ADVANCED TOPICS IN CACHES

Slides adapted by: Dr Sparsh Mittal

Exercise on Caches

- For the following access pattern, (i) indicate if each access is a hit or miss. (ii) What is the hit rate? Assume that the cache has 2 sets and is 2-way set-associative.
- Block A maps to set 0,
- B to set 1,
- C to set 0,
- D to set 1,
- E to set 0,
- F to set 1.
- Assume an LRU replacement policy.
-
- Access pattern: ABCBABECADBCAFDBCEA

How to Improve Cache Performance

- Three fundamental goals
- Reducing miss rate
- Reducing miss latency or miss cost
- Reducing hit latency or hit cost

Software Approaches for Higher Hit Rate

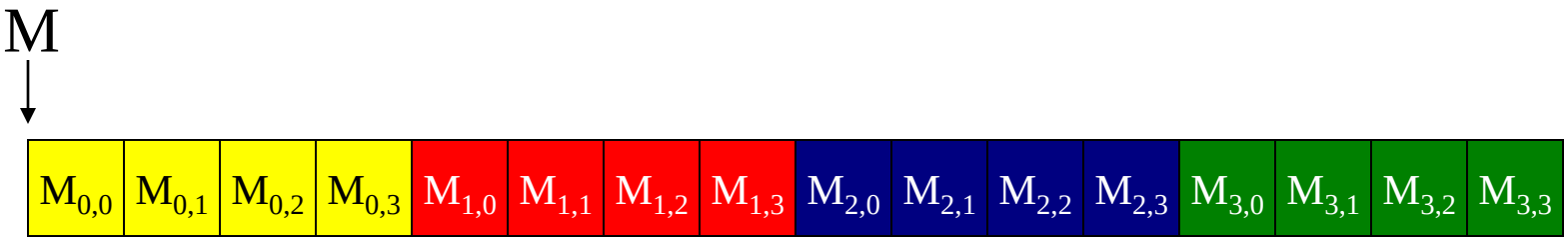
- Restructuring data access patterns
- Restructuring data layout

- Loop interchange
- Data structure separation/merging
- Blocking
- …

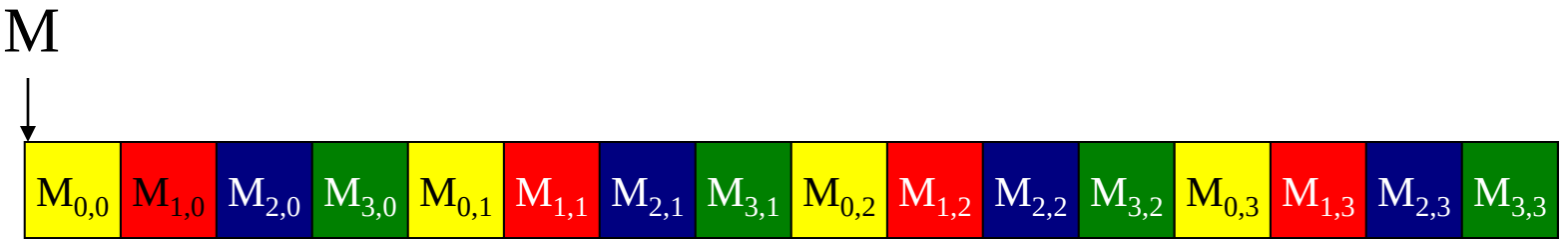
Row & Column-Major Layout

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Row -major



Column -major



Row & Column-Major Layout

- Row-major order is used in C/C++
- Column-major order is used in Fortran, MATLAB, GNU Octave, R and Scilab.

Restructuring Data Access Patterns (I)

- Idea: Restructure data layout or data access patterns
- Example: If column-major
  - x[i+1,j] follows x[i,j] in memory
  - x[i,j+1] is far away from x[i,j]

Poor code

```
for i = 1, rows
    for j = 1, columns
        sum = sum + x[i,j]
```

Better code

```
for j = 1, columns
    for i = 1, rows
        sum = sum + x[i,j]
```

- This is called loop interchange

Restructuring Data Layout (I)

```
struct Node {
    int key;
    char [256] name;
    char [256] school;
    struct Node* next;
}

while (node) {
    if (node→key == input-key) {
        // access other fields of node
    }
    node = node→next;
}
```

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1M nodes) and unique keys
- Why does the code on the left have poor cache hit rate?
  - "**Other fields**" occupy most of the cache line even though rarely accessed!

Restructuring Data Layout (II)

```
struct Node {
    int key;
    struct Node-data* node-data;
    struct Node* next;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
     if (node→key == input-key) {
          // access node→node-data
     }
     node = node→next;
}
```

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Better replacement/insertion policies
  - Non-blocking caches (handling multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches

Enabling High Bandwidth Memories

Multiple Instructions per Cycle

- Can generate multiple cache/memory accesses per cycle
- How do we ensure the cache/memory can handle multiple accesses in the same clock cycle?

- Solutions:
  - true multi-porting
  - virtual multi-porting (time sharing a port)
  - multiple cache copies
  - banking (interleaving)

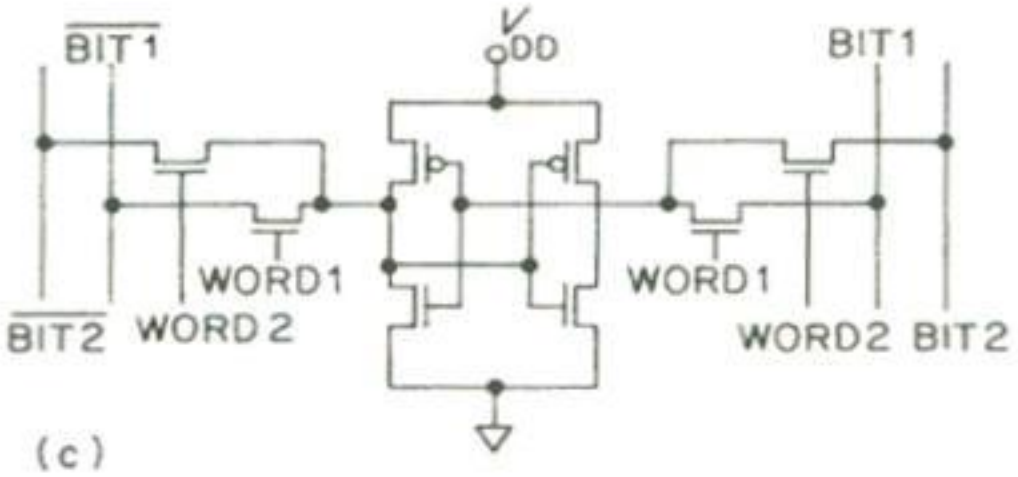
Handling Multiple Accesses per Cycle (I)

- True multiporting
  - Each memory cell has multiple read or write ports

+ Truly concurrent accesses (no conflicts on read accesses)

-- Expensive in terms of latency, power, area

  - What about read and write to the same location at the same time?
    - Peripheral logic needs to handle this

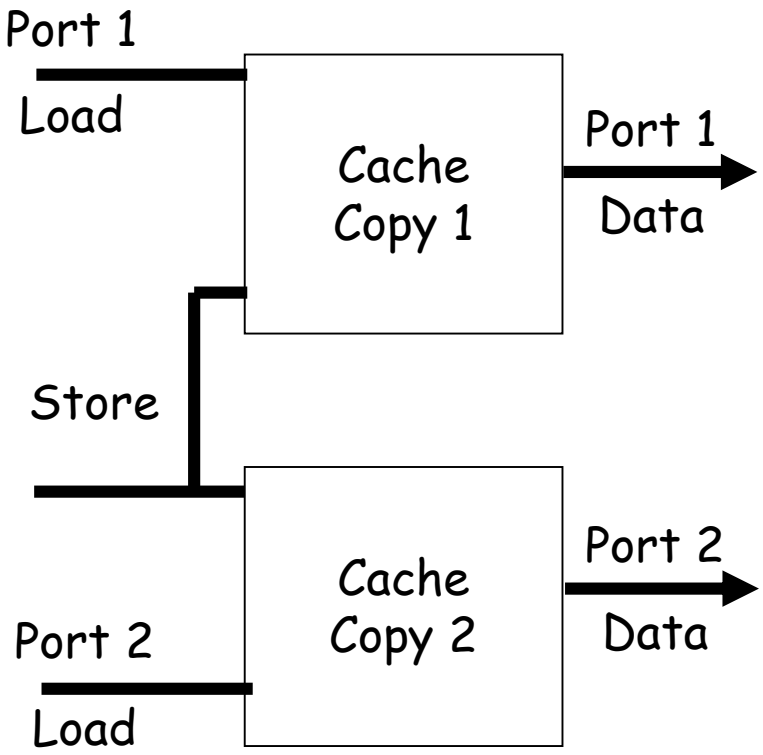


Handling Multiple Accesses per Cycle (II)

- Virtual multiporting
  - Time-share a single port
  - Each access needs to be (significantly) shorter than clock cycle
  - Used in Alpha 21264
  - Is this scalable?

Handling Multiple Accesses per Cycle (III)

- Multiple cache copies
  - Stores update both caches
  - Loads proceed in parallel
- Used in Alpha 21164
- Scalability?
  - Store operations form a bottleneck
  - Area proportional to “ports”



Handling Multiple Accesses per Cycle (III)

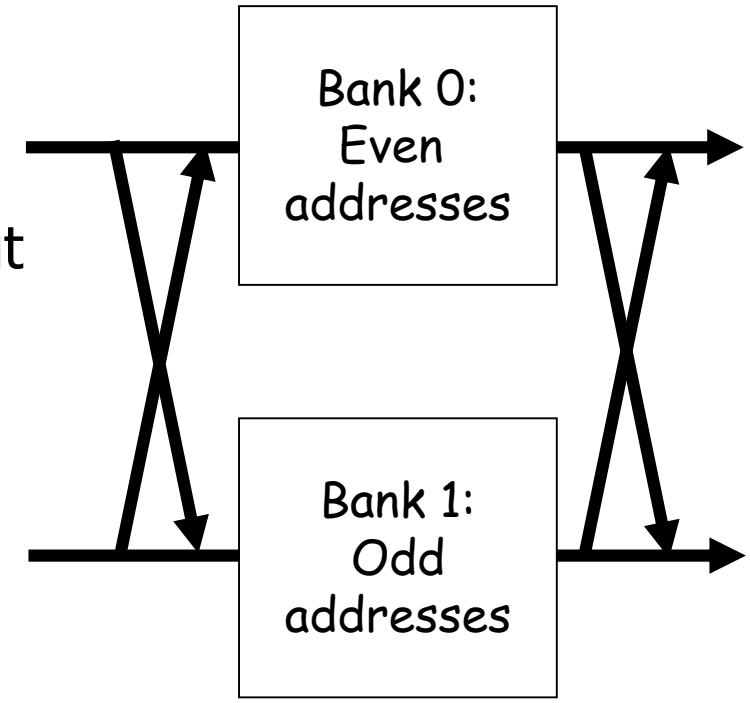
- Banking (Interleaving)
  - Bits in address determines which bank an address maps to
    - Address space partitioned into separate banks
    - Which bits to use for "bank address"?

  + No increase in data store area

  -- Cannot satisfy multiple accesses to the same bank

  -- Crossbar interconnect in input/output

- Bank conflicts
  - Two accesses are to the same bank



General Principle: Interleaving

- Interleaving (banking)
  - Problem: a single monolithic memory array takes long to access and does not enable multiple accesses in parallel
  - Goal: Reduce the latency of memory array access and enable multiple accesses in parallel
  - Idea: Divide the array into multiple banks that can be accessed independently (in the same cycle or in consecutive cycles)
    - Each bank is smaller than the entire memory storage
    - Accesses to different banks can be overlapped

Multi-Core Issues in Caching

Caches in Multi-Core Systems

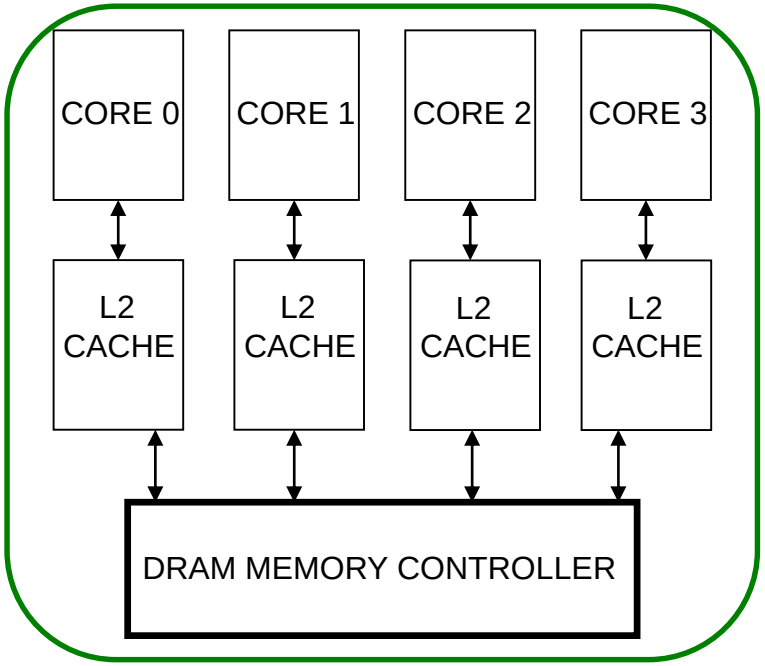
- Cache efficiency becomes even more important in a multi-core/multi-threaded system
  - Memory bandwidth is at premium
  - Cache space is a limited resource

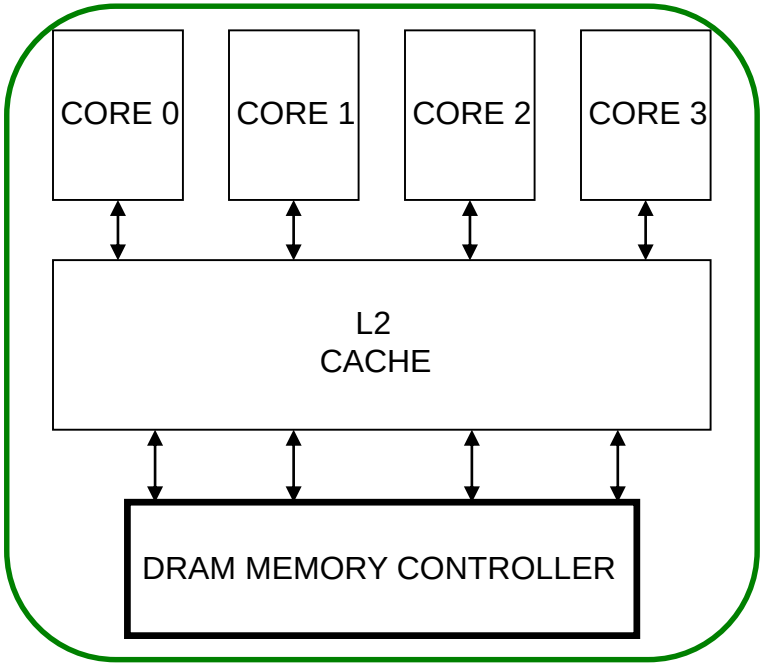
- How do we design the caches in a multi-core system?

- Many decisions
  - Shared vs. private caches
  - How to maximize performance of the entire system?
  - How to provide QoS to different threads in a shared cache?
  - How should space be allocated to threads in a shared cache?

Private vs. Shared Caches

- Private cache: Cache belongs to one core (a shared block can be in multiple caches)
- Shared cache: Cache is shared by multiple cores





Resource Sharing Concept and Advantages

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory

+ Resource sharing improves utilization/efficiency → throughput
  - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data
+ Reduces communication latency
  - For example, shared data kept in the same cache in multithreaded processors
+ Compatible with the shared memory model

Resource Sharing Disadvantages

- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

- Sometimes reduces each or some thread's performance
  - Thread performance can be worse than when it is run alone
- Eliminates performance isolation → inconsistent performance across runs
  - Thread performance depends on co-executing threads
- Uncontrolled (free-for-all) sharing degrades QoS
  - Causes unfairness, starvation

Need to efficiently and fairly utilize shared resources

Shared Caches Between Cores

- Advantages:
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
  - Easier to maintain coherence (a cache block is in a single location)
  - Shared data do not ping pong between caches

- Disadvantages
  - Slower access
  - Cores incur conflict misses due to other cores' accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

Shared Caches: How to Share?

- Free-for-all sharing
  - Placement/replacement policies are the same as a single core system (usually LRU or pseudo-LRU)
  - Not thread/application aware
  - An incoming block evicts a block regardless of which threads the blocks belong to

- Problems
  - Inefficient utilization of cache: LRU is not the best policy
  - A cache-unfriendly application can destroy the performance of a cache friendly application
  - Not all applications benefit equally from the same amount of cache: free-for-all might prioritize those that do not benefit
  - Reduced performance, reduced fairness

Example: Utility Based Shared Cache Partitioning

- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space

- The high-level idea can be applied to other shared resources as well.

CACHE COHERENCE

Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Coherence Defined

- Informally: Reads return most recently written value

- Formally:
  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value

  - P_1 writes X; P_2 reads X (sufficiently later)
    $\Rightarrow$ read returns written value

  - P_1 writes X, P_2 writes X
    $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X

This slide is out of course

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence

1. Snooping protocols
   - Each cache monitors bus reads/writes

2. Directory-based protocols
   - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

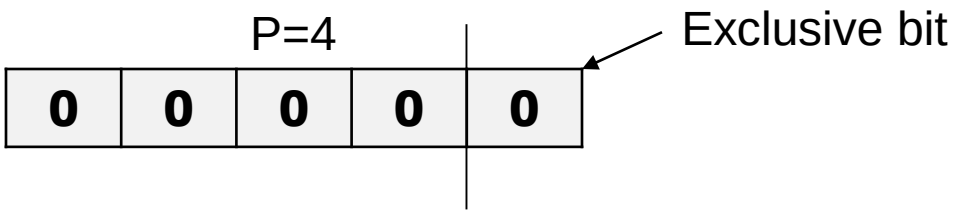
CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		1
CPU B read X	Cache miss for X	1	1	1

Directory Based Coherence

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

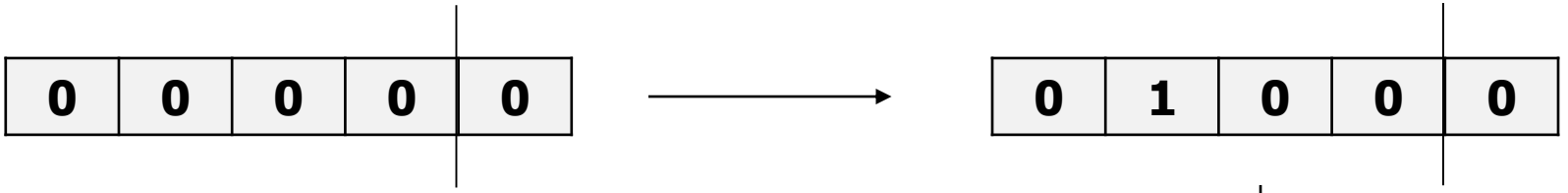
- An example mechanism:
  - For each cache block in memory, store P+1 bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that a cache has the only copy of the block and can update it without notifying others
  - On a read: set the cache's bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an "exclusive bit" associated with each block in each cache (so that the cache can update the exclusive block silently)

Directory Based Coherence Example (I)

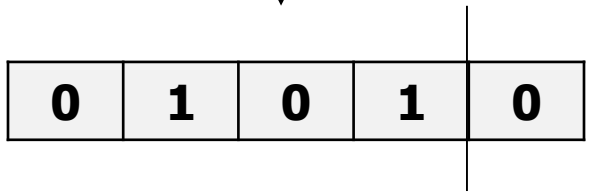


No core has the block A

1. P1 has a read miss to block A

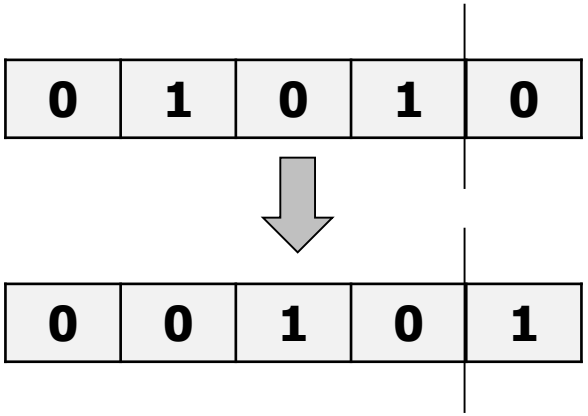


2. P3 has a read miss



Directory Based Coherence Example (II)

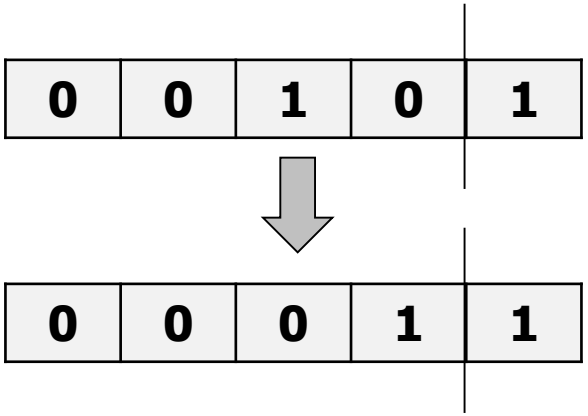
3. P2 has a write miss



a. Invalidate P1 and P3's copies
b. Write request => P2 has exclusive copy of block now
c. => So we set the exclusive bit
d. Now P2 can modify the block without notifying any other processor or directory
e. => P2 needs to have a bit in its cache notifying that it can perform exclusive updates to that block
f. => We need an **exclusive bit** for each cache block

Directory Based Coherence Example (III)

3. P3 has a write miss



a. Memory controller requests block from P2
b. Memory controller gives block to P3
c. P2's copy is invalidated

4. P2 has a read miss
=> P3 supplies it