Roadmap

- Memory & Data
- Integers/FP
- Machine code
- Assembly programming (x86)
- Procedures/Stacks
- Memory and caches  ➞ We are here!
- Processes
- Virtual Memory
- Memory Allocation
- Java vs C

Making memory accesses fast!

- Memory hierarchy, caches, locality
- Cache organization
- Program optimizations that consider caches
How does execution time grow with SIZE?

```c
int array[SIZE];
int A = 0;

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

---

Actual Data

![Plot](image-url)
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus bandwidth evolved much slower

Core 2 Duo:
Can process at least 256 Bytes/cycle

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100 cycles

Main Memory

Problem: lots of waiting on memory

Solution: caches
Cache

- **English definition**: a hidden storage space for provisions, weapons, and/or treasures

- **CSE Definition**: computer memory with short access time used for the storage of frequently or recently used instructions or data (i-cache and d-cache)

  more generally,

  used to optimize data transfers between system elements with different characteristics (network interface cache, I/O cache, etc.)

---

General Cache Mechanics

<table>
<thead>
<tr>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”
General Cache Concepts: **Hit**

Data in block $b$ is needed

Request: 14

Cache

| 8 | 9 | 14 | 3 |

Memory

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
General Cache Concepts: **Hit**

Request: 14

Data in block b is needed

Block b is in cache: Hit!

<table>
<thead>
<tr>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 9 14 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td></td>
<td>4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>8 9 10 11</td>
</tr>
<tr>
<td></td>
<td>12 13 14 15</td>
</tr>
</tbody>
</table>

General Cache Concepts: **Miss**

Request: 12

Data in block b is needed

<table>
<thead>
<tr>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 9 14 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td></td>
<td>4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>8 9 10 11</td>
</tr>
<tr>
<td></td>
<td>12 13 14 15</td>
</tr>
</tbody>
</table>
General Cache Concepts: **Miss**

Data in block b is needed

Block b is not in cache: **Miss!**

Request: 12

Cache

| 8 | 9 | 14 | 3 |

Memory

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

---

General Cache Concepts: **Miss**

Data in block b is needed

Block b is fetched from memory

Request: 12

Cache

| 8 | 9 | 14 | 3 |

Memory

| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
General Cache Concepts: **Miss**

**Data in block b is needed**

**Block b is not in cache:** *Miss!*

**Block b is fetched from memory**

**Block b is stored in cache**

- **Placement policy:** determines where b goes
- **Replacement policy:** determines which block gets evicted (victim)

---

**Lets think about those numbers**

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

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles
Let's think about those numbers

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

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles
  
  - Average 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”

Why Caches Work

- **Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently
Why Caches Work

- **Locality:** Programs tend to use data and instructions with 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
  - Why is this important?

- **Spatial locality?**
Why Caches Work

- **Locality:** Programs tend to use data and instructions with 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 *tend* to be referenced close together in time
  - How do caches take advantage of this?

Example: Locality?

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

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

- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

- **Being able to assess the locality of code is a crucial skill for a programmer**
Locality Example #1

```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;
}
```

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;
}
```
Locality Example #3

```c
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

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

Memory Hierarchies

- Some fundamental and enduring properties of hardware and software systems:
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
    - Well-written programs tend to exhibit good locality

- These properties complement each other beautifully

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy
An Example Memory Hierarchy

- **L0:** CPU registers
  - Hold words retrieved from L1 cache

- **L1:** on-chip L1 cache (SRAM)
  - Hold cache lines retrieved from L2 cache
  - Smaller, faster, costlier per byte

- **L2:** off-chip L2 cache (SRAM)
  - Hold cache lines retrieved from main memory
  - Larger, slower, cheaper per byte

- **L3:** main memory (DRAM)
  - Hold disk blocks retrieved from local disks
  - Local disks hold files retrieved from disks on remote network servers

- **L4:** local secondary storage (local disks)
  - Hold files retrieved from local disks

- **L5:** remote secondary storage (distributed file systems, web servers)
  - Hold files retrieved from remote networks

### Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses)
  - $= 1 – \text{hit rate}$
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.

- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (trend: increasing!)
Examples of Caching in the Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4/8-byte words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-bytes block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-bytes block</td>
<td>Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware+OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>File system client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web server</td>
</tr>
</tbody>
</table>

Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

Throughput: 16 B/cycle
Latency: 3 cycles

Throughput: 8 B/cycle
Latency: 14 cycles

Throughput: 2 B/cycle
Latency: 100 cycles

Throughput: 1 B/30 cycles
Latency: millions

~4 MB
L2 unified cache

~4 GB
Main Memory

~500 GB
Disk
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes just one) of the available cache slots
    - if one (e.g., block \( i \) must be placed in slot \( i \) mod size),
      - direct-mapped
    - if more than one,
      - \( n \)-way set-associative (where \( n \) is a power of 2)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time

- **Capacity miss**
  - Occurs when the set of active cache blocks (the working set) is larger than the cache (just won’t fit)

---

General Cache Organization \( (S, E, B) \)

- \( E = 2^e \) lines per set
- \( S = 2^s \) sets
- \( B = 2^b \) bytes data block per cache line (the data)
- \( cache size: S \times E \times B \) data bytes
**Cache Read**

- \( E = 2^s \) lines per set
- \( S = 2^s \) sets

Address of word:
- \( t \) bits
- \( s \) bits
- \( b \) bits

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

**Example: Direct-Mapped Cache (E = 1)**

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

Address of int:
- \( t \) bits
- \( 0...01 \) 100

Find set
Example: Direct-Mapped Cache (E = 1)

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

No match: old line is evicted and replaced
Example (for E = 1)

```c
int sum_array_rows(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;
}
```

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

Assume sum, i, j in registers
Address of an aligned element
of a: aa...aaaaaaaaxyyyyy000

Assume: cold (empty) cache
3 bits for set, 5 bits for byte
aa...aaaa xxy yyyy000

32 B = 4 doubles
4 misses per row of array
4*16 = 64 misses
every access a miss
16*16 = 256 misses

Example (for E = 1)

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

if x and y have unaligned starting addresses,
e.g., &x[0] = 0, &y[0] = 144

Autumn 2012
Memory Organization
E-way Set-Associative Cache (Here: E = 2)

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

Address of short int:

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

find set

Autumn 2012
Memory Organization
E-way Set-Associative Cache (Here: E = 2)

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

No match:
• One line in set is selected for eviction and replacement
• Replacement policies: random, least recently used (LRU), ...

Example (for 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 have aligned starting addresses, e.g., &x[0] = 0, &y[0] = 128
still can fit both because 2 lines in each set
Fully Set-Associative Caches ($S = 1$)

- All lines in one single set, $S = 1$
  - $E = C / B$, where $C$ is total cache size
  - Since, $S = (C / B) / E$, therefore, $S = 1$

- Direct-mapped caches have $E = 1$
  - $S = (C / B) / E = C / B$

- Tags are more expensive in associative caches
  - Fully-associative cache, $C / B$ tag comparators
  - Direct-mapped cache, 1 tag comparator
  - In general, $E$-way set-associative caches, $E$ tag comparators

- Tag size, assuming $m$ address bits ($m = 32$ for IA32)
  - $m - \log_2 S - \log_2 B$

Typical Memory Hierarchy (Intel Core i7)

- L0: CPU registers (optimized by compiler)
- L1: on-chip L1 cache (SRAM) 8-way associative in Intel Core i7
- L2: off-chip L2 cache (SRAM) 8-way associative in Intel Core i7
- L3: off-chip cache L3 shared by multiple cores (SRAM) 16-way associative in Intel Core i7
- L4: main memory (DRAM)
- L5: local secondary storage (local disks)
- L6: remote secondary storage (distributed file systems, web servers)
What about writes?

- Multiple copies of data exist:
  - L1, L2, Main Memory, Disk

- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (to indicate if line is different from memory or not)

- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)

- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

Software Caches are More Flexible

- Examples
  - File system buffer caches, web browser 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
Optimizations for the Memory Hierarchy

- **Write code that has locality**
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- **How to achieve?**
  - Proper choice of algorithm
  - Loop transformations

- **Cache versus register-level optimization:**
  - In both cases locality desirable
  - Register space much smaller
  - Requires scalar replacement to exploit temporal locality
  - Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)

---

**Example: Matrix Multiplication**

```c
s = (double *) calloc(sizeof(double), n*n);

/* 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];
}
```

\[
\begin{array}{ccc}
  c & = & a \times b \\
  i & \times & j \\
\end{array}
\]
Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size $C << n$ (much smaller than $n$)

- First iteration:
  - $n/8 + n = 9n/8$ misses (omitting matrix $c$)
  - Afterwards in cache: (schematic)

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

- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$
Blocked Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

/* 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; i++)
                        for (j1 = j; j1 < j+B; j++)
                            for (k1 = k; k1 < k+B; k++)
                                c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}

---

Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Four blocks fit into cache: $4B^2 < C$

- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)
Cache Miss Analysis

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

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

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

Summary

- No blocking: $(9/8) \times n^3$
- Blocking: $1/(4B) \times n^3$
- If $B = 8$ difference is $4 \times 8 \times 9 / 8 = 36x$
- If $B = 16$ difference is $4 \times 16 \times 9 / 8 = 72x$

- Suggests largest possible block size B, but limit $4B^2 < C$!
  (can possibly be relaxed a bit, but there is a limit for B)

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^3$, computation $2n^3$
    - Every array element used $O(n)$ times!
  - But program has to be written properly
The Memory Mountain

Read throughput (MB/s)

Stride (words)

Working set size (bytes)

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified L2 cache

Mem

L1

L2