Caches: replacement policy, memory policy, and cache hierarchies

Yipeng Huang

Rutgers University

April 8, 2021

Table of contents

Looking ahead

Class plan

1. Thursday, 4/8: PA5 cache simulator and performance released.
2. Thursday, 4/8: PA4 binary bomb lab due.
3. Monday, 4/12: Quiz due.

Table of contents

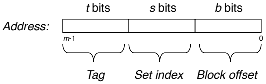
Cache placement policy (how to find data at address for read and write hit)

Several designs for caches

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

Cache design options use m-bit memory addresses differently

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



m = 48bits, or 64bits, or 32bits

Figure: Memory addresses. Image credit CS:APP

Fully associative cache

s=0; S=2^s=1 sets, therefore, no hash function involved

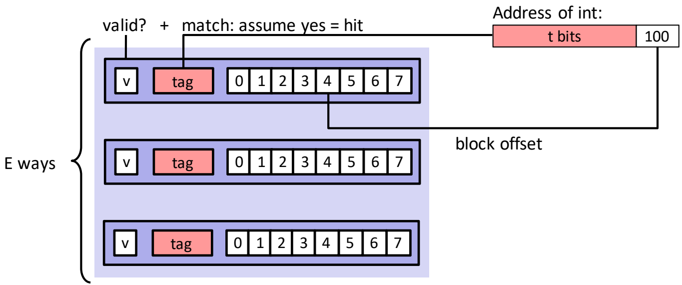


Figure: Fully associative cache. Image credit CS:APP

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.

Direct-mapped cache

E=1

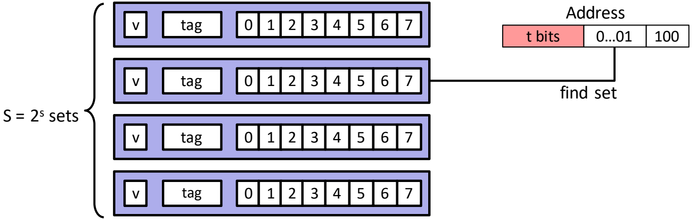


Figure: Direct-mapped cache. Image credit CS:APP

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.

Direct-mapped cache

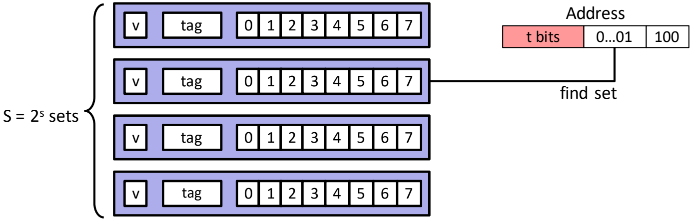


Figure: Direct-mapped cache. Image credit CS:APP

Let's see textbook slides for a simulation

E-way set-associative cache

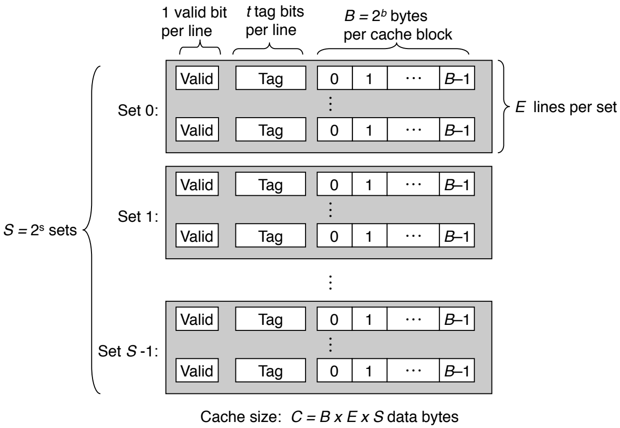


Figure: Direct-mapped cache. Image credit CS:APP

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.

E-way set-associative cache

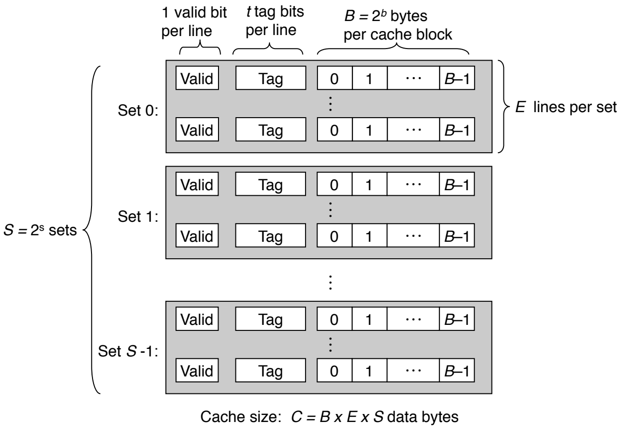
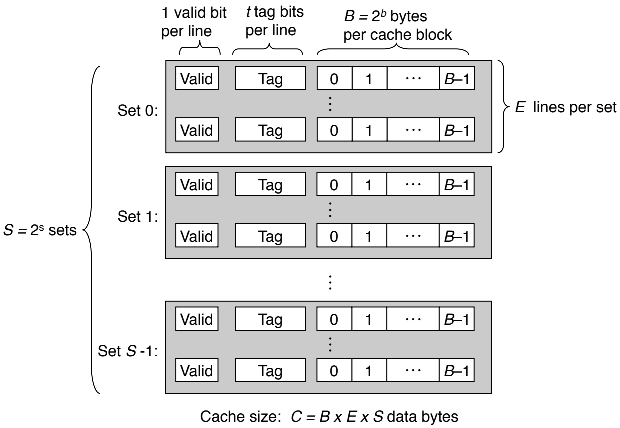


Figure: Direct-mapped cache. Image credit CS:APP

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

- $C = 32$KB L1 data cache per core
- $S = 64 = 2^6$ sets/cache ($s = 6$ bits)
- $E = 8 = 2^3$ ways/set
- $B = 64 = 2^6$ bytes/block ($b = 6$ bits)
- $C = S * E * B$
- Assuming memory addresses are $m = 48$, then tag size $t = m - s - b = 48 - 6 - 6 = 36$ 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

Direct-mapped cache

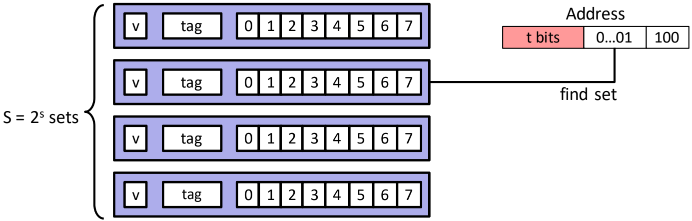


Figure: Direct-mapped cache. Image credit CS:APP

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.

Associative caches

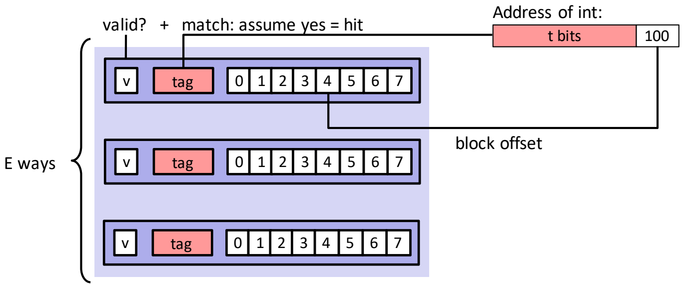


Figure: Fully associative cache. Image credit CS:APP

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-capcity 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

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 not load block into cache.

Typical designs:

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

READ / LOAD from memory: movq (0x00) %eax
WRITE / STORE to memory: movq %eax (0x00)

Table of contents

Multilevel cache hierarchies

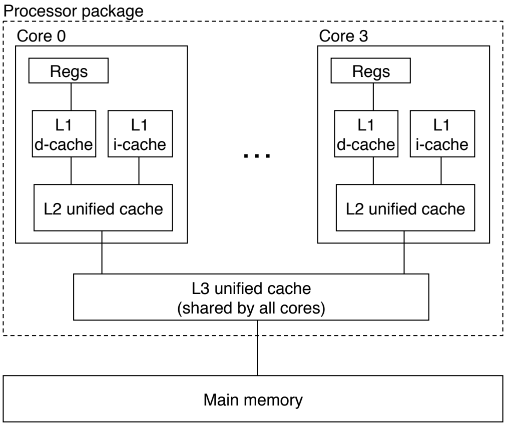


Figure: Intel Core i7 cache hierarchy. Image credit CS:APP

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.

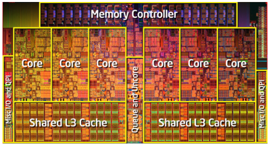


Figure: Intel 2020 Gulftown die shot. Image credit AnandTech