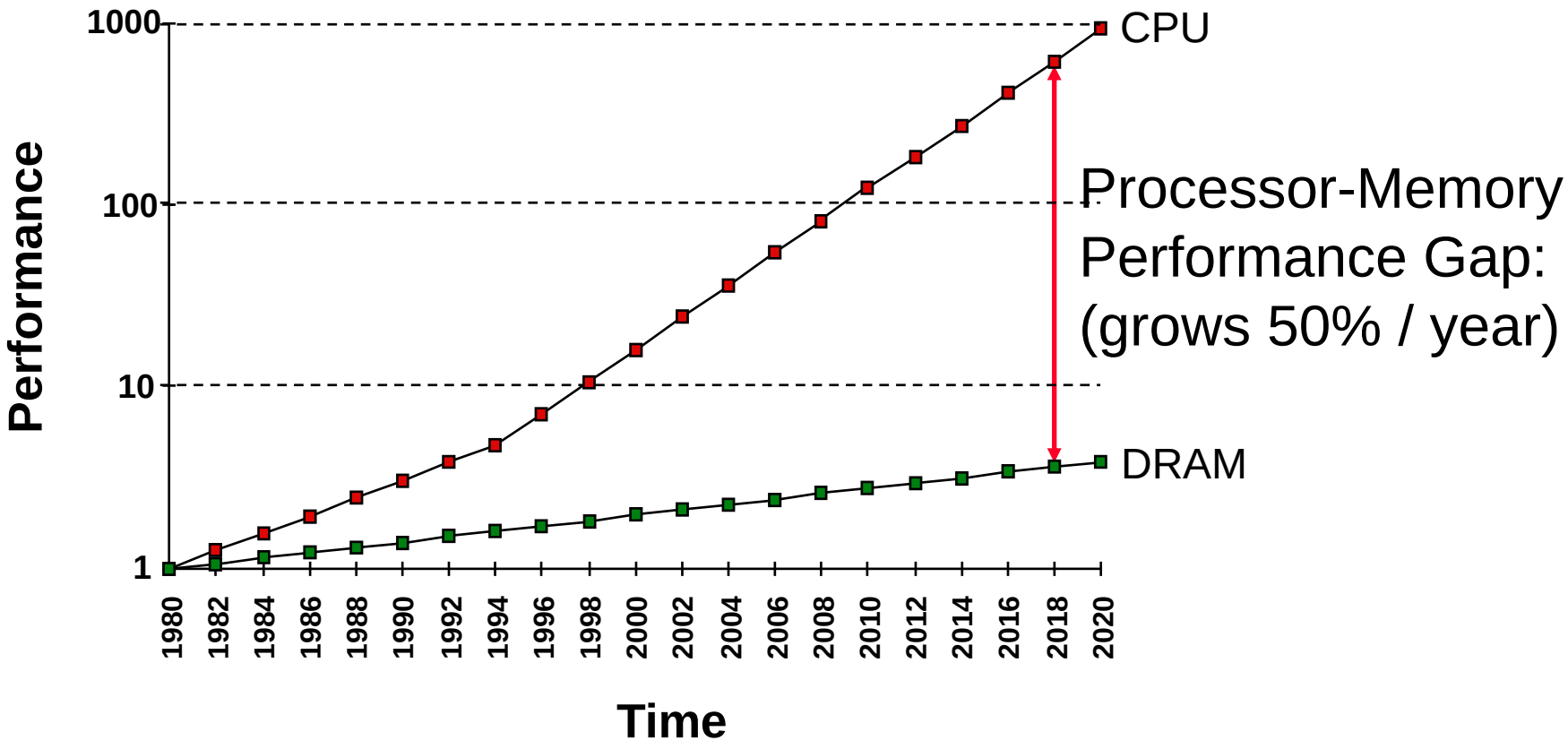
Computer Structure

Cache Memory

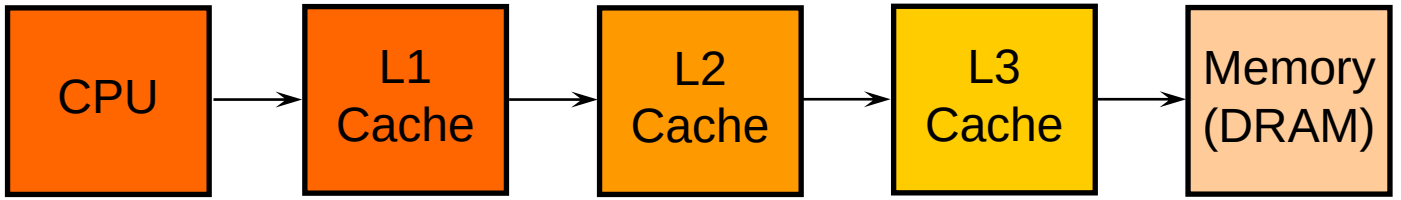
Lihu Rappoport and Adi Yoaz

Processor – Memory Gap



Memory Trade-Offs

- **Large (dense) memories are slow**
- **Fast memories are small, expensive and consume high power**
- **Goal:** give the processor a feeling that it has a memory which is large (dense), fast, consumes low power, and cheap
- **Solution: a Hierarchy of memories**



Speed:	Fastest	⟶	Slowest
Size:	Smallest	⟶	Biggest
Cost:	Highest	⟶	Lowest
Power:	Highest	⟶	Lowest

Why Hierarchy Works

- **Temporal Locality (Locality in Time):**
  - If an item is referenced, it will tend to be referenced again soon
  - Example: code and variables in loops

  ⇒ **Keep recently accessed data closer to the processor**

- **Spatial Locality (Locality in Space):**
  - If an item is referenced, nearby items tend to be referenced soon
  - Example: scanning an array

  ⇒ **Move contiguous blocks closer to the processor**

- **Locality + smaller HW is faster + Amdahl's law**
  ⇒ **memory hierarchy**

Memory Hierarchy: Terminology

- **For each memory level define the following**
  - Hit: data of the requested address is available in the cache level
  - Miss: data of the requested address is not available in the cache level
  - Hit Rate: the fraction of accesses that hit in that cache level
    - For L1 cache: #L1 hits / #data accesses in the program
    - For L2 cache: #L2 hits / #L1 misses
  - Hit Time: time from data request till data received when hitting in the cache level; includes the time to determine hit/miss
  - Miss Rate = 1 – (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- **Average memory-access time =**

$$t_{effective} = (\text{Hit time} \times \text{Hit Rate}) + (\text{Miss Time} \times \text{Miss rate})$$
$$= (\text{Hit time} \times \text{Hit Rate}) + (\text{Miss Time} \times (1 - \text{Hit rate}))$$

  - If hit rate is close to 1, $t_{effective}$ is close to Hit time

Effective Memory Access Time

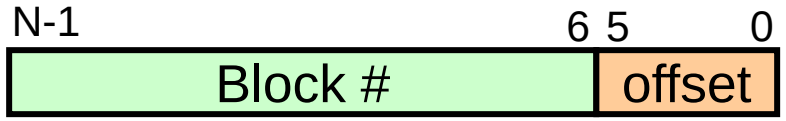
- **Cache – holds a subset of the memory**
  - Hopefully – the subset being used now
- **Effective memory access time**
  - $t_{effective} = (t_{cache} \times \text{Hit Rate}) + (t_{mem} \times (1 - \text{Hit rate}))$
  - t_{mem} includes the time it takes to detect a cache miss
- **Example**
  - Assume $t_{cache} = 10$ nsec and $t_{mem} = 100$ nsec

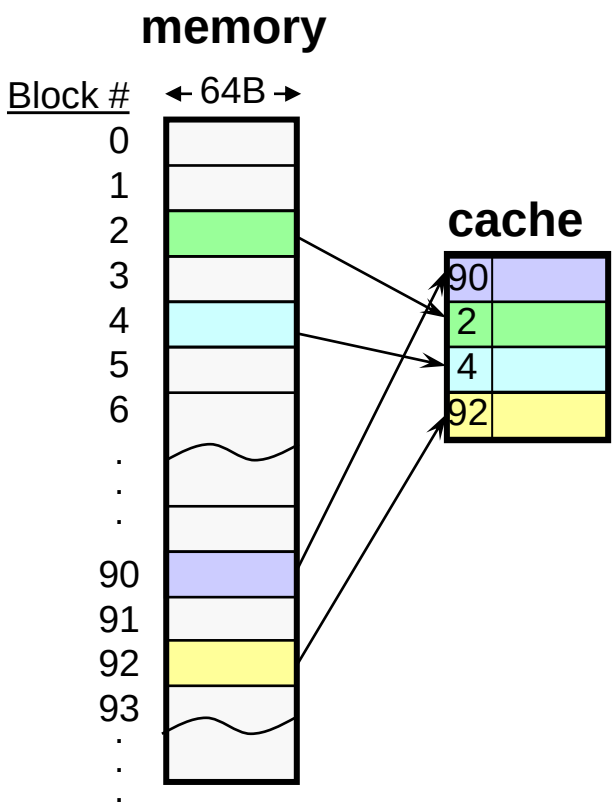
Hit Rate	t_{eff} (nsec)
0	100
50	55
90	20
99	10.9
99.9	10.1

  - t_{mem}/t_{cache} goes up $\Rightarrow$ more important that hit-rate closer to 1

Cache – Main Idea

- **The cache holds a small part of the entire memory**
  - Need to map parts of the memory into the cache
- **Main memory is (logically) partitioned into blocks**
  - Typical block size is 32 or 64 bytes
  - Blocks are aligned
- **The cache partitioned into cache lines**
  - Each cache line holds a memory block
  - Only a subset of the memory blocks are mapped to the cache at a given time
  - The cache views an address as

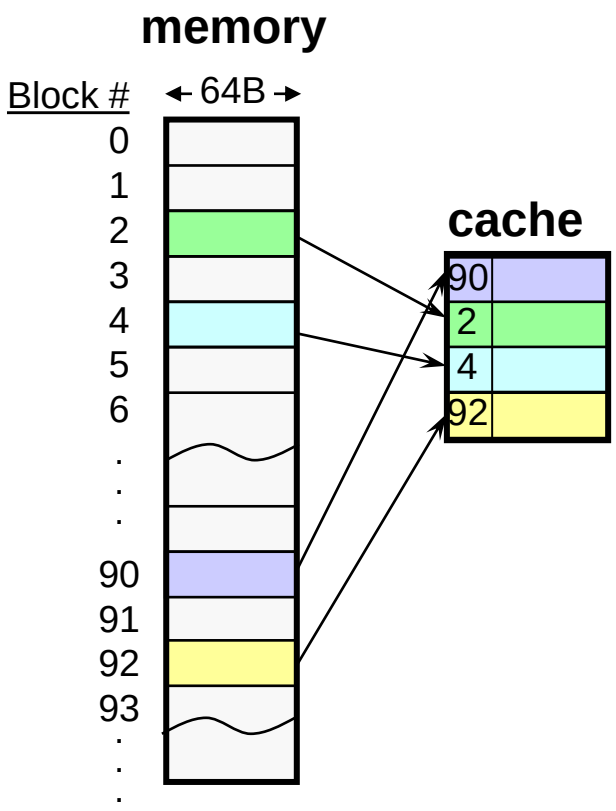




Cache Lookup

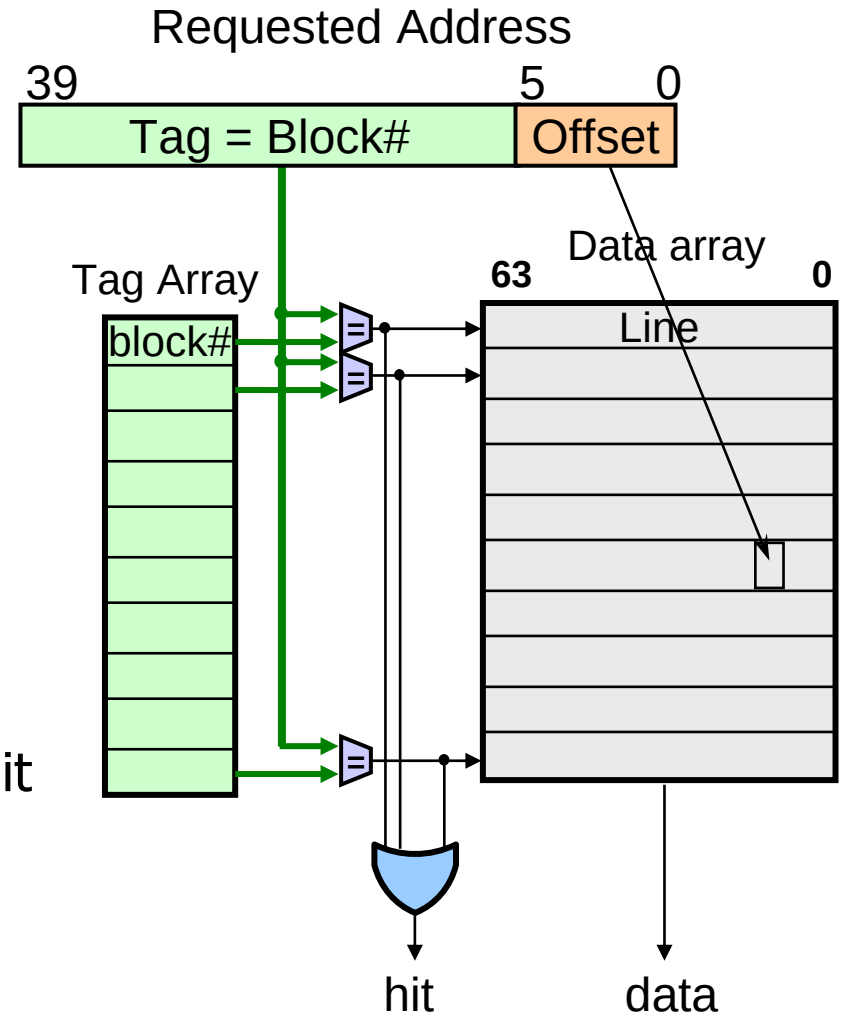
- **Cache hit**
  - Block is mapped to the cache –
    return data according to block's offset

- **Cache miss**
  - Block is not mapped to the cache
    $\Rightarrow$ do a cache line fill
    - Fetch block into fill buffer
      - may require few bus cycle
    - Write fill buffer into cache
  - May need to evict another block
    from the cache
    - Make room for the new block



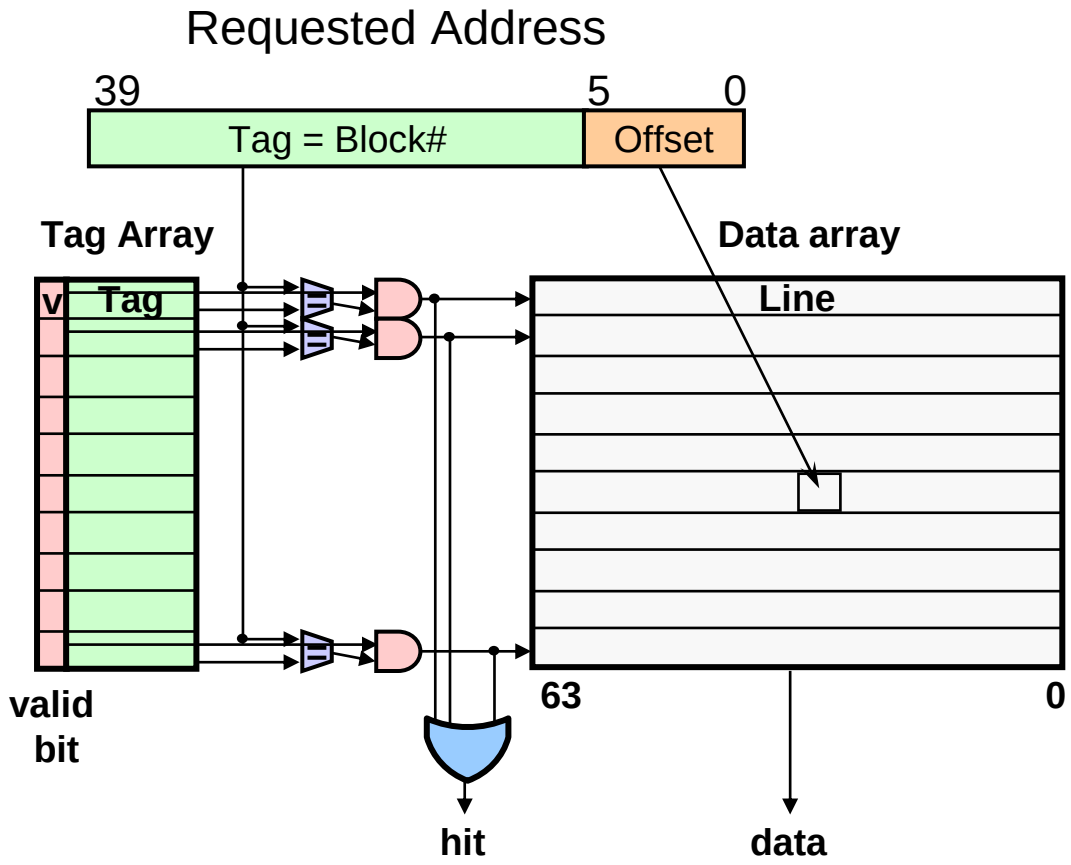
Fully Associative Cache

- **An address is partitioned to**
  - Address m.s.bits: memory block number
  - Address l.s.bits: offset within block
- **Each memory block may be mapped to each of the cache lines**
  - Lookup block in all lines
- **Each cache line has a tag**
  - All tags are compared to the block# in parallel
  - Need a comparator per line
    - Takes area and power
  - If one of the tags matches the block#, we have a hit
    - Supply data according to offset



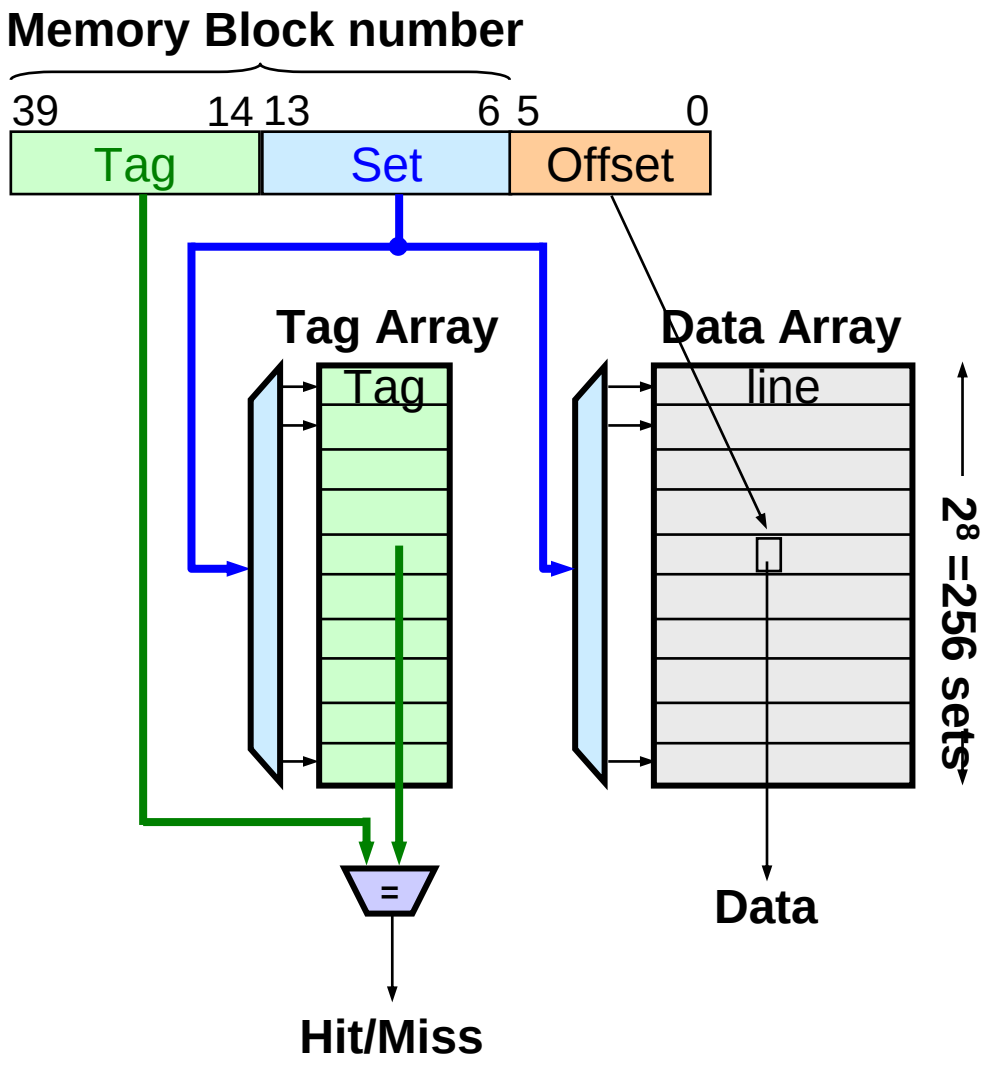
Valid Bit

- **Initially cache is empty**
  - Need to have a "line valid" indication – line *valid bit*
- **A line may also be invalidated**



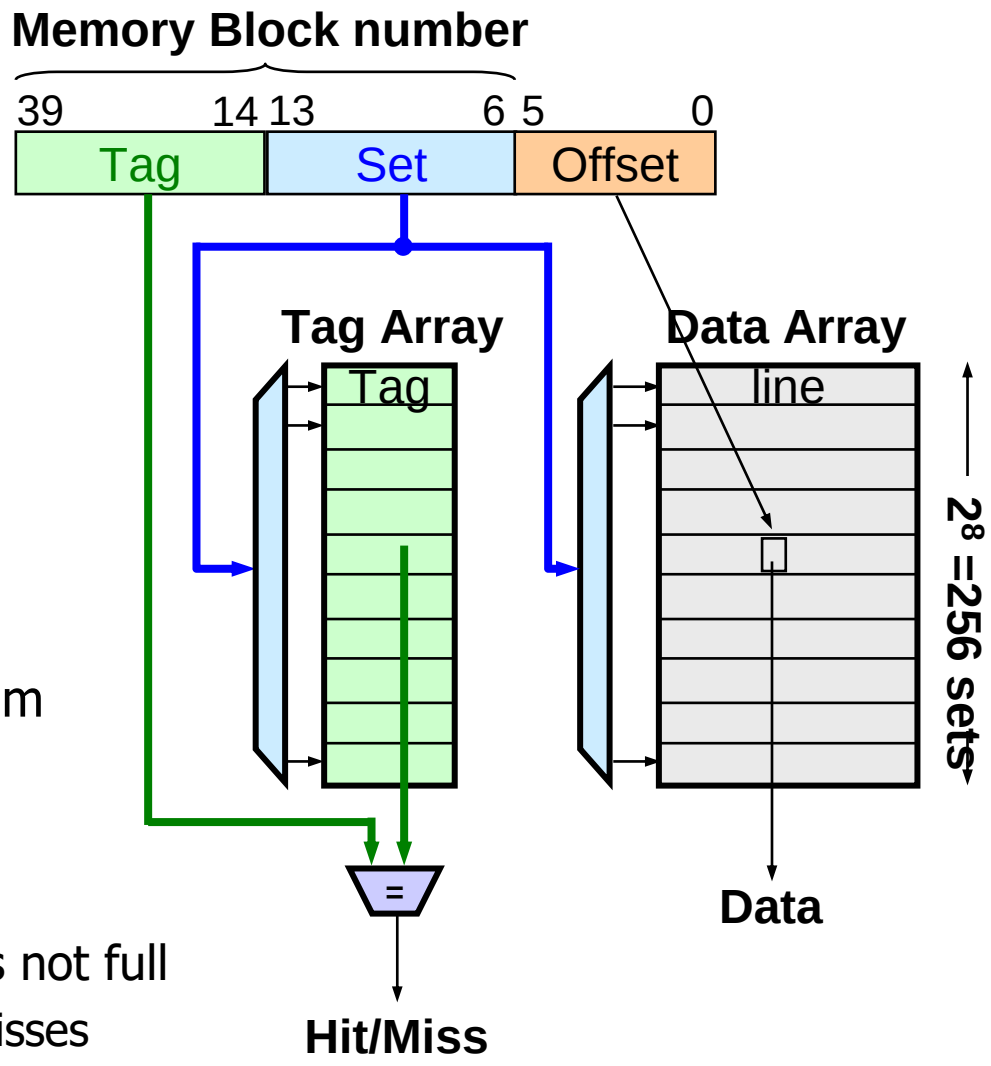
Direct Map Cache

- **Direct Map cache:** each memory block is mapped to a specific cache line
- **The block number is split into *set* and *tag***
  - Set – the *n* lower bits of the memory block number
    - Determines to which of the 2^n cache lines the memory block is mapped to
  - Tag – the upper memory block number bits
    - Compared to only to the tag stored in the cache for the given set ⇒ a single comparator needed



Direct Map Cache

- **Cache size = #sets × block size**
  - In the example: $2^8 \times 2^6$ Bytes = 2^{14} Bytes
- **Two memory blocks with the same set bits are mapped to the same cache line**
  - Only one can reside in the cache at a given time
  - The minimum distance between conflicting memory blocks is the cache size (2^{14} Bytes in the example)
- **Advantages**
  - Low power and area
  - Easy replacement algorithm – only one possible victim
- **Disadvantage – lower hits rate than a F.A. cache**
  - Line replacements due to "set conflict misses"
    - A victim may be needed even the when the cache is not full
    - Unlike Fully Associative cache that has only capacity misses



Direct Map Cache – Example

Line Size: 32 bytes ⇒ 5 Offset bits

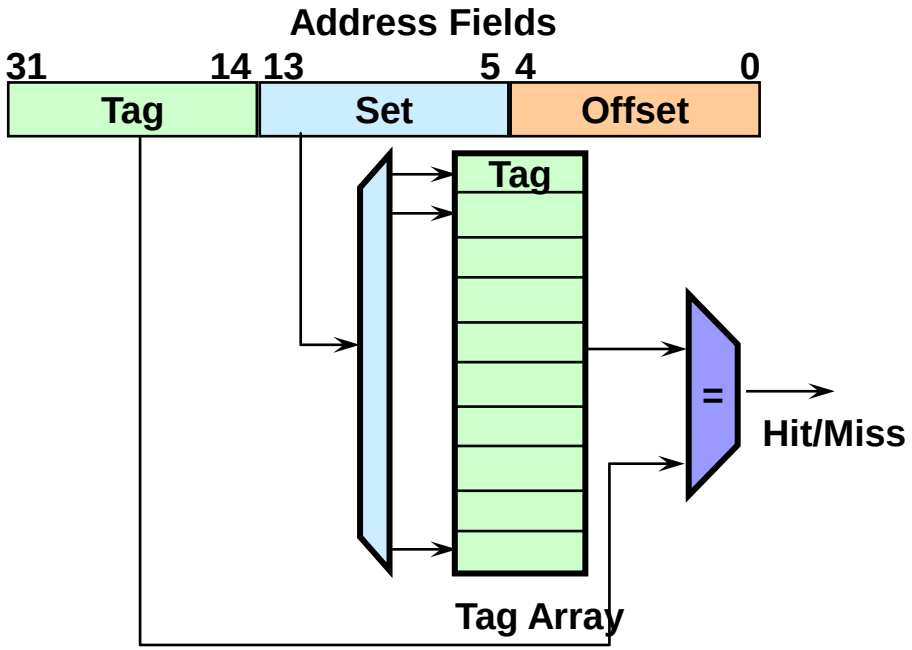
Cache Size: 16KB = 2^{14} Bytes

#lines = cache size / line size
= $2^{14}/2^5=2^9=512$

#sets = #lines

#set bits = 9 bits

#Tag bits
= 32 – (#set bits + #offset bits)
= 32 – (9+5) = 18 bits



Lookup Address: 0x12345678

0001 0010 0011 0100 0101 0110 0111 1000

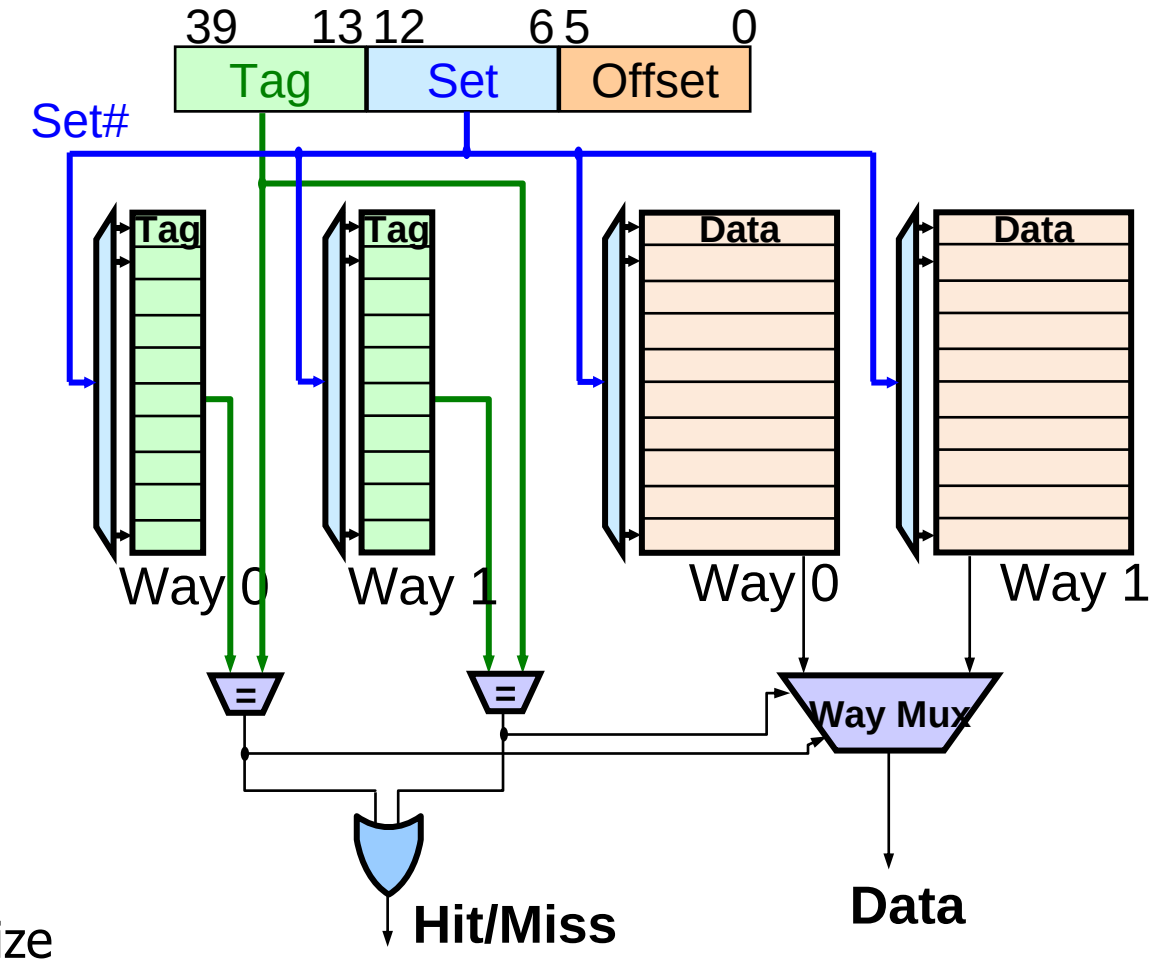
tag= 0x048D1

set= 0x0B3

offset= 0x18

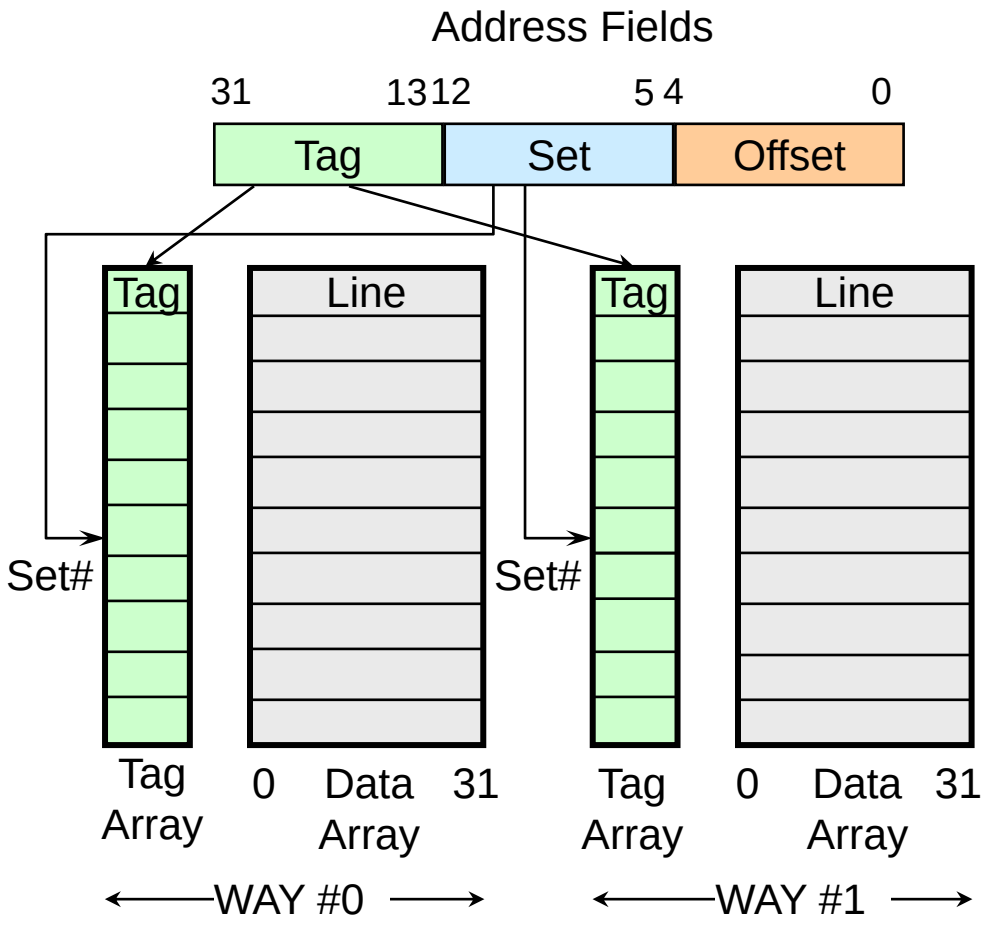
2-Way Set Associative Cache

- **Split the cache lines to way 0 and way 1**
  - Compared to a direct map cache of the same size
    - Each way has half the sets
    - The set field is one bit shorter
    - The tag field is one bit longer
  - Each memory block can reside in the specific set, either in way 0 or in way 1
  - 2 comparators
  - Way mux selects data between the 2 ways
- **Cache size = #ways × #sets × block size**
  - In the example: $2 \times 2^7 \times 2^6$ Bytes = 2^{14} Bytes
- **Two memory blocks with the same set bits can still reside in the cache**
  - The minimum distance between conflicting memory blocks is the <u>way</u> size = ½ the cache size
  - Fewer cases of "set conflict misses"



2-Way Set Associative Cache

- **Each set holds two lines (*way 0* and *way 1*)**
  - Each block can be mapped into one of two lines in the appropriate set



Example:

Line Size: 32 bytes = 2^5 bytes

Cache Size 16KB = 2^{14} bytes

#lines = cache size / line size =
$2^{14} / 2^5 = 2^9$ =512 lines

#ways 2

#sets = #lines/#ways = 256 = 2^8 sets

Offset bits $\log_2$(line size) = 5 bits

Set bits $\log_2$(#sets) = 8 bits

Tag bits 32 – (5+8) = 19 bits

Address 0x12345678

0001 0010 0011 0100 0101 0110 0111 1000

Offset: 1 1000 = 0x18

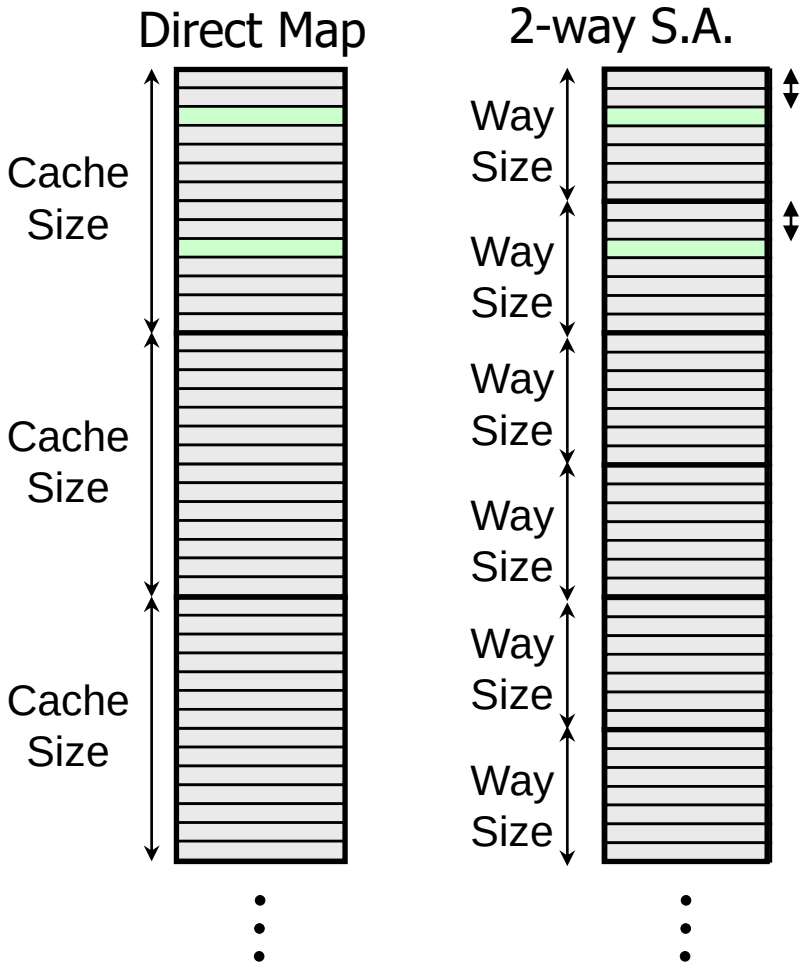
Set: 1011 0011 = 0x0B3

Tag: 000 1001 0001 1010 0010 = 0x091A2

2-Way Set Associative vs. Direct Map Cache

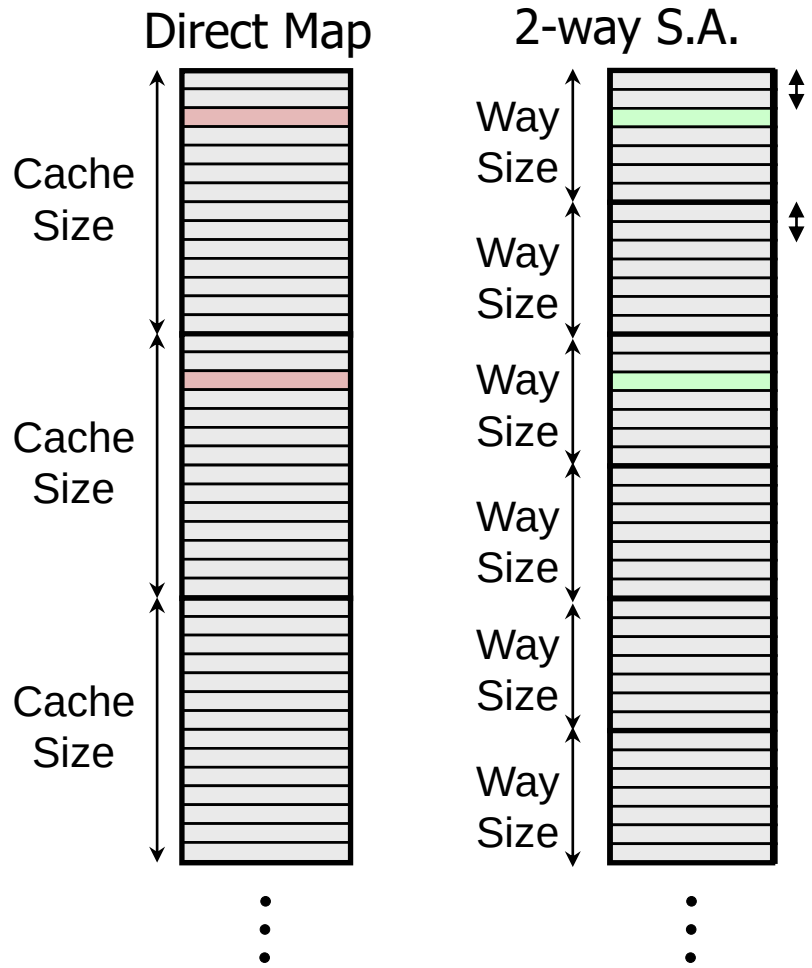
Different set, so both can reside in the cache

Same set, still both can reside in the cache



Same set, both <u>cannot</u> reside in the cache

Same set, still both can reside in the cache



Cache Replacement

- **Direct map**
  - A new block is mapped to a single line in the cache
  - Old line is evicted (re-written to memory if needed)
- **N-way set associative cache**
  - Choose a victim from all ways in the appropriate set
- **Replacement algorithms**
  - Optimal replacement algorithm
  - FIFO – First In First Out
  - Random
  - LRU – Least Recently used
- **LRU is the best**
  - but not that much better even than random

LRU Implementation

- **2 ways**
  - 1 bit per set to mark the latest way accessed in the set
  - Evict the way which is not pointed by the bit
- **k-way set associative LRU**
  - For each set maintain the way access order ⇒ per set: $\log_2 k$ bit counter per way
    - E.g.: 4 ways: 4×2 bits per set; 8 ways: 8×3 bits per set
    - Many bits, complicated update scheme
  - When way *i* is accessed

```
X = Counter[i]
Counter[i] = k-1 // make way i the MRU way
for (j = 0 to k-1)
        if ((j≠i) AND (Counter[j]>X)) Counter[j]--;
```

  - When replacement is needed: evict way with counter = 0

Initial State

Way	0	1	2	3
Count	0	1	2	3

Access way 2

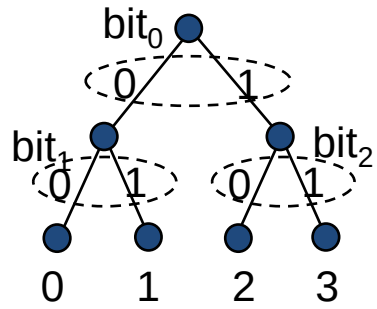
Way	0	1	2	3
Count	0	1	3	2

Access way 0

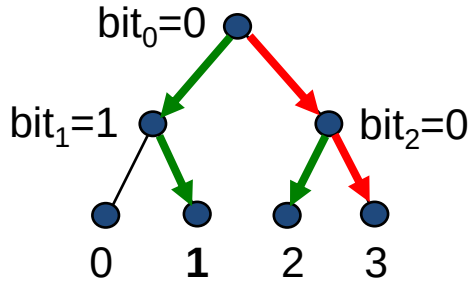
Way	0	1	2	3
Count	3	0	2	1

Pseudo LRU (PLRU)

- **PLRU records a partial order using a tree structure**
  - The ways of a set are in the leaves; the PLRU bits are the internal nodes
    - For *n* leaves (ways) there are *n*-1 internal nodes (PLRU bits)
- **Example: 4-ways, using 3 bits per-set:**
  - Bit_0 specifies the access order between ways {0,1} and {2,3}
  - Bit_1 specifies the access order between ways 0 and 1
  - Bit_2 specifies the access order between ways 2 and 3



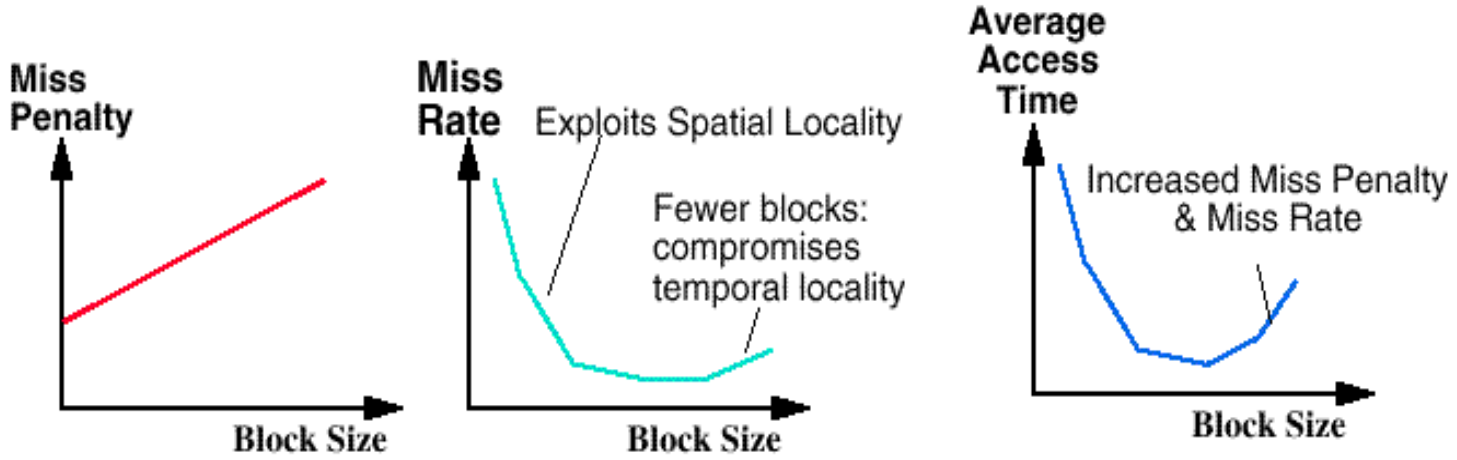
- **When accessing a way update the PLRU bits to point to it (0-left, 1-right)**
  - E.g., after accessing ways 0→3→2→1 the tree looks as follows:
    - bit_0=0 points to the pair with way 1 (the MRU way)
    - bit_1=1 points to way 1 (the MRU way)
    - bit_2=0 points to way 2, which was accessed after way 3
  - The way access order information is partial
    - Cannot say if way 0 was accessed before/after ways 2 and 3



- **Victim selection: follow the opposite directions pointed by the PLRU bits (0-right, 1-left)**
  - Way 3 selected as victim, even though way 0 is the LRU way

Cache Line Size

- **Larger line size takes advantage of spatial locality**
  - Too big blocks: may fetch unused data
  - Possibly evicting useful data ⇒ miss rate goes up
- **Larger line size means larger miss penalty**
  - Takes longer time to perform a cache line fill
    - Using critical chunk first reduces the issue
  - Takes longer to evict (when using write back update policy)



Ave. Access Time = hit time×(1– miss rate) + miss penalty × miss rate

Effect of Cache on Performance

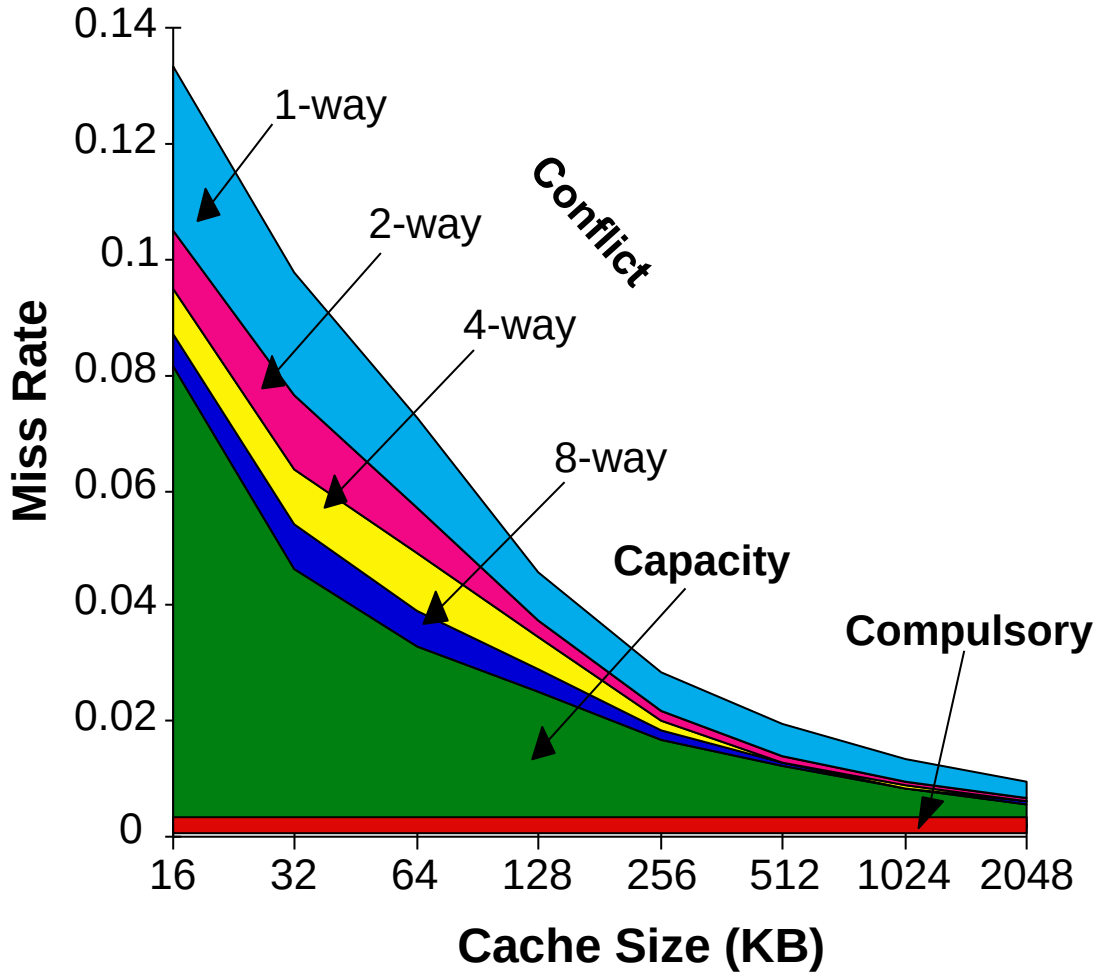
- **MPI – miss per instruction**
  - MPI = #cache misses / #instructions
    = #cache misses / #mem access ×
    #mem access / #instructions
  - More correlative to performance than cache miss rate
    - Takes into account also frequency of memory accesses

- **Memory stall cycles**
  = #memory accesses × miss rate × miss penalty
  = IC × MPI × miss penalty

- **CPU time**
  = (CPU execution cycles + Memory stall cycles) × cycle time
  = IC × ($CPI_{execution}$ + MPI × Miss penalty) × cycle time

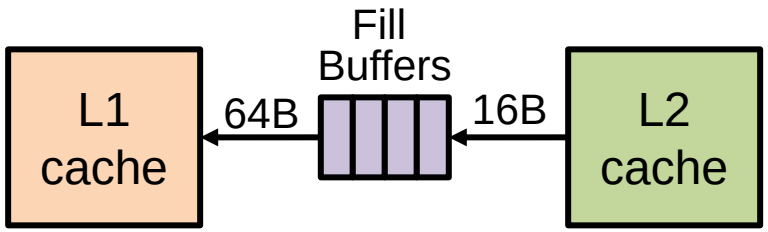
Classifying Misses: 3 Cs

- **Compulsory**
  - First access to a block
  - Solution: predict future block accesses and prefetch
- **Capacity**
  - total data size > cache size ⇒ blocks are replaced
  - Solution: increase cache size, stream buffers, prefetch
- **Conflict**
  - Block replaced due to set conflict
  - A cache with the same size, but with higher associativity would not have this miss
  - Solution: increase associativity, victim cache



Cache Line Fill and Fill Buffers

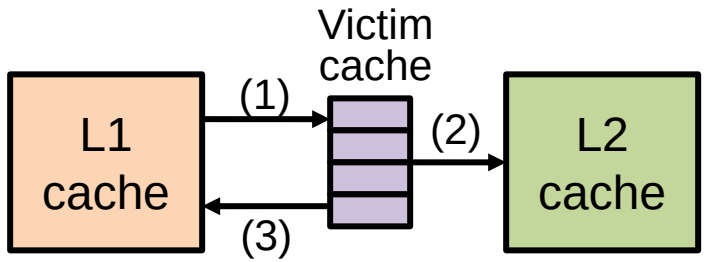
- **Before inserting a new line into cache**
  - Put new line in a fill buffer



- **The bus from the L2 to the L1 may be narrower than the cache line size**
  - E.g., for a bus width of 16B and a cache line 64B, it takes 4 bus cycles to get the full cache line
  - Get the chunk that contains the missed address (the critical chunk) first
  - The fill buffer accumulates the full line and than write it to the cache
  - Lookup the fill buffers in parallel with the cache, and supply data directly from the Fill Buffer
- **SW/compiler may provide a hint to the HW that data read by a load is read once – "streaming load"**
  - E.g., scanning-once through a large array
    $\Rightarrow$ the line is not expected to be accessed again
    $\Rightarrow$ no point in filling it into the cache, and evicting another line that could possibly be needed again
  - Get the data directly from the Fill Buffer, without ever filling the line into the cache

Victim Cache

- **Per-set pressure may be non-uniform**
  - Some sets may have more conflict misses than others
  - Solution: allocate ways to sets dynamically
  - Especially effective for direct mapped cache
    - Combine the fast hit time of a direct mapped cache and still reduce conflict misses



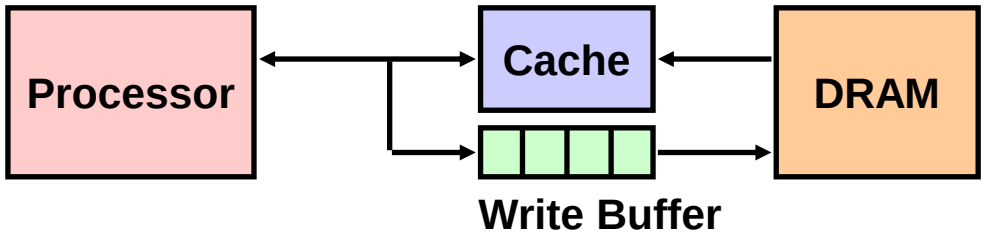
- **Victim cache gives a 2nd chance to evicted lines**
  - (1) A line evicted from L1 cache is placed in the Victim Cache
  - (2) If the Victim Cache is full ⇒ evict its LRU line to the L2 cache
- **On L1 cache lookup, lookup the Victim Cache in parallel**
  - Data supplied to the core directly from Victim Cache at same access time as L1 cache hit
  - (3) On victim cache hit move the line back from the Victim Cache into the L1 cache
    - Evicted line moved to the Victim Cache

Memory Update Policy on Writes

- **Write back – cheaper writes**
  - "Store" operations that hit the cache write only to the cache
    - Main memory is not accessed in case of write hit
    - Line is marked as *modified* or *dirty*
  - Modified cache line written to memory only when it is evicted
    - Saves memory accesses when a line is updated many times
    - On eviction, the entire line must be written to memory
      - There is no indication which bytes within the line were modified
- **Write through – cheaper evictions**
  - Stores that hit the cache write both to cache and to memory
    - Need to write only the bytes that were changed (not entire line)
  - The ratio between reads and write is ~ 4:1
    - While writing to memory, the processor goes on reading from cache
  - No need to evict blocks from the cache (never dirty)
  - Use write buffers so that don't wait for lower level memory

Write Buffer for Write Through

- **A Write Buffer between the Cache and Memory**
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory



- **Work ok if store frequency << 1/DRAM write cycle**
  - Otherwise store buffer overflows no matter how big it is
- **Write combining**
  - combine writes in the write buffer
- **On cache miss need to lookup write buffer**

Cache Write Miss

- **The processor is not waiting for data**
  - ⇒ continues its work

- **Option 1: Write allocate: fetch the line into the cache**
  - Goes with write back policy, assuming more writes are needed
  - hoping that subsequent writes to the line hit the cache

- **Option 2: Write no allocate: do not fetch line into cache**
  - Goes with write through policy
  - subsequent writes would update memory anyhow

Prefetching

- **Hardware Data Prefetching – predict future data accesses**
  - Next sequential / Streaming
    - Triggered by an ascending access to very recently loaded data
    - Fetches the next line, assuming a streaming load
  - Stride prefetcher
    - Tracks individual load instructions, detecting a regular stride
    - Prefetch address = current address + stride
    - Detects strides of up to 2K bytes, both forward and backward
  - General pattern prefetcher
    - Identifies patterns of Load address distances
- **Data has to be prefetched before it is needed**
  - The prefetcher aggressiveness has to be tuned
  - How fast and how far ahead to issue the prefetch request, such that data arrives on time, before it is needed

Prefetching (cont.)

- **Prefetching relies on extra memory bandwidth**
  - Too aggressive / inaccurate prefetching slows down demand fetches
    - Hurts performance
- **Software Prefetching**
  - Special prefetching instructions that cannot cause faults
- **Instruction Prefetching**
  - On a cache miss, prefetch sequential cache lines into stream buffers
  - Branch predictor directed prefetching
    - Let branch predictor run ahead

Code Optimizations: Merging Arrays

- **Merge 2 arrays into a single array of compound elements**

```
/* Before: two sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: One array of structures */
struct merge {
        int val;
        int key;
} merged_array[SIZE];
```

- **Reduce conflicts between `val` and `key`**
- **Improves spatial locality**

Code Optimizations: Loop Fusion

- **Combine 2 independent loops that have same looping and some variables overlap**
- Assume each element in `a` is 4 bytes, 32KB cache, 32 B / line

```
for (i = 0; i < 10000; i++)
  a[i] = 1 / a[i];
for (i = 0; i < 10000; i++)
  sum = sum + a[i];
```

In the example only load accesses are taken into account for hit rate calculation

  - First loop: hit 7/8 of iterations
  - Second loop: array > cache ⇒ same hit rate as in 1st loop

- Fuse the loops

```
for (i = 0; i < 10000; i++) {
  a[i] = 1 / a[i];
  sum = sum + a[i];
}
```

  - First line: hit 7/8 of iterations
  - Second line: hit all

Code Optimizations: Loop Interchange

- **2-dimension array in memory:**

`x[0][0] x[0][1] … x[0][99] x[1][0] x[1][1] …`

```
/* Original program: access are 100 bytes apparat */
/* x[0][0] x[1][0] … x[4999][0] x[0][1] x[1][1] … */
for (j = 0; j < 100; j++)
    for (i = 0; i < 5000; i++)
          x[i][j] = 2 * x[i][j];
```

```
/* Reversing the loops order: access are adjacent */
/* Improved spatial locality                      */
for (i = 0; i < 5000; i++)
    for (j = 0; j < 100; j++)
          x[i][j] = 2 * x[i][j];
```

Improving Cache Performance

- **Reduce cache miss rate**
  - Larger cache
  - Reduce compulsory misses
    - Larger Block Size
    - HW Prefetching (Instr, Data)
    - SW Prefetching (Data)
  - Reduce conflict misses
    - Higher Associativity
    - Victim Cache
  - Stream buffers
    - Reduce cache thrashing
  - Compiler Optimizations
- **Reduce the miss penalty**
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit under Miss, Miss under Miss)
  - Second Level Cache
- **Reduce cache hit time**
  - On-chip caches
  - Smaller size cache (hit time increases with cache size)
  - Direct map cache (hit time increases with associativity)

Separate Instruction / Data Caches

- **Support parallel instruction fetch and data access**
  - Fetch and data access are done at different pipe-stages
  - Locate the I-cache close to the fetch logic, and the D-cache close to the memory logic
- **Each cache has its own characteristics**
  - Instruction cache is a read only cache, while data cache supports both reads and writes
    - No need to write back instruction cache line to L2 when evicted
  - Different access patterns ⇒ different prefetch schemes

- **When separating the caches, need to support self modifying code**
  - A Store instruction may write to a memory location that holds code, and modify the code
  - Whenever executing a Store ⇒ snoop the instruction cache
    - Add snoop port to the I-cache tag array – otherwise snoops would stall fetch
  - If the snoop hits on the I-cache ⇒ invalidate the line
    - Flush the pipeline – it main contain stale code, and re-fetch the instructions following the store

Non-Blocking Cache

- **Hit Under Miss**
  - Allow cache hits while one miss is in progress
  - Another miss has to wait
  - Relevant for an out-of-order execution CPU

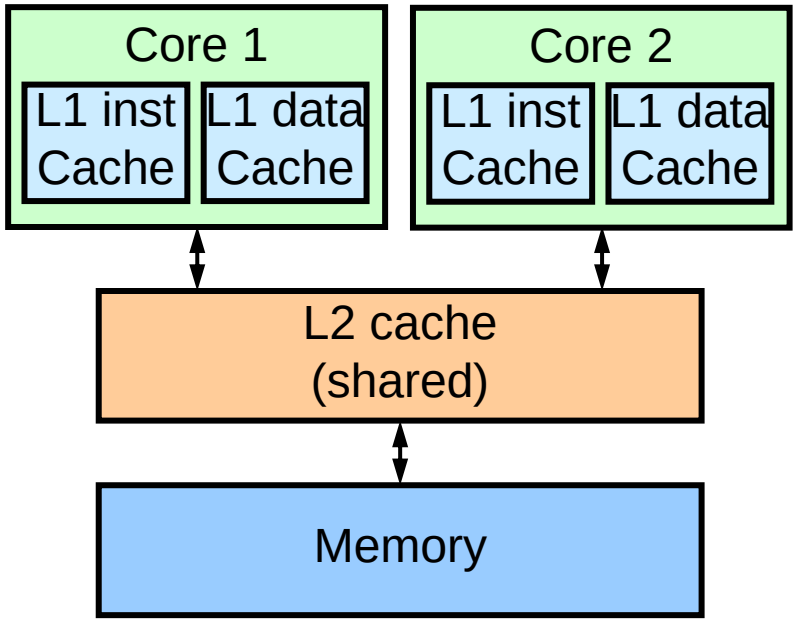
- **Miss Under Miss, Hit Under Multiple Misses**
  - Allow hits and misses when other misses in progress
  - Memory system must allow multiple pending requests
  - Manage a list of outstanding cache misses
    - When miss is served and data gets back, update list

Multi-ported Cache

- **N-ported cache enables *n* accesses in parallel**
  - Parallelize cache access in different pipeline stages
  - Parallelize cache access in a super-scalar processors
- **About doubles the cache die size**
- **Possible solution: banked cache**
  - Each line is divided to *n* banks
  - Can fetch data from $k \leq n$ different banks in possibly different lines

L2 Cache

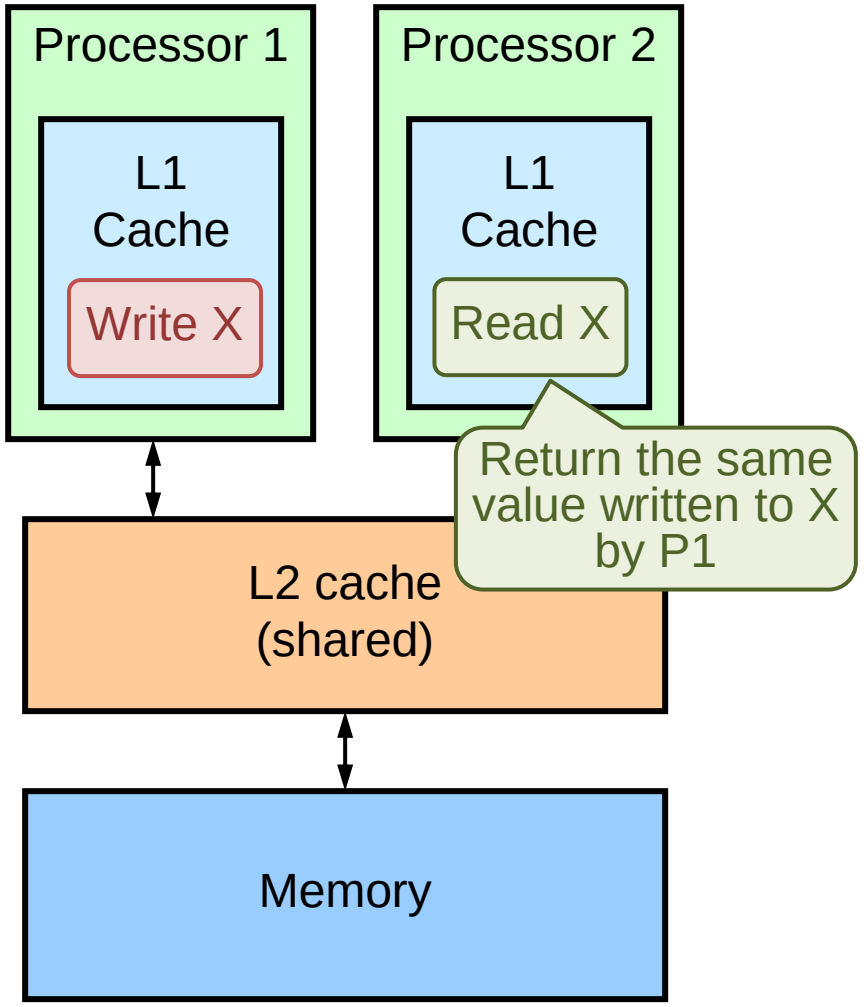
- **L2 cache is bigger, but with higher latency**
  - Reduces L1 miss penalty – saves access to memory
  - L2 cache may private per core, or shared by a few cores
  - L2 holds both code and data



- **L2 inclusive of all the L1s in all the cores**
  - All addresses in L1 are also contained in L2
    - Address evicted from L2 ⇒ snoop invalidate it in L1
  - Data in L1 may be more updated than in L2
    - When evicting a modified line from L1 ⇒ write to L2
    - When evicting a line from L2 which is modified in L1
      - L2 needs to get the latest data from L1 before evicting it to memory
        ⇒ L2 sends snoop invalidate to L1, L1 respond by sending the data to L2
  - Since L2 contains L1 it needs to be significantly larger
    - e.g., if L2 is only 2× L1, half of L2 is duplicated in L1

Multi-processor System

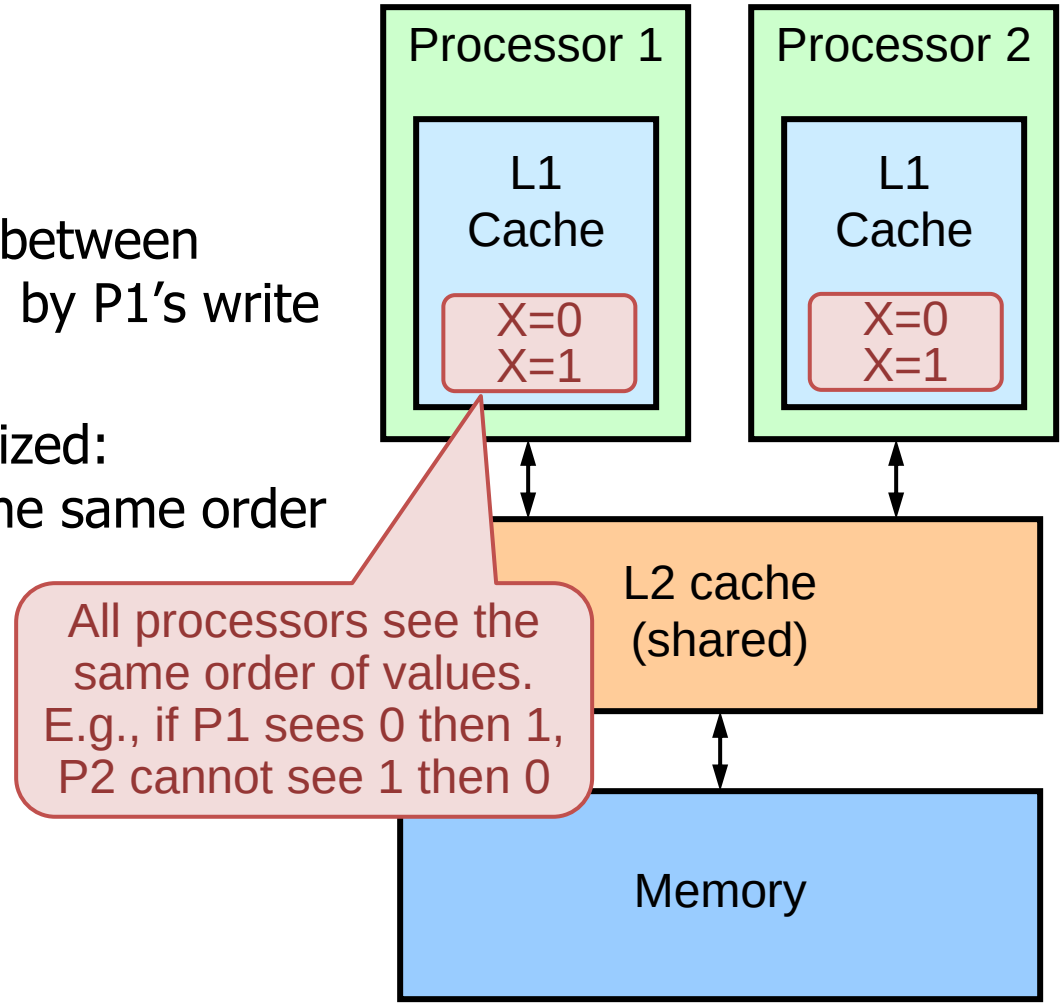
- **A memory system is *coherent* if**
  1. If P1 writes to address X,
     and later-on P2 reads X,
     and there are no other writes to X in between
     ⇒ P2's read returns the value written by P1's write



Multi-processor System

- **A memory system is *coherent* if**
  1. If P1 writes to address X,
     and later on P2 reads X,
     and there are no other writes to X in between
     ⇒ P2's read returns the value written by P1's write

  2. Writes to the same location are serialized:
     two writes to location X are seen in the same order
     by all processors

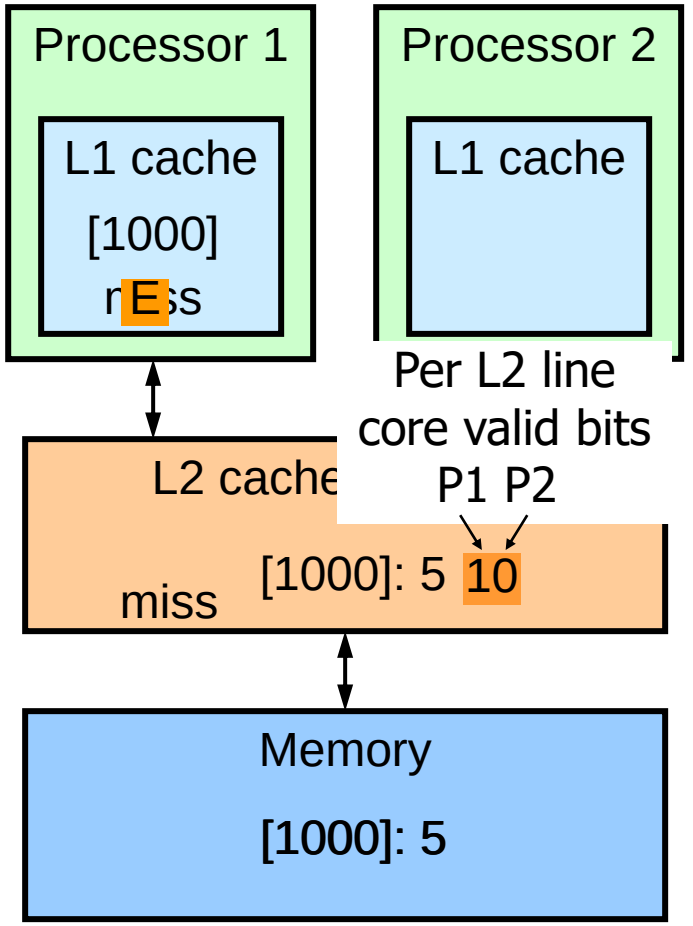


MESI Protocol

- **Each cache line can be in one of 4 states**
  - Invalid – Line's data is not valid
  - Shared – Line is valid and not dirty,
    copies may exist in other processors
  - Exclusive – Line is valid and not dirty,
    other processors do not have the line in their local caches
  - Modified – Line is valid and dirty,
    other processors do not have the line in their local caches

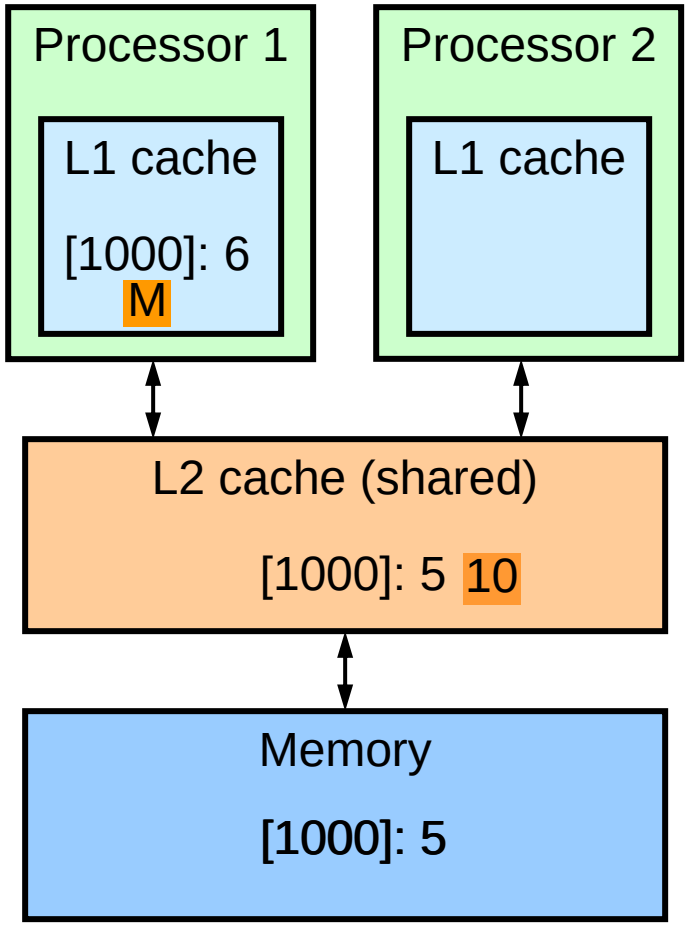
Multi-processor System: Example

- **P1 reads address 1000**



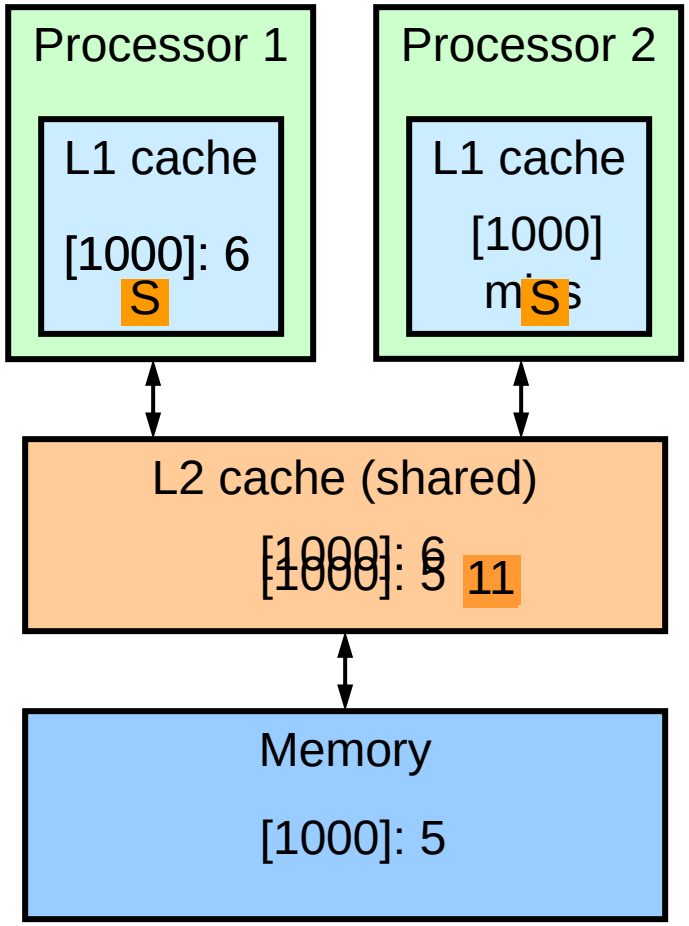
Multi-processor System: Example

- **P1 reads address 1000**
- **P1 writes address 1000**



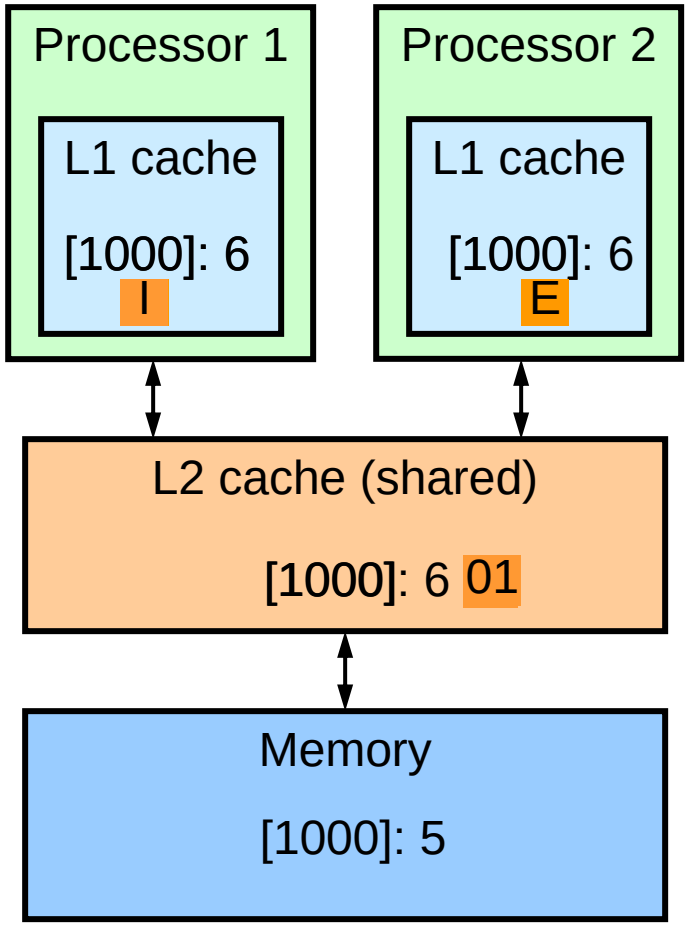
Multi-processor System: Example

- **P1 reads address 1000**
- **P1 writes address 1000**
- **P2 reads address 1000**
  - L2 snoops address 1000
  - P1 writes back address 1000
  - P2 gets address 1000



Multi-processor System: Example

- **P1 reads address 1000**
- **P1 writes address 1000**
- **P2 reads address 1000**
  - L2 snoops address 1000
  - P1 writes back address 1000
  - P2 gets address 1000
- **P2 requests for ownership with write intent for address 1000**
  - L2 snoop invalidates address 1000 in P1
  - L2 grants ownership on address 1000 to P2



Multi-processor System: Inclusion

- **L2 needs to keep track of the presence of each line in each of the processors**
  - Determine if it needs to send a snoop to a processor
  - Determine in what state to provide a requested line (S,E)
  - ⇒ Need to guarantee that the L1 caches in each processor are inclusive of the L2 cache

- **When L2 evicts a line**
  - L2 sends a snoop invalidate to all processors that have it
  - If the line is modified in the L1 cache of one of the processors (in which case it exist only in that processor)
    - The processor responds by sending the updated value to L2
    - When the line is evicted from L2, the updated value gets written to memory

MESI Protocol States

State	Valid	Modified	Copies may exist in other processors
Invalid	No	N.A.	possibly
Shared	Yes	No	possibly
Exclusive	Yes	No	No
Modified	Yes	Yes	No

- **A modified line must be exclusive**
  - Otherwise, another processor which has the line will be using stale data
  - Therefore, before modifying a line, a processor must request ownership of the line

Non-Inclusive Cache

- **An inclusive L2 cache works well when L2 >> L1**
  - Assume L1 cache size is ¼ of the L2 cache
  - ¼ of the L2 cache holds a copy of the L1 ⇒ significant overall cache capacity is wasted
  - When the ratio between L2 size and L1 size is not high ⇒ use a non-inclusive cache

- **Non-inclusive L2 cache ⇒ a Snoop Filter maintains MESI protocol**
  - The snoop filter is inclusive of the L1 caches of all cores, and maintains coherency between the cores
  - The snoop filter essentially has a tag array and per-core valid bits, but no data

- **Non-inclusive L2 cache ⇒ L1 and L2 are mutually exclusive**
  - In case of L1 miss which misses also in L2 ⇒
    bring the data to L1 only (and the tag into the snoop filter)
  - In case of L1 miss which hits in the L2 ⇒ bring data to L1, and invalidate it from L2
  - L1 victim is put into L2 ⇒ L2 behaves like a victim cache for the L1

Memory Ordering and Global Observation

- **Assume memory address A is shared by two processors, P1 and P2, and a third processor, P3, sends a Request For Ownership (RFO) for address A**
  - The snoop filter sees that address A is shared, and sends a snoop invalidate to P1 and P2
  - Since for sure the data in P1 and P2 is not modified, can the snoop filter acknowledge the RFO to P3 in parallel to sending the snoop invalidate to P1 and P2 ?
- **Getting ownership requires both data and Global observation**
  - A write to a location in memory is said to be *globally observed* when a subsequent read of the location by any processor returns the value written by the write
  - The data may be sent to P3 immediately, although in case of a non-inclusive L2, it may have to be supplied from L3 or from DRAM
  - The GO cannot be sent until the other cores are invalidated and responded to the snoops
  - Otherwise, memory ordering violation could occur: P3 may write a new value to address A, while P1 and P2 are still using the previous value of address A in their calculations
  - Memory ordering requires that any processor asking for the data of address A after P3 gets GO, must get and use the new value of address A

Backup

Cache Optimization Summary

	Technique	Miss Rate	Miss Penalty	Hit Time
miss rate	Larger Block Size	+	–	
	Higher Associativity	+		–
	Victim Caches	+		
	Pseudo-Associative Caches	+		
	HW Prefetching of Instr/Data	+		
	Compiler Controlled Prefetching	+		
	Compiler Reduce Misses	+		
miss penalty	Priority to Read Misses		+	
	Sub-block Placement		+	+
	Early Restart & Critical Word 1st		+	
	Non-Blocking Caches		+	
	Second Level Caches		+	

X86 Memory Ordering Rules

1. Loads are not reordered with other loads
2. Stores are not reordered with other stores
3. Stores are not reordered with older loads
4. Loads may be reordered with older stores to different locations but not with older stores to the same location
5. In a multiprocessor system, memory ordering obeys causality
   (memory ordering respects transitive visibility)
6. In a multiprocessor system, stores to the same location have a total order
7. In a multiprocessor system, locked instructions have a total order
8. Loads and stores are not reordered with locked instructions

X86 Memory Ordering Examples

- **Loads are not reordered with other loads and stores are not reordered with other stores**
  - I.e., loads are seen in program order, and stores are seen in program order
  - Initially x = y = 0
  - r1 == 1 and r2 == 0 is not allowed
  - The stores cannot be reordered at processor 0 and the loads cannot be reordered at processor 1

Processor 0	Processor 1
Store [X] ← 1 //S1 Store [Y] ← 1 //S2	r1 ← Load [Y] //L1 r2 ← Load [X] //L2

  - r1 == 1 implies that the store to Y (S2) executed before the load from Y (L1)
  - In turn, this implies that the store to X (S1) executed before the load from X (L2), which implies r2 == 1

- **Stores are not reordered with older loads**
  - Initially x = y = 0
  - r1 == 1 and r2 == 1 is not allowed
  - Store S1 is not reordered with the load L1
  - Store S2 is not reordered with the load L2
  - If r1 == 1, then L1 must see the store S2 (S2 is before L1)
  - This implies that the load L2 appears to be ordered before the store S1, implying r2 == 0

Processor 0	Processor 1
r1 ← Load [X] //L1 Store [Y] ← 1 //S1	r1 ← Load [Y] //L2 Store [X] ← 1 //S2

X86 Memory Ordering Examples

- **Loads may be reordered with older stores to different locations**
  - Initially x = y = 0
  - r1 == 0 and r2 == 0 is allowed
  - At processor 0, the load L1 and the store S1 are to different locations ⇒ may be reordered
  - Similarly, at processor 1, L2 and S2 may be reordered
  - Any interleaving of the operations is allowed. One such interleaving has the two loads preceding the two stores. This would result in each load returning value 0.

Processor 0	Processor 1
Store [X] ← 1 //S1 r1 ← Load [Y] //L1	Store [Y] ← 1 //S2 r2 ← Load [X] //L2

- **Loads are not reordered with older stores to the same location**
  - nitially x = y = 0
  - Must have r1 == 1 and r2 == 1

Processor 0	Processor 1
Store [X] ← 1 //S1 r1 ← Load [X] //L1	Store [Y] ← 1 //S2 r2 ← Load [Y] //L2

X86 Memory Ordering Examples

- **Stores are transitively visible**
  - i.e., stores that are causally related appear to execute in an order consistent with the causal relation
  - Initially x = y = 0
  - r1 == 1, r2 == 1, r3 == 0 is not allowed

Processor 0	Processor 1	Processor 2
Store [X] ← 1 //S1	r1 ← Load [X] //L1 Store [Y] ← 1 //S2	r2 ← Load [Y] //L2 r3 ← Load [X] //L3

  - If r1 == 1, store S1 causally precedes load L2 and, therefore, the subsequent store M3.
  - Memory ordering rules ensure that S1 appears to execute before S2 with respect to processor 2; It also prevents the loads L2 and L3 from being reordered
  - Since r2 == 1, S2 appears to precede L2 and, therefore, S1 must appear to precede L2 at processor 2. This implies that r3 == 1.

X86 Memory Ordering Examples

- **Total order on stores to the same location**
  - Any two stores to the same memory location (even by different processors) must appear to all processors to execute in the same order

Processor 0	Processor 1	Processor 2	Processor 3
Store [X] ← 1 //S1	Store [X] ← 2 //S2	r1 ← Load [X] //L1 r2 ← Load [X] //L2	r3 ← Load [X] //L3 r4 ← Load [X] //L4

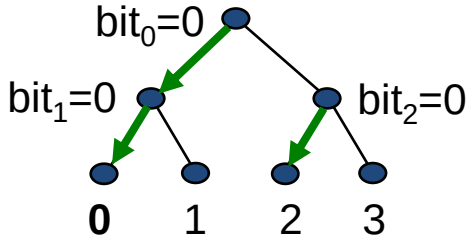
  - r1 == 1, r2 == 2, r3 == 2, r4 == 1 is not allowed
  - Processor 2 and processor 3 must agree on the order of the stores S1 and S2
  - Suppose S1 appears to execute first (i.e., S2 overwrites x with 2)
  - The loads within each processor (1 and 2) cannot be reordered
  - Since r3 == 2, the store S2 must have preceded the load L3
  - Since S1 appears to execute before S2, L4 cannot return 1, the value written by S1

Observation and Global Observation

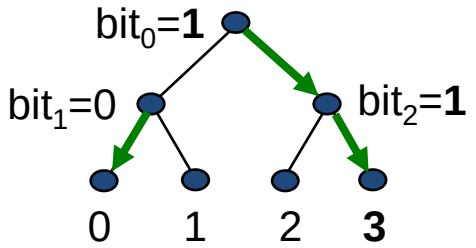
- A precise definition of visibility of a memory access is necessary for defining an ordering model
- Define a point of "Global Observation" (aka GO) when new data is visible from writes
- A write to a location in memory is said to be *observed* by a processor when a subsequent read of the location by the same processor returns the value written by the write
- A write to a location in memory is said to be *globally observed* when a subsequent read of the location by any processor returns the value written by the write
- A read from a location in memory is said to be *observed* by a processor when a subsequent write to the location by the same processor does not affect the value returned by the read
- A read from a location in memory is said to be *globally observed* when a subsequent write to the location by any processor does not affect the value returned by the read
- The concept of *observation* applies to both shared and non-shared memory, while the concept of *global observation* only applies to shared memory

PLRU Example: Update

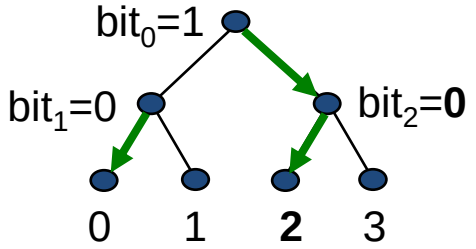
- **PLRU tree after accessing ways 0→3→2→1**
  - Assume initially all 3 PLU bits are 0s
  - After accessing way 0, no change to the bits



  - After accessing way 3: set bit_0=1, bit_2=1



  - After accessing way 2: set bit_2=0



  - After accessing way 1: set bit_0=0, bit_2=1