An Introduction to the Memory Hierarchy

CS 532

Karen L. Karavanic

Includes material from Hennessy & Patterson, Bryant & O'Hallaron

Portland State
UNIVERSITY

The Single Core Era
The Memory Hierarchy

- Why have a hierarchy of memory?
- How does caching work?
- How does caching affect performance ?

Locality of Reference (review)

- Programs tend to use data and instructions with addresses near or equal to those they have recently used
- Temporal locality: recently referenced addresses are likely be referenced again soon
  - Ex: Loops
- Spatial locality: if an address is referenced, *nearby* addresses are likely to be referenced soon
  - Ex: reference successive elements of an array

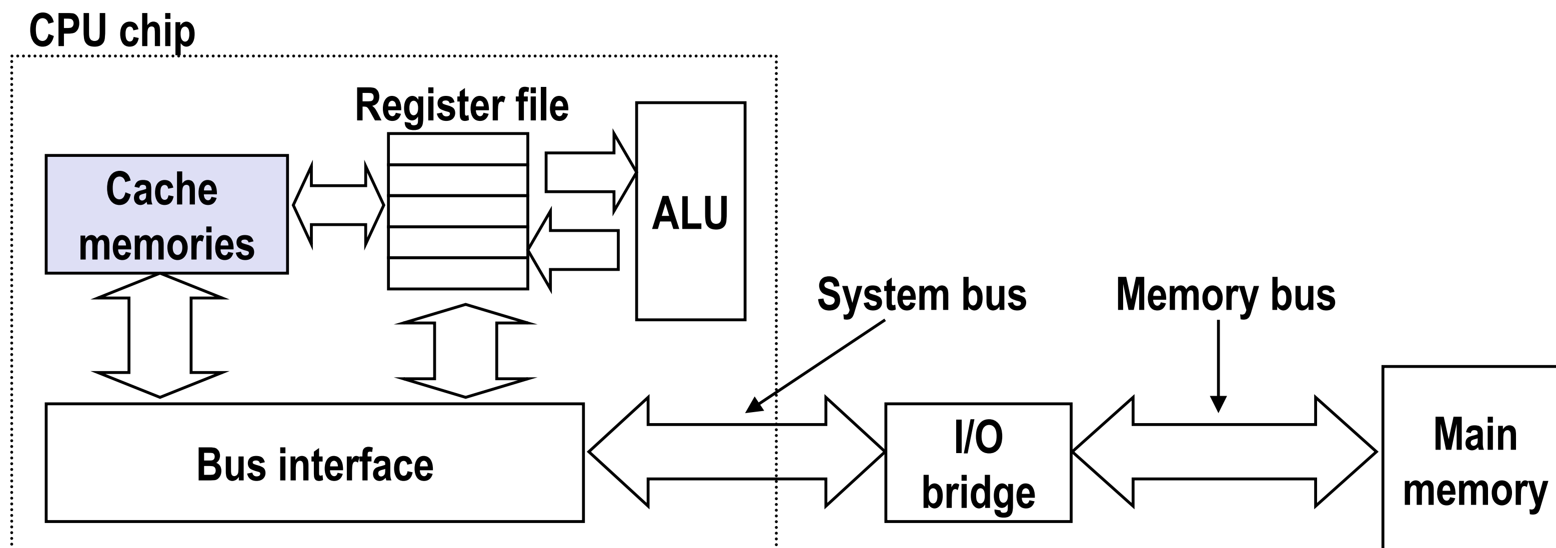
Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern). **Spatial locality**
  - Reference variable `sum` each iteration. **Temporal locality**
- **Instruction references**
  - Reference instructions in sequence. **Spatial locality**
  - Cycle through loop repeatedly. **Temporal locality**

Cache Memories

- Cache memories **are small, fast SRAM-based memories managed automatically in hardware.**
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.**
- **Typical single core system structure:**

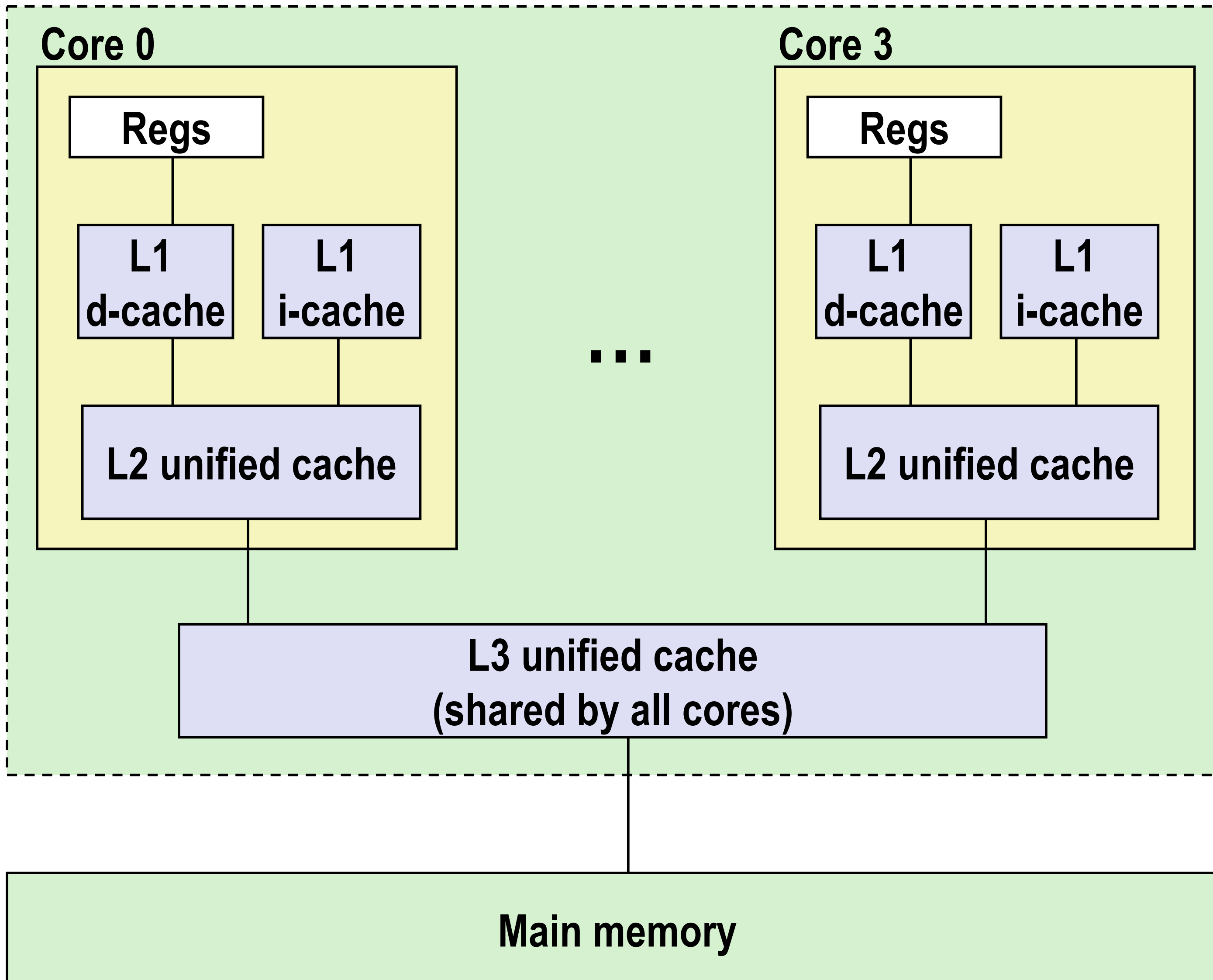


From Bryant & O'Hallaron, Computer Systems

Caches

- *Cache*: A smaller, faster storage device used to hold temporary copies of data fetched from a larger, slower storage device
- Memory hierarchies
  - For each *k*, the faster, smaller device at level *k* serves as a cache for the larger, slower device at level *k + 1*
  - Programs tend to access data at level k more than the data at k + 1 (why?)
- Goals:
  - Hide latency of storage devices
  - Decrease storage cost

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

From Bryant & O'Hallaron, Computer Systems

General Cache Concepts

Cache: 4 | 9 | 10 | 3

Smaller, faster, more expensive memory caches a subset of the blocks

10

Data is copied in block-sized transfer units (called a *block* or *line*)

Memory:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Larger, slower, cheaper memory viewed as partitioned into "blocks"

Modified From Bryant & O'Hallaron, Computer Systems

General Cache Concepts: Cache Hit

Request: 14

Cache

8	9	14	3

Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

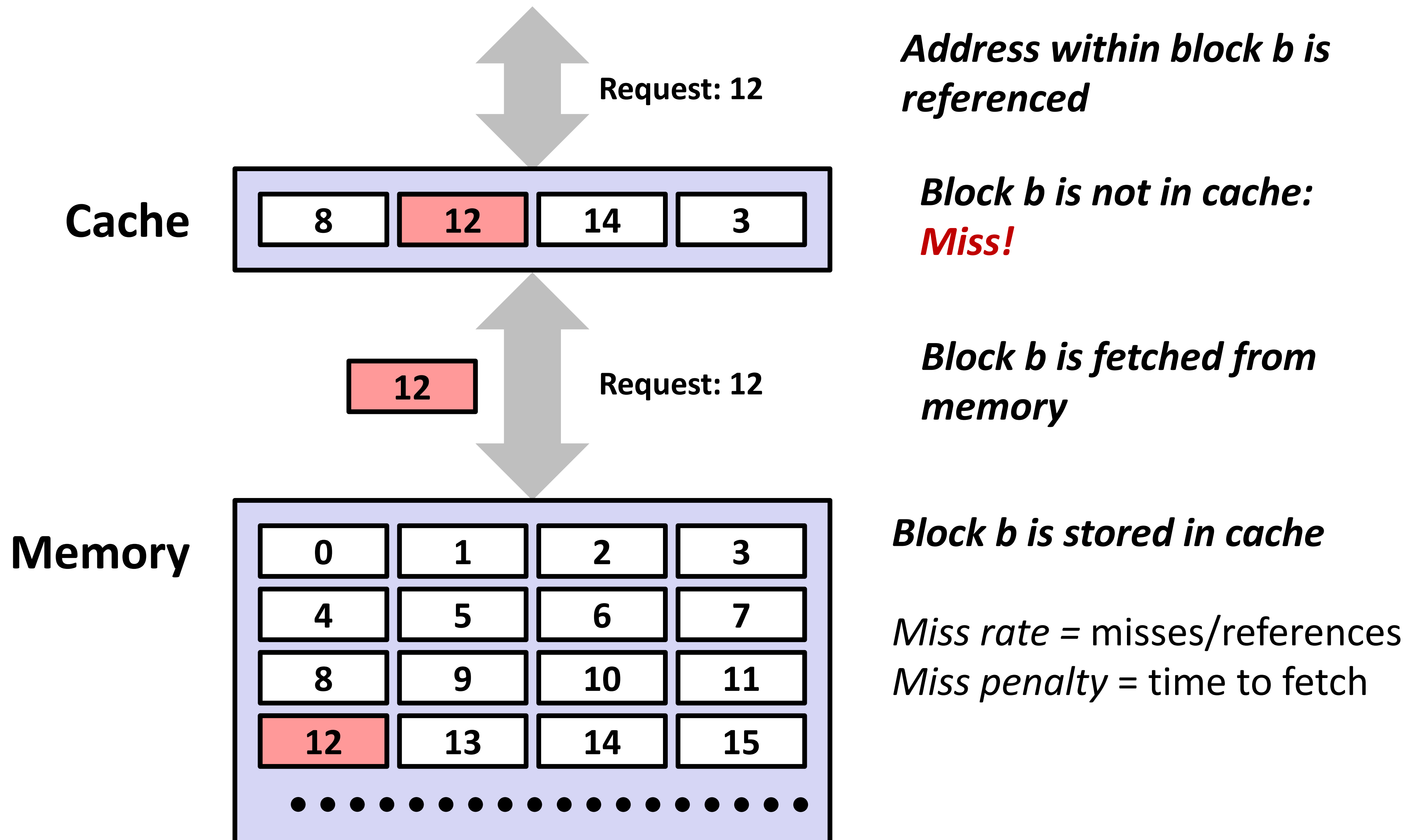
● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ●

Address within block b is referenced

Block b is in cache: Cache Hit!

Modified From Bryant & O'Hallaron, Computer Systems

General Cache Concepts: Miss



Address within block b is referenced

Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache

- *Miss rate* = misses/references
- *Miss penalty* = time to fetch

From Bryant & O'Hallaron, Computer Systems

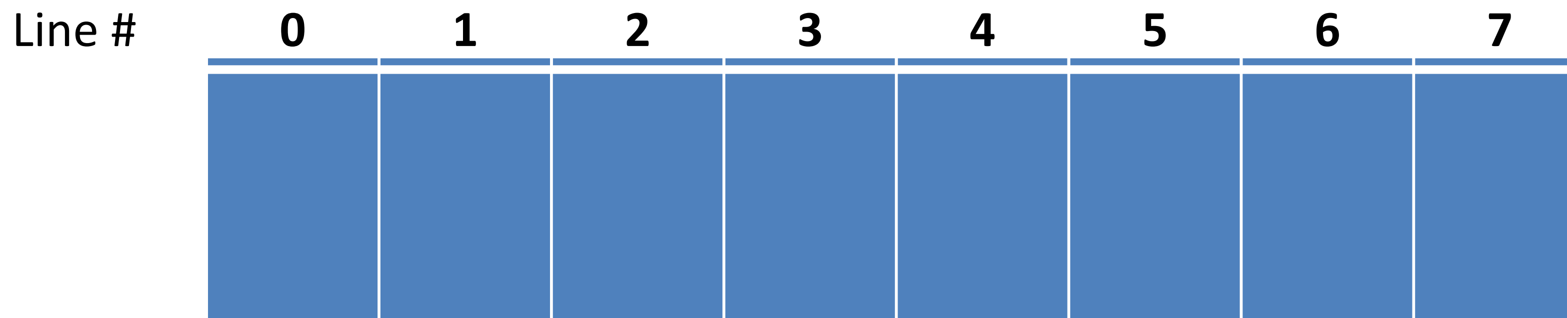
Questions

- Why are cache lines larger than one *int* or one *float*?
- We have seen L1, L2, and L3 cache. What does LLC stand for?
- In 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second level cache. What are the miss rates?
- How might cache performance be affected as we increase the value of x in this code?

```
sum = 0;
for (i = 0; i < n; i = i + x)
 sum += a[i];
return sum;
```

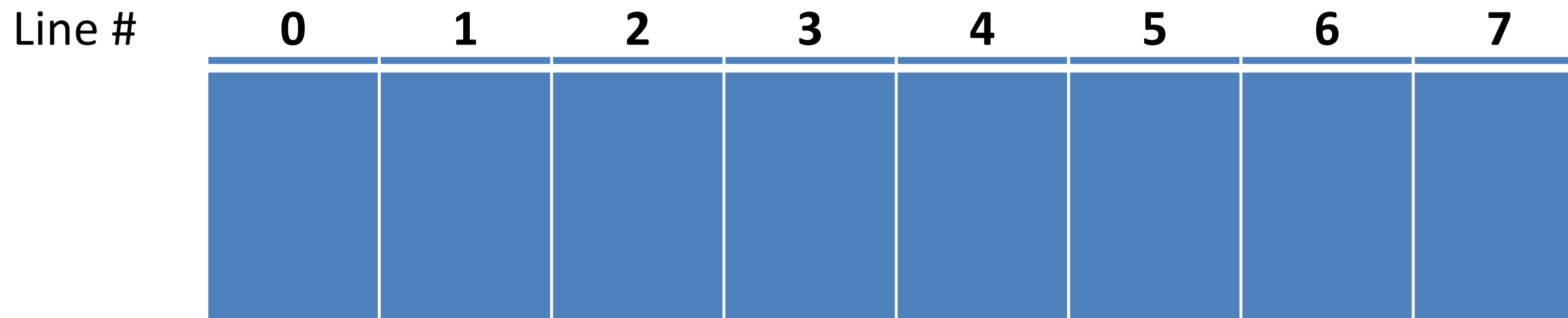
Cache Design: Key Questions

- Block Placement:
  - Given a cache with *n* lines, where can a particular block e.g. block address=12, be placed?
- Block Identification:
  - How do we find a particular block?
- Block Replacement:
  - Which block to evict and replace on a miss?
- Write Strategy:
  - What happens on a write?

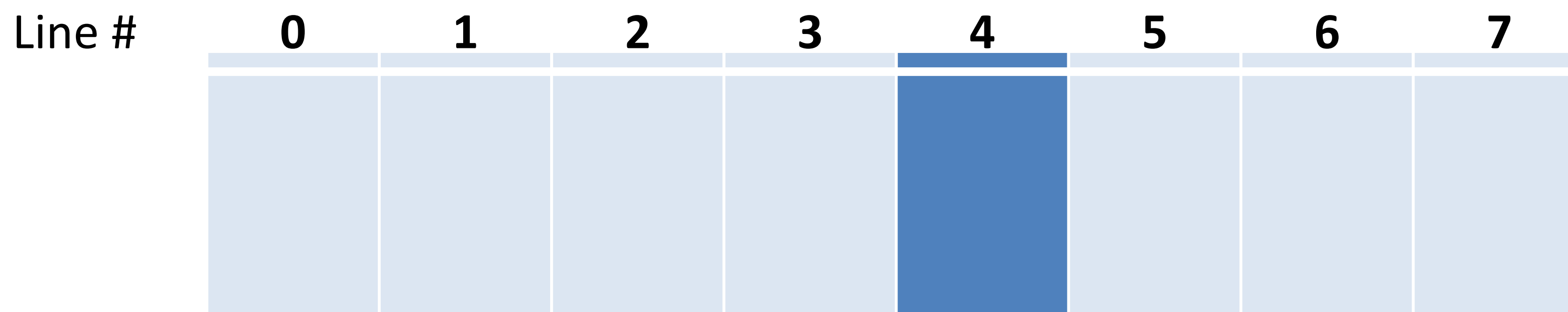


Block Placement: 12

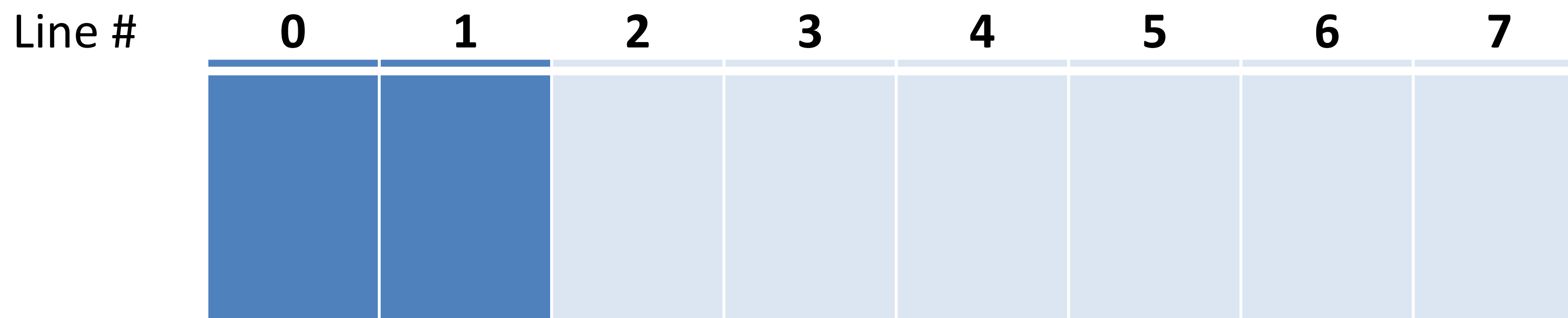
- Fully Associative: anywhere



- Direct mapped: 12 mod 8 = 4



- *n*-way Set Associative: 12 mod *n* = Set 0; Set size = 8/*n*



Block Identification

- Fully Associative: Check every cache line in the cache for a match using its *tag*:
  - Address: address of currently resident block
  - Valid bit: is there valid data?
- Direct Mapped: Compute the one cache line from the address and check for match
- Set Associative: Compute the Set, check every cache line in the set for a match
  - Index: set number

Block Replacement

- Direct Mapped: if the one eligible line is in use, evict that block
- Fully or Set Associative:
  - Random
    - Inexpensive
  - Least Recently Used (LRU)
    - Temporal and some Spatial Locality
    - Expensive -> Use Approximation
  - First In First Out (FIFO)
    - Captures some Temporal Locality
    - Less Expensive

Write Strategy

- Locate block within the cache
- Write through OR Write back
- Write through – data written to the cache AND memory simultaneously
  - Easier to implement
  - Keeps cache consistent with lower levels of hierarchy
- Write back – Data written ONLY to the cache. Data is written to memory when the block is replaced in / evicted from the cache
  - Dirty bit: set on write to signal need to write back to memory on replacement
  - Reduces memory traffic

Performance Implications

- Average memory access time

  = Hit time + Miss Rate * Miss penalty
- Reducing the Miss Rate
  - Larger block size, larger cache size, larger associativity
- Reducing the Miss Penalty
  - Multilevel caches, giving reads priority over writes
- Reducing the time to Hit in the Cache
  - Avoid address translation when indexing the cache

Reducing the Miss Rate – Miss Types

- Compulsory miss
  - First time a block is accessed ("cold-start misses")
- Capacity miss
  - Occurs when the set of cache blocks needed is larger than the cache. Replacement -> extra misses
- Conflict miss
  - Too many blocks map to same set ("collision miss") -> replacement -> extra misses