Memory Hierarchy and 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];
    }
}
```

Reality

![Graph showing the relationship between TIME and SIZE]
Processor-memory bottleneck

Processor performance doubled about every 18 months

Main Memory

Cache

Bandwidth: 256 bytes/cycle Latency: 3 cycles

Bus bandwidth evolved much slower

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

CPU

1. Request data in block b.

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

Cache hit

CPU

1. Request: 14

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

CPU

Request: 14

Cache

8 9 14 3

Memory

8 9 10 11

12 13 14 15

***************

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

Data is moved in block units

Block: unit of data in cache and memory. (a.k.a. line)

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

0 1 2 3

4 5 6 7

8 9 10 11

12 13 14 15

***************
**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?

**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.

**Memory hierarchy**

Why does it work?

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.

**Locality #1: Basic iteration over array**

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

What is stored in memory?
Locality #2: iteration over 2D array

```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;
}
```

Locality #3: iteration over 2D array

```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;
}
```

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 x 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 * 1 cycle) + (0.03 * 100 cycles) = 3.97 cycles
- 99% hits: (0.93 * 1 cycle) + (0.01 * 100 cycles) = 1.93 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!)

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

<table>
<thead>
<tr>
<th>Block ID</th>
<th>Offset within block</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010000</td>
<td>0000</td>
</tr>
<tr>
<td>00010000</td>
<td>0001</td>
</tr>
<tr>
<td>00010000</td>
<td>0010</td>
</tr>
<tr>
<td>00010000</td>
<td>0011</td>
</tr>
<tr>
<td>00010000</td>
<td>0100</td>
</tr>
<tr>
<td>00010000</td>
<td>0101</td>
</tr>
<tr>
<td>00010000</td>
<td>0110</td>
</tr>
<tr>
<td>00010000</td>
<td>0111</td>
</tr>
<tr>
<td>00010000</td>
<td>1000</td>
</tr>
<tr>
<td>00010000</td>
<td>1001</td>
</tr>
<tr>
<td>00010000</td>
<td>1010</td>
</tr>
<tr>
<td>00010000</td>
<td>1011</td>
</tr>
<tr>
<td>00010000</td>
<td>1100</td>
</tr>
<tr>
<td>00010000</td>
<td>1101</td>
</tr>
<tr>
<td>00010000</td>
<td>1110</td>
</tr>
<tr>
<td>00010000</td>
<td>1111</td>
</tr>
</tbody>
</table>

Placement policy

Mapping:
index(Block ID) = ???

Small, fixed number of block slots.

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

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

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

**Placement: mapping ambiguity?**

- **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

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

**Address = tag, index, offset**

- **a-bit Address**
  - Tag
  - Index
  - Offset
  - (a-s-b) bits
  - s bits
  - b bits

- **Block ID**
  - Offset within block
  - log(# cache slots)
  - Block ID bits - Index bits
  - # address bits

**Where within a block?**

- Disambiguates slot contents.

**What slot in the cache?**

- Tag
  - Index
  - Offset
  - Block ID
  - Offset within block
  - log(block size) = b

**Placement: tags resolve ambiguity**

- **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

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

**Block ID bits not used for index.**
**Cache size puzzle**

Cache starts *empty.*

Access (address, hit/miss) stream:

\[(0xA, \text{miss}), (0xB, \text{hit}), (0xC, \text{miss})\]

What could the block size be?

1. First, convert the hex to integers
2. Remember that blocks must be aligned to the block size
3. Hint: there are two possible block sizes!

---

**Example memory hierarchy**

**Software caches**

Examples

- File system buffer caches, web browser caches, database caches, network CDN caches, etc.

Some design differences

- 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
- 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
Example: Matrix Multiplication

```c
// Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k]*b[k*n + j];
}
```

Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size C is much smaller than n

Other iterations:
- Again:
  - n/8 + n = 9n/8 misses
  - (omitting matrix c)

Total misses:
- 9n/8 * n² = (9/8) * n³

Blocked Matrix Multiplication

```c
// Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
```
Cache Miss Analysis

Assume:
- Cache block = 64 bytes = 8 doubles
- Cache size $C \ll n$ (much smaller than $n$)
- Three blocks fit into cache: $3B^2 < C$

Other (block) iterations:
- Same as first iteration
- $2n/B * B^2/8 = nB/4$

Total misses:
- $nB/4 * (n/B)^2 = n^3/(4B)$

Summary

No blocking: $(9/8) * n^3$
Blocking: $1/(4B) * n^3$

If $B = 8$, difference is $4 * 8^3 / 8 = 36x$
If $B = 16$, difference is $4 * 16^3 / 8 = 72x$

Reason for dramatic difference:
Matrix multiplication has inherent temporal locality:
Every array element used $O(n)$ times!
But program has to be written properly

Exercise: order these 3 functions by locality

```c
typedef struct {
    int vel[3];
    int acc[3];
} point;

void clear1(point *p, int n) {
    for (i=0; i<n; i++)
        for (j=0; j<3; j++)
            p[i].vel[j] = 0;
    for (i=0; i<n; i++)
        for (j=0; j<3; j++)
            p[i].acc[j] = 0;
}

void clear2(point *p, int n) {
    for (i=0; i<n; i++)
        for (j=0; j<3; j++)
            p[i].vel[j] = 0;
    for (i=0; i<n; i++)
        for (j=0; j<3; j++)
            p[i].acc[j] = 0;
}

void clear3(point *p, int n) {
    for (j=0; j<3; j++)
        for (i=0; i<n; i++)
            p[i].vel[j] = 0;
    for (j=0; j<3; j++)
        for (i=0; i<n; i++)
            p[i].acc[j] = 0;
}
```

```c
#define N 100
point p[N];
```