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: 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];
  }
}
```

TIME

SIZE
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  
*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)

Data is moved in block units

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

---

**CPU**

**Cache**

8 9 14 3

**Memory**

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

---

Stores subset of memory blocks.
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.

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

---

**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

Data:

Instructions:

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

What is stored in memory?
### 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;
}
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
</tr>
<tr>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
</tr>
</tbody>
</table>
Locality #3

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 $100 \times \text{hit cost}$.  

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: $1 \text{ cycle} + 0.03 \times 100 \text{ cycles} = 4 \text{ cycles}$  
- 99% hits: $1 \text{ cycle} + 0.01 \times 100 \text{ cycles} = 2 \text{ cycles}$

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
  - explicitly program-controlled
  - small, fast, power-hungry, expensive

- main memory
  - (DRAM)
  - large, slow, power-efficient, cheap

- persistent storage
  - (hard disk, flash, over network, cloud, etc.)

- L3 cache
  - (SRAM, off-chip)

- L2 cache
  - (SRAM, on-chip)

- L1 cache
  - (SRAM, on-chip)

program sees “memory”
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: 00010010
- Block ID: address bits - offset bits
- Offset within block: \( \log_2(\text{block size}) \)

Note: drawing address order differently from here on!

\*Remember withinSameBlock? (Pointers Lab)\*
Placement Policy

Memory

<table>
<thead>
<tr>
<th>Block ID</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

Mapping:
index(Block ID) = ???

Cache

Index
00
01
10
11

Small, fixed number of block slots.

Large, fixed number of block slots.

S = # slots = 4
Placement: *Direct-Mapped*

**Memory**

<table>
<thead>
<tr>
<th>Block ID</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
</table>

**Mapping:**

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

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

**Cache**

- \( S = \# \text{ slots} = 4 \)
Placement: mapping ambiguity

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

Which block is in slot 2?

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

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

Tag | Index

00010010 \textit{full byte address}

Block ID | Offset within block

Address bits - Offset bits

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

\# address bits
Why not this mapping?
index(Block ID) = 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
A puzzle.

Cache starts *empty*.

Access (address, hit/miss) stream:

(10, miss), (11, hit), (12, 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:
index(Block ID) = Block ID mod S

Sets:
- 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 each

Sets S = # slots in cache

Direct mapped

Fully associative

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*

<table>
<thead>
<tr>
<th>E-way</th>
<th>S slots</th>
<th>16-bit Address</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-way</td>
<td>8 sets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-way</td>
<td>4 sets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-way</td>
<td>2 sets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>E = 1-way</th>
<th>S = 8 sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>E = 2-way</th>
<th>S = 4 sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>E = 4-way</th>
<th>S = 2 sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

tag bits | _____ | tag bits | _____ | tag bits | _____ |
set index bits | _____ | set index bits | _____ | set index bits | _____ |
block offset bits | _____ | block offset bits | _____ | block offset bits | _____ |
index(0x1833) | _____ | index(0x1833) | _____ | 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 \((S, E, B)\)

- **Powers of 2**
- **\(E\) lines per set** ("E-way")
- **\(S\) sets**
- **set**
- **block/line**

**cache capacity:**
\[S \times E \times B\] data bytes

**address size:**
\[t + s + b\] address bits

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

$E = 2^e \text{ lines per set}$

$S = 2^s \text{ sets}$

Locate set by index
Hit if any block in set: is valid; and
has matching tag
Get data at offset in block

Address of byte in memory:
- $t$ bits
- $s$ bits
- $b$ bits

tag
set
index
block
offset
data begins at this offset

valid bit

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

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

### Diagram

- $S = 2^5$ sets
- Address of int:
  - $t$ bits
  - 0...01
  - 100

- Find set

- Cache structure with tags and valid bits.
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?

Index | Tag | Valid | B0 | B1 | B2 | B3
--- | --- | --- | --- | --- | --- | ---
0 | 19 | 1 | 99 | 11 | 23 | 11
1 | 15 | 0 | – | – | – | –
2 | 1B | 1 | 00 | 02 | 04 | 08
3 | 36 | 0 | – | – | – | –
4 | 32 | 1 | 43 | 6D | 8F | 09
5 | 0D | 1 | 36 | 72 | F0 | 1D
6 | 31 | 0 | – | – | – | –
7 | 16 | 1 | 11 | C2 | DF | 03

Index | Tag | Valid | B0 | B1 | B2 | B3
--- | --- | --- | --- | --- | --- | ---
8 | 24 | 1 | 3A | 00 | 51 | 89
9 | 2D | 0 | – | – | – | –
A | 2D | 1 | 93 | 15 | DA | 3B
B | 0B | 0 | – | – | – | –
C | 12 | 0 | – | – | – | –
D | 16 | 1 | 04 | 96 | 34 | 15
E | 13 | 1 | 83 | 77 | 1B | D3
F | 14 | 0 | – | – | – | –
Example (E = 1)

Locals in registers.
Assume \( a \) is aligned such that
\[ &a[r][c] \] 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 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
\[ \text{aa...arrr rcc cc000} \]
\[ \text{0,0: aa...a000 000 00000} \]

<table>
<thead>
<tr>
<th></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
every access a miss
16*16 = 256 misses

|        | 0,0 | 0,1 | 0,2 | 0,3 |
| 3,0    | 3,1 | 3,2 | 3,3 |

32 bytes = 4 doubles
4 misses per row of array
4*16 = 64 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;
}
```

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

**Address bits:**
- \(B = \) [4 bits]
- \(S = \) [3 bits]

**Addresses as bits**
- \(0x00000000\): [4 bits]
- \(0x00000080\): [4 bits]
- \(0x000000A0\): [4 bits]

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

Address of int:

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

Find set:
Cache Read: Set-Associative (Example: E = 2)

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

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

Cache

U    0xCAFE

Memory

T    0xFAEC
U    0xCAFE

Tag

Dirty bit

eax =
ecx = T
edx = U

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.

- eax =
- ecx = T
- edx = U

Cache

Memory

T

U

0xFACE

0xCafe
Write-back, write-allocate example

```
eax = 0xCAFE
cx = 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.
   b. Evict T (dirty: write back).
   c. Fill U.
   d. Set %eax.
5. **DONE.**
### Example Memory Hierarchy

**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**

#### 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.

---

[49]
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