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];
    }
}
```
Processor-memory bottleneck

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

Main Memory

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)

Cache hit

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

Data is moved in block units

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

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

Memory

Request: 14

CPU

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.

**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:

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

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;
}
```

What is stored in memory?

**Locality #2**

row-major M x N 2D array in C

```
a[0][0] a[0][1] a[0][2] a[0][3]
a[1][0] a[1][1] a[1][2] a[1][3]
```

```
```
Locality #3

```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;
}
```

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

Cost of cache misses

Miss cost could be $100 \times$ 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: $(0.97 \times 1 \text{ cycle}) + (0.03 \times 100 \text{ cycles}) = 3.97 \text{ cycles}$
99% hits: $(0.93 \times 1 \text{ cycle}) + (0.01 \times 100 \text{ cycles}) = 1.93 \text{ cycles}$

Memory hierarchy

Why does it work?

small, fast, power-hungry, expensive

L1 cache (SRAM, on-chip)
<16MB, 0.5-25ns access, 5K-15K MBps

L2 cache (SRAM, on-chip)

L3 cache (SRAM, off-chip)

main memory (DRAM)
<~64MB, 80-250ns, 1K-5K MBps

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

large, slow, power-efficient, cheap

explicitly program-controlled

program sees "memory"
**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)!

**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
  - address bits - offset bits
  - log₂(block size)

**Memory Hierarchy and Cache**

**Placement policy**

**Mapping:**
- index(Block ID) = ???

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

Which block is in slot 2?

---

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

Tag

Index

Offset

(a-s-b) bits

s bits

b bits

Block ID bits - Index bits

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

Tag

Index

\(\text{0001010} \) full byte address

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: set-associative

One index per set of block slots.
Store block in any slot within set.

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  Tag  Index  Offset

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

index(1101) = ____

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?

Example: tag, index, offset? #2

E-way set-associative
S slots
16-byte blocks

E = 1-way
S = 8 sets

E = 2-way
S = 4 sets

E = 4-way
S = 2 sets

General cache organization (S, E, B)

E lines per set ("E-way")

Power of 2

S sets

block/line

set

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 lines per set**
- **S = 2^s sets**
- **B = 2^b bytes of data per cache line (the data block)**

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

**Memory Hierarchy and Cache**

**Cache read: direct-mapped (E = 1)**

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

**Direct-mapped cache practice**

- **Access 0x354**
- **Access 0xA20**

*offset bits? Index bits? Tag bits?*
**Example #1 (E = 1)**

Locals in registers.
Assume \( a \) is aligned such that
\[ a[r][c] = aa...arrr rcc ccc000 \]

```
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;
}
```

```
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;
}
```

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

Assume: cold (empty) cache

3-bit set index, 5-bit offset
\[ 0,0: a...a rrr rcc ccc000 \]

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

```
for (int i = 0; i < 8; i++) {
    sum += x[i]*y[i];
}
return sum;
```

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

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

```
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?
```

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

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

```
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;
}
```

**Memory Hierarchy and Cache**

```
```

```
```

```
```

```
```

```
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. \texttt{mov }$T, \%ecx
2. \texttt{mov }$U, \%edx
3. \texttt{mov }$0x\text{FEED}, \(%ecx)
   a. Miss on T.
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.
5. DONE.

Example memory hierarchy

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

Processor package

Core 0
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)
- Regs

Core 3
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache
- Regs

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