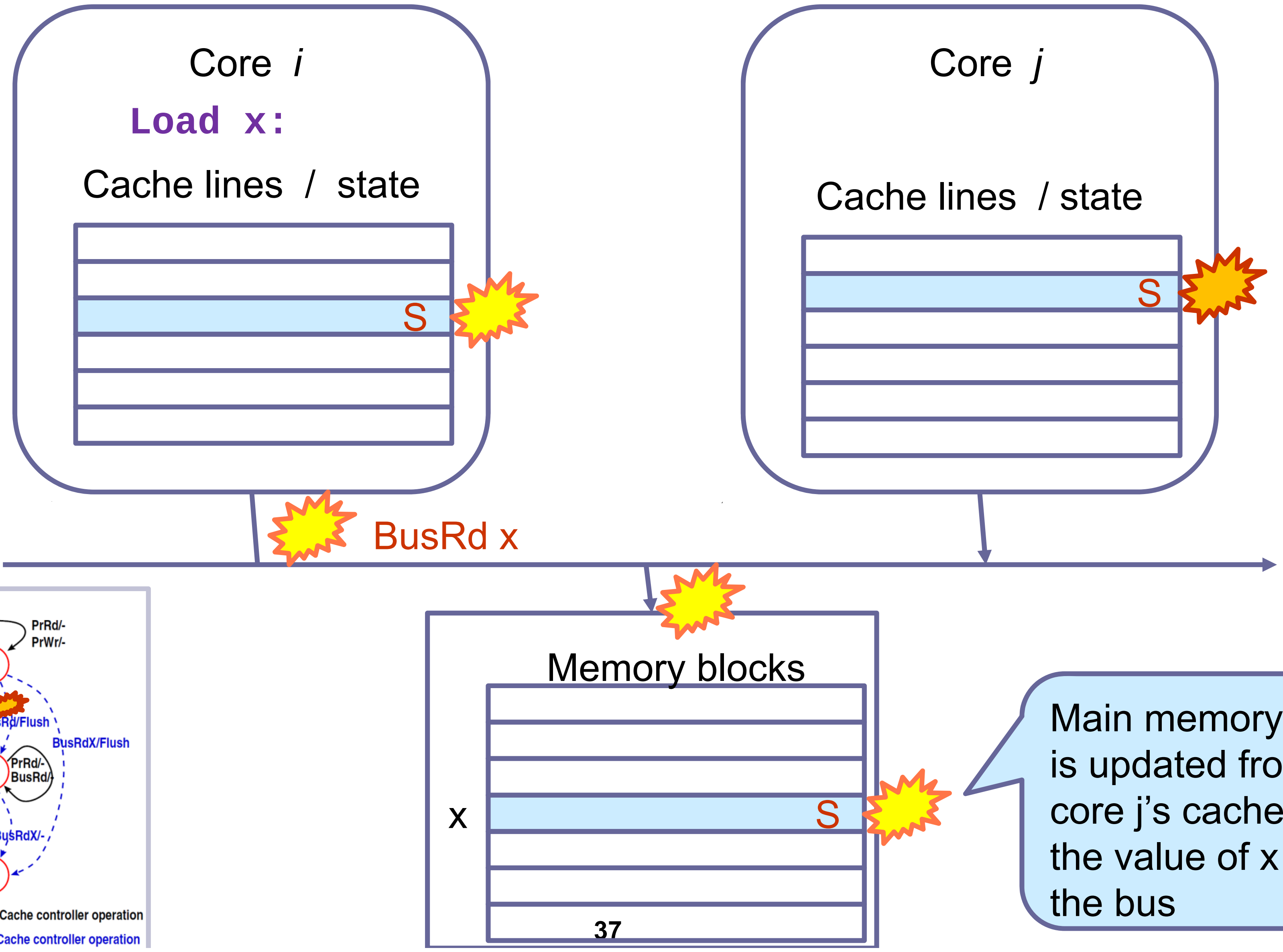


Write requires invalidation

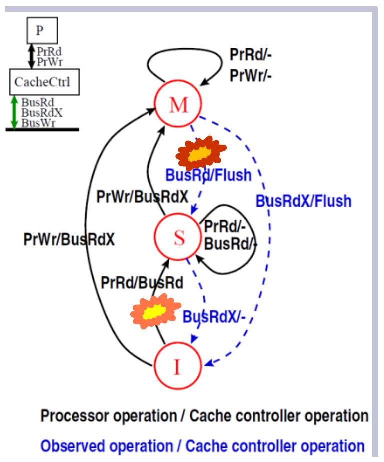




Read triggers updating

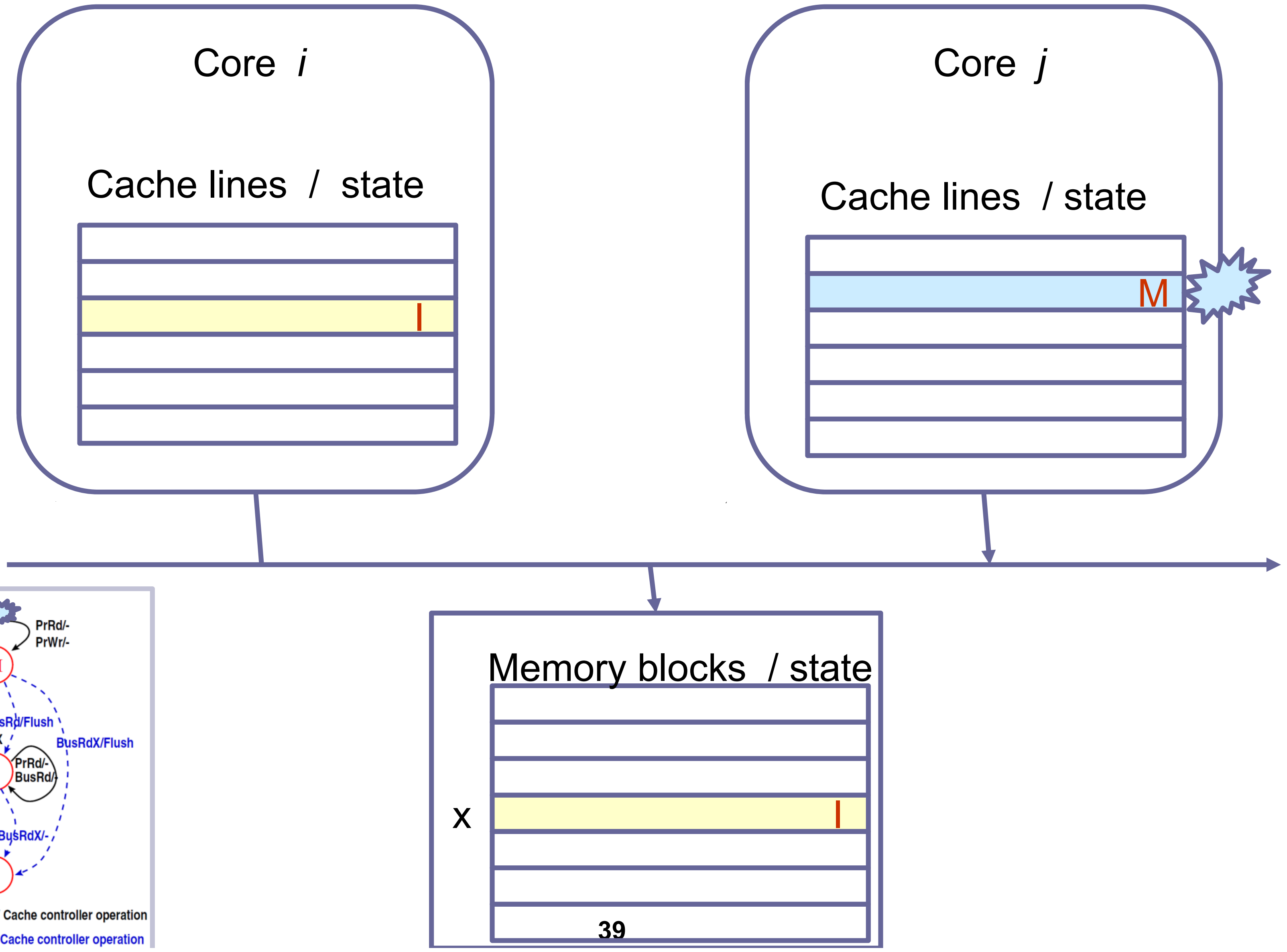


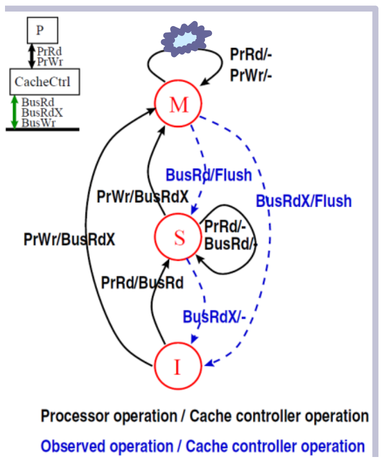
Main memory copy is updated from core j's cache when the value of x is on the bus



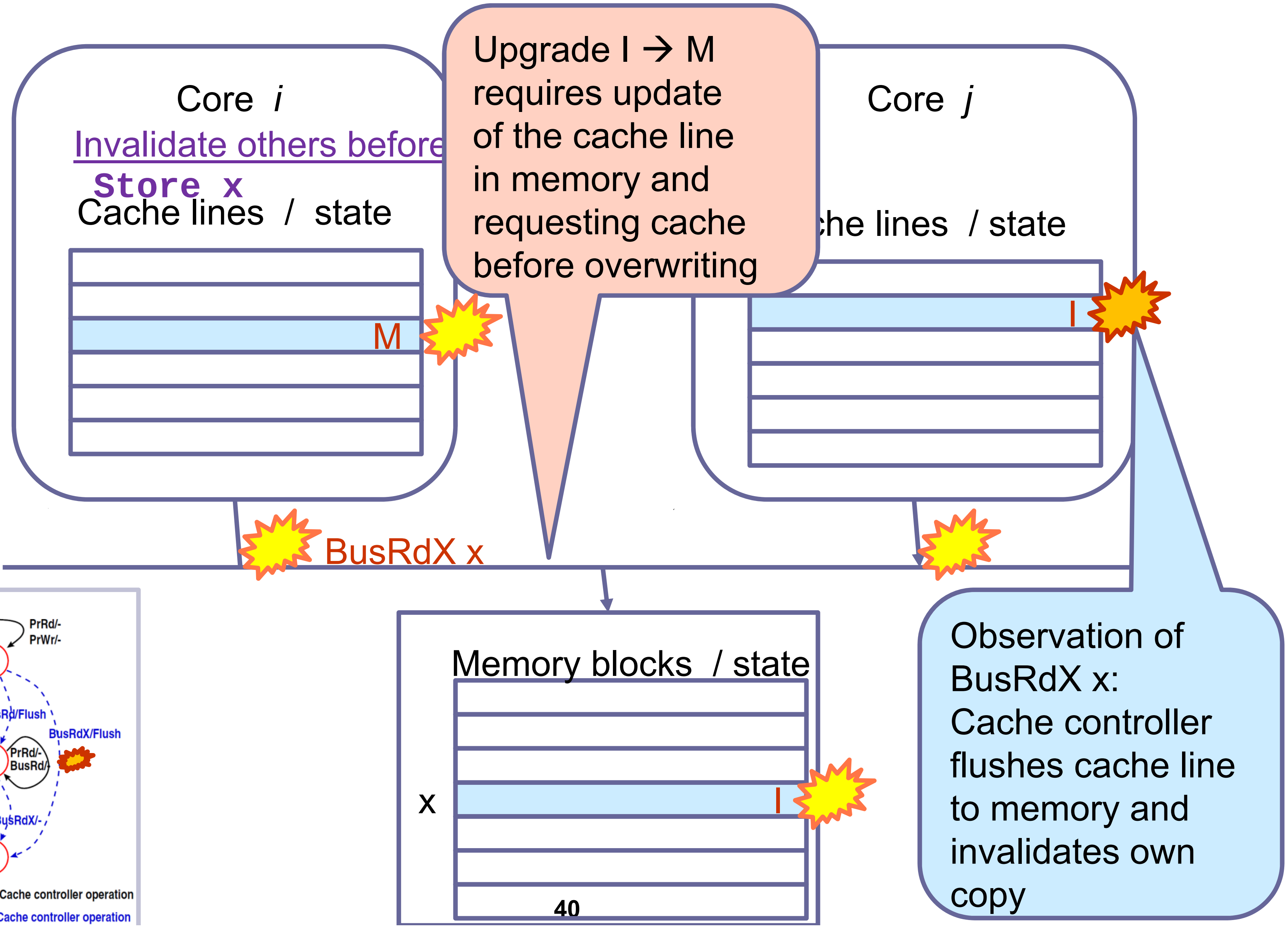
Another MSI Example

Another scenario: Core *j* has written…



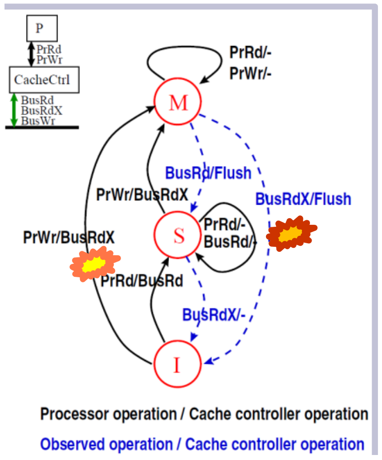


Now Core *i* wants to write…

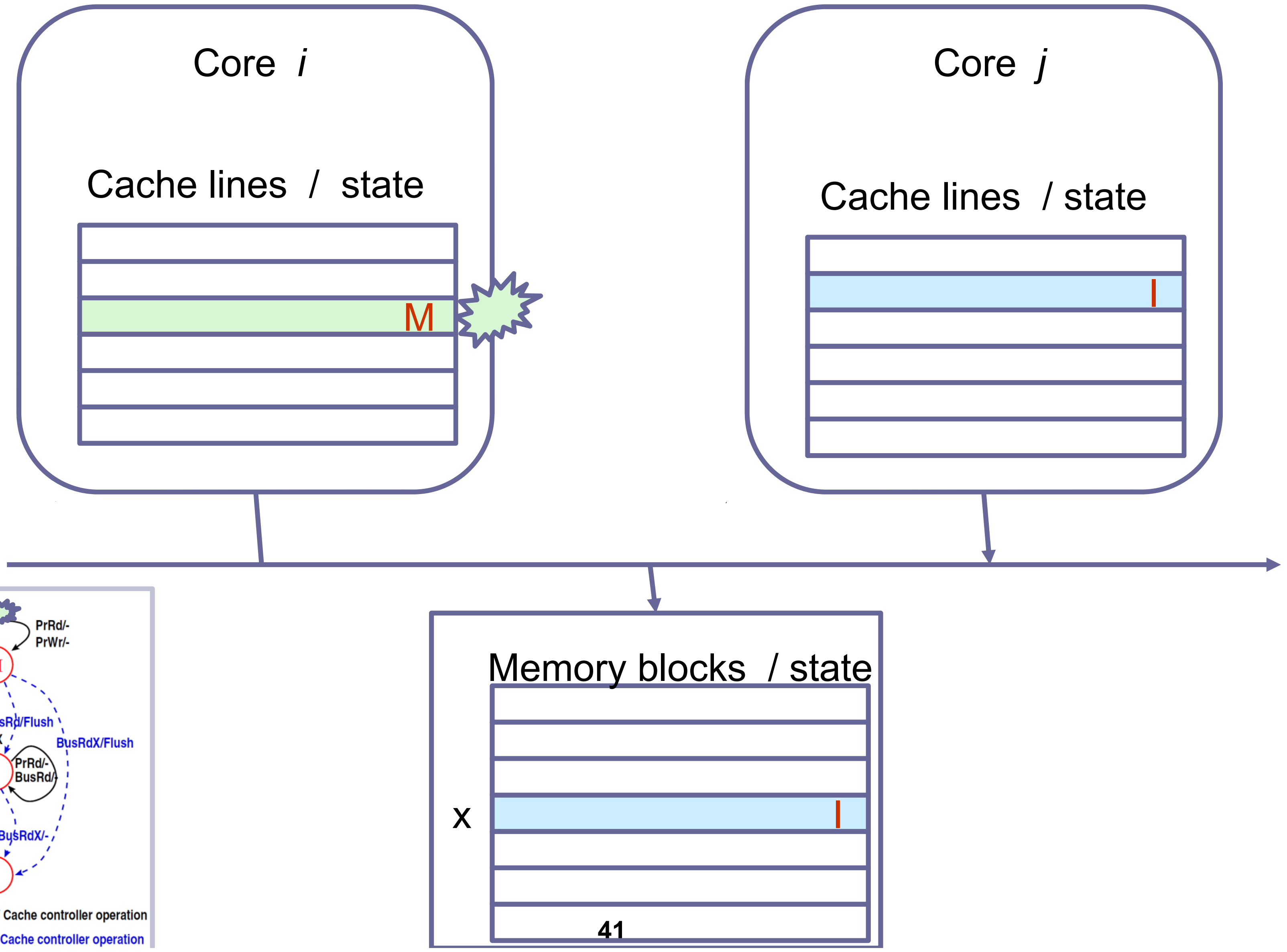


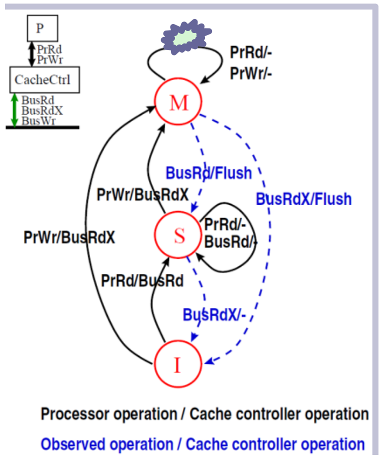
Upgrade I → M requires update of the cache line in memory and requesting cache before overwriting

Observation of BusRdX x: Cache controller flushes cache line to memory and invalidates own copy



Now Core *i* can write…





MESI-Protocol

E-state helps to reduce bus traffic in sequential computations where data is **not shared**.

MSI protocol:

2 bus operations (BusRd, BusRdX) required

if a processor first reads (→ *S*), then writes (→ *M*) a memory location,

even if no other processor works on this data.

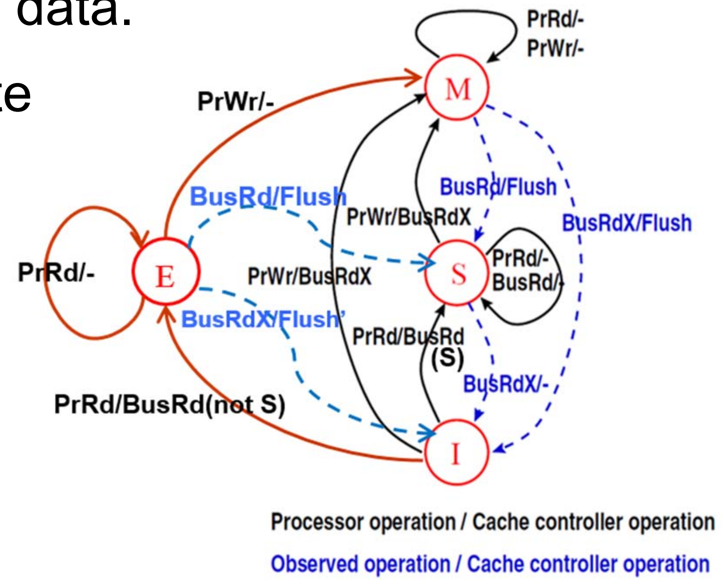
→ generalization to MESI-protocol with new state

E (exclusive)

no other cache has a copy of this block,

this copy is not modified and same data

in MM.



Modifications in MSI-protocol:

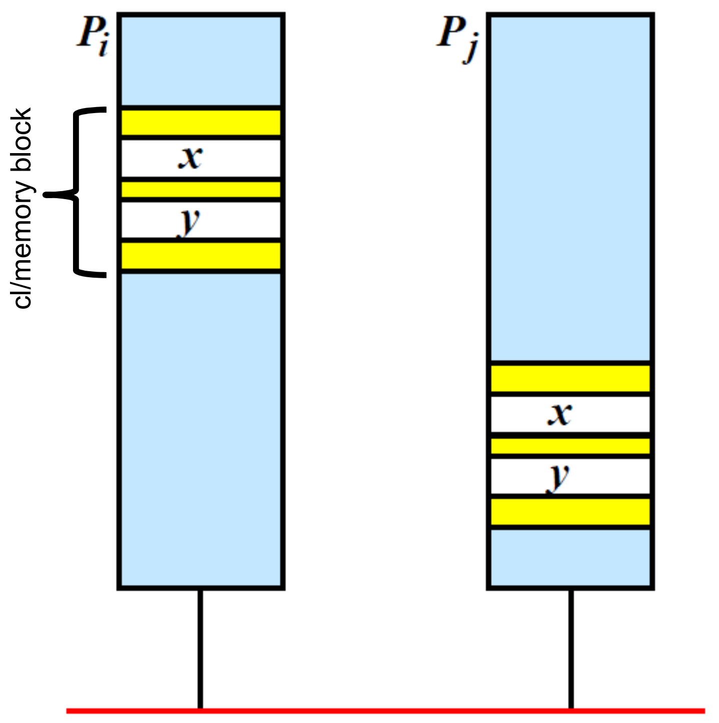
+ PrRd to a <u>non-cached</u> address (BusRd): → *E* (not *S*)

+ PrWr to *E*-address: local change to *M*, write (**no bus operation**)

+ read access from another processor to *E*-address (BusRd/Flush): → *S*

MESI supported by Intel Pentium, MIPS R4400, IBM, PowerPC, …

Performance Issue: False Sharing

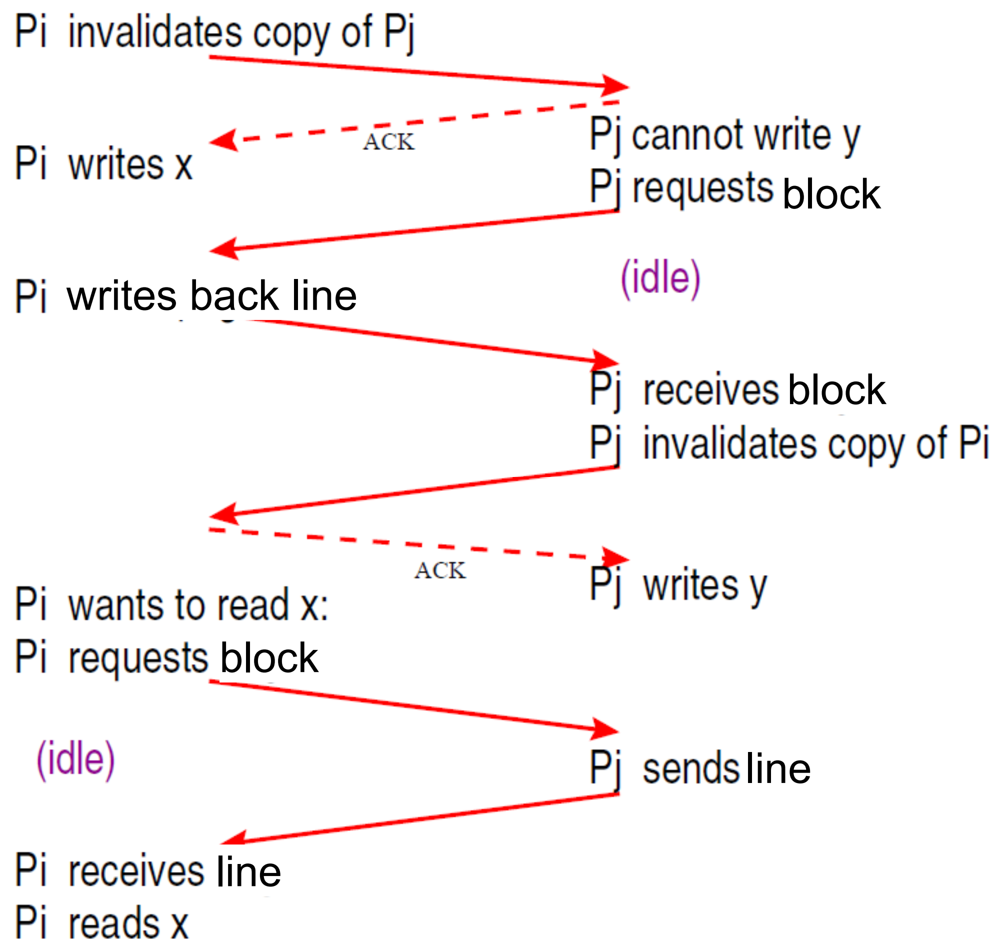


P_i

```
...
*x = ...
... = x
```

P_j

```
...
*y = ...
...
```



Invalidation causes an extra cache miss
Cache lines or memory blocks are the units of sharing
→ sequentialization, thrashing

How to Avoid False Sharing?

- Smaller cache lines
  - false sharing less probable, but
  - more administrative effort
- Programmer or compiler gives hints for data placement
  - padding, data privatization
- Time slices for exclusive use of CL:

  each line stays for $\geq d$ time units at one processor

How to reduce performance penalty of false sharing?

- Use weaker consistency models
  - programming more complicated and error-prone

Shared Memory Consistency Models

- Problem of memory inconsistencies when memory-access order differs from the program execution-order.
- Improve performance at the cost of error-prone programming
- different memory consistency models:

  Strict consistency

  Sequential consistency

  Causal consistency

  Superstep consistency

  "PRAM" consistency

  Weak consistency

  Release consistency / Barrier consistency

  Lazy Release consistency

  Entry consistency

  Others (processor consistency, total/partial store ordering etc.)

[Culler et al.'98, Ch. 9.1], [Gharachorloo/Adve'96]

Consistency Models: Strict Consistency

Strict consistency: no memory inconsistency

Read (x) returns the value that was most recently (→global time) written to *x*.

Writes must be seen instantaneously by all processors.

realized in classical uniprocessors and SB-PRAM;

in DSM physically impossible without additional synchronization

P_1 ⟵ 3m distance ⟶ P_2

```
t1 : *x = ...
     ...                                   t1 + 1ns: ... = x
```

Transport of *x* from P_1 to P_2 with speed 10*c*

Consistency Models: Sequential Consistency

Sequential consistency [Lamport'79] no memory inconsistency

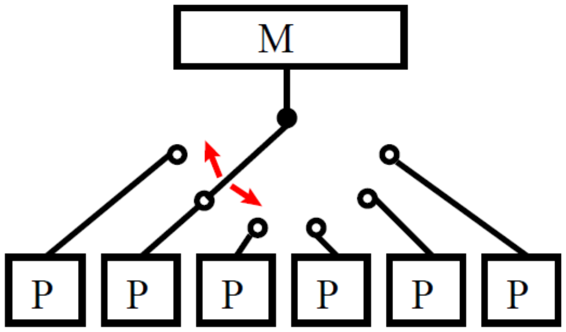
+ all memory accesses are ordered in some sequential order and seen the same way by all processors

+ all read and write accesses of a processor appear in program order

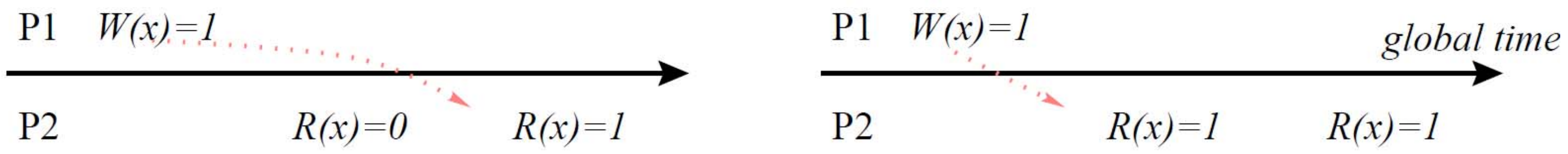
+ otherwise, arbitrary delays possible

Read/writes are atomic.

Acces mode.



Not deterministic:



Consistency Models: Weak Consistency

- memory consistency must be guaranteed by programmer not HW

+ Classification of shared variables (and their accesses):

  synchronization variables (locks, semaphores)

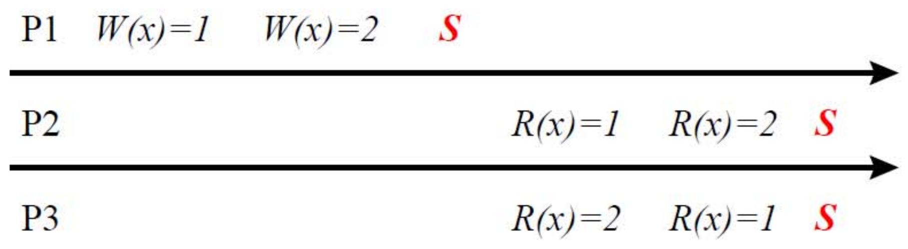
  → always consistent, atomic access

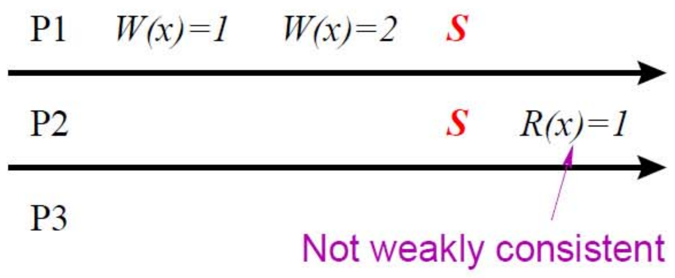
  other shared variables

  → kept consistent by the user, using synchronizations

+ Accesses to synchronisation variables are sequentially consistent

- Accesses to other shared variables may be seen in different order on different processors.



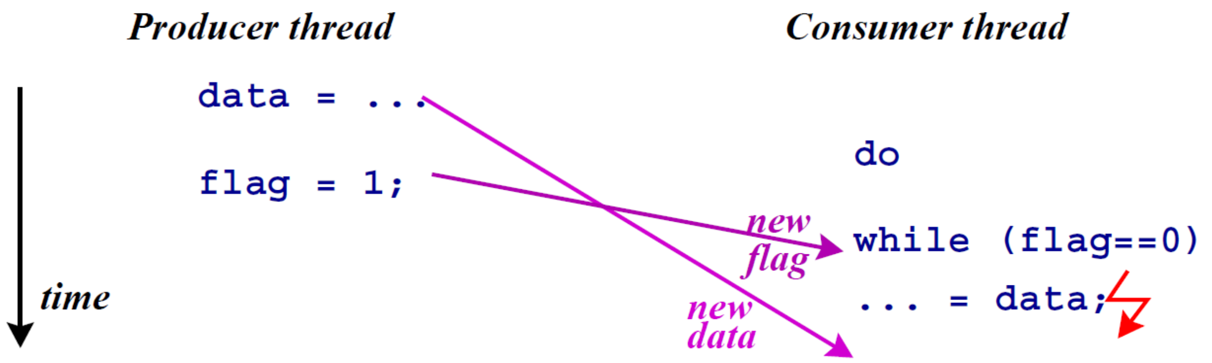


S is a flush operation.

Consistency Models: Weak Consistency in OpenMP

OpenMP implements weak consistency. Inconsistencies may occur due to

+ register allocation
+ compiler optimizations
+ caches with write buffers



Need explicit „memory fence“ to control consistency: flush directive

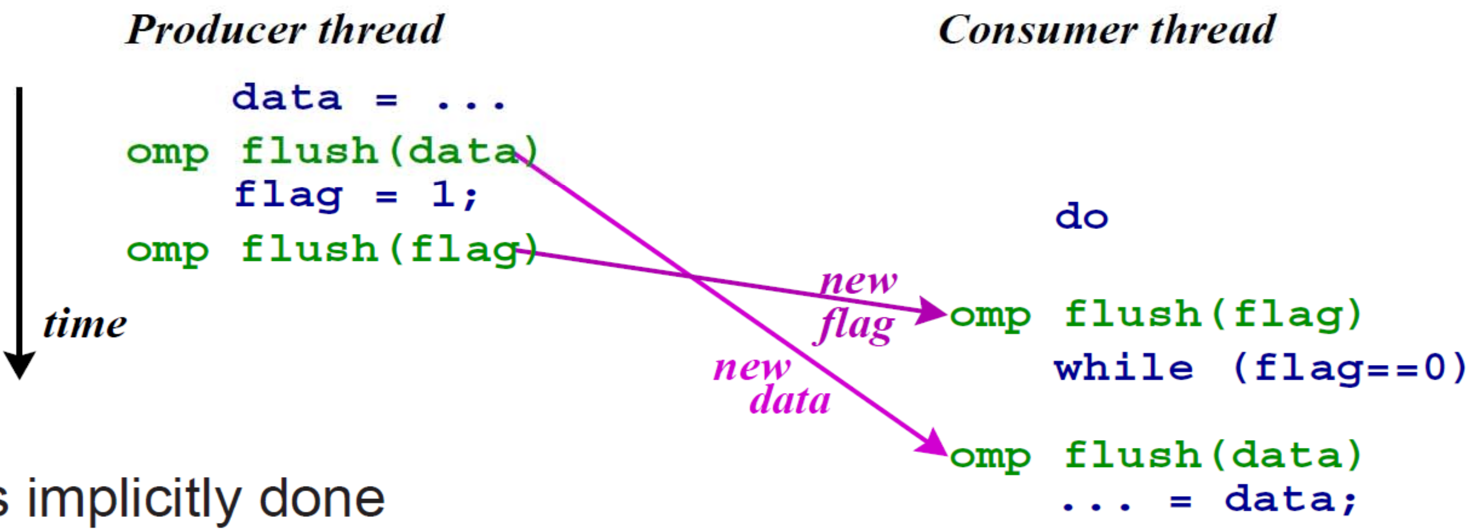
- write back register contents to memory
- forbid code moving compiler optimizations
- flush cache write buffers to memory
- re-read flushed values from memory

Consistency Models: Weak Consistency in OpenMP (cont.)

`!$omp flush (` ***shvarlist*** `)`

creates for the executing processor a consistent memory view
for the shared variables in ***shvarlist***.

If no parameter: create consistency of all accessible shared variables.



```
Producer thread:
    data = ...
omp flush(data)
    flag = 1;
omp flush(flag)

Consumer thread:
    do
omp flush(flag)
    while (flag==0)
omp flush(data)
    ... = data;
```

A flush is implicitly done

at `barrier`, `critical`, `end critical`, `end parallel`,

and at `end do`, `end section`, `end single`

if no `nowait` parameter is given

Summary

- Overview of architectures for shared memory parallel computers
- Caches to reduce latency for data acesses
- Cache coherence to keep data consistent that is stored in caches and main memory
- False sharing and what to do about it
- Memory consistency models.

Further Reading

- D. Culler et al. *Parallel Computer Architecture, a Hardware/Software Approach.* Morgan Kaufmann, 1998.
- J. Hennessy, D. Patterson: *Computer Architecture, a Quantitative Approach,* Second edition (1996) or later. Morgan Kaufmann.
- S. Adve, K. Gharachorloo: Shared memory consistency models: a tutorial. IEEE Computer, 1996.

Sources for Slides (Acknowledgements)

- Christoph Kessler, PELAB / IDA,Linköping University, Sweden
- Thomas Fahringer, UIBK