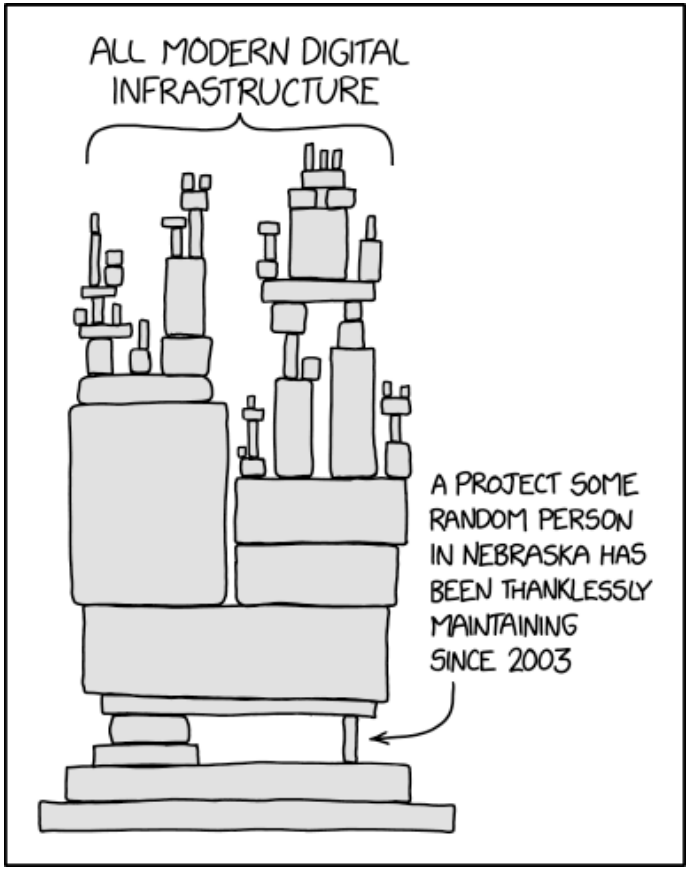
Caches I

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



Administrivia

- Today
  - HW12 due (11:59pm)
  - **Lab2 due (11:59pm)**
- Monday, 7/22
  - RD15 due (1pm)
  - HW13 due (11:59pm)
  - **Quiz 2 released (11:59pm)**
    - Same rules apply as Quiz 1
- Wednesday, 7/24
  - RD16 due (1pm)
  - HW14 due (11:59pm)

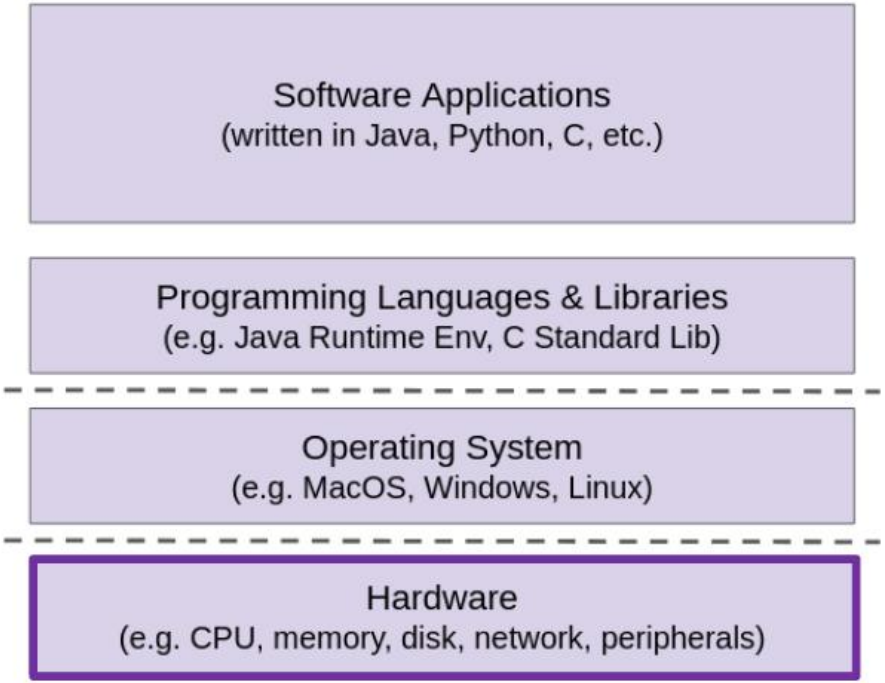
Quiz 1 Grades Released!

- Regrade requests will open tonight at 11:59pm
- If you received a message about possible academic misconduct, **please email me**
- And even though it's cheesy…
  - Your success in life is not defined by grades
  - ***You*** are not defined by grades
  - We know it seems critically important right now, but we promise, the numbers on a transcript will fade with time.

Topic Group 3: Scale & Coherence

- **How do we make memory accesses faster?**
- How do programs manage large amounts of memory?
- How does your computer run multiple programs at once?

Starting with caches, which are implemented in hardware.

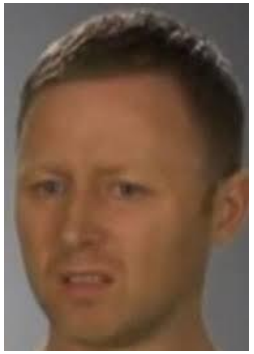


Aside: Units and Prefixes

kilo = 10^3 = 1000 kibi = 2^{10} = 1024

- Traditional prefixes represent powers of 10, we define new ones for base 2
  - Ex: 1 **Kibi**byte = 2^{10} bytes ≈ 10^3 bytes = 1 **Kilo**byte
- SI prefixes are *ambiguous* if base 10 or base 2 (does 'k' stand for kilo or kibi?)
- IEC prefixes are *unambiguously* base 2

↑ people often say "kilo" when they mean "kibi"



SI Symbol	SI Prefix	SI size	IEC symbol	IEC Prefix	IEC Size
K	Kilo-	10^3	Ki	Kibi-	2^{10}
M	Mega-	10^6	Mi	Mebi-	2^{20}
G	Giga-	10^9	Gi	Gibi-	2^{30}
T	Tera-	10^{12}	Ti	Tebi-	2^{40}
P	Peta-	10^{15}	Pi	Pebi-	2^{50}
E	Exa-	10^{18}	Ei	Exbi-	2^{60}
Z	Zetta-	10^{21}	Zi	Zebi-	2^{70}
Y	Yotta-	10^{24}	Yi	Yobi-	2^{80}

How to Remember?

- You can always look it up :)
  - It’s on the midterm reference sheet

- Mnemonics
  - **K**iller **M**echanical **G**iraffe **T**eaches **P**et **E**xtinct **Z**ebra to **Y**odel
  - **K**irby **M**issed **G**anondorf **T**erribly, **P**otentially **E**xterminating **Z**elda and **Y**oshi
  - From xkcd: **K**arl **M**arx **G**ave **T**he **P**roletariat **E**leven **Z**eppelins, **Y**o
    - https://xkcd.com/992/

Review Questions

1. Convert the following to or from IEC:
   a. 512 Mi-students $Mi = 2^{20}$ $512 = 2^9$ $2^9 \cdot 2^{20} = \boxed{2^{29} \text{ students}}$
   b. 2^{33} cats $= 2^3 \cdot 2^{30} = 2^3$ Gi-cats $= \boxed{8 \text{ Gi-cats}}$ (2^{30} = 1 Gi)
2. Compute the average memory access time (AMAT) for a system with the following properties:
   a. Hit time of 2 ns
   b. Miss rate of 1%
   c. Miss penalty of 300 ns

$AMAT = HT + MR \cdot MP$

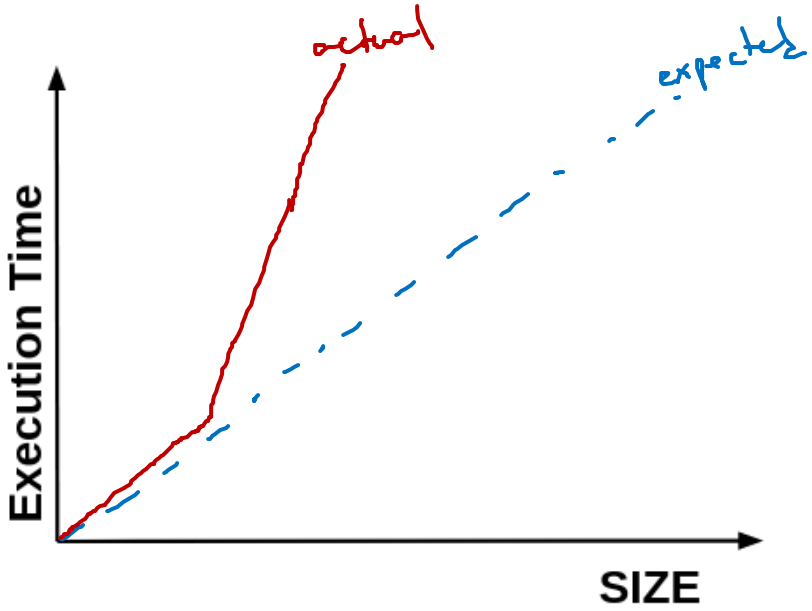
$= 2\text{ns} + 0.01 \cdot 300\text{ns}$

$= 2\text{ns} + 3\text{ns}$

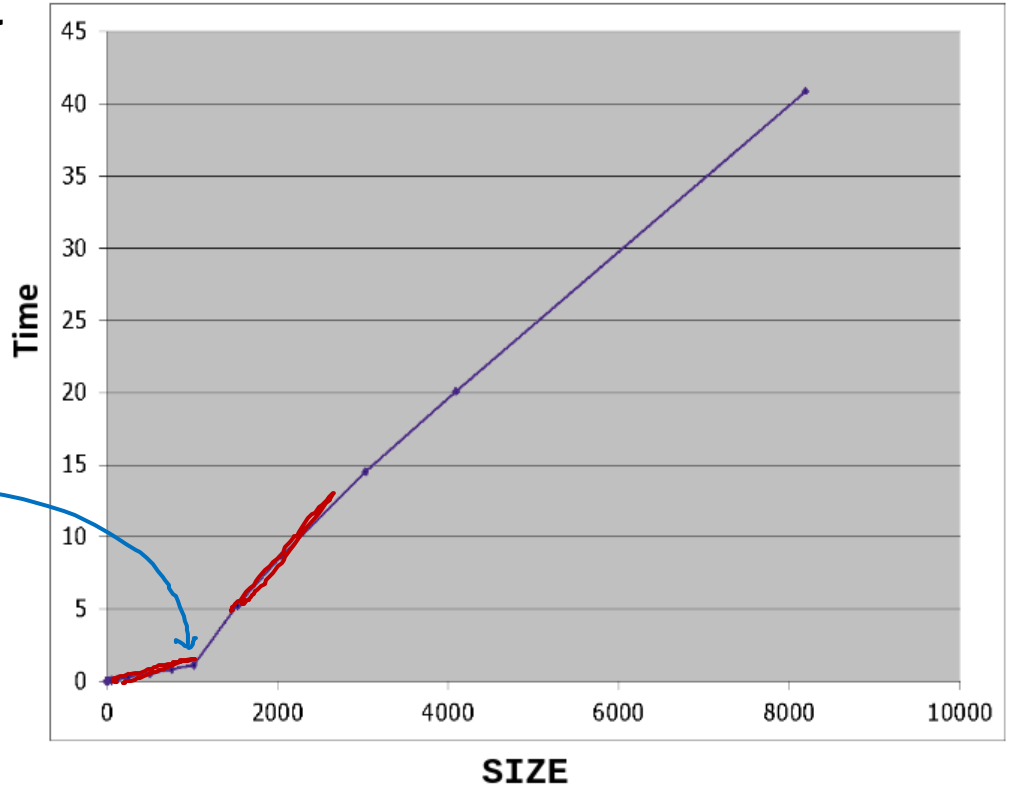
$= \boxed{5\text{ns}}$

How does execution time grow with SIZE?

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



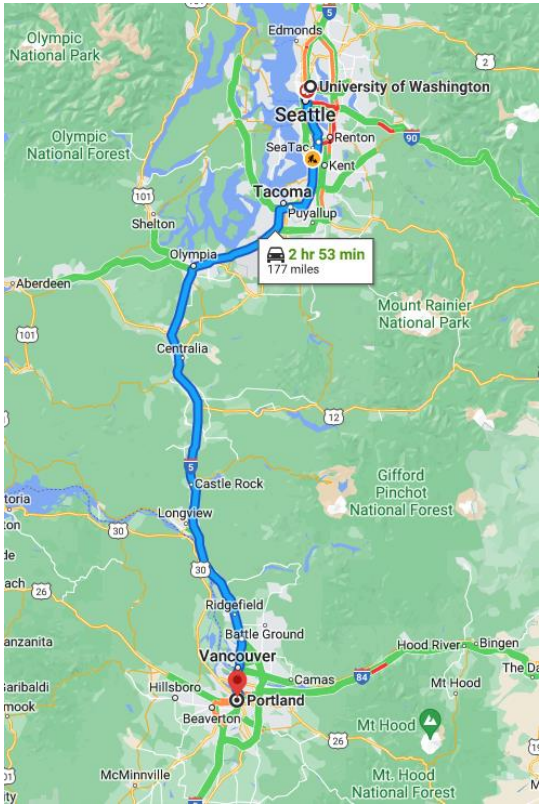
Actual Data



point where we exceed cache size

An Analogy

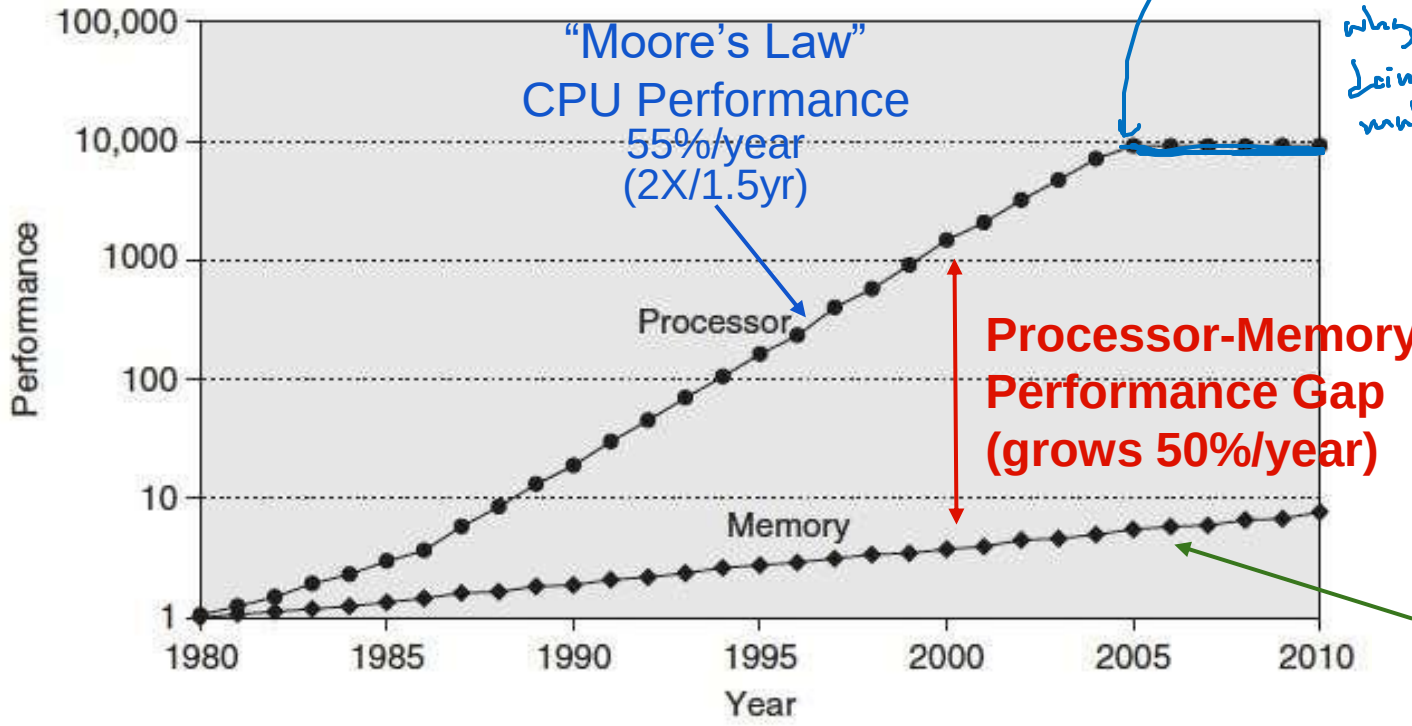

U-HAUL
U-HAUL


Olympic National Park
Edmonds
University of Washington
Seattle
Olympic National Forest
SeaTac
Renton
Kent
Tacoma
Puyallup
Shelton
Olympia
2 hr 53 min
177 miles
Aberdeen
Mount Rainier National Park
Centralia
Castle Rock
Gifford Pinchot National Forest
Longview
Ridgefield
Battle Ground
Vancouver
Hood River
Bingen
Camas
Mt Hood
Hillsboro
Portland
Beaverton
Mt Hood
McMinnville
Mt. Hood National Forest

Caches

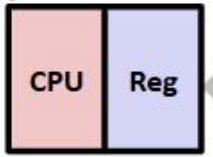
- **Cache basics**
- **Principle of locality**
- **Memory hierarchies**
- Cache organization
- Program optimizations that consider caches

Problem: Processor-Memory Bottleneck



A Very Silly Analogy

Processor performance, very fast



Bus latency / bandwidth much slower

Main Memory

Core 2 Duo:

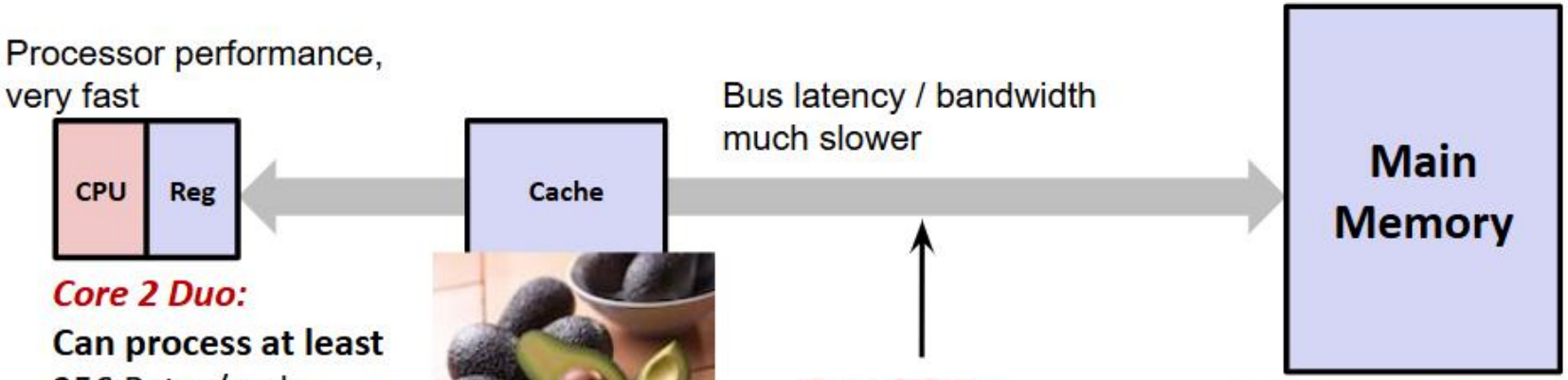
Can process at least
256 Bytes/cycle

Core 2 Duo:

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

Problem: memory is slow

A Very Silly Analogy (pt 2)



Solution: caches!

Cache 💰

- Pronounced “cash”
  - Often abbreviated to ‘$”
- English: hidden storage space for provisions, weapons, or treasures
- Computer: Memory with short access time used for the storage of frequently or recently used instructions or data
  - I-cache for instructions
  - d-cache for data
  - *More generally*: Used to optimize data transfers between any system elements with different characteristics (network interface cache, I/O cache, etc.)

If caches are so much faster, why do we need memory?

others exist too, but these are the most common right now

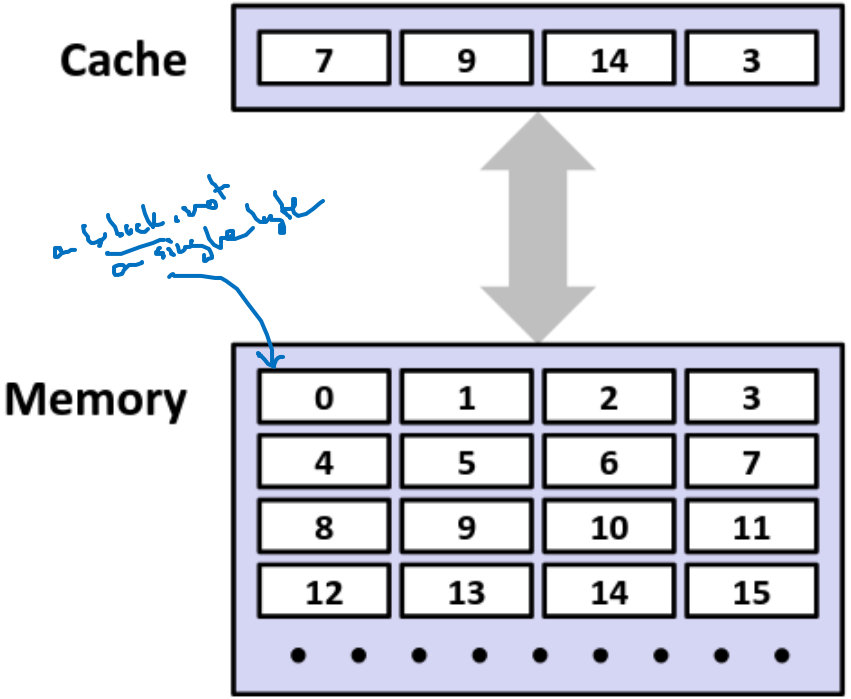
- Two common memory technologies
  - **DRAM**: high-capacity, cheap, energy efficient, but *slow*
  - **SRAM**: much faster, but less energy efficient and ***expensive***
- We can't afford to have all our computer's memory be SRAM
  - Use DRAM to provide large amounts of memory for cheap
  - Have a small SRAM cache for speed

SRAM

DRAM

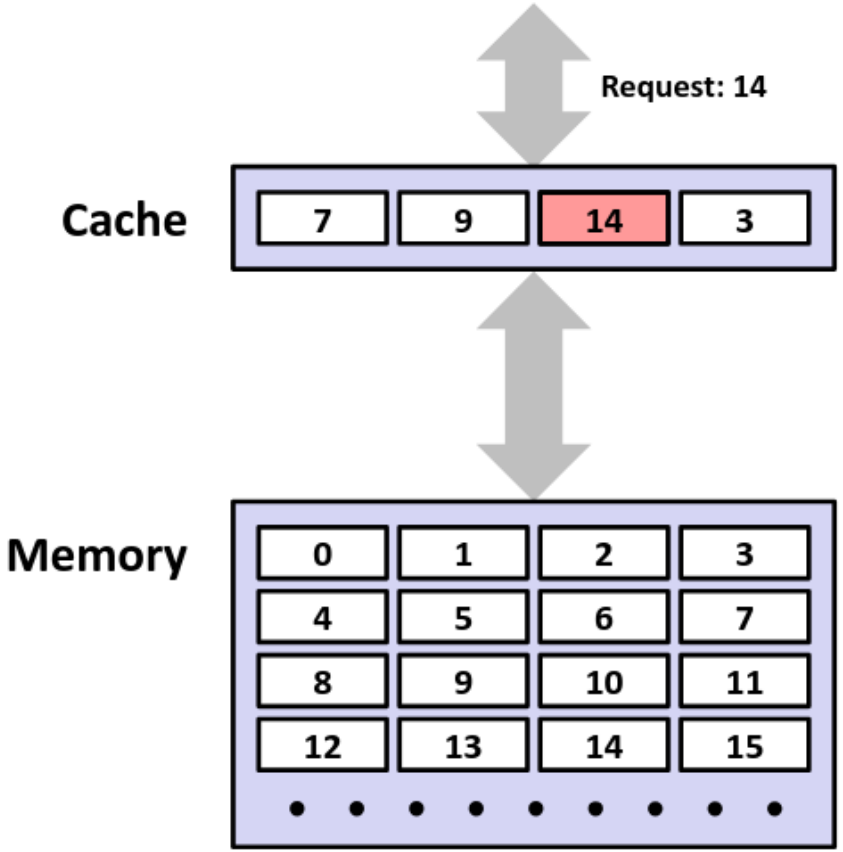
General Cache Mechanics

- Memory
  - Slower, larger, cheaper
  - Partition into “**blocks**”
- Cache
  - Smaller, faster, more expensive
  - Stores a subset of **blocks** from memory
  - Data is copied in *block-sized chunks*



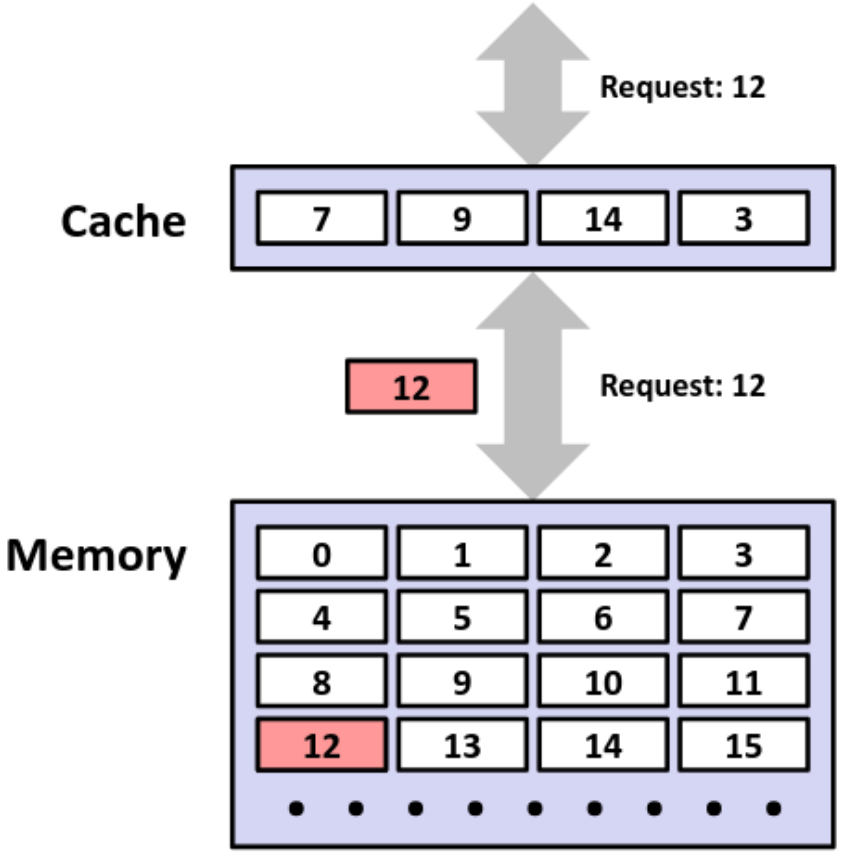
Cache Mechanics: Hit

- Data we need is in block *b*
- Block *b* is in the cache
  - ***Hit!***
- Data is returned to the CPU



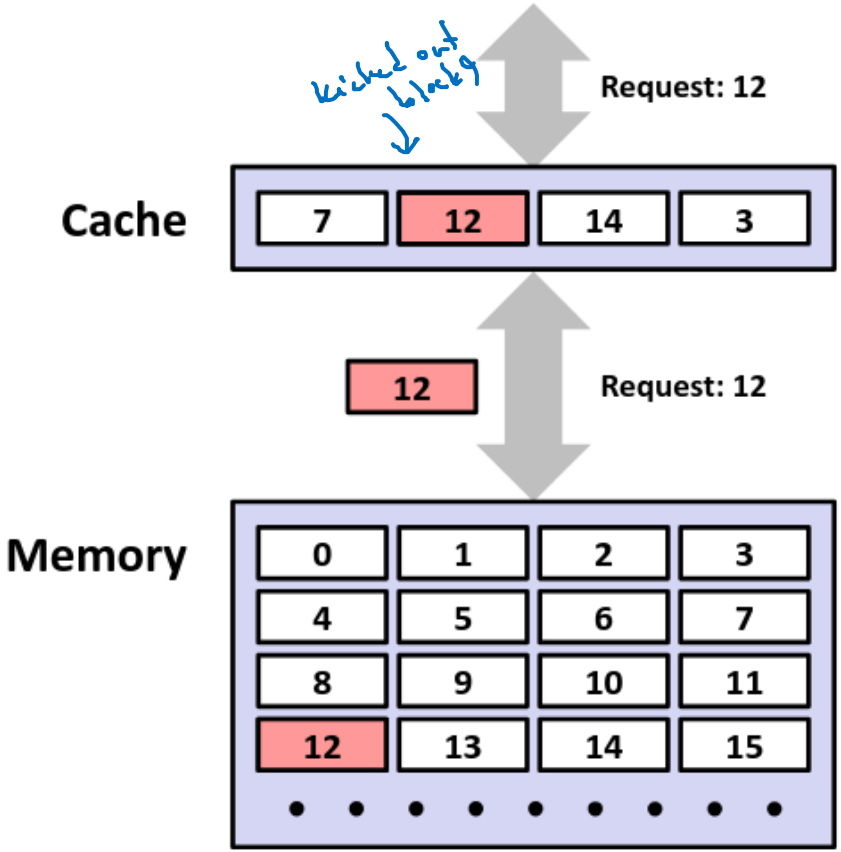
Cache Mechanics: Miss

- Data we need is in block *b*
- Block *b* is not in the cache already
  - ***Miss!***
- Block *b* is fetched from memory
- Block *b* is written into the cache



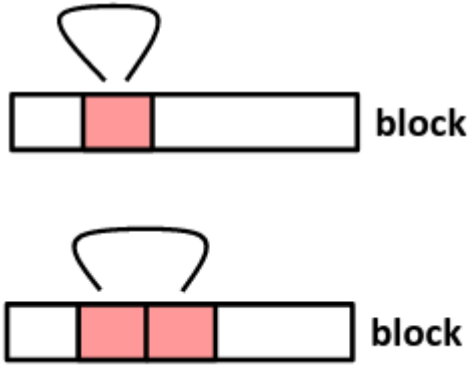
Cache Mechanics: Miss (pt 2)

- Data we need is in block *b*
- Block *b* is not in the cache already
  - ***Miss!***
- Block *b* is fetched from memory
- Block *b* is written into the cache
  - **Placement policy** decides where it goes
  - **Replacement policy** decides what we kick out
- Data is returned to the CPU



Why Caches Work

- Takes advantage of **locality**
  - Common patterns in how programs access data
- **Temporal locality**
  - If a program accesses data once, it's likely to access it again
- **Spatial locality**
  - If a program accesses some data, it's likely to access other data that's *nearby* in memory



How do caches take advantage of this?

Locality Example

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

Data

- **Temporal**: `sum`, `i`, and `n` are accessed every loop iteration
- **Spacial**: consecutive elements of `a`

Instructions

- **Temporal**: loop body code
- **Instructions**: instructions executed in sequence

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;
}
```

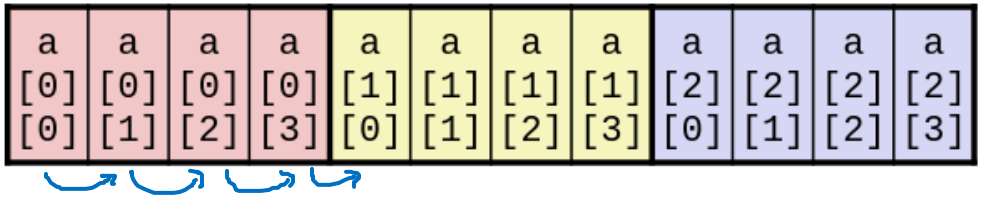
Access pattern:

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]
9) a[2][0]
10) a[2][1]
11) a[2][2]
12) a[2][3]

Stride? 1

Good cache performance. use entire block before moving on to the next

Layout in Memory



Locality Example 2

```
int sum_array_rows(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;
}
```

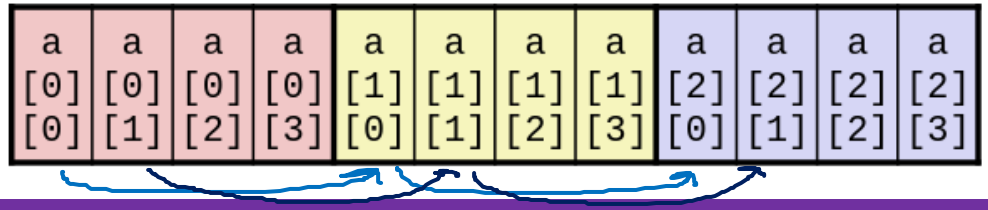
Access pattern:

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]

Stride? 4

Bad cache performance :(

Layout in Memory

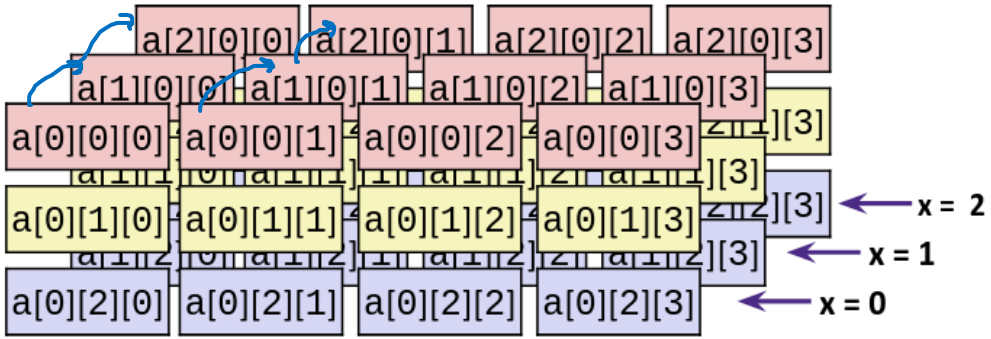


Locality Example 3

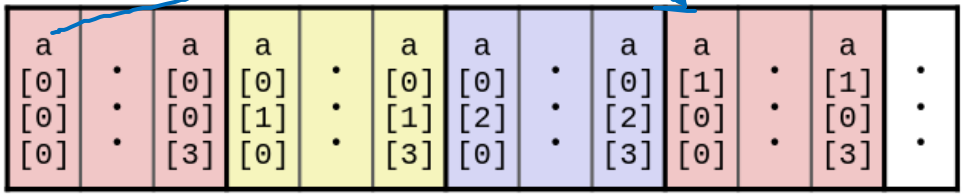
really bad cache performance!

```
int sum_array_3D(int a[X][Y][Z])
{
    int i, j, k, sum = 0;

    for (i = 0; i < Y; i++)
        for (j = 0; j < Z; j++)
            for (k = 0; k < X; k++)
                sum += a[k][i][j];

    return sum;
}
```

Access pattern:



Layout in Memory



a[0][0][0]
a[1][0][0]
a[2][0][0]

Cache Performance Metrics

- **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
  - Total miss time = Hit Time + Miss Penalty

Cache Performance

- **Average Memory Access Time (AMAT)**: average time to access data, considering both cache hits and misses
  - **AMAT = Hit time + Miss Rate × Miss Penalty**
    (abbreviated **AMAT = HT + MR×MP**)

Each access has 100% chance of incurring HT
and MR chance of incurring MP

Practice Questions

amount of time for CPU to do 1 task

1. Processor specs: 200 ps clock, MP of 50 clock cycles, MR of 0.02 misses/instruction, and HT of 1 clock cycle
   A) What is the AMAT (in clock cycles)? HT + MR · MP = 1 + 0.02 · 50 = 1+1 = 2 cycles
   2 cycles · 200 ps/cycle = 400 ps
2. Which of these improvements would result in the lowest AMAT?
   A) 190 ps clock same # of cycles → 2 · 190 = 380 ps
   B) Miss penalty of 40 clock cycles 1+0.02·40 = 1.8 cycles · 200 ps/cycle = 360 ps
   C) Miss rate of 0.015 misses/instruction
   1+0.015·50 = 1.75 cycles · 200 ps/cycle = 350 ps

Cache Performance (pt 2)

- Misses have a *much* larger effect on AMAT than hits!
  - Going to memory could be 100x slower than accessing the cache (measured in *clock cycles*)
- High **miss rate** or **miss penalty** hurt AMAT the most

Ex: Assume HT of 1 clock cycle and MP of 100 clock cycles MP = 1 − HR

If HR = 99%, AMAT = 2 cycles MP = 0.01, 1 + 0.01 · 100 = 2

If HR = 97%, AMAT = 4 cycles MP = 0.03, 1 + 0.03 · 100 = 4

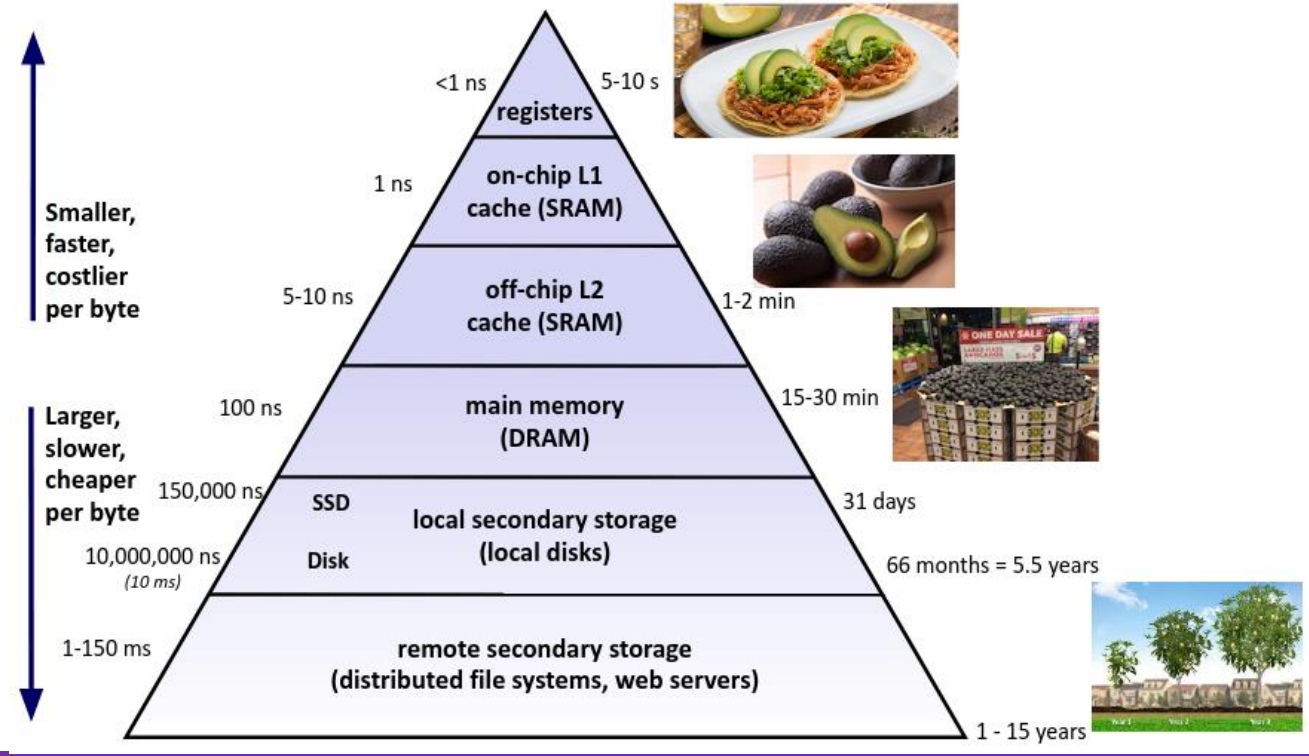
Increasing hit rate by only 2% cut acces time in ***half***!

Can we have more than one cache? Yes!

- Why would we want to do that?
  - Avoid going to memory at all costs!
- Typical performance numbers
  - Miss Rate
    - L1: 3-10%
    - L2: very small (likely <1%)
  - Hit Time
    - L1: 4 clock cycles
    - L2: 10 clock cycles
  - Miss Penalty
    - 50-200 cycles for missing in L2 going to main memory)
    - Trend: increasing!

these numbers are outdated!
- trade secrets

An Example Memory Hierarchy



Learning About Your Machine

- **Linux**:
  - `lscpu`
  - `ls /sys/devices/system/cpu/cpu0/cache/index0/`
  - Example: `cat /sys/devices/system/cpu/cpu0/cache/index*/size`
- **Windows**:
  - `wmic memcache get <query>` (all values in KB)
  - Example: `wmic memcache get MaxCacheSize`
- Modern processor specs: http://www.7-cpu.com/

Summary

- **Memory Hierarchy**
  - Higher levels are faster, smaller, and more expensive
    - Contain the most used data from lower levels
  - Exploits **temporal and spatial locality**
  - **Caches** are intermediate levels between memory and the CPU
- Cache Performance
  - Ideal case: data found in cache (hit)
  - Bad case: not found in cache (miss), go to next level in hierarchy
  - **Average Memory Access Time (AMAT) = HT + MR × MP**
    - Hurt my high **miss rate** and **miss penalty**