Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimizations that consider caches

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

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**

![Diagram of cache and memory with data transfer](image)

- **Cache**
  - Smaller, faster, more expensive memory caches a subset of the blocks
  - Data is copied in block-sized transfer units

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

Request: 14

Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss

Request: 12

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)
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)

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
  - Average access time:
    - 97% hits: $1 \text{ cycle} + 0.03 \times 100 \text{ cycles} = 4 \text{ cycles}$
    - 99% hits: $1 \text{ cycle} + 0.01 \times 100 \text{ cycles} = 2 \text{ cycles}$

- This is why “miss rate” is used instead of “hit rate”
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 \((i \mod \text{size})\)), **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)

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
Example: Locality?

```c
int 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;
}
```

```
0: a[0][0]
1: a[0][1]
2: a[0][2]
3: a[0][3]
4: a[1][0]
5: a[1][1]
6: a[1][2]
7: a[1][3]
8: a[2][0]
9: a[2][1]
10: a[2][2]
11: a[2][3]
12: a[2][3]
```

`stride-1`
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: registers
L1: on-chip L1 cache (SRAM)
L2: off-chip L2 cache (SRAM)
L3: main memory (DRAM)
L4: local secondary storage (local disks)
L5: remote secondary storage (distributed file systems, web servers)

CPU registers hold words retrieved from L1 cache
L1 cache holds cache lines retrieved from L2 cache
L2 cache holds cache lines retrieved from main memory
Main memory holds disk blocks retrieved from local disks
Local disks hold files retrieved from disks on remote network servers

Smaller, faster, costlier per byte
Larger, slower, cheaper per byte
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-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>1,000,000</td>
<td>File system client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>1,000,000</td>
<td>Web system client</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:
L1 D-cache: 16 B/cycle, 3 cycles
L1 unified cache: 8 B/cycle, 14 cycles
L2 unified cache: 2 B/cycle, 100 cycles
Main Memory: 1 B/30 cycles, millions
Disk: ~500 GB

Latency:
L1 D-cache: 32 KB
L1 unified cache: ~4 MB
L2 unified cache: ~4 GB
Main Memory: ~4 GB
Disk: ~500 GB

Not drawn to scale
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

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

Address of word:

- t bits
- s bits
- b bits

- data begins at this offset

• 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:

\[ \begin{array}{c|c|c|c|c|c|c|c}
   & t \text{ bits} & 0 & 01 & 100 \\
\hline
\text{valid?} & \text{match: assume yes = hit} & \\
\text{tag} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

**S** = \(2^E\) sets

block offset

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;
}
```
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

E-way Set-Associative Cache (Here: E = 2)

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

Address of short int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

Find set
E-way Set-Associative Cache (Here: E = 2)

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

Address of short int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

Compare both:

valid? + match: yes = hit

Block offset:

V tag 0 1 2 3 4 5 6 7

V tag 0 1 2 3 4 5 6 7

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
  - $S = 1 = (C / B) / E$

- 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)

- CPU registers (optimized by compiler)
- on-chip L1 cache (SRAM)
  - 8-way associative in Intel Core i7
- off-chip L2 cache (SRAM)
  - 8-way associative in Intel Core i7
- off-chip L3 cache shared by multiple cores (SRAM)
  - 16-way associative in Intel Core i7
- main memory (DRAM)
- local secondary storage (local disks)
- 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 (line 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

---

The Memory Mountain

<table>
<thead>
<tr>
<th>Working set size (bytes)</th>
<th>Stride (words)</th>
<th>Read throughput (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B</td>
<td>1</td>
<td>1200</td>
</tr>
<tr>
<td>16B</td>
<td>1</td>
<td>1100</td>
</tr>
<tr>
<td>32B</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>64B</td>
<td>1</td>
<td>900</td>
</tr>
<tr>
<td>128B</td>
<td>1</td>
<td>800</td>
</tr>
<tr>
<td>256B</td>
<td>1</td>
<td>700</td>
</tr>
<tr>
<td>512B</td>
<td>1</td>
<td>600</td>
</tr>
<tr>
<td>1KB</td>
<td>1</td>
<td>500</td>
</tr>
<tr>
<td>2KB</td>
<td>1</td>
<td>400</td>
</tr>
<tr>
<td>4KB</td>
<td>1</td>
<td>300</td>
</tr>
<tr>
<td>8KB</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>16KB</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>32KB</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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
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
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++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k]*b[k*n + j];
}
```

\[
\begin{align*}
C &= A \times B \\
C[i] &= \sum_{k=0}^{n-1} A[i][k] \times B[k][j]
\end{align*}
\]
Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (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:
  - $9n/8 \times n^2 = (9/8) \times n^3$
**Blocked Matrix Multiplication**

```c
double *c = 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: $3n^2$, computation $2n^3$
    - Every array elements used $O(n)$ times!
  - But program has to be written properly