Memory Hierarchy and Cache

Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming

How does execution time grow with SIZE?

```cpp
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];
    }
}
```
### Processor-memory bottleneck

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

- Bandwidth: 256 bytes/cycle
- Latency: 1-few cycles

Bus bandwidth: 2 Bytes/cycle
Latency: 100 cycles

**Solution:** caches

### 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.

### General cache mechanics

**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).

- Data is moved in block units

### Cache hit

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

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

Instructions:

```
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;
}
```

What is stored in memory?

```
row-major M x N 2D array in C
```

Locality #2

Placement Policy: where to put block in cache

Replacement Policy: which block to evict

*Assessing locality in code is an important programming skill.*
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;
}
```

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

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;
}
```

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

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!)

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)
- L2 cache (SRAM, on-chip)
- L3 cache (SRAM, off-chip)
- main memory (DRAM)
- persistent storage (hard disk, flash, over network, cloud, etc.)

program sees “memory”

 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)

Mapping:
index(Block ID) = ???
**Placement: direct-mapped**

Mapping:

$\text{index(Block ID)} = \text{Block ID} \mod S$

*(easy for power-of-2 block sizes...)*

**Placement: mapping ambiguity?**

Which block is in slot 2?

**Placement: tags resolve ambiguity**

Mapping:

$\text{index(Block ID)} = \text{Block ID} \mod S$

Block ID bits not used for index.

**Address = tag, index, offset**

Disambiguates slot contents.

What slot in the cache?

Where within a block?

a-bit Address

$(a-s-b)$ bits  
$s$ bits  
b bits

Block ID bits - Index bits  
$\log_2(\text{# cache slots})$

Tag

Index

Full byte address

Block ID

Address bits - Offset bits

$\log_2(\text{block size}) = b$

$\# \text{ address bits}$
### Puzzle #1

Cache starts **empty**.

Access (address, hit/miss) stream:

$$(10, \text{miss}), (11, \text{hit}), (12, \text{miss})$$

What could the block size be?

---

### Placement: direct-mapping conflicts

What happens when accessing in repeated pattern:

$$0010, 0110, 0010, 0110, 0010...?$$

**cache conflict**

Every access suffers a miss, evicts cache line needed by next access.

---

### 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?

- **1-way**
  - 8 sets, 1 block each
- **2-way**
  - 4 sets, 2 blocks each
- **4-way**
  - 2 sets, 4 blocks each
- **8-way**
  - 1 set, 8 blocks

Common: least recently used (LRU)

but hardware may implement “not most recently used”
Example: tag, index, offset? #1

<table>
<thead>
<tr>
<th>4-bit Address</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

Direct-mapped
4 slots
2-byte blocks

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:
(10, miss); (12, miss); (10, miss)

associativity of cache?

Example: tag, index, offset? #2

E-way set-associative
S slots
16-byte blocks

<table>
<thead>
<tr>
<th>16-bit Address</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

E = 1-way
S = 8 sets

E = 2-way
S = 4 sets

E = 4-way
S = 2 sets

tag bits
set index bits
block offset bits

index(0x1833)

General cache organization (S, E, B)

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

Power of 2

S sets

set

block/line

valid bit

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

cache capacity:
\( S \times E \times B \) data bytes
address size:
\( t + s + b \) address bits

Memory Hierarchy and Cache 30

Memory Hierarchy and Cache 31

Memory Hierarchy and Cache 32
Cache read

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

Address of int:
- Offset bits?  Index bits?  Tag bits?

Cache read: direct-mapped \( (E = 1) \)

This cache:
- Block size: 8 bytes
- Associativity: 1 block per set (direct mapped)
**Memory Hierarchy and Cache**

**Example #1 (E = 1)**

Locals in registers.
Assume `a` is aligned such that `a[r][c]` is aligned

```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;
}
```

```c
int sum_array_cols(double a[16][16]) {
    double sum = 0;
    for (int c = 0; c < 16; c++) {
        for (int r = 0; r < 16; r++) {
            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)**

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

**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 (Example: E = 2)**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cache/memory not involved

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

Example memory hierarchy

Processor package

Core 0
- L1 d-cache
- L1 i-cache
- L2 unified cache

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

L3 unified cache (shared by all cores)

Main memory

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

L1 i-cache and d-cache:
- 32 KB, 8-way,
- Access: 4 cycles

L2 unified cache:
- 256 KB, 8-way,
- Access: 11 cycles

L3 unified cache:
- 8 MB, 16-way,
- Access: 30-40 cycles

Block size: 64 bytes for all caches.

slower, but more likely to hit

(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