Memory Hierarchy: Cache

Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming

How does execution time grow with SIZE?

```java
int[] array = new int[SIZE];
fillArrayRandomly(array);
int s = 0;

for (int i = 0; i < 200000; i++) {
    for (int j = 0; j < SIZE; j++) {
        s += array[j];
    }
}
```

reality beyond O(...)
**Processor-Memory Bottleneck**

Processor performance doubled about every 18 months

- **CPU**: Instruction fetching
- **Reg**: Instruction register
- **Cache**: Instruction cache

**Main Memory**

Bus bandwidth evolved much slower

---

**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**

- **CPU**: Instruction fetching
- **Cache**: Instruction cache
- **Memory**: Main memory

**Block**: unit of data in cache and memory.

- Smaller, faster, more expensive
- Stores subset of memory blocks (lines)

**Data is moved in block units**

**Memory**: Partitioned into blocks (lines)

---

**Cache Hit**

1. *Request data in block b.*
2. *Cache hit:* Block b is in cache.
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.


Example: Locality?

```c
int sum_array_rows(int a[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```

Data:
Temporal: sum referenced in each iteration
Spatial: array a[] accessed in `stride-1` pattern

Instructions:
Temporal: execute loop repeatedly
Spatial: execute instructions in sequence

Assessing locality in code is an important programming skill.
**Locality Example #2**

```c
int sum_array_cols(int a[M][N]) {
    int i, j, sum = 0;
    for (j = 0; j < N; j++) {
        for (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**

**Huge difference between a hit and a miss**
Could be 100x, if just L1 and main memory

**99% hits could be twice as good as 97%. How?**
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

**This is why "miss rate" is used instead of "hit rate"**

**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?

program sees “memory”; hardware manages caching transparently

small, fast, power-hungry, expensive

explicitly program-controlled

large, slow, power-efficient, cheap

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

explicitly
program-controlled

Cache Organization: Key Points

Block

Fixed-size unit of data in memory/cache

Placement Policy

Where should a given block be stored in the cache?

- 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 memory into fixed-size aligned blocks. power of 2

Example: block size = 8

full byte address

00010010

Block ID

offset within block

log₂(block size)

address bits - offset bits

Memory

Note: drawing addresses order differently from here out...

Placement Policy

Mapping:

index(Block ID) = ???

Large, fixed number of block slots.

Small, 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...)

- Memory
- Cache

**Block ID**

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

**Index**

-00
-01
-10
-11

**S = # slots = 4**

---

Placement: mapping ambiguity

**Mapping:**

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

- Memory
- Cache

**Block ID**

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

**Index**

-00
-01
-10
-11

**S = # slots = 4**

- Which block is in slot 2?

---

Placement: Tags resolve ambiguity

**Mapping:**

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

- Memory
- Cache

**Block ID**

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

**Index**

-00
-01
-10
-11

**Block ID bits not used for index.**

---

Address = Tag, Index, Offset

- Disambiguates slot contents.
- What slot in the cache?

**a-bit Address**

- Tag
- Index
- Offset

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

- Where within a block?

**Block ID**

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

**Block ID bits - Index bits**

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

**Offset within block**

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

**# address bits**

**full byte address**

- 00010 010
**Placement: Direct-Mapped**

- Why not this mapping?
  \[ \text{index(Block ID)} = \frac{\text{Block ID}}{S} \]
  (still easy for power-of-2 block sizes...)

**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:
  \[ 0010, 0110, 0010, 0110, 0010, \ldots \]?

**Placement: Set Associative**

- S = \# sets in cache

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

**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

- direct mapped
- fully associative
Example: Tag, Index, Offset?

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?

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
E = 2-way
S = 4 sets
E = 4-way
S = 2 sets

Replacement Policy

If set is full, what block should be replaced?
Common: least recently used (LRU)
(hardware usually implements “not most recently used”)

1-way associativity
8 sets, 1 block each
2-way associativity
4 sets, 2 blocks each
4-way associativity
2 sets, 4 blocks each

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**
  - Cache size: \( S \times E \times B \) data bytes
  - Address size: \( t + s + b \) address bits
  - Valid bit
  - \( B = 2^b \) bytes of data per cache line (the data block)

Cache Read

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

Read: Direct-Mapped (E = 1)

- Direct-mapped: One line per set
- Assume: cache block size 8 bytes

- Address of int: 4 bytes
- Find set
  - \( t \) bits
  - \( 0.01 \) 100

- If no match: old line is evicted and replaced
Assume:

- cache
- block size
- 8 bytes

**Example (E = 1)**

int sum_array_row(double a[16][16]){  
    int i, j;  
    double sum = 0;  
    for (i = 0; i < 16; i++)  
        for (j = 0; j < 16; j++)  
            sum += a[i][j];  
    return sum;  
}

int sum_array_cols(double a[16][16]){  
    int i, j;  
    double sum = 0;  
    for (i = 0; i < 16; i++)  
        for (j = 0; j < 16; j++)  
            sum += a[i][j];  
    return sum;  
}

**Read: Set-Associative (Example: E = 2)**

E = 2: Two lines per set  
Assume: cache block size 8 bytes

Example (E = 1)

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

Assume:

- sum, j in registers  
- Address of an aligned element of a: aa...arrrreec000

Assume: cold (empty) cache  
3-bit set index, 5-bit offset

<table>
<thead>
<tr>
<th>0,0: a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0: a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>0,0: a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

if x and y have aligned starting addresses,  
e.g., &x[0] = 0, &y[0] = 128

40, 41, 42, 43

If x and y have unaligned starting addresses,  
e.g., &x[0] = 0, &y[0] = 160

40, 41, 42, 43

Read: Set-Associative (Example: E = 2)

E = 2: Two lines per set  
Assume: cache block size 8 bytes

Address of int:  
1 bits 0..1 100

**Example (E = 1)**

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

Example (E = 1)

Address bits:  
B = 5  
S = 0: 128: 160:

If x and y have aligned starting addresses,  
e.g., &x[0] = 0, &y[0] = 128

40, 41, 42, 43

If x and y have unaligned starting addresses,  
e.g., &x[0] = 0, &y[0] = 160

40, 41, 42, 43

If no match:  
- One line in set is selected for eviction and replacement  
- Replacement policies: random, least recently used (LRU), ...
Example (E = 2)

```c
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;
    for (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
- first access to a block

Conflict miss
- cache has space for all needed blocks, but multiple blocks map to same slot e.g., referencing blocks 0, 8, 8, ... would miss every time
- increasing associativity can reduce conflict misses

Capacity miss
- working set of active cache blocks is larger than the cache

What about writes?

Multiple copies of data exist:
- L1, L2, possibly L3, main memory

Write-hit policy
- Write-through: write immediately to memory, all caches in between.
- Write-back: defer write to memory until line is evicted (replaced)
  - Need a dirty bit to indicate if line is different from memory or not

Write-miss policy
- Write-allocate: load into cache, update line in cache.
  - Good if more nearby writes or reads follow
- No-write-allocate: just write immediately to memory.

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

Write-back, write-allocate example

Cache

<table>
<thead>
<tr>
<th>U</th>
<th>0xCAFE</th>
</tr>
</thead>
</table>

Memory

<table>
<thead>
<tr>
<th>T</th>
<th>0xFACE</th>
</tr>
</thead>
</table>

| U | 0xCAFE |

Write-back, write-allocate example

1. `mov $T, %ecx`  
   `ecx = T`
2. `mov SU, %edx`  
   `edx = U`
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

Intel Core i7 circa 2011

- 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
- so, no placement restrictions
- index structures like hash tables are common (for placement)
- Often use complex replacement policies
- misses are very expensive when disk or network involved
- worth thousands of cycles to avoid them
- Not necessarily constrained to single "block" transfers
- may fetch or write-back in larger units, opportunistically
Cache-Friendly Code

Locality, locality, locality. ← locality

Programmer can optimize for cache performance
- Data structure organization
- Data access patterns
  - Nested loops
  - Blocking (see CSAPP 6.5)

All systems favor “cache-friendly code”
- Performance is hardware-specific
  - Cache size, line size, associativity, etc.
- Generic rules still capture most of advantages
  - Keep working set small (temporal locality)
  - Use small strides (spatial locality)
  - Focus on inner loop code