The memory hierarchy: Cache placement, replacement, and write policies

Yipeng Huang
Rutgers University
April 12, 2022
Table of contents

Announcements

Cache placement policy (how to find data at address for read and write hit)
  Fully associative cache
  Direct-mapped cache
  Set-associative cache

Cache performance metrics: hits, misses, evictions
  Cache hits
  Cache misses

Cache replacement policy (how to find space for read and write miss)
  Direct-mapped cache need no cache replacement policy
  Associative caches need a cache replacement policy (e.g., FIFO, LRU)

Policies for writes from CPU to memory

Multilevel cache hierarchies
Announcements

Class session plan

- Today, Tuesday, 4/12: Cache placement, replacement, and write policies (Book chapters 6.4).
- Thursday, 4/14: Cache-friendly code–loop interchange, cache blocking (Book chapters 6.5 and 6.6), and cache oblivious algorithms.
Table of contents

Announcements

Cache placement policy (how to find data at address for read and write hit)
  Fully associative cache
  Direct-mapped cache
  Set-associative cache

Cache performance metrics: hits, misses, evictions
  Cache hits
  Cache misses

Cache replacement policy (how to find space for read and write miss)
  Direct-mapped cache need no cache replacement policy
  Associative caches need a cache replacement policy (e.g., FIFO, LRU)

Policies for writes from CPU to memory

Multilevel cache hierarchies
Cache placement policy (how to find data at address for read and write hit)

Several designs for caches

- Fully associative cache
- Direct-mapped cache
- N-way set-associative cache

Cache design options use $m$-bit memory addresses differently

- $t$-bit tag
- $s$-bit set index
- $b$-bit block offset

Figure: Memory addresses. Image credit CS:APP
Fully associative cache

- $m$-bit memory address split into:
  - $t$-bit tag
  - $b$-bit block offset

Figure: Fully associative cache. Image credit CS:APP
Fully associative cache

Figure: Fully associative cache. Image credit CS:APP

$b$-bit block offset

- Here, $b = 3$
- The number of bytes in a block is $B = 2^b = 2^3 = 8$
- A block is the minimum number of bytes that can be cached
- Good for capturing spatial locality, short strides
Fully associative cache

Figure: Fully associative cache. Image credit CS:APP

t-bit tag

- here, $t = m - b = m - 3$
- When CPU wants to read from or write to memory, all $t$-bits in tag need to match for read/write hit.
Fully associative cache

Full associativity

- Blocks can go into any of $E$ ways
- Here, $E = 3$
- Good for capturing temporal locality: cache hits can happen even with intervening reads and writes to other tags.

Figure: Fully associative cache. Image credit CS:APP
Fully associative cache

- **Capacity of cache**
  - Total capacity of fully associative cache in bytes: \( C = EB = E \times 2^b \)
  - Here, \( C = E \times 2^b = 3 \times 2^3 = 24 \) bytes
Fully associative cache

Strengths

- Blocks can go into any of $E$-ways.
- Hit rate is only limited by total capacity.

Weaknesses

- Searching across all valid tags is expensive.
- Figuring out which block to evict on read/write miss is also expensive.
- Requires a lot of combinational logic.

Figure: Fully associative cache. Image credit CS:APP
Direct-mapped cache

$m$-bit memory address split into:

- $t$-bit tag
- $s$-bit set index
- $b$-bit block offset

Figure: Direct-mapped cache. Image credit CS:APP
Direct-mapped cache

$S = 2^s$ sets

$b$-bit block offset

- here, $b = 3$
- The number of bytes in a block is $B = 2^b = 2^3 = 8$
- A block is the minimum number of bytes that can be cached
- Good for capturing spatial locality, short strides

**Figure:** Direct-mapped cache. Image credit CS:APP
Direct-mapped cache

$s = 2^s$ sets

- $s$-bit set index
  - Here, $s = 2$
  - The number of sets in cache is $S = 2^s = 2^2 = 4$
  - A hash function that limits exactly where a block can go
  - Good for further increasing ability to exploit spatial locality, short strides

Figure: Direct-mapped cache. Image credit CS:APP
Direct-mapped cache

$S = 2^s$ sets

$0 \ldots 01 \ 100$

Address

$t$-bit tag

- here,
  $t = m - s - b = m - 2 - 3$

- When CPU wants to read from or write to memory, all $t$-bits in tag need to match for read/write hit.

Figure: Direct-mapped cache. Image credit CS:APP
Direct-mapped cache

- In direct-mapped cache, blocks can go into only one of $E = 1$ way

Figure: Direct-mapped cache. Image credit CS:APP
Direct-mapped cache

Figure: Direct-mapped cache. Image credit CS:APP

Capacity of cache

- Total capacity of fully associative cache in bytes:
  \[ C = SEB = 2^s \times E \times 2^b \]
- Here, \[ C = 2^s \times E \times 2^b = 2^2 \times 1 \times 2^3 = 32 \text{ bytes} \]
Direct-mapped cache

Strengths

- Simple to implement.
- No need to search across tags.

Weaknesses

- Can lead to surprising thrashing of cache with unfortunate access patterns.
- Unexpected conflict misses independent of cache capacity.

Figure: Direct-mapped cache. Image credit CS:APP
**E-way set-associative cache**

- **Strengths**
  - Blocks can go into any of $E$-ways, increases ability to support temporal locality, thereby increasing hit rate.
  - Only need to search across $E$ tags. Avoids costly searching across all valid tags.
  - Avoids conflict misses due to unfortunate access patterns.

**Figure:** Direct-mapped cache. Image credit CS:APP

- $S = 2^s$ sets
- $E$ lines per set
- $B = 2^b$ bytes per cache block
- Valid
- Tag
- $0 \ 1 \ \cdots \ B-1$

Cache size: $C = B \times E \times S$ data bytes
**E-way set-associative cache**

- **Set 0:**
  - Valid: 1
  - Tag: 0 1 ... B-1
  - B = 2^b bytes per cache block

- **Set 1:**
  - Valid: 1
  - Tag: 0 1 ... B-1

- **Set S-1:**
  - Valid: 1
  - Tag: 0 1 ... B-1

**Cache size:** \( C = B \times E \times S \) data bytes

**Figure:** Direct-mapped cache. Image credit CS:APP

---

**Used in practice in, e.g., a recent Intel Core i7:**

- **C = 32KB L1 data cache per core**
- **S = 64 = 2^6 sets/cache**
  
  \( (s = 6 \text{ bits}) \)
- **E = 8 = 2^3 ways/set**
- **B = 64 = 2^6 bytes/block**
  
  \( (b = 6 \text{ bits}) \)
- **C = S \times E \times B**
- **Assuming memory addresses are** \( m = 48 \),
  then tag size
  
  \[ t = m - s - b = 48 - 6 - 6 = 36 \text{ bits}. \]
E-way set-associative cache

Let's see textbook slides for a simulation

Figure: Direct-mapped cache. Image credit CS:APP
Table of contents

Announcements

Cache placement policy (how to find data at address for read and write hit)
   Fully associative cache
   Direct-mapped cache
   Set-associative cache

Cache performance metrics: hits, misses, evictions
   Cache hits
   Cache misses

Cache replacement policy (how to find space for read and write miss)
   Direct-mapped cache need no cache replacement policy
   Associative caches need a cache replacement policy (e.g., FIFO, LRU)

Policies for writes from CPU to memory

Multilevel cache hierarchies
Cache hits

Memory access is serviced from cache

- Hit rate = \( \frac{\text{Number of hits}}{\text{Number of memory accesses}} \)
- Hit time: latency to access cache (4 cycles for L1, 10 cycles for L2)
Cache misses: metrics

Memory access cannot be serviced from cache

- Miss rate = \( \frac{\text{Number of misses}}{\text{Number of memory accesses}} \)
- Miss penalty (miss time): latency to access next level cache or memory (up to 200 cycles for memory).
- Average memory access time = hit time + miss rate \( \times \) miss penalty
Cache misses: Classification

Compulsory misses
▶ First access to a block of memory will miss because cache is cold.

Conflict misses
▶ Multiple blocks map (hash) to the same cache set.
▶ Fully associative caches have no such conflict misses.

Capacity misses
▶ Occurs when the set of active cache blocks (working set) is larger than the cache.
▶ Direct mapped caches have no such capacity misses.
Table of contents

Announcements

Cache placement policy (how to find data at address for read and write hit)
   Fully associative cache
   Direct-mapped cache
   Set-associative cache

Cache performance metrics: hits, misses, evictions
   Cache hits
   Cache misses

Cache replacement policy (how to find space for read and write miss)
   Direct-mapped cache need no cache replacement policy
   Associative caches need a cache replacement policy (e.g., FIFO, LRU)

Policies for writes from CPU to memory

Multilevel cache hierarchies
Direct-mapped cache

No need for replacement policy

- The number of sets in cache is $S = 2^s = 2^2 = 4$.
- A hash function that limits exactly where a block can go.
- In direct-mapped cache, blocks can go into only one of $E = 1$ way.
- No cache replacement policy is needed.

Figure: Direct-mapped cache. Image credit CS:APP
Associative caches

Figure: Fully associative cache. Image credit CS:APP

Needs replacement policy

- Blocks can go into any of E ways
- Here, $E = 3$
- Good for capturing temporal locality.
- If all ways/lines/blocks are occupied, and a cache miss happens, which way/line/block will be the victim and get evicted for replacement?
Cache replacement policies for associative caches

**FIFO: First-in, first-out**
- Evict the cache line that was placed the longest ago.
- Each cache set essentially becomes limited-capacity queue.

**LRU: Least Recently Used**
- Evict the cache line that was last accessed longest ago.
- Needs a counter on each cache line, and/or a global counter (e.g., program counter).
Table of contents

Announcements

Cache placement policy (how to find data at address for read and write hit)
  Fully associative cache
  Direct-mapped cache
  Set-associative cache

Cache performance metrics: hits, misses, evictions
  Cache hits
  Cache misses

Cache replacement policy (how to find space for read and write miss)
  Direct-mapped cache need no cache replacement policy
  Associative caches need a cache replacement policy (e.g., FIFO, LRU)

Policies for writes from CPU to memory

Multilevel cache hierarchies
Policies for writes from CPU to memory

How to deal with write-hit?

- **Write-through.** Simple. Writes update both cache and memory. Costly memory bus traffic.

- **Write-back.** Complex. Writes update only cache and set a dirty bit; memory updated only upon eviction. Reduces memory bus traffic. (For multi-core CPUs, motivates complex cache coherence protocols)

How to deal with write-miss?

- **No-write-allocate.** Simple. Write-misses do not load block into cache. But write-misses are not mitigated via cache support.

- **Write-allocate.** Complex. Write-misses will load block into cache.

Typical designs:

- **Simple:** write-through + no-write-allocate.
- **Complex:** write-back + write-allocate.
Table of contents

Announcements

Cache placement policy (how to find data at address for read and write hit)
   Fully associative cache
   Direct-mapped cache
   Set-associative cache

Cache performance metrics: hits, misses, evictions
   Cache hits
   Cache misses

Cache replacement policy (how to find space for read and write miss)
   Direct-mapped cache need no cache replacement policy
   Associative caches need a cache replacement policy (e.g., FIFO, LRU)

Policies for writes from CPU to memory

Multilevel cache hierarchies
Multilevel cache hierarchies

Small fast caches nested inside large slow caches

- L1 data and instruction cache: 32KB, 64 set, 8-way associative, 64B block, 4 cycle latency.
- L2 cache: 256KB, 512 set, 8-way associative, 64B block, 10 cycle latency.
- L3 cache: 8MB, 8192 set, 16-way associative, 64B block, 40-75 cycle latency.

Notice how latency cost increases as $E$-way associativity increases.

Figure: Intel Core i7 cache hierarchy. Image credit CS:APP

Figure: Intel 2020 Gulftown die shot. Image credit AnandTech