Memory Hierarchy and Cache

Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming

How does execution time grow with SIZE?

```c
int array[SIZE];
fillArrayRandomly(array);
int s = 0;

for (int i = 0; i < 200000; i++) {
    for (int j = 0; j < SIZE; j++) {
        s += array[j];
    }
}
```

Reality

![Graph showing execution time growth with SIZE]
**Processor-memory bottleneck**

Processor performance doubled about every 18 months. Bus bandwidth evolved much slower.

- **Main Memory**
- **CPU**
- **Reg**
- **Cache**

**Bus bandwidth:** 256 bytes/cycle, **Latency:** 1-few cycles

**Bandwidth:** 2 Bytes/cycle, **Latency:** 100 cycles

**Solution: caches**

**General cache mechanics**

**Cache**

```
  8 9 14 3
```

**Memory**

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

Block: unit of data in cache and memory. (a.k.a. line)

Smaller, faster, more expensive. Stores subset of memory blocks. (lines)

Larger, slower, cheaper. Partitioned into blocks (lines).

**Cache hit**

```
CPU
```

Request: 14

1. **Request data in block b.**
2. **Cache hit:** Block b is in cache.

```
Cache
```

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

```
Memory
```

```
8 9 14 3
4 5 6 7
8 9 10 11
12 13 14 15
```

**Cache**

**English:**

*n.* a hidden storage space for provisions, weapons, or treasures

*v.* to store away in hiding for future use

**Computer Science:**

*n.* a computer memory with short access time used to store frequently or recently used instructions or data

*v.* to store [data/instructions] temporarily for later quick retrieval

Also used more broadly in CS: software caches, file caches, etc.
Locality: why caches work

Programs tend to use data and instructions at addresses near or equal to those they have used recently.

Temporal locality:
Recently referenced items are likely to be referenced again in the near future.

Spatial locality:
Items with nearby addresses are likely to be referenced close together in time.

How do caches exploit temporal and spatial locality?

Locality #1

Data:

Instructions:

Locality #2

What is stored in memory?

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

```c
int sum_array_rows(int a[M][N]) {
    int sum = 0;
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```
**Locality #3**

```c
int sum_array_cols(int a[M][N]) {
    int sum = 0;
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < M; i++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```

**Locality #4**

```c
int sum_array_3d(int a[M][N][N]) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < M; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

What is "wrong" with this code?  How can it be fixed?

**Cost of cache misses**

Miss cost could be 100 × hit cost.

99% hits could be twice as good as 97%.  How?

Assume cache hit time of 1 cycle, miss penalty of 100 cycles

Mean access time:

- 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
- 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

**Cache performance metrics**

**Miss Rate**

Fraction of memory accesses to data not in cache (misses / accesses)

Typically: 3% - 10% for L1; maybe < 1% for L2, depending on size, etc.

**Hit Time**

Time to find and deliver a block in the cache to the processor.

Typically: 1 - 2 clock cycles for L1; 5 - 20 clock cycles for L2

**Miss Penalty**

Additional time required on cache miss = main memory access time

Typically 50 - 200 cycles for L2 (trend: increasing!)
**Memory Hierarchy**

- Why does it work?
  - registers: small, fast, power-hungry, expensive
  - L1 cache (SRAM, on-chip): program sees “memory” explicitly
  - L2 cache (SRAM, on-chip)
  - L3 cache (SRAM, off-chip)
  - main memory (DRAM)
  - persistent storage (hard disk, flash, over network, cloud, etc.)

**Cache organization**

- Block
  - Fixed-size unit of data in memory/cache

**Placement Policy**
- Where in the cache should a given block be stored?
  - direct-mapped, set associative

**Replacement Policy**
- What if there is no room in the cache for requested data?
  - least recently used, most recently used

**Write Policy**
- When should writes update lower levels of memory hierarchy?
  - write back, write through, write allocate, no write allocate

**Blocks**

- Divide address space into fixed-size aligned blocks. power of 2
  - Example: block size = 8
    - full byte address: 00010010
    - Block ID: address bits - offset bits
      -offset within block: log₂(block size)

**Placement policy**

- Mapping: index(Block ID) = ???
  - Note: drawing address order differently from here on!

- Small, fixed number of block slots.
  - Large, fixed number of block slots.
**Placement: direct-mapped**

- Memory: 
  - Block ID: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
- **Mapping:** index(Block ID) = Block ID mod S
  
- **Cache:** 
  - Index: 00, 01, 10, 11
  - S = # slots = 4
  
- (easy for power-of-2 block sizes...)

**Placement: mapping ambiguity?**

- Memory: 
  - Block ID: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
- **Mapping:** index(Block ID) = Block ID mod S

- **Cache:** 
  - Index: 00, 01, 10, 11
  - S = # slots = 4

- Which block is in slot 2?

**Placement: tags resolve ambiguity**

- Memory: 
  - Block ID: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

- **Mapping:** index(Block ID) = Block ID mod S

- Cache: 
  - Index: 00, 01, 10, 11
  - Block ID bits not used for index.

**Address = tag, index, offset**

- **Data:**
  - Disambiguates slot contents.
  - What slot in the cache?
  - Where within a block?
  - a-bit Address: Tag, Index, Offset
  - (a-s-b) bits, s bits, b bits

- **Block ID:**
  - Address bits - Offset bits
  - \(\log_2(\text{block size}) = b\)
  - \# address bits

- **Tag, Index:**
  - full address of individual byte in memory

- **Offset within block:**
  - \(\log_2(\text{block size}) = b\)
Puzzle #1

Cache starts empty.
Access (address, hit/miss) stream:

(0xA, miss), (0xB, hit), (0xC, miss)

What could the block size be?

Placement: set-associative

One index per set of block slots.
Store block in any slot within set.

Mapping:
\[ \text{index(Block ID)} = \text{Block ID} \mod S \]

Replacement policy: if set is full, what block should be replaced?
Common: least recently used (LRU)
but hardware may implement “not most recently used”
**Example: tag, index, offset? #1**

4-bit Address | Tag | Index | Offset
--- | --- | --- | ---

Direct-mapped  | tag bits | ____
4 slots        | set index bits | ____
2-byte blocks  | block offset bits | ____

index(1101) = ____

---

**Replacement policy**

If set is full, what block should be replaced?

Common: least recently used (LRU)

(but hardware usually implements “not most recently used”)

Another puzzle: Cache starts empty, uses LRU.

Access (address, hit/miss) stream:

(0xA, miss); (0xB, miss); (0xA, miss)

---

**Example: tag, index, offset? #2**

$E$-way set-associative

$S$ slots

16-byte blocks

**General cache organization ($S$, $E$, $B$)**

$E$ lines per set ("E-way")

$S$ sets

$B$ = 2$^B$ bytes of data per cache line (the data block)

**address size:**

$t + s + b$ address bits

---

**Power of 2**

---

**cache capacity:**

$S \times E \times B$ data bytes

---

**valid bit**

---

**set**

---

**block/line**

---

---
Cache read

E lines per set

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

Address of byte in memory:

\[ t \text{ bits} \quad s \text{ bits} \quad b \text{ bits} \]

Find the set by index. If any block in the set:

- is valid;
- and
- has matching tag

Get data at offset in block.

B = \( 2^b \) bytes of data per cache line (the data block)

S = 2^t \text{ sets}

Valid bit

Cache read: direct-mapped (E = 1)

This cache:

- Block size: 8 bytes
- Associativity: 1 block per set (direct mapped)

Address of int:

\[ t \text{ bits} \quad 0..100 \]

Find set

valid? + match?: yes = hit

If no match: old line is evicted and replaced

Direct-mapped cache practice

12-bit address
16 lines, 4-byte block size
Direct mapped
Offset bits? Index bits? Tag bits?

Access 0x354
Access 0xA20
Example #1 (E = 1)

Locals in registers.
Assume a is aligned such that
\[ a[r][c] = a...arrr \quad rrrr \quad cccc \quad 000 \]

```c
int sum_array_rows(double a[16][16]) {
    double sum = 0;
    for (int r = 0; r < 16; r++) {
        for (int c = 0; c < 16; c++) {
            sum += a[r][c];
        }
    }
    return sum;
}
```

Example #2 (E = 1)

```c
int dotprod(int x[8], int y[8]) {
    int sum = 0;
    for (int i = 0; i < 8; i++) {
        sum += x[i]*y[i];
    }
    return sum;
}
```

Cache read: set-associative (Example: E = 2)

This cache:
- Block size: 8 bytes
- Associativity: 2 blocks per set

![Cache read: set-associative diagram](image)
Example #3 \((E = 2)\)

```c
float dotprod(float x[8], float y[8]) {
    float sum = 0;
    for (int i = 0; i < 8; i++) {
        sum += x[i]*y[i];
    }
    return sum;
}
```

If \(x\) and \(y\) aligned, e.g. \&\(x[0] = 0\), \&\(y[0] = 128\), can still fit both because each set has space for two blocks/lines.

Types of Cache Misses

Cold (compulsory) miss

Conflict miss

Capacity miss

Which ones can we mitigate/eliminate? How?

Writing to cache

Multiple copies of data exist, must be kept in sync.

Write-hit policy

- Write-through:
- Write-back: needs a dirty bit

Write-miss policy

- Write-allocate:
- No-write-allocate:

Typical caches:

- Write-back + Write-allocate, usually
- Write-through + No-write-allocate, occasionally

Write-back, write-allocate example

1. mov $T, %ecx
2. mov $U, %edx
3. mov $0xFEED, (%ecx)
a. Miss on T.

Cache/memory not involved
Write-back, write-allocate example

1. `mov $T, %ecx`
2. `mov $U, %edx`
3. `mov $0xFEED, (%ecx)`
   a. Miss on T.
   c. Fill T (write-allocate).
   d. Write T in cache (dirty).
4. `mov (%edx), %eax`
   a. Miss on U.
   c. Fill T (write-allocate).
   d. Write T in cache (dirty).

Example memory hierarchy

Typical laptop/desktop processor (c.a. 201_)

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)
- Main memory

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

(L Aside) Software caches

Examples
- File system buffer caches, web browser caches, database caches, network CDN caches, etc.

Some design differences
- Almost always fully-associative
- Often use complex replacement policies
- Not necessarily constrained to single “block” transfers
Cache-friendly code

Locality, locality, locality.

Programmer can optimize for cache performance
- Data structure layout
- Data access patterns
  - Nested loops
  - Blocking (see CSAPP 6.5)

All systems favor "cache-friendly code"
- Performance is hardware-specific
- Generic rules capture most advantages
  - Keep working set small (temporal locality)
  - Use small strides (spatial locality)
  - Focus on inner loop code