Caches I
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun
Administrivia

- Homework 2 due today at 5pm
- Lab 3 due next Thursday

- Midterm will be graded over the weekend
  - Posted solutions not set in stone
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
How does execution time grow with SIZE?

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

for (int i = 0; i < 200000; i++) {
    for (int j = 0; j < SIZE; j++) {
        sum += array[j]; // execute SIZE \times 200000 times
    }
}
```

**Plot**

- **Time** vs. **SIZE**

- * expect linear
Actual Data

![Graph showing the relationship between cache size and time. The graph has a kink at a certain size, indicating a change in behavior.]
Making memory accesses fast!

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

1989 first Intel CPU with cache on chip
1998 Pentium III has two cache levels on chip

“Moore’s Law”

Processor-Memory Performance Gap (grows 50%/year)

µProc 55%/year (2X/1.5yr)

DRAM 7%/year (2X/10yrs)
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower

Main Memory

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

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100-200 cycles (30-60ns)

Problem: lots of waiting on memory

cycle: single machine step (fixed-time)
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower

Main Memory

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

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100-200 cycles (30-60ns)

Solution: caches

cycle: single machine step (fixed-time)
Cache 💰

- **Pronunciation**: “cash”
  - We abbreviate this as “$”

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

- **Computer**: Memory with short access time used for the storage of frequently or recently used instructions (i-cache/I$) or data (d-cache/D$)
  - More generally: Used to optimize data transfers between any system elements with different characteristics (network interface cache, I/O cache, etc.)
General Cache Mechanics

- Larger, slower, cheaper memory.
- Viewed as partitioned into “blocks” or “lines”.

Larger, faster, more expensive memory.
Caches a subset of the blocks (a.k.a. lines)

Data is copied in block-sized transfer units.

Cache

| 7 | 9 | 14 | 3 |

Memory

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

Data in block b is needed

Block b is in cache: **Hit!**
General Cache Concepts: **Miss**

1. **Data in block b is needed**
2. **Block b is fetched from memory**
3. **Block b is stored in cache**
   - Placement policy: determines where b goes
   - Replacement policy: determines which block gets evicted (victim)
4. **Data returned to CPU**
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 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: Any 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
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 #1

**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;
}

**M = 3, N=4**

Access Pattern:

```
1) a[0][0]  2) a[0][1]  3) a[0][2]  4) a[0][3]
```

```
5) a[1][0]  6) a[1][1]  7) a[1][2]  8) a[1][3]
```

```
```

**Note:** 76 is just one possible starting address of array a.
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 #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;
}
```

M = 3, N=4

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

Layout in Memory:
```
\begin{array}{cccc}
  a[0][0] & a[0][1] & a[0][2] & a[0][3] \\
  a[1][0] & a[1][1] & a[1][2] & a[1][3] \\
\end{array}
```

Stride = ?
Locality Example #3

What is wrong with this code?

```
int sum_array_3D(int a[M][N][L])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < L; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];

    return sum;
}
```

How can it be fixed?

```
int sum_array_3D(int a[M][N][L])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < L; j++)
            sum += a[k][i][j];

    return sum;
}
```
Locality Example #3

```c
int sum_array_3D(int a[M][N][L])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < L; 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?

```
inner loop:   k = stride-N+L
              i = stride-L
              j = stride-1
```
Cache Performance Metrics

- Huge difference between a cache hit and a cache miss
  - Could be 100x speed difference between accessing cache and main memory (measured in clock cycles)

- Miss Rate (MR)
  - Fraction of memory references not found in cache (misses / accesses) = 1 - Hit Rate

- Hit Time (HT)
  - Time to deliver a block in the cache to the processor
    - Includes time to determine whether the block is in the cache

- Miss Penalty (MP)
  - Additional time required because of a miss
Cache Performance

- Two things hurt the performance of a cache:
  - Miss rate and miss penalty

- Average Memory Access Time (AMAT): average time to access memory considering both hits and misses

  \[
  \text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}
  \]

  (abbreviated \(\text{AMAT} = \text{HT} + \text{MR} \times \text{MP}\))

- 99% hit rate twice as good as 97% hit rate!

  - Assume HT of 1 clock cycle and MP of 100 clock cycles
    - 97%: \(\text{AMAT} = 1 + (1 - 0.97) \times 100 = 1 + 7 = 41 \text{ clock cycles}\)
    - 99%: \(\text{AMAT} = 1 + (1 - 0.99) \times 100 = 1 + 1 = 2 \text{ clock cycles}\)
Peer Instruction Question

- **Processor specs:** 200 ps clock, MP of 50 clock cycles, MR of 0.02 misses/instruction, and HT of 1 clock cycle

  \[
  \text{AMAT} = \left( 1 + 50 \times 0.02 \right) = 1 + 1 = 2 \text{ clock cycle, } 200 \text{ ps}
  \]

- **Which improvement would be best?**
  - 190 ps clock (overclocking)
    \[
    1 + 50 \times 0.02 = 1 + 1 = 2 \text{ clock cycle, } 190 \text{ ps}
    \]
  - MP of 40 clock cycles (changing mean - smaller)
    \[
    1 + 40 \times 0.02 = 1.8 \text{ clock cycle, } 360 \text{ ps}
    \]
  - MR of 0.015 misses/instruction (better locality in code)
    \[
    1 + 50 \times 0.015 = 1.75 \text{ clock cycle, } 350 \text{ ps}
    \]
Can we have more than one cache?

- Why would we want to do that?
  - Avoid going to memory!

- Typical performance numbers:
  - Miss Rate
    - L1 MR = 3-10%
    - L2 MR = Quite small (e.g., < 1%), depending on parameters, etc.
  - Hit Time
    - L1 HT = 4 clock cycles
    - L2 HT = 10 clock cycles
  - Miss Penalty
    - P = 50-200 cycles for missing in L2 & going to main memory
    - Trend: increasing!
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

- **CPU**
  - Registers: <1 ns
  - On-chip L1 cache (SRAM): 1 ns
  - Off-chip L2 cache (SRAM): 5-10 ns
- **Main Memory** (DRAM):
  - Local secondary storage (local disks):
    - SSD: 100 ns
    - Disk: 150,000 ns
  - Remote secondary storage (distributed file systems, web servers):
    - 10,000,000 ns (10 ms)
  - Remote secondary storage:
    - 1-150 ms
  - 1-5 minutes
  - 1-2 minutes
  - 15-30 minutes
  - 1-2 minutes
  - 15-30 minutes
  - 31 days
  - 66 months = 1.3 years
  - 1 - 15 years
An Example Memory Hierarchy

- 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
An Example Memory Hierarchy

- **Registers**
  - Larger, slower, cheaper per byte
  - Explicitly program-controlled (e.g. refer to exactly `%rax, %rbx`)

- **On-chip L1 cache (SRAM)**
  - Smaller, faster, costlier per byte
  - Program sees "memory"; hardware manages caching transparently

- **Off-chip L2 cache (SRAM)**

- **Main memory (DRAM)**

- **Local secondary storage (local disks)**

- **Remote secondary storage (distributed file systems, web servers)**
Memory Hierarchies

- **Fundamental idea of a memory hierarchy:**
  - For each level $k$, the faster, smaller device at level $k$ serves as a cache for the larger, slower device at level $k+1$

- **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
- Registers
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3
- Registers
- L1 d-cache
- L1 i-cache
- L2 unified cache

L3 unified cache (shared by all cores)

Main memory

Block size:
- 64 bytes for all caches.

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
Summary

- **Memory Hierarchy**
  - Successively higher levels contain “most used” data from lower levels
  - Exploits *temporal and spatial locality*
  - Caches are intermediate storage levels used to optimize data transfers between any system elements with different characteristics

- **Cache Performance**
  - Ideal case: found in cache (hit)
  - Bad case: not found in cache (miss), search in next level
  - Average Memory Access Time (AMAT) = HT + MR × MP
    - Hurt by Miss Rate and Miss Penalty