CS 240 Stage 3
Abstractions for Practical Systems

Caching and the memory hierarchy
Operating systems and the process model
Virtual memory
Dynamic memory allocation
Victory lap

How does execution time grow with SIZE?

```plaintext
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];
    }
}
```
**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).

**Processor-memory bottleneck**

Processor performance doubled about every 18 months

Bus bandwidth evolved much slower

---

**Solution:** caches
**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**

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

Data:

Instructions:

What is stored in memory?
Locality #2

```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;
    }
}
```

row-major M x N 2D array in C

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;
    }
}
```

row-major M x N 2D array in C

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 \times 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 \times 100 cycles = 4 cycles
- 99% hits: 1 cycle + 0.01 \times 100 cycles = 2 cycles

hit/miss rates
### 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.)

### 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: 00010000
- Offset within block: log₂(block size)
Placement policy

Small, fixed number of block slots.

Large, fixed number of block slots.

Placement: direct-mapped

(S = # slots = 4)

(index(Block ID) = Block ID mod S)

Tag
Data

Block ID bits not used for index.

Placement: mapping ambiguity?

(index(Block ID) = ???)

Which block is in slot 2?

Placement: tags resolve ambiguity

(index(Block ID) = Block ID mod S)
Address = tag, index, offset

Disambiguates slot contents.  What slot in the cache?

a-bit Address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a-s-b) bits</td>
<td>s bits</td>
<td>b bits</td>
</tr>
</tbody>
</table>

Where within a block?

Block ID bits - Index bits  log₂(# cache slots)

Tag

Index

00010 010  full byte address

Block ID

Address bits - Offset bits  log₂(block size) = b

# address bits

Puzzle #1

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

(10, miss), (11, hit), (12, miss)

What could the block size be?

Placement: direct-mapped

Why not this mapping?
index(Block ID) = Block ID / S
(still easy for power-of-2 block sizes...)

Memory Hierarchy and Cache

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.

Set

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

**Mapping:**

index(Block ID) = Block ID mod $S$

- 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

$S = \# slots in cache$

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**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

Direct-mapped tag bits

4 slots set index bits

2-byte blocks block offset bits

index(1101) = ____

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

$E$-way set-associative

$E$ slots

16-byte blocks

**16-bit Address**

<table>
<thead>
<tr>
<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)

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?
**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)

**Cache capacity:**
$S \times E \times B$ data bytes

**Address size:**
t + s + b address bits

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

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

- **S** = $2^s$ sets

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

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

**Address of int:**

- **t bits**
- **s bits**
- **b bits**

- **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?

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>0</td>
<td>02</td>
<td>04</td>
<td>08</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>00</td>
<td>36</td>
<td>72</td>
<td>FA</td>
<td>1D</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

Access 0x354

Access 0xA20

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

Locals in registers.

Assume a is aligned such that

\[ a[a][c] = aa...a rrrr cccc 000 \]

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

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

**Example #2 (E = 1)**

block = 16 bytes; 8 sets in cache

How many block offset bits?

How many set index bits?

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

16 bytes = 4 ints

If x and y are mutually aligned, e.g., 0x00, 0x80

If x and y are mutually unaligned, e.g., 0x00, 0xA0

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

This cache:

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

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

**Assume:** cold (empty) cache

32 bytes = 4 doubles every access a miss

16*16 = 256 misses

16 sets

32 bytes = 4 doubles

4 misses per row of array

4*16 = 64 misses

32 bytes = 4 doubles
Cache read: set-associative (Example: E = 2)

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

Address of int:
- t bits: 0...01 100

If no match: Evict and replace one line in set.

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?

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;
}
```

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.

**Example memory hierarchy**

**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.

**Typical laptop/desktop processor**

- Core 0
  - 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: 35-40 cycles
  - Block size: 64 bytes for all caches.
  - Slower, but more likely to hit

- Core 3
  - 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: 35-40 cycles
  - Block size: 64 bytes for all caches.
  - Slower, but more likely to hit

**Memory Hierarchy and Cache**
(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