Making memory accesses fast!

- What we want: Memories that are
  - **Big**
    - Fast
    - Cheap
- Hardware: Pick any two
  - So we’ll be clever...
    - Pick “fast and cheap” (but not big), and
    - “big and cheap” (but not fast)

Problem: Processor-Memory Bottleneck

Solution: caches

Cycle time vs. Memory Access Time

- Processor vs Memory Performance

1980: no cache in microprocessor;
1995 2-level cache

How does cache affect performance?

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

**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 (l-cache and d-cache)
- Used to optimize data transfers between system elements with different characteristics (main memory cache, network interface cache, I/O cache, etc.)

**General Cache Mechanics**

- Smaller, faster, more expensive memory caches a subset of the blocks
- 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
  - Placement policy: determines where b goes
  - Replacement policy: determines which block gets evicted (victim)
Cache Organization Questions

- How can the processor quickly determine whether a memory reference is a hit or a miss?
- If there's a miss, where in the cache should we put the data we have to retrieve from the layer(s) above?
  - Which data item currently in the cache should we overwrite with the new data?
- On a read miss, should the new data be put in the cache at all?
- On a write miss, should the written data be (a) put in the cache, or (b) written to the higher layer(s), or (c) both?

Core i7-8700K (2017)

Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches

Example: Locality?

```
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 (and blocks are larger than one word)

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

- A high cache hit rate is essential to good performance
- Both the hardware designer and the programmer are concerned with it
- Being able to assess the locality of code is a crucial skill for a programmer

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?
Another Locality Example

```c
int sum_array_2d(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            sum += a[j][i];
    return sum;
}
```

What is “wrong” with this code?
• How can it be fixed?

• Array is stored in row major order
• Accesses are in column major order
• No spatial locality

Memory and Caches
• Cache basics
• Principle of locality
• Memory hierarchies
• Cache organization
• Program optimizations that consider caches

Cost of Cache Misses
• 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 = 1.03 cycles
    • 99% Hits: 1 cycle + 0.01 * 100 cycles = 1.01 cycles

  • This is why “miss rate” is used instead of “hit rate”

Cache Performance Metrics
• Miss Rate
  • Fraction of memory references not found in cache (misses / accesses)
    • 1 - hit rate
  • Typical numbers (in percentages):
    • 3% - 10% for L1
    • Can be worse for higher level

• 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 hit times: 1 - 2 clock cycles for L1

• Miss Penalty
  • Additional time required because of a miss
    • Typically 50 - 200 cycles

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
Memory Hierarchies

- Fundamental idea of a memory hierarchy:
  - Each level, k, serves as a cache for the larger, slower, level, k+1, behind it

- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

- Big idea: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Intel Core i7 Cache Hierarchy

Processor package

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

Core 3
- L1 cache
- L2 unified cache
- L3 unified cache (shared by all cores)

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

Main memory

Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches

An Example Memory Hierarchy

- Smaller, faster, cache per byte
- Larger, slower, cheaper per byte

Remote secondary storage (distributed file systems, web servers)

Intel i7 Die

Core i7-8700K (2017)
Putting data in the cache

Looking up data in the cache: How?

Looking up data: Tags

Fully Associative Caches

Direct Mapped Caches
Direct Mapped Caches

Address → tag | index

- Q: Why use low order bits for index and high bits for tag, rather than vice versa?
  - tag | index, or
  - index | tag

- A: Programs tend to sweep through memory sequentially
  - e.g., instructions
  - e.g., arrays
- AND by using the low order bits as the index, sequential memory makes use of the entire cache

A small optimization

- We don't need to save the low order bits of the address in the tag storage, because they're implied by the line number

Exploiting Spatial Locality – Cache Blocks

What’s a cache block? (or cache line)

What’s a cache block? (or cache line)
Set Associative Caches

- What if we could store data in any place in the cache?
- That's called a fully-associative cache
- But that might slow down caches... so we do something in between.

Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
- Block size is 16 bytes.
- 0x1833 in binary is 00...01 1000 0011 0011.

Example placement in set-associative caches

- Block size is 16 bytes.
- 0x1833 in binary is 00...01 1000 0011 0011.

Example placement in set-associative caches

- Block replacement

Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization (part 2)
- Program optimizations that consider caches
Cache Read

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

Example: Direct-Mapped Cache (E = 1)

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

- Address of byte in memory:
  - Tag
  - 0...s bits
  - Block
- Data begins at this offset

Example: Direct-Mapped Cache (E = 1)

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

- Address of int:
  - Tag
  - 0...s bits

Example: Direct-Mapped Cache (E = 1)

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

- Address of short int:
  - Tag
  - 0...b bits

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

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

- Address of short int:
  - Tag
  - 0...b bits

Example: Direct-Mapped Cache (E = 1)

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

- 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

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

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

- 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
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 j, 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, 1, 2, ... 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)

What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory
- What is the main problem with that?

What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until line is evicted)
    - 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 (just write immediately to memory)
- Typical caches:
  - Write-back + Write-allocate, usually
  - Write-through + No-write-allocate, occasionally
Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches

Why Caches Work

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

  ![Temporal locality block diagram](image)

- **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

Optimizations for the Memory Hierarchy

- Write code that has good 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

Example: Matrix Multiplication

```c
double *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];
}
```

(Simplistic) Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 64 bytes = 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)

(Simplistic) Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n (much smaller than n)
- Other iterations:
  - Again: n/8 + n = 9n/8 misses (omitting matrix c)
- Total misses:
  - 9n/8 * n^3 = 9/8 * n^4
Blocked Matrix Multiplication

\[ c = (\text{double} *) \text{calloc}(\text{sizeof(double)}, n \times n); \]

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

(Simplistic) Cache Miss Analysis

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

• First (block) iteration:
  • \(B^2/8\) misses for each block
  • \(2n/B \times B^2/8 = nB/4\) (omitting matrix c)

• Afterwards in cache (schematic)

Summary

• No blocking: \(\lfloor n/8 \rfloor \times n^3\)
• Blocking: \(1/(4B) \times n^3\)
  • (No blocking)/Blocking = 9B/2
    • If \(B = 8\) ratio is 36x
    • If \(B = 16\) ratio is 72x

• Suggests using largest possible block size B (but limit \(3B^2 < C\))

• Reason for this difference:
  Matrix multiplication has inherent temporal locality:
  • Input data: \(3n^2\), computation \(2n^3\)
  • Every array element used \(O(n)\) times!
  • But program has to be written properly

Cache-Friendly Code

• Programmer can optimize for cache performance
  • How data structures are organized
  • Enhance spatial locality
  • How data are accessed (e.g., nested loop structure)
  • Enhance temporal locality
  • All systems favor “cache-friendly code”
  • Keep working set reasonably small (temporal locality)
  • Use small strides (spatial locality)
  • Focus on inner loop code

  • Getting absolute optimum performance is very platform specific
    • Cache sizes, line sizes, associativities, etc.