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

Memory Hierarchy and Cache

Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming

How does execution time grow with SIZE?

```java
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 the relationship between time and size.](image)

## Processor-memory bottleneck

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

![Diagram illustrating the bandwidth and latency of different memory hierarchies.](image)

**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).
**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?

**Cache miss**

1. **Request data in block b.**

2. **Cache miss:** Block is not in cache

3. **Cache eviction:** Evict a block to make room, maybe store to memory.

4. **Cache fill:** Fetch block from memory, store in cache.

**Locality #1**

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

What is stored in memory?

Data:

```
n
```

Instructions:
Locality #2

row-major M x N 2D array in C

```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

row-major M x N 2D array in C

```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;
}
```

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/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**
- **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**
- **block ID**
- **offset within block**

Note: drawing address order differently from here on!
**Placement policy**

- **Small, fixed number of block slots.**
- **Large, fixed number of block slots.**

**Placement: direct-mapped**

- **S = # slots = 4**
- **Easy for power-of-2 block sizes...**

**Placement: mapping ambiguity?**

- **Which block is in slot 2?**

**Placement: tags resolve ambiguity**

- 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**

- Tag
- Index
- Offset

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

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

**Block ID**

- Offset within block  \( \log_2(\text{block size}) = b \)

- # 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?
  - \( \text{index(Block ID)} = \text{Block ID} / S \)
  - (still easy for power-of-2 block sizes...)

**Memory**

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

**Cache**

- Index
  - 00
  - 01
  - 10
  - 11

---

**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.  

<table>
<thead>
<tr>
<th><strong>Set</strong></th>
<th><strong>1-way</strong></th>
<th><strong>2-way</strong></th>
<th><strong>4-way</strong></th>
<th><strong>8-way</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
<td>1 set, 8 blocks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Sets</strong></th>
<th><strong>S = # slot sets in cache</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>direct mapped</td>
<td></td>
</tr>
<tr>
<td>fully associative</td>
<td></td>
</tr>
</tbody>
</table>

**Mapping:**  

index(Block ID) = 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  

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

Direct-mapped  
tag bits __________  
set index bits ______  
2-byte blocks block offset bits ______

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

\( E \)-way set-associative \( S \) slots  

<table>
<thead>
<tr>
<th><strong>Set</strong></th>
<th><strong>E = 1-way</strong></th>
<th><strong>E = 2-way</strong></th>
<th><strong>E = 4-way</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( S = 8 ) sets</td>
<td>( S = 4 ) sets</td>
<td>( S = 2 ) sets</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Tag, Index, Offset</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>tag bits</td>
</tr>
<tr>
<td>index(0x1833)</td>
</tr>
</tbody>
</table>

**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  

( )

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)

- Set index:
- Valid bit
- Block offset

**Address of byte in memory:**

$t$ bits  $s$ bits  $b$ bits

**Locate set by index**

Hit if any block in set:

- **is valid**; and
- **has matching tag**

**Get data at offset in block**

**If no match:**

Old line is evicted and replaced
### Direct-mapped cache practice

- **12-bit address**
- **16 lines, 4-byte block size**

**Direct mapped**

<table>
<thead>
<tr>
<th>Offset bits?</th>
<th>Index bits?</th>
<th>Tag bits?</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Access 0x354

**Access 0xA20**

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

- **Block = 16 bytes; 8 sets in cache**
- **How many block offset bits?**
- **How many set index bits?**

```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;
}
```

### Example #1 (E = 1)

**Locals in registers.**

- **Assume a is aligned such that**
  ```c
  &a[r][c] is aa...a rrrr cccc 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;
}
```

```c
int sum_array_cols(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;
}
```

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

**This block:**
- **8 bytes**
- **Associativity: 2 blocks per set**

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

#### Example: E = 2

```
\begin{array}{c|c}
|   |   |   |   |   |   |
\hline
0 | 1 | 2 | 3 | 4 | 5 \hline
1 | 2 | 3 | 4 | 5 | 6 \hline
2 | 3 | 4 | 5 | 6 | 7 \hline
3 | 4 | 5 | 6 | 7 | 8 \hline
\end{array}
```

**Find set**

```
\begin{array}{c|c}
|   |   |   |   |   |   |
\hline
0 | 1 | 2 | 3 | 4 | 5 \hline
1 | 2 | 3 | 4 | 5 | 6 \hline
2 | 3 | 4 | 5 | 6 | 7 \hline
3 | 4 | 5 | 6 | 7 | 8 \hline
\end{array}
```
Cache read: set-associative (Example: E = 2)

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

Address of int:

If no match: Evict and replace one line in 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

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

Example memory hierarchy

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