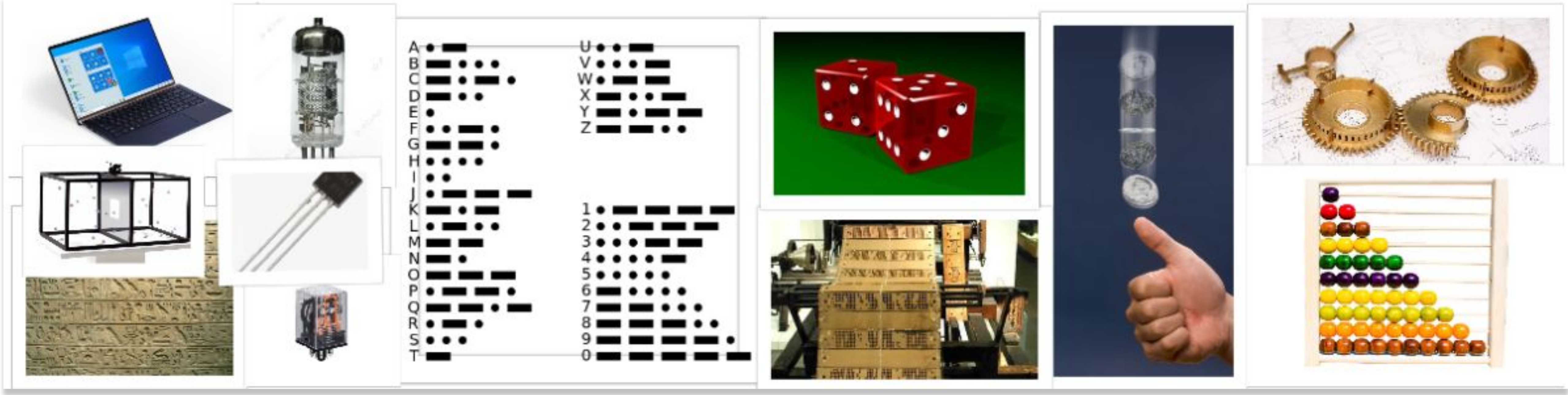


Computer Architecture

Memory/Storage Hierarchy

&

Cache Replacement Policies and Read & Write Strategies

Design goals for cache memory

- Where to place the incoming blocks from main memory?
  - Block placement policies
- How to find a block in the cache?
  - Block identification
- Which block should be replaced on a miss?
  - Block replacement policies
- What happens on a cache write?
  - Write strategy

Block replacement

- Cache's size is smaller, compared with the main memory

- How about the replacement policies for the direct mapping cache?
  - Only one choice and easy
- How about the replacement policies for the set associative mapping cache?
  - A set of choices and complicated
  - Which parameters can be used to decide the block?
  - Locality

Block replacement Algorithms/Policies

- Random
- First In First Out (FIFO)
- Last In First Out (LIFO)
- Least Recently Used (LRU)
- Pseudo-LRU (PLRU)
- Optimal
- Hybrid

Random Replacement Algorithm

- Needs a pseudo-random number generator

- Flip a coin or through a dice
    - Take the decision
    - Computers do it using random number generator

- For each set, generate block no. between *a* (min) and *b* (max)
    - a + [rand() % (b-a+1)]

- Does it take any advantages of locality?
    - No

First-in First-out (FIFO) Algorithm

- Replace/evict the block which is in the cache longest period of time (order)

- Does it take any advantages of locality?
  - Yes

- How does one implement it?
  - A queue for storing the references order of the blocks
  - Operations
    - En-queue
    - De-queue [used for evict out]
- Maintain an queue for each set
- Overhead
- Does it always match the temporal locality characteristic of the program?
  - Some memory location such as global variables can be accessed continuously

Least Recently Used Algorithm

- Replace/evict the block which is LRU
  - block that hasn't been accessed (**read/write**) by the processor in the longest period of time
- How does one implement it for associativity = 2?
  - A bit per cache line
  - Set/Clear on each access

LRU=0	**Way-0**	**LRU=1**	**Way-1**

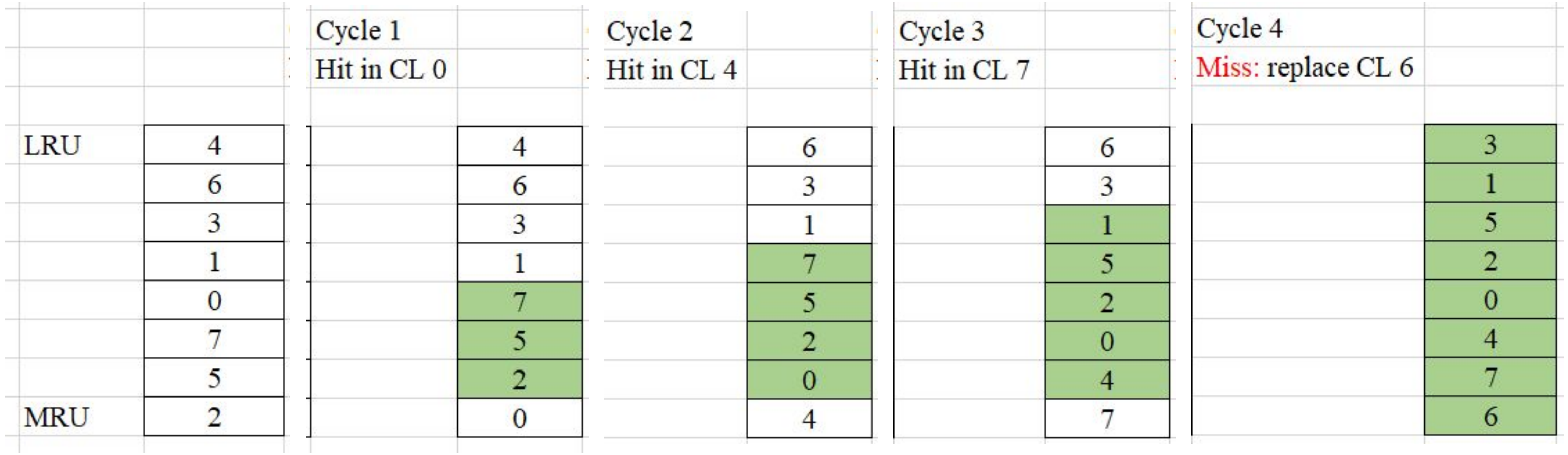
- How does one implement it for associativity >2?

Least Recently Used Algorithm

- If associativity > 2
  - LRU is difficult/expensive
    - Record timestamps? How many bits?
    - Find minimum timestamp on each replacement
    - Sorted list? Re-sort on every access, hit or miss?
  - Is there a way-out?
    - Shift-register-based implementation
    - Every time a block is referenced as hit or miss, it placed on the head on the ordered list, while other blocks in the set are pushed down the list

Least Recently Used Algorithm

- Shift-register-based implementation



		Cycle 1 Hit in CL 0	Cycle 2 Hit in CL 4	Cycle 3 Hit in CL 7	Cycle 4 Miss: replace CL 6
LRU	4	4	6	6	3
	6	6	3	3	1
	3	3	1	1	5
	1	1	7	5	2
	0	7	5	2	0
	7	5	2	0	4
	5	2	0	4	7
MRU	2	0	4	7	6

CL: Cache Line, Hit or miss can happen at CL <i>

Least Recently Used Algorithm

- Overhead for LRU
  - Maintaining the linked list
- Is there a better way-out?
  - Not-most-recently-used (NMRU)
    - History maintain for which block is accessed most recently
    - Evict out the block randomly from the other blocks
  - Researchers have proposed practical implementation of LRU or pseudo LRU algorithm

Pseudo LRU (PLRU)

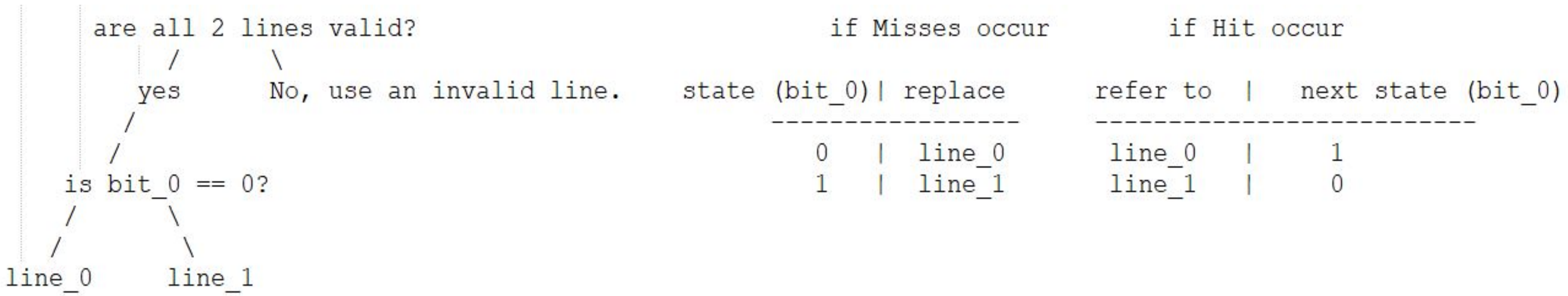
- Tree-PLRU
- Bit-PLRU

Tree-PLRU:

two-way set associative – one bit
indicates which line of the two has been reference more recently

four-way set associative - three bits
each bit represents one branch point in a binary decision tree; let 1 represent that the left side has been referenced more recently than the right side, and 0 vice-versa

Tree-PLRU: two-way set associative - one bit



```
     are all 2 lines valid?                          if Misses occur         if Hit occur
          /      \
        yes       No, use an invalid line.    state (bit_0)| replace      refer to  |   next state (bit_0)
       /                                             -----------------     --------------------------
      /                                                 0    |  line_0      line_0   |      1
   is bit_0 == 0?                                       1    |  line_1      line_1   |      0
    /      \
   /        \
line_0     line_1
```

each bit represents one branch point in a binary decision tree; let 1 represent that the left side has been referenced more recently than the right side, and 0 represent that the right has been referenced more recently than left side.

Example

- see in the CachePLRUReplacementAlgos.xlsx

Tree-PLRU: four-way set associative - three bits

each bit represents one branch point in a binary decision tree; let 1 represent that the left side has been referenced more recently than the right side, and 0 vice-versa

```
              are all 4 lines valid?
                   /       \
                 yes        no, use an invalid line
                  |
                  |
                  |
             bit_0 == 0?
               /        \
              y          n
             /            \
      bit_1 == 0?    bit_2 == 0?
        /     \        /     \
       y       n      y       n
      /         \    /         \
    line_0  line_1  line_2  line_3
```

If Miss occurs

```
 state | replace
-------+--------
  00x  |  line_0
  01x  |  line_1
  1x0  |  line_2
  1x1  |  line_3

  ('x' means
  don't care)
```

If Hit occurs

```
 ref to | next state
--------+-----------
 line_0 |    11_
 line_1 |    10_
 line_2 |    0_1
 line_3 |    0_0

('_' means unchanged)
```

Need (n-1) bit for each set, where n is the # of way

Overhead = (n-1) * K,
No of set is K

The representation of the states: bit_0, bit_1, bit_2

Example

- see in the <u>CachePLRUReplacementAlgos.xlsx</u>

Bit-PLRU

- Bit-PLRU stores one status bit for each cache line. These bits are called MRU-bits.
- Every access to a line sets its MRU-bit to 1, indicating that the line was recently used. Whenever the last remaining 0 bit of a set's status bits is set to 1, all other bits are reset to 0.
- At cache misses, the line with lowest index whose MRU-bit is 0 is replaced.

Example

- see in the CachePLRUReplacementAlgos.xlsx

Optimal Replacement Policy

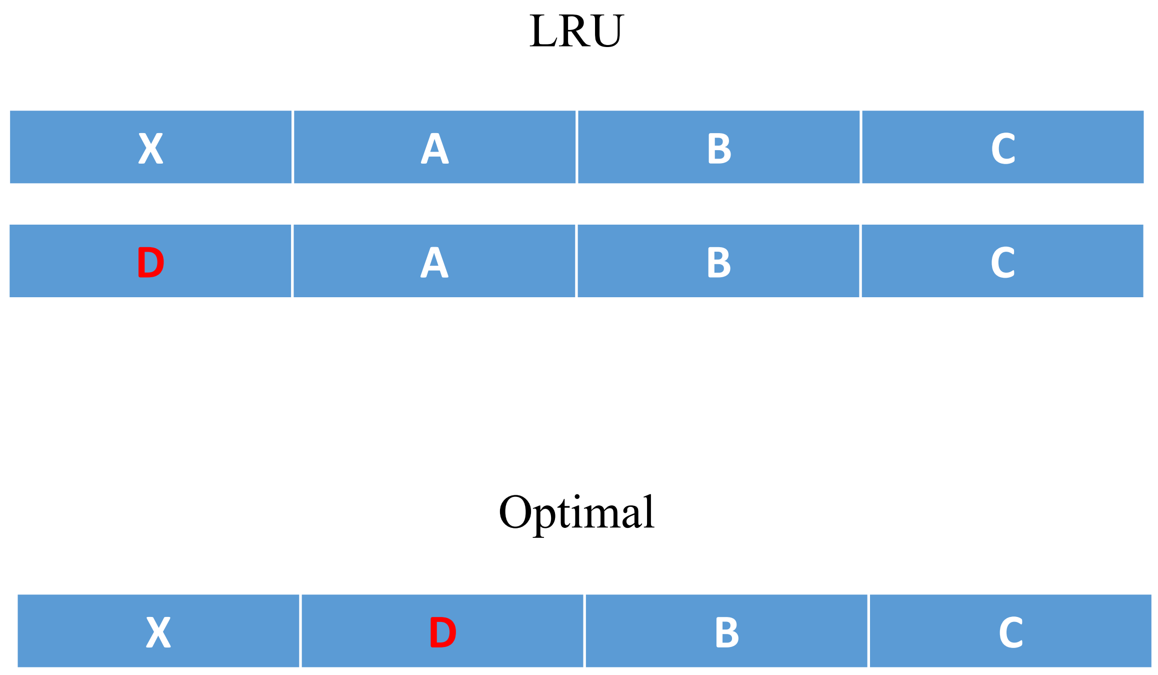
Evict the block with longest reuse distance

- Need future's knowledge

Can we build it?

- General case
  - No
- Special case
  - Yes
    - Trace

- Optimal better than LRU
  - 4-way set associative
  - LRU
  - Ref. X, A, B, C, D, X



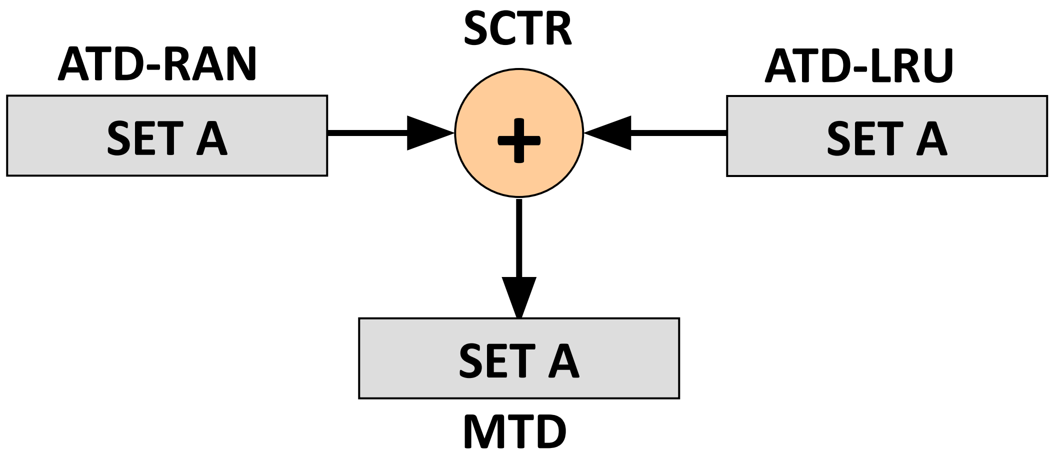
Cache Hybrid Replacement Algorithm

Hybrid Replacement Algorithm

- Example
  - 4-way set associative
  - LRU
  - Cyclic Ref. X, A, B, C, D, X, …
- Set thrashing: When the "program working set" in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
  - In practice
    - Depends on workload
    - Avg. hit rate of LRU and Random are similar
- Combined of two approaches: LRU & Random
- How to choose between LRU & Random?
  - Set sampling

Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.
Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.

Tournament Selection (TSEL) of Replacement Policies for a Single Set



Auxiliary Tag Directory (ATD)
Main Tag Directory (MDT)

If MSB of SCTR is 1, MTD uses RAN else MTD use LRU

ATD-RAN	ATD-LRU	Saturating Counter (SCTR)
HIT	HIT	Unchanged
MISS	MISS	Unchanged
HIT	MISS	+= Cost of Miss in ATD-LRU
MISS	HIT	-= Cost of Miss in ATD-RAN

Updation to the SCTR would happen based on MTD

Qureshi et al., “A Case for MLP-Aware Cache Replacement,“ ISCA 2006.

Tournament Selection (TSEL) of Replacement Policies for a Single Set

Only accesses that result in a miss for MTD are serviced by the memory system. If an access results in a hit for MTD but a miss for either ATD-RAN or ATD-LRU, then it is not serviced by the memory system. Instead, the ATD that incurred the miss finds a replacement victim using its replacement policy. The tag field associated with the replacement victim of the ATD is updated. The value of cost-q associated with the block is obtained from the corresponding tag-directory entry in MTD.

Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Cache Architecture: Read and Write Operations

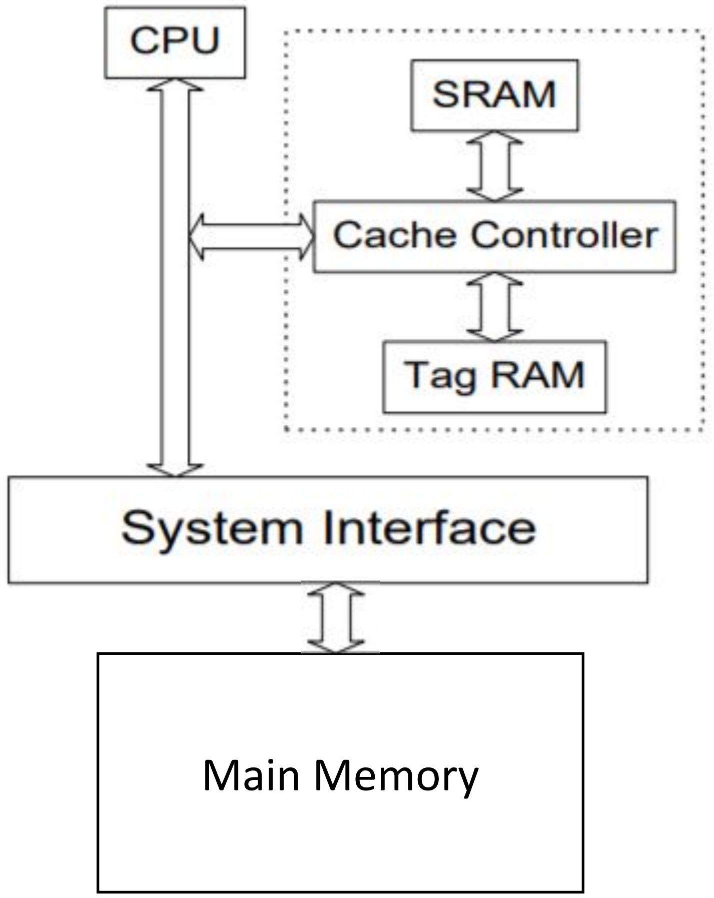
Cache Architecture: Read and Write Operations

- Caches have two characteristics
  - a read architecture
  - a write policy.
- The read architectures
  - Look Aside or
  - Look Through
- The write policies
  - Write-Back or
  - Write-Through
- Possible cases:

	Look Aside	Look through
Write-back	Yes	Yes
Write-through	Yes	Yes

Cache Read Architecture: Look aside cache

- Cache unit **sits in parallel with main** memory
- Both the main memory and the **cache see a bus cycle at the same time**. Hence the name "look aside."

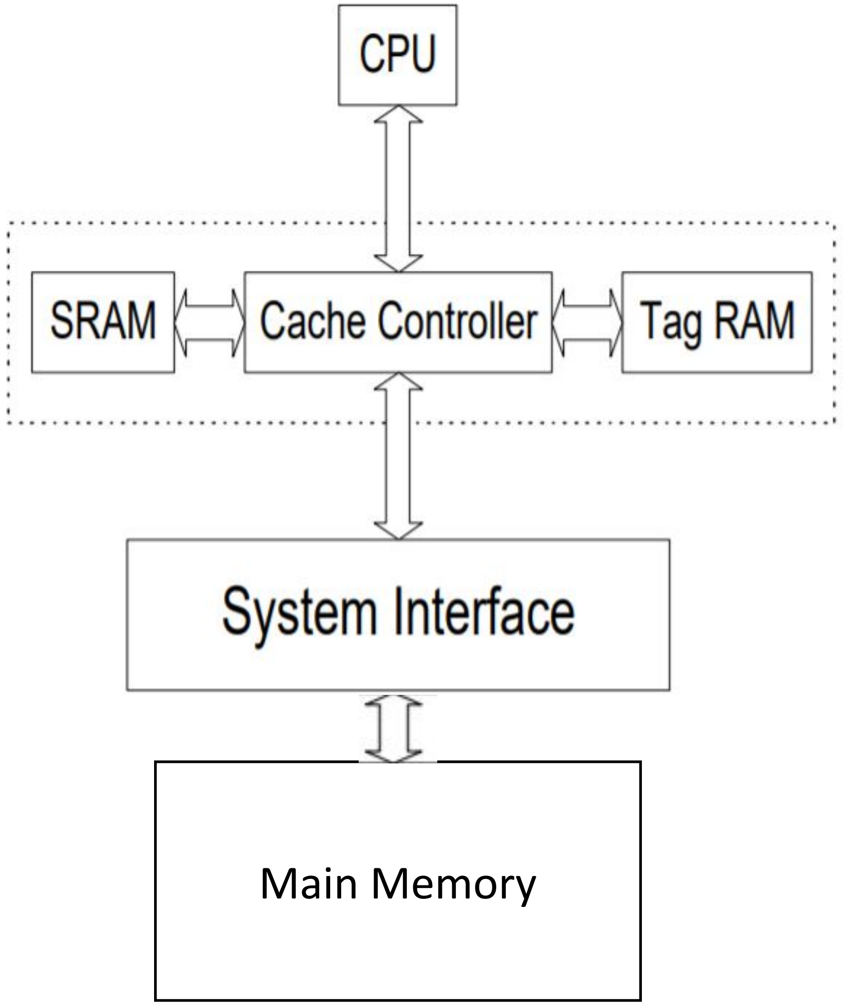


Cache Read Architecture: Look aside cache

- Processor starts a read cycle
- Hit:
  - Cache will respond to the read cycle and terminate the bus cycle
- Miss:
  - Main memory will respond to the processor and terminate the bus cycle
  - The data will be stored in the cache also
- Less complex & less expensive
- Access time is fast
- Drawback is processor cannot access cache while another bus master accessing main memory

Cache Read Architecture: Look through cache

- Cache sees the processors bus cycle before allowing it to pass on to the system bus. Hence "look through."



Cache Read Architecture: Look through cache

- Processor starts a read cycle
- Hit:
  - Cache will respond to the read cycle without starting an access to main memory
- Miss:
  - Cache passes the bus cycle onto the system bus and main memory responds
  - The data will be stored in the cache also
- Complex & expensive
- Access time is slow
- Drawback of look aside architecture has removed

Cache Write Architecture: Write Back

- That is, when the processor starts a write cycle the cache receives the data and terminates the cycle. The cache then writes the data back to main memory when the system bus is available. This method provides the greatest performance by allowing the processor to continue its tasks while main memory is updated at a later time. However, controlling writes to main memory increase the cache's complexity and cost.
- Dirty or modified bit is needed to identify a modified cache block

Cache Write Architecture: Write Through

- The processor writes through the cache to main memory. The cache may update its contents, however the write cycle does not end until the data is stored into main memory. This method is less complex and therefore less expensive to implement. The performance with a Write-Through policy is lower since the processor must wait for main memory to accept the data.

- When the processor must wait for writes to complete during write through, the processor is said to write stall.

- A <u>common optimization to</u> reduce write stalls is a write buffer, which allows the processor to continue as soon as the data are written to the buffer, thereby overlapping processor execution with memory updating.

Write misses: Write allocate and No-Write allocate

- Because the data are not needed on a write, there are two options on a write miss:
- Write allocate—The block is allocated on a write miss, followed by the preceding write hit actions. In this natural option, write misses act like read misses.
- No-write allocate—This apparently unusual alternative is write misses do not affect the cache. Instead, the block is modified only in the lower-level memory.
- Thus, blocks stay out of the cache in no-write allocate until the program tries to read the blocks, but even blocks that are only written will still be in the cache with write allocate.

Example

- Assume a fully associative write-back cache with many cache entries that starts empty. Following is a sequence of five memory operations (the address is in squarebrackets):

  Write Mem[100];

  Write Mem[100];

  Read Mem[200];

  Write Mem[200];

  Write Mem[100].

- What are the number of hits and misses when using no-write allocate versus write allocate?

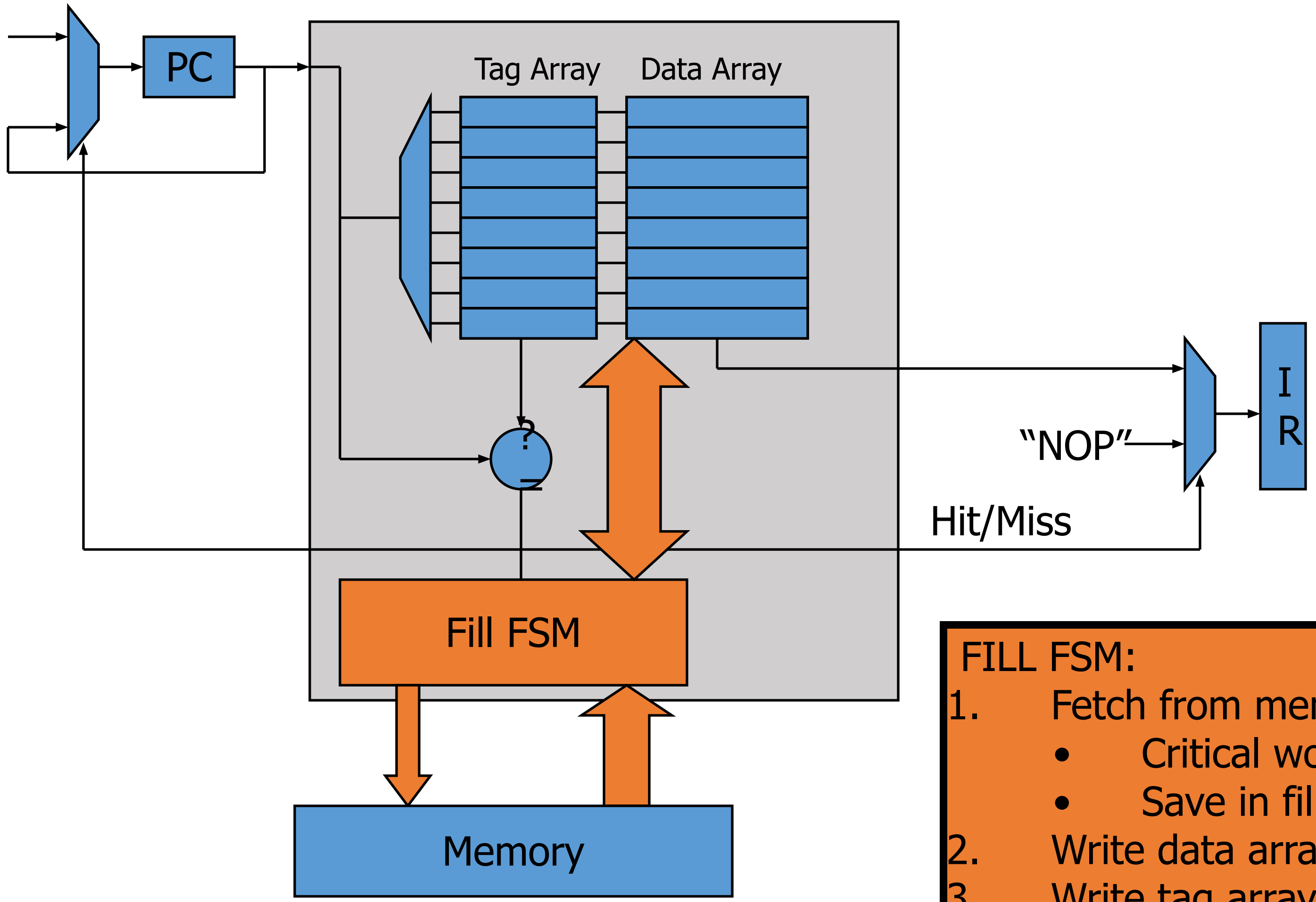
Example

- For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. <u>The last write to 100 is still a miss</u>. The result for no-write allocate is four misses and one hit.
- For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits because 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

Write allocate Vs. No-write allocate

- Either write miss policy could be used with write through or write back.
- Usually, write-back caches use write allocate, hoping that subsequent writes to that block will be captured by the cache.
- Write-through caches often use no-write allocate. The reasoning is that even if there are subsequent writes to that block, the writes must still go to the lower-level memory, so what’s to be gained?

I-Caches and Pipelining



FILL FSM:
1. Fetch from memory
   - Critical word first
   - Save in fill buffer
2. Write data array
3. Write tag array
4. Miss condition ends

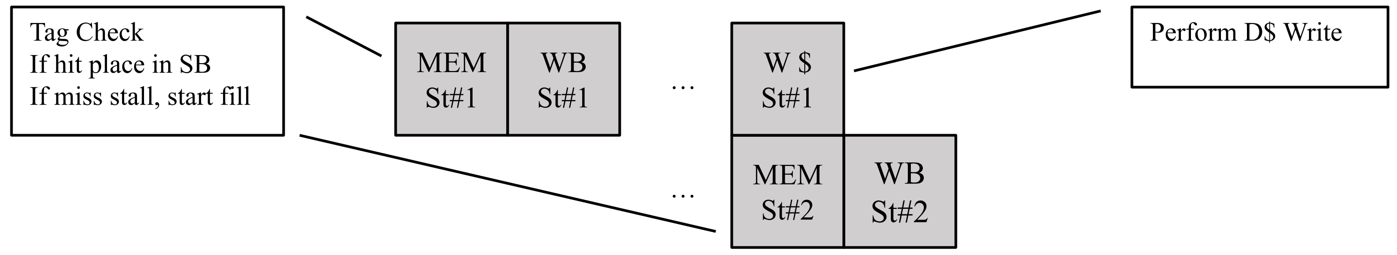
D-Caches and Pipelining

- Pipelining loads from cache
  - Hit/Miss signal from cache
  - Stalls pipeline or inject NOPs?
    - Hard to do in current real designs, since wires are too slow for global stall signals
  - Instead, treat more like branch misprediction
    - Cancel/flush pipeline
    - Restart when cache fill logic is done

D-Caches and Pipelining

- Stores more difficult
  - MEM stage:
    - Perform tag check
    - Only enable write on a hit
    - On a miss, must not write (data corruption)
  - Problem:
    - Must do tag check and data array access sequentially
    - This will hurt cycle time or force extra pipeline stage
    - Extra pipeline stage delays loads as well: IPC hit!

Solution: Pipelining Writes



- Store #1 performs tag check only in MEM stage
  - <value, address, cache way> placed in store buffer (SB)
- When store #2 reaches MEM stage
  - Store #1 writes to data cache
- In the meantime, must handle RAW to store buffer
  - Pending write in SB to address A
  - Newer loads must check SB for conflict
  - Stall/flush SB, or forward directly from SB
- Any load miss must also flush SB first
  - Otherwise SB D$ write may be to wrong line
- Can expand to >1 entry to overlap store misses

Cache hierarchy performance

Cache Misses and Performance

- How does this affect performance?
- Performance = Time / Program

$$= \frac{\textbf{Instructions}}{\textbf{Program}} \times \frac{\textbf{Cycles}}{\textbf{Instruction}} \times \frac{\textbf{Tim}}{\textbf{Cycle}}$$

(code size) **(CPI)** **(cycle time)**

- Cache organization affects cycle time
  - Hit latency
- Cache misses affect **CPI**

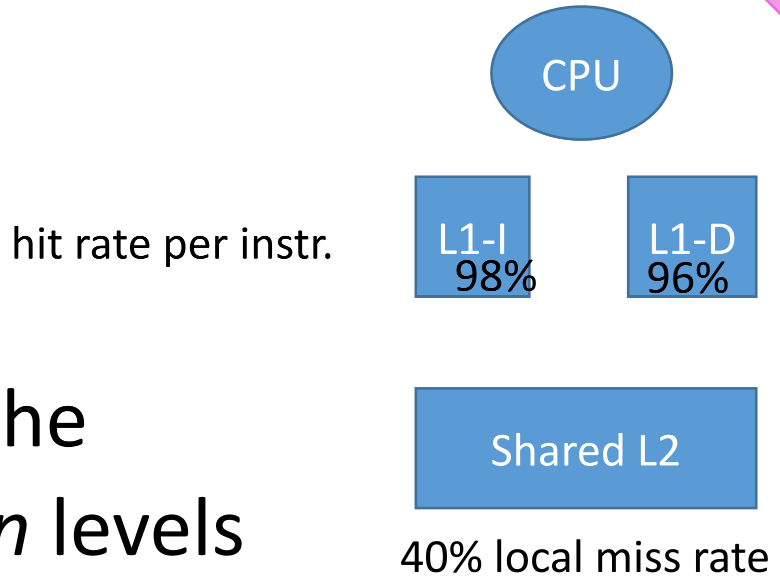
Cache Misses and CPI

$$
\begin{aligned}
CPI &= \frac{cycles}{inst} = \frac{cycles_{hit}}{inst} + \frac{cycles_{miss}}{inst} \\
&= \frac{cycles_{hit}}{inst} + \frac{cycles}{miss} \times \frac{miss}{inst} \\
&= \frac{cycles_{hit}}{inst} + Miss\_penalty \times Miss\_rate
\end{aligned}
$$

- Cycles spent handling misses are strictly additive
- Miss_penalty is recursively defined at next level of cache hierarchy as weighted sum of hit latency and miss latency

Cache Misses and CPI

$$CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l$$



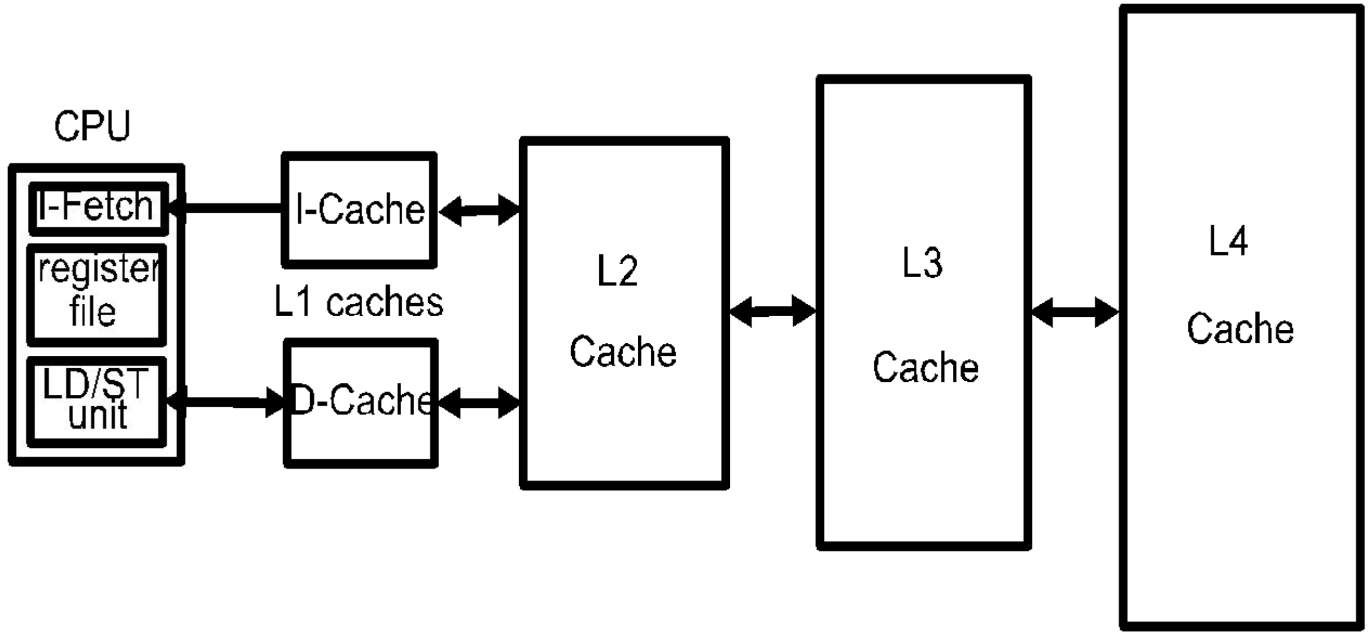
- P_l is miss penalty at each of *n* levels of cache
- MPI_l is miss rate per instruction at each of *n* levels of cache
- Miss rate specification:
  - Per instruction: easy to incorporate in CPI
  - Per reference: must convert to per instruction
    - Local: misses per local reference
    - Global: misses per fetch or load or store

Cache Misses and Performance

- CPI equation
  - Only holds for misses that cannot be overlapped with other activity
  - Store misses often overlapped
    - Place store in store queue
    - Wait for miss to complete
    - Perform store
    - Allow subsequent instructions to continue in parallel
  - Modern out-of-order processors also do this for loads
    - Cache performance modeling requires detailed modeling of entire processor core

Memory Inclusion

Memory Inclusion



Memory Inclusion

- Instruction and Data accessed by a processor must be present in the main memory
- The block present in the cache level must have a copy in the main memory
- A design choice
    - Whether or not a cache includes higher-level caches
- A cache at *level j* is said to include another cache at *level i*, j > *i*, if
    - Any memory location cached at *level i* is also cached at *level j*
    - The state of the copy at *level j* includes the state of the copy at *level i*

Memory Inclusion

- A cache at *level j* is said to include another cache at *level i*, j > i, if
  - Any memory location cached at *level i* is also cached at *level j*
  - The state of the copy at *level j* includes the state of the copy at *level i*

    Level i's access rights are the same as access rights at *level j*
    OR
    Level i's access rights are more restrictive than access rights at *level j*

Example: If the copy is readable and writeable at *level i*, then the copy must be readable and writeable at *level j*.
If the copy is read-only at *level j* then the copy cannot be writable at *level i*.

Memory Inclusion

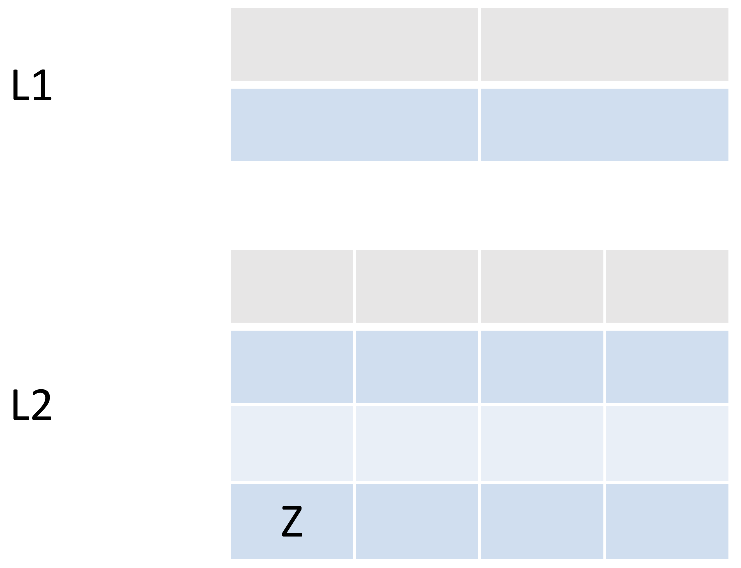
- Inclusion is a very convenient property of cache hierarchies, especially to help maintain coherence
  - *Single core*
  - *Multi-core*
  - *Allocation*
  - *Eviction*
- The major *disadvantage is cache* space usage
- To avoid such a waste of cache space, some hierarchies enforce *exclusion*

Memory Exclusion

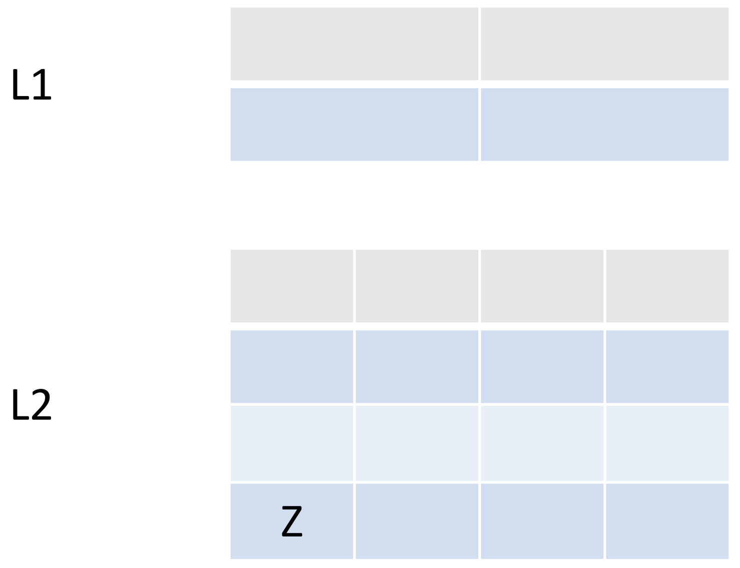
- To avoid such a waste of cache space, some hierarchies enforce *exclusion*
- If a block is present at a cache level L_i, then it is not present at other cache levels
- Whenever a block is brought at L_1 (on miss), it must be invalidated at all other levels L_i's, i>1
- Whenever a block is replaced at L_i then possibly allocated at the next level, L_{i+1}

Memory Inclusion & Exclusion: Examples

The initial state of the L1 and L2 caches: block Z is already in L2



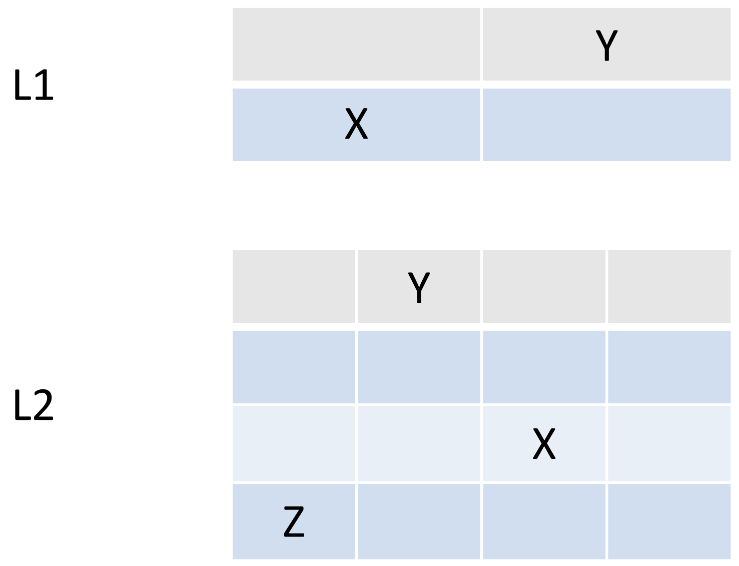
Inclusive L2



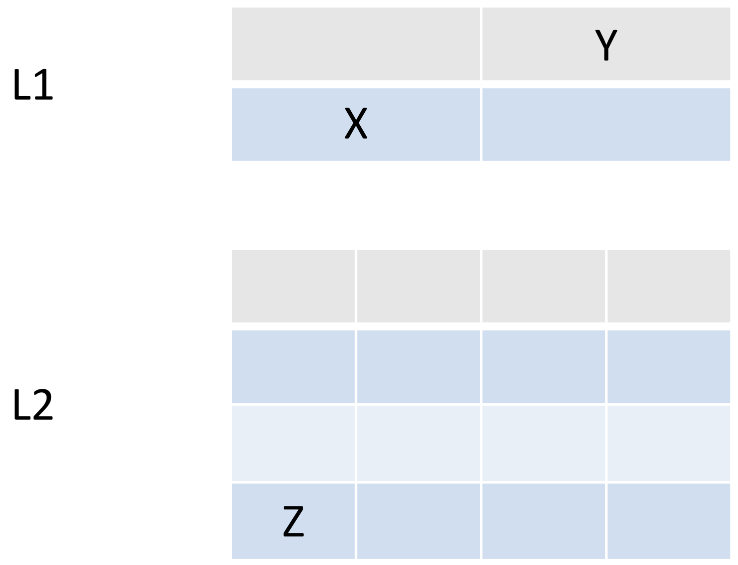
Exclusive L2

Memory Inclusion & Exclusion: Examples

After an L1 and L2 cache miss on blocks X and Y:



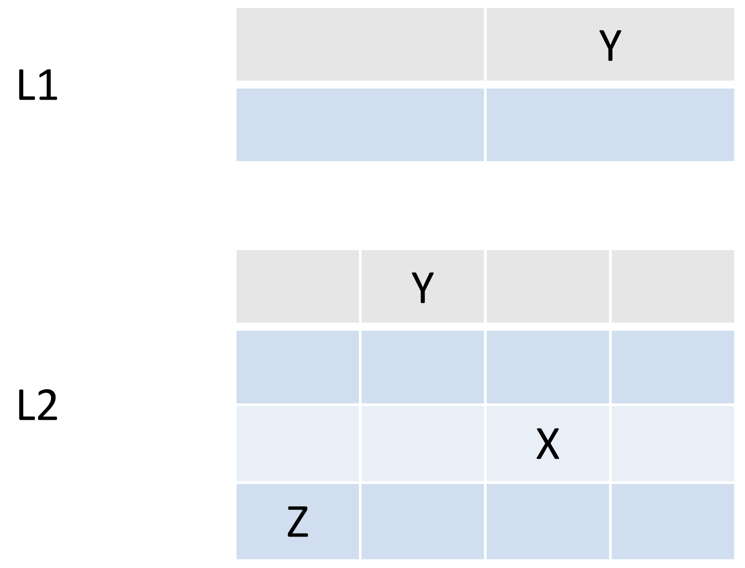
Inclusive L2



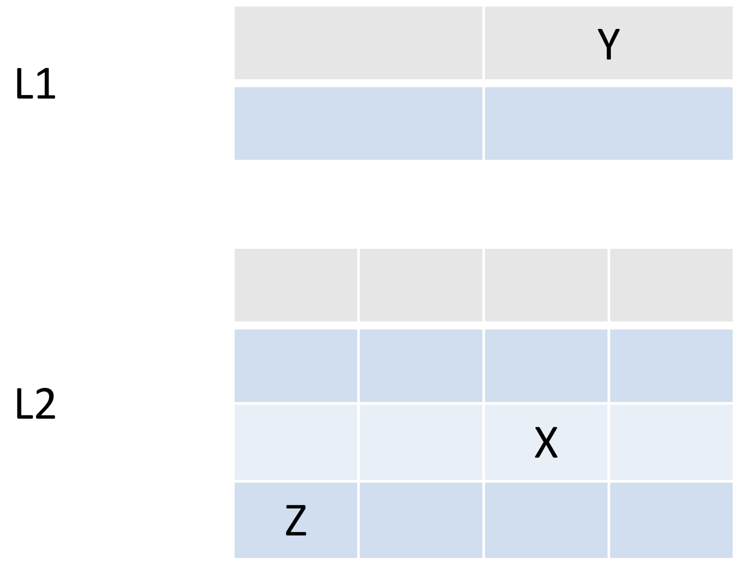
Exclusive L2

Memory Inclusion: Examples

After block X is evicted from the L1 cache:



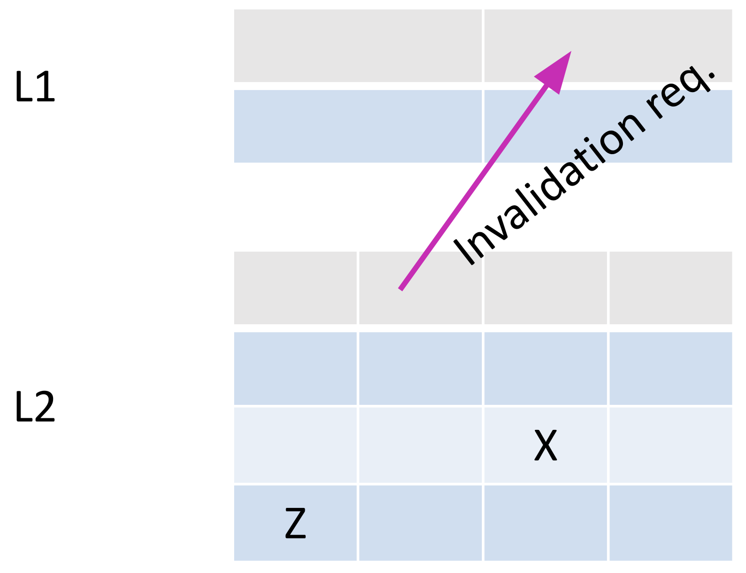
Inclusive L2



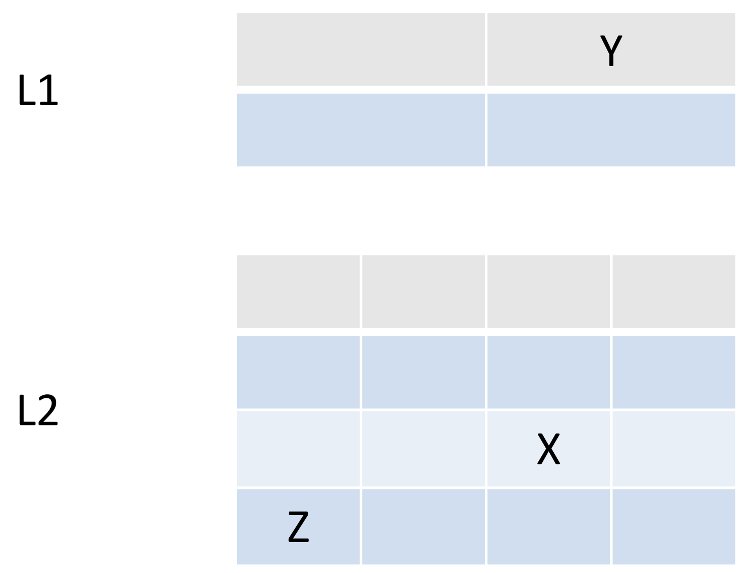
Exclusive L2

Memory Inclusion: Examples

After block Y is evicted from the ***L2*** *cache:*



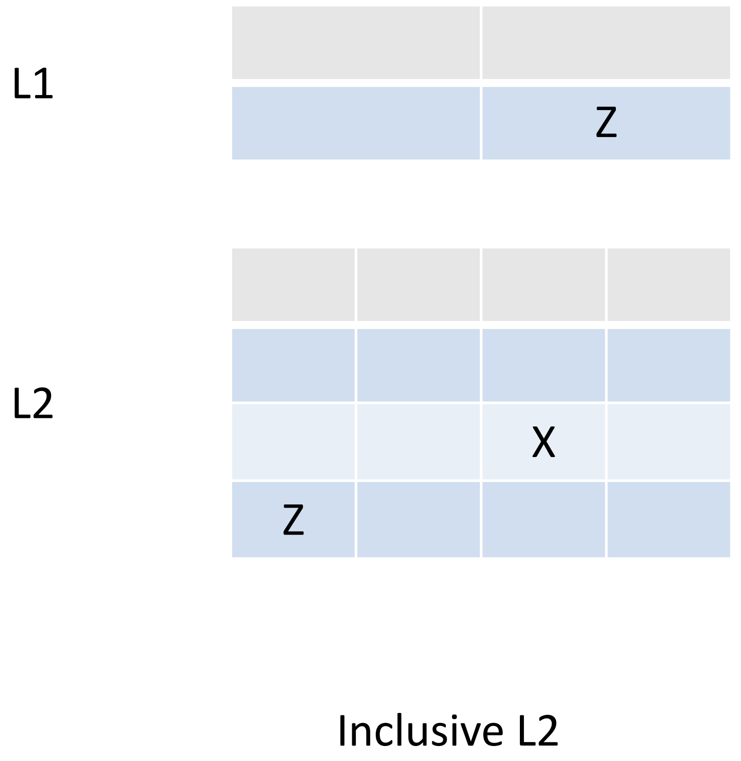
Inclusive L2

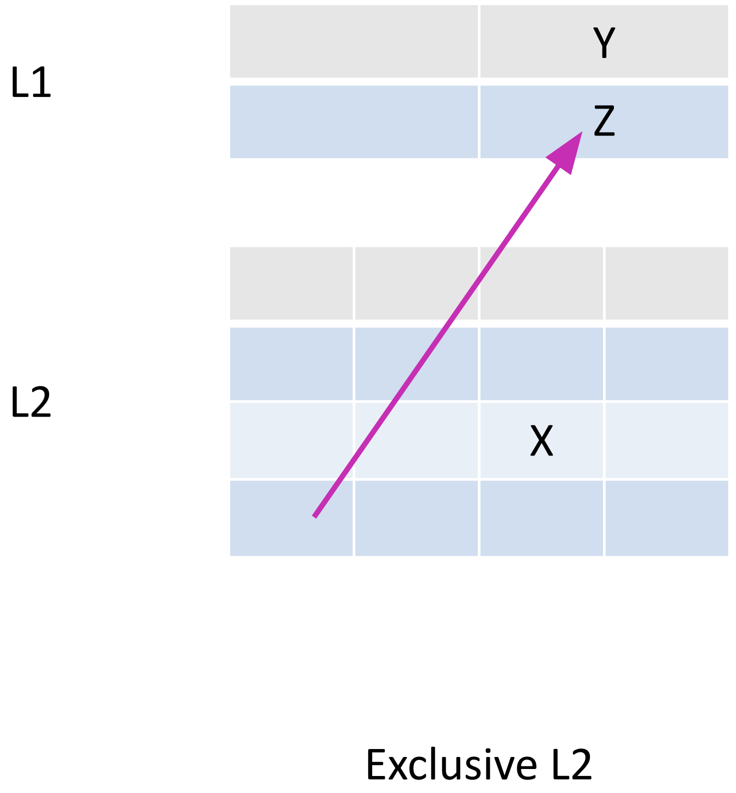


Exclusive L2

Memory Inclusion: Examples

After L1 cache miss on block Z:





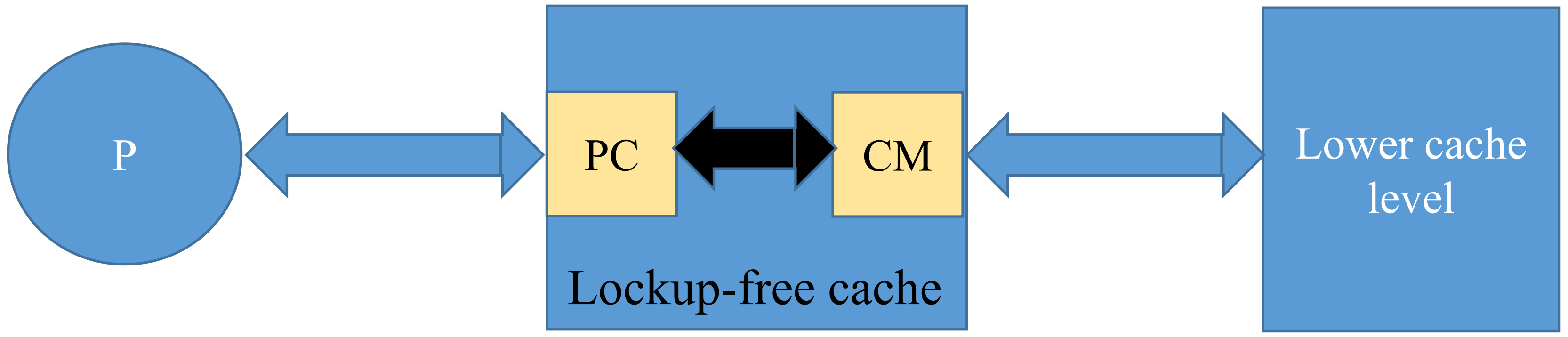
Non-blocking or lockup-free caches

Non-blocking or lockup-free caches

- Caches allow several concurrent misses
  - D-cache’s misses on load
  - Out-of-order processor
  - Hit-under-miss policy
  - Modern microprocessor caches (L1 & L2) are lockup-free
- How does the cache controller allow this?

Non-blocking or lockup-free caches

- How does the cache controller allow this?



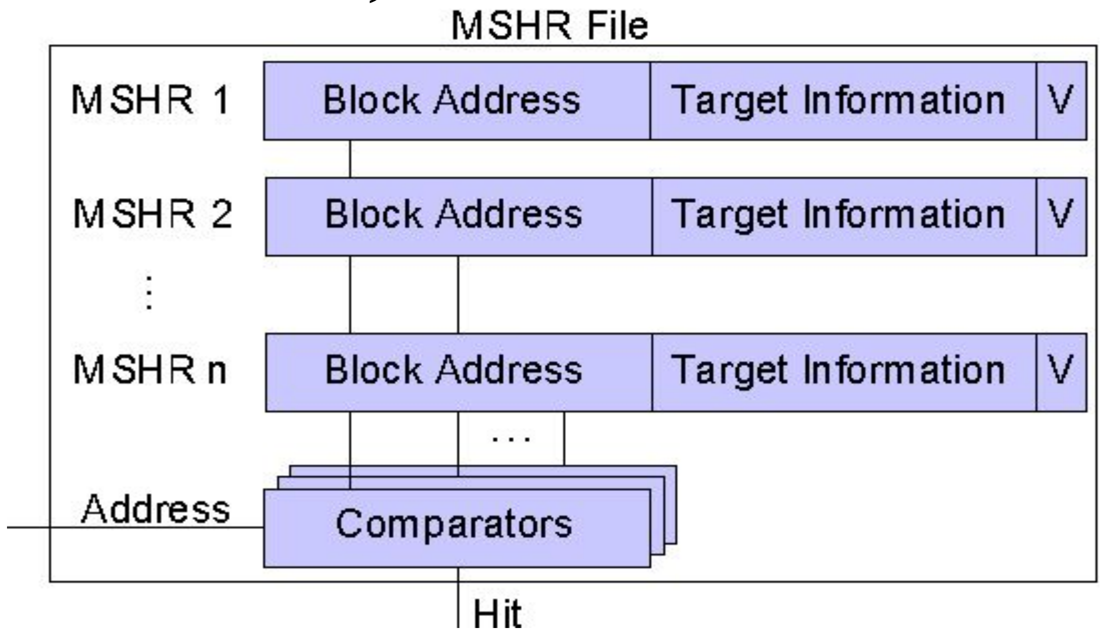
P: Processor
PC: Processor to cache interface
CM: Cache to memory interface

Non-blocking or lockup-free caches

- Get more than one cache miss request
- It continues to accept accesses to a block even if a miss to the block is in progress
- Many access may miss on a block whose miss is already pending in the cache
  - Why?
- The first miss is the *primary* miss and the rest are *secondary*
- How does the cache maintain such requests and send the data to CPU?
  - MSHR, Early restart and Critical word first

Non-blocking or lockup-free caches

- How does the cache maintain such requests?
- Encode the information related to a miss in a *missing status holding register* (MSHR)
- When a cache read miss occurs, an MSHR is associated with it



Non-blocking or lockup-free caches

- Inside the MSHR
- A bit indicating whether it is free or busy
- Information regarding which missing line is attached to it
  - A comparator
- A bit indicating whether the line is valid or not

Non-blocking or lockup-free caches

- Inside the MSHR
- For each word in the line
  - Fields for the destination register to load
  - Whether the data are valid or not
  - A bit for type of operation

Non-blocking or lockup-free caches

On a cache miss:

- Search MSHRs for a pending access to the same block
- Found: Allocate a load/store entry in the same MSHR entry
- Not found: Allocate a new MSHR
- No free entry: stall

When a block returns from the next level in memory

- Check which loads/stores waiting for it
- Forward data to the load/store unit
- Deallocate load/store entry in the MSHR entry
- Write block in cache or MSHR
- If last block, deallocate MSHR (after writing the block in cache)

Cache’s Design Parameters

Cache design

- Block size, Associativity
- Block organization
  - Direct-mapped
  - Fully associative
  - Set-associative
- Block replacement policy
  - Random
  - FIFO
  - LRU or Pseudo-LRU
- Write policy
  - Writeback
  - Write-through
    - Write-allocate
    - Write-no-allocate
- Read policy
  - Look aside
  - Look through

Cache design

- Inclusive cache
- Exclusive cache

Summary

- Replacement Algorithms
- Read architecture
- Write policies
- Write misses
- Performance evaluation
- Memory inclusion
- Lockup-free cache