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?

```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
Processor-Memory Bottleneck

- Processor performance doubled about every 18 months

- Bus bandwidth evolved much slower

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

- Bandwidth: 2 Bytes/cycle
  Latency: 100 cycles

Solution: caches
Cache

English:

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

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

Also used more broadly in CS: software caches, file caches, etc.
Cache Memory

**General Cache Mechanics**

- **CPU**
  - **Block:** unit of data in cache and memory. (a.k.a. line)
  - Smaller, faster, more expensive.
  - Stores **subset of memory blocks**. (lines)

- **Cache**
  - Data is moved in block units
  - 8 9 14 3

- **Memory**
  - Larger, slower, cheaper.
  - **Partitioned into blocks** (lines).
  - 0 1 2 3
  - 4 5 6 7
  - 8 9 10 11
  - 12 13 14 15
Cache Hit

1. Request data in block b.

2. Cache hit:
   Block b is in cache.
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.


Placement Policy: where to put block in cache

Replacement Policy: which block to evict
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

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

What is stored in memory?

Data:

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;
}
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?
Cost of Cache Misses

Miss cost could be $100 \times \text{hit cost}$.

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 \text{ cycles} = 4 \text{ cycles}
99% hits: 1 cycle + 0.01 \times 100 \text{ cycles} = 2 \text{ 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?

- Persistent storage (hard disk, flash, over network, cloud, etc.)
- Main memory (DRAM)
- L3 cache (SRAM, off-chip)
- L2 cache (SRAM, on-chip)
- L1 cache (SRAM, on-chip)
- Registers

Program sees “memory”

Small, fast, power-hungry, expensive

Large, slow, power-efficient, cheap

Explicitly program-controlled
Cache Organization: Key Points

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 address bits - offset bits

offset within block log₂(block size)

Note: drawing address order differently from here on!

remember withinSameBlock? (Pointers Lab)
Placement Policy

Memory

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

Mapping:
index(Block ID) = ???

Cache

Index
00
01
10
11

S = # slots = 4

Small, fixed number of block slots.

Large, fixed number of block slots.
Placement: **Direct-Mapped**

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

*easy for power-of-2 block sizes...*
Placement: mapping ambiguity

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

Memory

Cache

Which block is in slot 2?

S = # slots = 4
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

<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>

Block ID bits - Index bits

$\log_2(\# \text{ cache slots})$

Tag

Index

00010010  full byte address

Block ID

Address bits - Offset bits

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

Offset within block

# address bits

# address bits
memory (
still easy for power-of-2 block sizes...)

why not this mapping?
index(Block ID) = Block ID / S

(block id)

index

00
01
10
11

1000
1001
1010
1011
1100
1101
1110
1111

1000
1001
1010
1011
1100
1101
1110
1111

placement: direct-mapped
A puzzle.

Cache starts *empty*.

Access (address, hit/miss) stream:

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

What could the block size be?
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>Set</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>

**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 usually implements “not most recently used”
Example: Tag, Index, Offset?

4-bit Address | Tag | Index | Offset

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

index(1101) = _____
Example: Tag, Index, Offset?

E-way set-associative
S 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

Set

0
1
2
3
4
5
6
7

tag bits
set index bits
block offset bits
index(0x1833)

E = 2-way
S = 4 sets

Set

0
1
2
3

tag bits
set index bits
block offset bits
index(0x1833)

E = 4-way
S = 2 sets

Set

0
1

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

associativity of cache?
General Cache Organization (S, E, B)

- **S** sets
- **E** lines per set ("E-way")
- **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

- Power of 2
- valid bit
- set
- block/line
Cache Read

E lines per set

S = $2^s$ sets

Hit if any block in set:
is valid; and
has matching tag

Get data at offset in block

Locate set by index

Address of byte in memory:

- t bits
- s bits
- b bits

- tag
- set index
- block offset

data begins at this offset

B = $2^b$ bytes of data per cache line (the data block)
Cache Read: Direct-Mapped \((E = 1)\)

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

\[ S = 2^s \text{ sets} \]
Cache Read: Direct-Mapped \((E = 1)\)

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

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>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</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>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</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>

<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>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
**Example (E = 1)**

Locals in registers.

Assume \( a \) is aligned such that

\[ &a[r][c] \text{ is } aa...a \text{ rrrr cccc 000} \]

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

Assume: cold (empty) cache

3-bit set index, 5-bit offset

\[ aa...arrrr \text{ rcc ccc000} \]

\[ 0,0: aa...a000 \text{ 000 00000} \]

**Table:**

<table>
<thead>
<tr>
<th>Index</th>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,4</td>
<td>0,5</td>
<td>0,6</td>
<td>0,7</td>
<td></td>
</tr>
<tr>
<td>0,8</td>
<td>0,9</td>
<td>0,a</td>
<td>0,b</td>
<td></td>
</tr>
<tr>
<td>0,c</td>
<td>0,d</td>
<td>0,e</td>
<td>0,f</td>
<td></td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td></td>
</tr>
<tr>
<td>1,4</td>
<td>1,5</td>
<td>1,6</td>
<td>1,7</td>
<td></td>
</tr>
<tr>
<td>1,8</td>
<td>1,9</td>
<td>1,a</td>
<td>1,b</td>
<td></td>
</tr>
<tr>
<td>1,c</td>
<td>1,d</td>
<td>1,e</td>
<td>1,f</td>
<td></td>
</tr>
</tbody>
</table>

32 bytes = 4 doubles

4 misses per row of array

4*16 = 64 misses

16*16 = 256 misses
Example \((E = 1)\)

```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;
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if \(x\) and \(y\) are mutually aligned, e.g., 0x00, 0x80

### Address bits:

- \(B = 16\) bytes = 4 ints
- \(S = 8\) sets in cache

#### How many block offset bits?

#### How many set index bits?

```
block = 16 bytes; 8 sets in cache
```

#### Addresses as bits

- 0x00000000:
- 0x00000080:
- 0x000000a0:

```
x[0]: x[1]: x[2]: x[3]
x[4]: x[5]: x[6]: x[7]
y[0]: y[1]: y[2]: y[3]
y[4]: y[5]: y[6]: y[7]
```

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

Address of int:

```
| t bits | 0...01 | 100 |
```

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

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

eax =
ecx = T
edx = U

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

```
eax =
ecx = T
edx = U
```

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.
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.
   b. Evict T (dirty: write back).
   c. Fill U.
   d. Set %eax.
5. DONE.
Example Memory Hierarchy

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

Processor package

Core 0

- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

Core 3

- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache

Main memory

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