The memory hierarchy: Cache replacement and write policies, cache-friendly code

Yipeng Huang

Rutgers University

April 17, 2023
Table of contents

Announcements

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

Cache-friendly code
  Loop interchange
  Cache blocking

Multilevel cache hierarchies
  Cache oblivious algorithms

Memory hierarchy implications for software-hardware abstraction
Announcements

Class session plan

- Monday, 4/17: Cache-Friendly Code–cache blocking (Book chapters 6.5 and 6.6), cache oblivious algorithms
Table of contents

Announcements

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

Cache-friendly code
  Loop interchange
  Cache blocking

Multilevel cache hierarchies
  Cache oblivious algorithms

Memory hierarchy implications for software-hardware abstraction
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
**E-way set-associative cache**

\[ S = 2^e \text{ sets} \]

- **Set 0:**
  - Valid
  - Tag
  - 0 1 \ldots \ B-1

- **Set 1:**
  - Valid
  - Tag
  - 0 1 \ldots \ B-1

- **Set S-1:**
  - Valid
  - Tag
  - 0 1 \ldots \ B-1

\[ E \text{ lines per set} \]

**Cache size:**

\[ C = B \times E \times S \text{ data bytes} \]

**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
E-way set-associative cache

- Valid
- Tag
- 0 1 ⋯ B-1

Set 0:

Set 1:

Set S-1:

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

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

- \( C = 32\text{KB} \) 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)
- \( C = S \times E \times B \)
- Assuming memory addresses are \( m = 48 \), then tag size
  \( t = m - s - b = 48 - 6 - 6 = 36 \) bits.

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

- 1 valid bit per line
- $t$ tag bits per line
- $B = 2^b$ bytes per cache block

Set 0:
- Valid
- Tag
- 0 1 \ldots B-1

Set 1:
- Valid
- Tag
- 0 1 \ldots B-1

Set $S-1$:
- Valid
- Tag
- 0 1 \ldots B-1

$S = 2^s$ sets

$C = B \times E \times S$ data bytes

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)
  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

Cache-friendly code
  Loop interchange
  Cache blocking

Multilevel cache hierarchies
  Cache oblivious algorithms

Memory hierarchy implications for software-hardware abstraction
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)
  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

Cache-friendly code
  Loop interchange
  Cache blocking

Multilevel cache hierarchies
  Cache oblivious algorithms

Memory hierarchy implications for software-hardware abstraction
Direct-mapped cache

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

![Associative cache diagram](image)

**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)
  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

Cache-friendly code
  Loop interchange
  Cache blocking

Multilevel cache hierarchies
  Cache oblivious algorithms

Memory hierarchy implications for software-hardware abstraction
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)
  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

Cache-friendly code
  Loop interchange
  Cache blocking

Multilevel cache hierarchies
  Cache oblivious algorithms

Memory hierarchy implications for software-hardware abstraction
Cache-friendly code

Algorithms can be written so that they work well with caches
  ▶ Maximize hit rate
  ▶ Minimize miss rate
  ▶ Minimize eviction counts

Do so by:
  ▶ Increasing spatial locality.
  ▶ Increasing temporal locality.

Advanced optimizing compilers can automatically make such optimizations
  ▶ GCC optimizations
  ▶ -floop-interchange
  ▶ -floop-block
Loop interchange

Refer to textbook slides on "Rearranging loops to improve spatial locality"

- Loop interchange increases spatial locality.
- In PA5, fourth part "cacheBlocking" you can explore the impact of this on matrix multiplication.
- In practice, programmers do not have to worry about this optimization.
- Optimized automatically in GCC by compiler flag `-floop-interchange` and `-O3`. 
Cache blocking

Refer to textbook slides on "Using blocking to improve temporal locality"

- Cache blocking increases temporal locality.
- In PA5, fourth part "cacheBlocking" you can explore the impact of this on matrix multiplication.
- In practice, programmers do not have to worry about this optimization.
- Optimized automatically in GCC by compiler flag `-floop-block`. But it is not part of default optimizations such as `-O3` so you have to remember to set it.
Table of contents

Announcements

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

Cache-friendly code
  Loop interchange
  Cache blocking

Multilevel cache hierarchies
  Cache oblivious algorithms

Memory hierarchy implications for software-hardware abstraction
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
Cache oblivious algorithms

The challenge in writing code / compiling programs to take advantage of caches:

- Programmers do not easily have information about target machine.
- Compiling binaries for every envisioned target machine is costly.
Matrix transpose baseline algorithm: iteration

\[ A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \]

\[ B = A^T = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix} \]

```c
for ( size_t i=0; i<n; i++ ) {
    for ( size_t j=0; j<n; j++ ) {
        B[ j*n + i ] = A[ i*n + j ];
    }
}
```
Matrix transpose via recursion

\[ A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \]

\[ B = A^T = \begin{bmatrix} A_{0,0}^T & A_{1,0}^T \\ A_{0,1}^T & A_{1,1}^T \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix} \]

**Strategy:**

- Divide and conquer large matrix to transpose into smaller transpositions.
- After some recursion, problem will fit well inside cache capacity.
- Once enough locality exists within subroutine, switch to plain iterative approach.

**Advantages:**

- No need to know about cache capacity and parameters beforehand.
- Works well with deep multilevel cache hierarchies: different amounts of locality for each cache level.
Table of contents

Announcements

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

Cache-friendly code
  Loop interchange
  Cache blocking

Multilevel cache hierarchies
  Cache oblivious algorithms

Memory hierarchy implications for software-hardware abstraction
Memory hierarchy implications for software-hardware abstraction

It is not entirely true the architecture can hide details of microarchitecture
Even less true going forward. What to do?

Application level recommendations

▶ Use industrial strength, optimized libraries compiled for target machine.
▶ Lapack, Linpack, Matlab, Python SciPy, NumPy...
▶ https://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring11/slides/class08.pdf

Algorithm level recommendations
Deploy cache-oblivious algorithm implementations.

Compiler level recommendations

▶ Enable compiler optimizations—e.g., -O3, -floop-interchange, -floop-block.