Lecture 17: Memory Architecture

CSE 374: Intermediate Programming Concepts and Tools
Administrivia

Be kind to yourself and one another 😊

Reminder: Midpoint Deadline Friday November 6th at 9pm PST
Thought experiment

```java
public int sum1(int n, int m, int[][] table) {
    int output = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            output += table[i][j];
        }
    }
    return output;
}

public int sum2(int n, int m, int[][] table) {
    int output = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            output += table[j][i];
        }
    }
    return output;
}
```

What do these two methods do?

What is the big-$\Theta$ $\Theta(n^m)$?
Thought Experiment Graphed

Running sum1 vs sum2 on tables of size n x 4096

- Blue line: sum1
- Red line: sum2

Time (in ms) vs Table height
Incorrect Assumptions

- Accessing memory is a quick and constant-time operation. \[\text{Lies!}\]
- Sometimes accessing memory is cheaper and easier than at other times
- Sometimes accessing memory is very slow
# Memory Architecture

## What is it?

<table>
<thead>
<tr>
<th>What is it?</th>
<th>Typical Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>The brain of the computer!</td>
<td>32 bits</td>
<td>≈free</td>
</tr>
<tr>
<td>Extra memory to make accessing it faster</td>
<td>128KB</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>Extra memory to make accessing it faster</td>
<td>2MB</td>
<td>7 ns</td>
</tr>
<tr>
<td>Working memory, what your programs need</td>
<td>8GB</td>
<td>100 ns</td>
</tr>
<tr>
<td>Large, longtime storage</td>
<td>1 TB</td>
<td>8,000,000 ns</td>
</tr>
</tbody>
</table>

## Diagram

- **CPU Register**: The brain of the computer!
- **L1 Cache**: Extra memory to make accessing it faster
- **L2 Cache**: Extra memory to make accessing it faster
- **RAM**: Working memory, what your programs need
- **Disk**: Large, longtime storage
RAM (Random-Access Memory)

- RAM is where data gets stored for the programs you run. Think of it as the main memory storage location for your programs.

- RAM goes by a ton of different names: memory, main memory, RAM are all names for this same thing.
RAM can be represented as a huge array

RAM:
- addresses, storing stuff at specific locations
- random access

Arrays
- indices, storing stuff at specific locations
- random access

If you’re interested in deeper than this: https://www.youtube.com/watch?v=fpnE6UAfbtU or take some EE classes?
Processor – Memory Gap

“Moore’s Law”

μProc
55%/year
(2X/1.5yr)

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

1989 first Intel CPU with cache on chip
1998 Pentium III has two cache levels on chip
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)
Example Memory Hierarchy

- registers: <1 ns
- on-chip L1 cache (SRAM): 5-10 ns
- off-chip L2 cache (SRAM): 1 ns
- main memory (DRAM): 5-10 ns
- SSD: 100 ns
- local secondary storage (local disks): 15-30 min
- remote secondary storage (distributed file systems, web servers): 31 days
- Disk: 150,000 ns (10 ms)
- Larger, slower, cheaper per byte: 1,000,000,000 ns (10 ms)
- Smaller, faster, costlier per byte: 1-150 ms
- 66 months = 5.5 years
- 1-15 years
Example Memory Hierarchy

- **registers**: CPU registers hold words retrieved from L1 cache
- **on-chip L1 cache (SRAM)**: L1 cache holds cache lines retrieved from L2 cache
- **off-chip L2 cache (SRAM)**: L2 cache holds cache lines retrieved from main memory
- **main memory (DRAM)**: Main memory holds disk blocks retrieved from local disks
- **local secondary storage (local disks)**: Local disks hold files retrieved from disks on remote network servers
- **remote secondary storage (distributed file systems, web servers)**:
Example Memory Hierarchy

- registers
  - explicitly program-controlled (e.g., refer to exactly %rax, %rbx)

- on-chip L1 cache (SRAM)

- off-chip L2 cache (SRAM)

- main memory (DRAM)

- local secondary storage (local disks)

- remote secondary storage (distributed file systems, web servers)

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

Program sees "memory"; hardware manages caching transparently
**Review: Binary, Bits and Bytes**

- **binary**
  - A base-2 system of representing numbers using only 1s and 0s
  - vs decimal, base 10, which has 9 symbols

- **bit**
  - The smallest unit of computer memory represented as a single binary value either 0 or 1

---

**Decimal Break Down**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary Break Down</th>
<th>Binary</th>
<th>Binary Break Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>01111111</td>
<td></td>
</tr>
</tbody>
</table>
Memory Architecture

Takeaways:
- the more memory a layer can store, the slower it is (generally)
- accessing the disk is very slow

Computer Design Decisions
- Physics
  - Speed of light
  - Physical closeness to CPU
- Cost
  - “good enough” to achieve speed
  - Balance between speed and space
Locality

How does the OS minimize disk accesses?

**Spatial Locality**
Computers try to partition memory you are likely to use close by
- Arrays
- Fields

**Temporal Locality**
Computers assume the memory you have just accessed you will likely access again in the near future
Leveraging Spatial Locality

When looking up address in “slow layer”
- bring in more than you need based on what’s near by
- cost of bringing 1 byte vs several bytes is the same
- Data Carpool!
Leveraging Temporal Locality

When looking up address in “slow layer”

Once we load something into RAM or cache, keep it around or a while

- But these layers are smaller
  When do we “evict” memory to make room?
Moving Memory

Amount of memory moved from disk to RAM
- Called a “block” or “page”
  ≈4kb
  Smallest unit of data on disk

Amount of memory moved from RAM to Cache
- called a “cache line”
  ≈64 bytes

Operating System is the Memory Boss
- controls page and cache line size
- decides when to move data to cache or evict
Example Revisited

public int sum1(int n, int m, int[][] table) {
    int output = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            output += table[i][j];
        }
    }
    return output;
}

public int sum2(int n, int m, int[][] table) {
    int output = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            output += table[j][i];
        }
    }
    return output;
}

Why does sum1 run so much faster than sum2?
sum1 takes advantage of spatial and temporal locality
How memory is used and moves around
Solution to Mercy’s traveling problem

- If we know Mercy is going to keep eating tuna . . . Why not buy a bunch during a single trip and save them all somewhere closer than the store?

- Let’s get Mercy a refrigerator!
Recap + connecting analogy back to computer
CPU – kind of like the home/brain of your computer. Pretty much all computation is done here and data needs to move here to do anything significant with it (math, if checks, normal statement execution).

Data travels between RAM and the CPU, but it’s slow.

RAM
Bring a bunch of food back when you go all the way to the store.

After

CPU

Cache!

RAM

Bring a bunch of data back when you go all the way to RAM.
Cache

- Rough definition: a place to store some memory that’s smaller and closer to the CPU compared to RAM. Because caches are closer to the CPU (where your data generally needs to go to be computed / modified / acted on) getting data from cache to CPU is a lot quicker than from RAM to CPU. This means we love when the data we want to access is conveniently in the cache.

- Generally we always store some data here in hopes that it will be used in the future and that we save ourselves the distance / time it takes to go to RAM.

- Analogy from earlier: The refrigerator (a cache) in your house to store food closer to you than the store. Walking to your fridge is much quicker than walking to the store!
Bring a bunch of food back when you go all the way to the store

Cache!

Bring a bunch of data back when you go all the way to RAM

After CPU

This is a big idea!
How is a bunch of memory taken from RAM?

- Imagine you want to retrieve the 1 at index 4 in RAM.
- Your computer is smart enough to know to grab some of the surrounding data because computer designers think that it’s reasonably likely you’ll want to access that data too.
  - (You don’t have to do anything in your code for this to happen – it happens automatically every time you access data!)
- To answer the title question, technically the term / units of transfer is in terms of ‘blocks’. 

This is a big idea (continued)!

0 99 21 24 1 22 5 2 3 1 1 1 0 0 5 1 2 22 21 4
How is a bunch of memory taken from RAM? (continued)

original data (the 1) we wanted to look up gets passed back to the cpu.

all the data from the block gets brought to the cache.

0 99 21 24 1 22 5 2 3 1 1 1 0 0 5 1 2 22 21 4
How does this pattern of memory grabbing affect our programs?

- This should have a major impact on programming with arrays. Say we access an index of an array that is stored in RAM. Because we grab a whole bunch of contiguous memory even when we just access one index in RAM, we’ll probably be grabbing other nearby parts of our array and storing that in our cache for quick access later.

Imagine that the below memory is just an entire array of length 13, with some data in it.

```
| 0 | 99 | 21 | 24 | 1  | 22 | 5  | 2  | 3  | 1  | 1  | 1  | 0  |
```

Just by accessing one element we bring the nearby elements back with us to the cache. In this case, it’s almost all of the array!
Another demo, but timed

- [https://repl.it/repls/MistyroseLinedTransformation](https://repl.it/repls/MistyroseLinedTransformation)

- (takes about 15 seconds to run)