The memory hierarchy: Locality and cache memories

Yipeng Huang

Rutgers University

April 4, 2024
Table of contents

Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality

Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches

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

PA5: Simulating a cache and optimizing programs for caches
Assembly programming view of computer: CPU and memory.

Full view of computer architecture and systems: +caches, +storage, +network

Figure: Memory hierarchy. Image credit CS:APP
Cache, memory, storage, and network hierarchy trends

Topic of this chapter:
- Technology trends that drive CPU-memory gap.
- How to create illusion of fast access to capacious data.

**Figure:** Widening gap: CPU processing time vs. memory access time. Image credit CS:APP
Static random-access memory (registers, caches)

- SRAM is bistable logic
- Access time: 1 to 10 CPU clock cycles
- Implemented in the same transistor technology as CPUs, so improvement has matched pace.

Figure: SRAM operating principle. Image credit Wikimedia
Dynamic random-access memory (main memory)

- Needs refreshing every 10s of milliseconds
- 8GB typical in laptop; 1TB on each ilab machine
- Access time: 100 CPU clock cycles
- Memory gap: DRAM technological improvement slower relative to CPU/SRAM.

Figure: DRAM operating principle. Image credit ocw.mit.edu
CPU / DRAM main memory interface

- DDR4 bus standard supports 25.6GB/s transfer rate

**Figure**: Memory Bus. Image credit CS:APP

**Figure**: Intel 2020 Gulftown die shot. Image credit AnandTech
Solid state and hard disk drives (storage)

Technology

▶ SSD: flash nonvolatile memory stores data as charge.
▶ HDD: magnetic orientation.
▶ Access time: 100K CPU clock cycles

For in-depth on storage, file systems, and operating systems, take:

▶ CS214 Systems Programming
▶ CS416 Operating Systems Design

Since summer 2021, LCSR (admins of iLab) have moved the storage systems that supports everyone’s home directories to SSD. https://resources.cs.rutgers.edu/docs/file-storage/storage-technology-options/
I/O interfaces

Storage interfaces

- SATA 3.0 interface (6Gb/s transfer rate) typical
- PCIe (15.8 GB/s) becoming commonplace for SSD
- But interface data rate is rarely the bottleneck.

For in-depth on computer network layers, take:

- CS352 Internet Technology

Figure: I/O Bus. Image credit CS:APP
Cache, memory, storage, and network hierarchy trends

Figure: Widening gap: CPU processing time vs. memory access time. Image credit CS:APP

Topic of this chapter:

- Technology trends that drive CPU-memory gap.
- How to create illusion of fast access to capacious data.
Table of contents

Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality

Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches

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

PA5: Simulating a cache and optimizing programs for caches
Locality: How to create illusion of fast access to capacious data

From the perspective of memory hierarchy, locality is using the data in at any particular level more frequently than accessing storage at next slower level.

First, let’s experience the puzzling effect of locality in `sumArray.c`

- `sumArrayRows()`
- `sumArrayCols()`

Well-written programs maximize locality

- Spatial locality
- Temporal locality
Spatial locality

Programs tend to access adjacent data.

Example: stride 1 memory access in \texttt{a} and \texttt{b}.
Temporal locality

Example:

```c
double dotProduct ( double a[N], double b[N],
) {
 double sum = 0.0;
 for (size_t i = 0; i < N; i++) {
    sum += a[i] * b[i];
 }
 return sum;
}
```

Programs tend to access data over and over.

Example: `sum` gets accessed $N$ times in iteration.
Table of contents

Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality

Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches

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

PA5: Simulating a cache and optimizing programs for caches
CPU / cache / DRAM main memory interface

**Figure:** Cache resides on CPU chip close to register file. Image credit CS:APP

**Figure:** Cache stores a temporary copy from the slower main memory. Image credit CS:APP
CPU / cache / DRAM main memory interactions

Figure: Cache stores a temporary copy from the slower main memory. Image credit CS:APP

When CPU loads (LD) from memory
- Cache read hit
- Cache read miss

When CPU stores (ST) to memory
- Cache write hit
- Cache write miss
Table of contents

Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality

Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches

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

PA5: Simulating a cache and optimizing programs for caches
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

$E$ ways

Valid? + match: assume yes = hit

Block offset

Address of int: $t$ bits 100

$b$-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: Fully associative cache. Image credit CS:APP
Fully associative cache

$t$-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

- 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: $C = E \times B = E \times 2^b$
- Here, $C = E \times 2^b = 3 \times 2^3 = 24$ bytes

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

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.
- Requires a lot of combinational logic.
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

$\begin{array}{c|cccccc}
\text{v} & \text{tag} & 0 & 1 & 2 & 3 & 4 \\
\hline
\text{v} & \text{tag} & 0 & 1 & 2 & 3 & 4 \\
\text{v} & \text{tag} & 0 & 1 & 2 & 3 & 4 \\
\end{array}$

$\begin{array}{c|cccccc}
\text{v} & \text{tag} & 5 & 6 & 7 & 0 & 1 \\
\hline
\text{v} & \text{tag} & 5 & 6 & 7 & 0 & 1 \\
\text{v} & \text{tag} & 5 & 6 & 7 & 0 & 1 \\
\end{array}$

$b$-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

$S = 2^s = 2^2 = 4$

- The number of sets in cache is

- A hash function that limits exactly where a block can go

- Good for further increasing ability to exploit spatial locality, short strides
Direct-mapped cache

$t \text{-bit tag}$

- 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.
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**

- 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

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

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

- **Set S-1:**
  - **Valid**
  - **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 ($s = 6$ 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**

| Valid | Tag | 0 | 1 | \cdots | B-1 |
|-------|-----|---|----|--------|
|       |     |   |    |        |      |
|       |     |   |    |        |      |

Set 0:

Set 1:

Set \(S-1\):

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

\(S = 2^s\) sets

\(B = 2^b\) bytes per cache block

1 valid bit per line

\(t\) tag bits per line

\(E\) lines per set

\(E\) lines per set

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

Let’s see textbook slides for a simulation
Table of contents

Cache, memory, storage, and network hierarchy trends
  Static random-access memory (registers, caches)
  Dynamic random-access memory (main memory)
  Solid state and hard disk drives (storage)

Locality: How to create illusion of fast access to capacious data
  Spatial locality
  Temporal locality

Caches: motivation
  Hardware caches supports software locality
  Software locality exploits hardware caches

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

PA5: Simulating a cache and optimizing programs for caches
PA5: Simulating a cache and optimizing programs for caches

Write a cache simulator
1. fullyAssociative
2. directMapped
3. setAssociative

Optimize some code for better cache performance
1. cacheBlocking
2. cacheOblivious
PA5: Simulating a cache and optimizing programs for caches

A tour of files in the package

- trace files
- csim-ref