Memory Hierarchy and Cache

Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming

https://cs.wellesley.edu/~cs240/
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];
    }
}
```

Memory Hierarchy and Cache
Reality

![Graph showing the relationship between size and time. The graph illustrates a linear increase in time as the size increases.](image-url)
Processor-memory bottleneck

Processor performance doubled about every 18 months

Bus bandwidth evolved much slower

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

**Example**

**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).

**Cache**

8 9 14 3

**Memory**

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.

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

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

Data:

Instructions:

What is stored in memory?
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

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

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

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

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**
  - 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**
  - large, slow, power-efficient, cheap

Program sees "memory" explicitly
program-controlled
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

address bits - offset bits

offset within block

log₂(block size)

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

Memory Mapping:
\[ \text{index(Block ID)} = ??? \]

Small, fixed number of block slots.

Large, fixed number of block slots.
Placement: *direct-mapped*

Mapping:

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

(easy for power-of-2 block sizes...)
Placement: mapping ambiguity?

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

Which block is in slot 2?
Placement: tags resolve ambiguity

Memory

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

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

Cache
Tag
Data
00
00
01
11
10
01
11
01

Block ID bits not used for index.
Address = tag, index, offset

- Disambiguates slot contents.
- What slot in the cache?
- Where within a block?

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

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

<table>
<thead>
<tr>
<th>00010010</th>
</tr>
</thead>
</table>

Full address of individual byte in memory

Block ID
- Address bits - Offset bits
- \(\log_2(\text{block size}) = b\)

# address bits
Placement: direct-mapped

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

Memory

Index

Cache
Puzzle #1

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

(0xA, miss), (0xB, hit), (0xC, 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 \( \text{mod} \ S \)

- **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

direct mapped

fully associative

**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 | Tag | Index | Offset |

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

index(1101) = _____
Example: tag, index, offset? #2

**E-way set-associative**

S slots

16-byte blocks

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

<table>
<thead>
<tr>
<th>Set</th>
<th>Set</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</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>

16-bit Address

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

<table>
<thead>
<tr>
<th>tag bits</th>
<th>set index bits</th>
<th>block offset bits</th>
<th>index(0x1833)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory Hierarchy and Cache 31
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:
(0xA, miss); (0xB, miss); (0xA, miss)

associativity of cache?
General cache organization (S, E, B)

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

S = 2^s sets

E lines per set

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:

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)

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

```
<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<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>
<tbody>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

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

find set
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>1B</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>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>1B</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>

Access 0x354
Access 0xA20
Example #1 \((E = 1)\)

**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

\(aa...arrr\ rcc\ cc000\)

\(0,0: aa...a000\ 000\ 0000\)

32 bytes = 4 doubles

Every access a miss

16*16 = 256 misses

32 bytes = 4 doubles

4 misses per row of array

4*16 = 64 misses
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;
}
```

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

**Address bits:**
- \(B = \) ____________
- \(S = \) ____________

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

- **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:
\[
\begin{array}{c|c|c}
\text{t bits} & 0...01 & 100 \\
\end{array}
\]

find set

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

Memory Hierarchy and Cache
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 #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

```
eax = 0xCAFE
ecx = T
edx = U
```

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

Cache

```
U 0xCAFE 0
```

Memory

```
T 0xFACE
U 0xCAFE
```
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.
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.

eax = 0xCAFE
ecx = T
edx = U

Cache

Memory

T
U

0xCAFE
0xFEED
0xCAFE

dirty bit
tag
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
(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