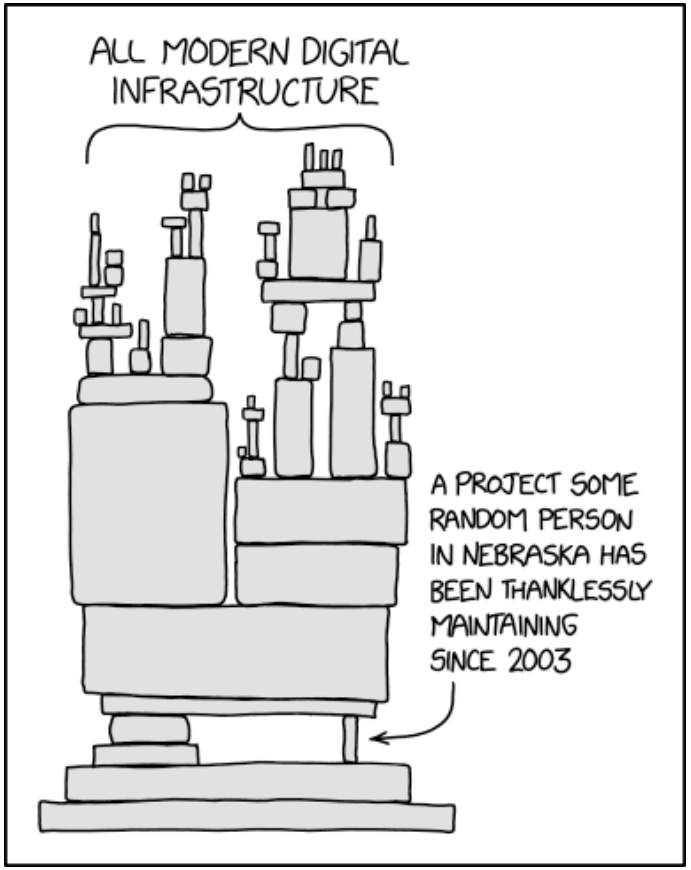
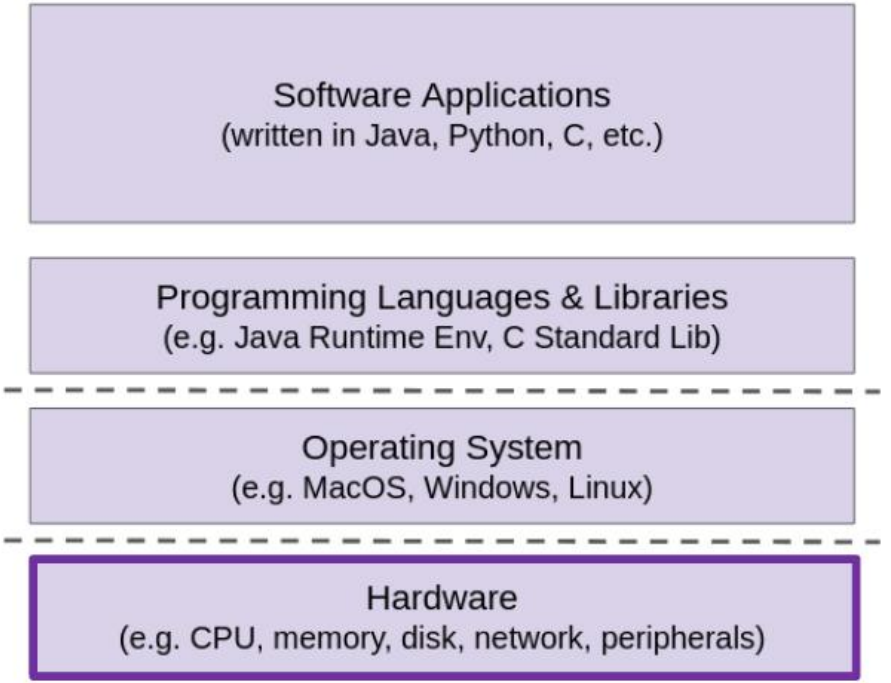
Caches I


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

Topic: Scale & Coherence

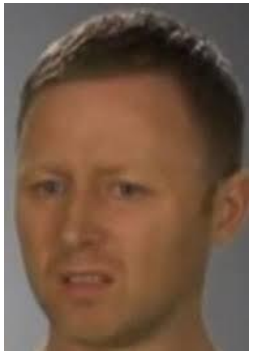
- **How do we make memory accesses faster?**
- How do programs manage large amounts of memory?
- How does your computer run multiple programs at once?

Starting with caches, which are implemented in hardware.



Aside: Units and Prefixes

- 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



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}

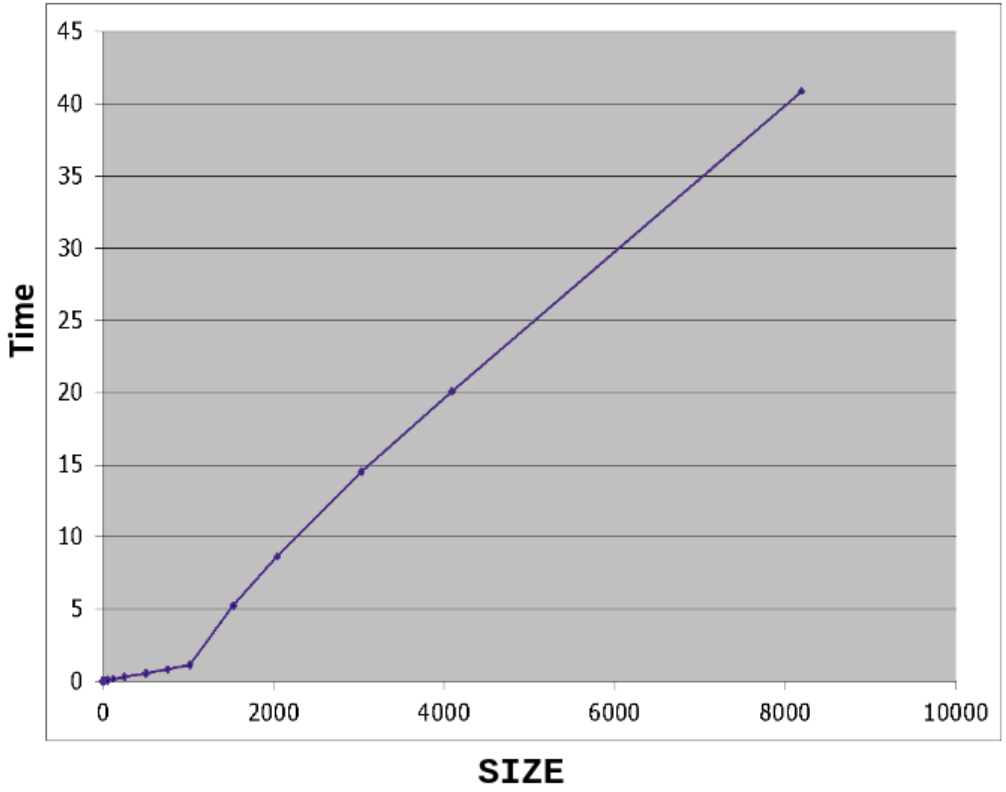
How to Remember?

- You can always look it up :)
- 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/

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];
    }
}
```

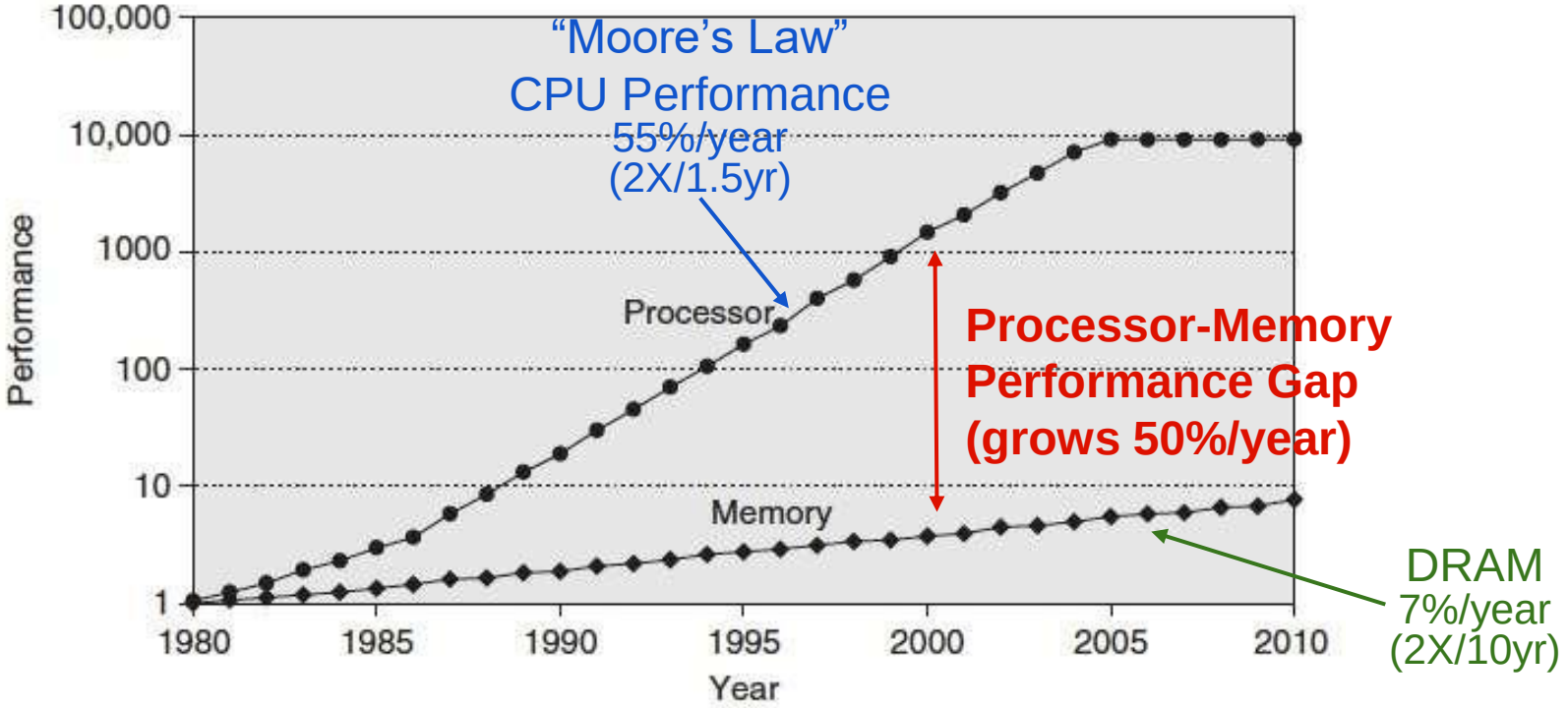
Actual Data



Caches

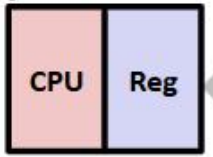
- **Cache basics**
- **Principle of locality**
- **Memory hierarchies**
- Cache organization
- Program optimizations that consider caches

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)

Processor performance, very fast

CPU | Reg

Cache

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)

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?

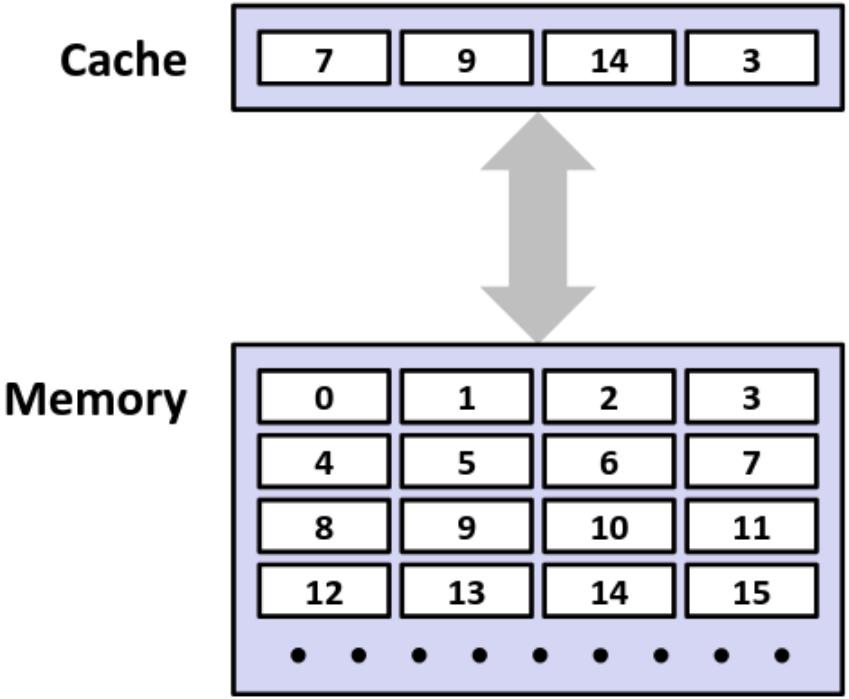
- Two common memory technologies
  - **DRAM**: high-capacity, cheap, energy efficient, but *slow*
  - **SRAM**: much faster, but less energy efficient and ***expensive***
- We can’t afford to have all our computer’s memory be SRAM
  - Use DRAM to provide large amounts of memory for cheap
  - Have a small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

SRAM

DRAM

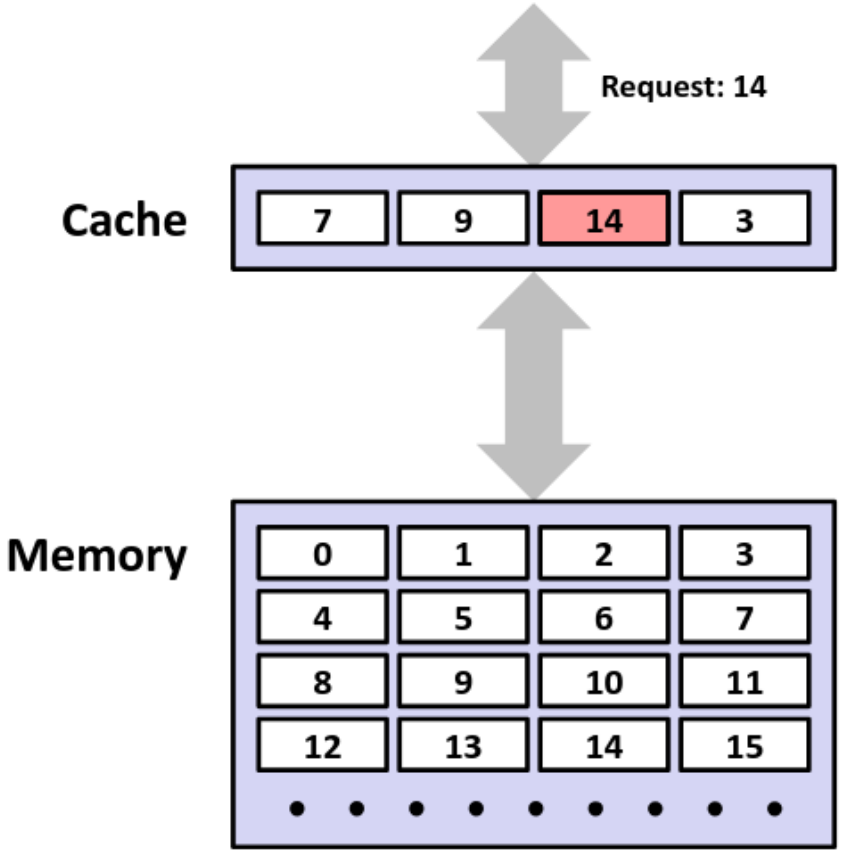
General Cache Mechanics

- 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*



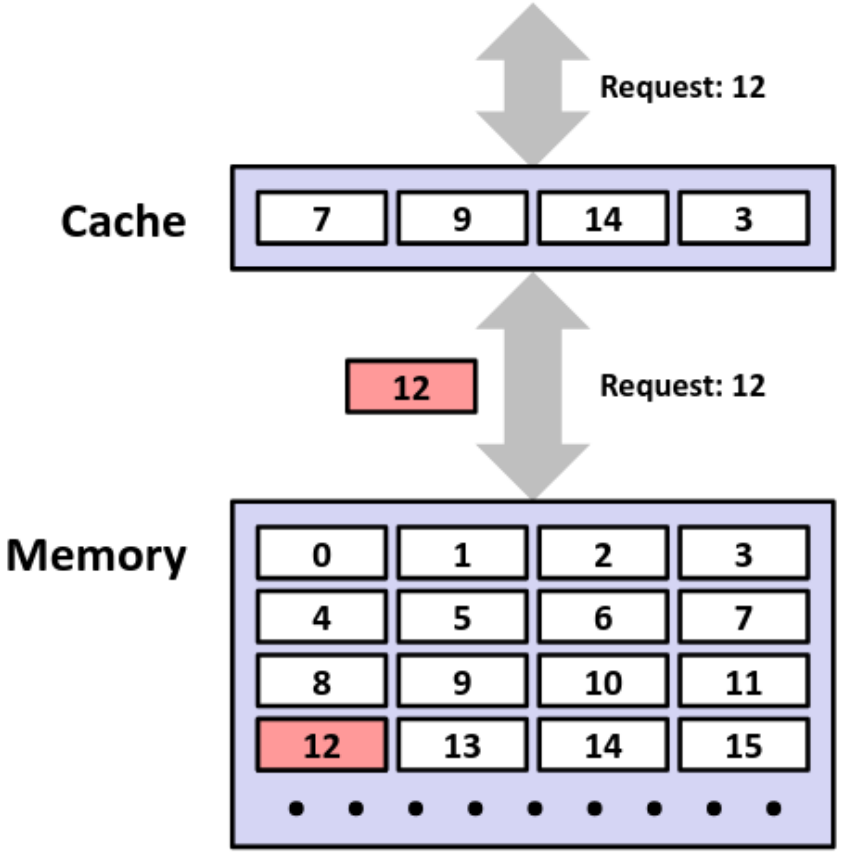
Cache Mechanics: Hit

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



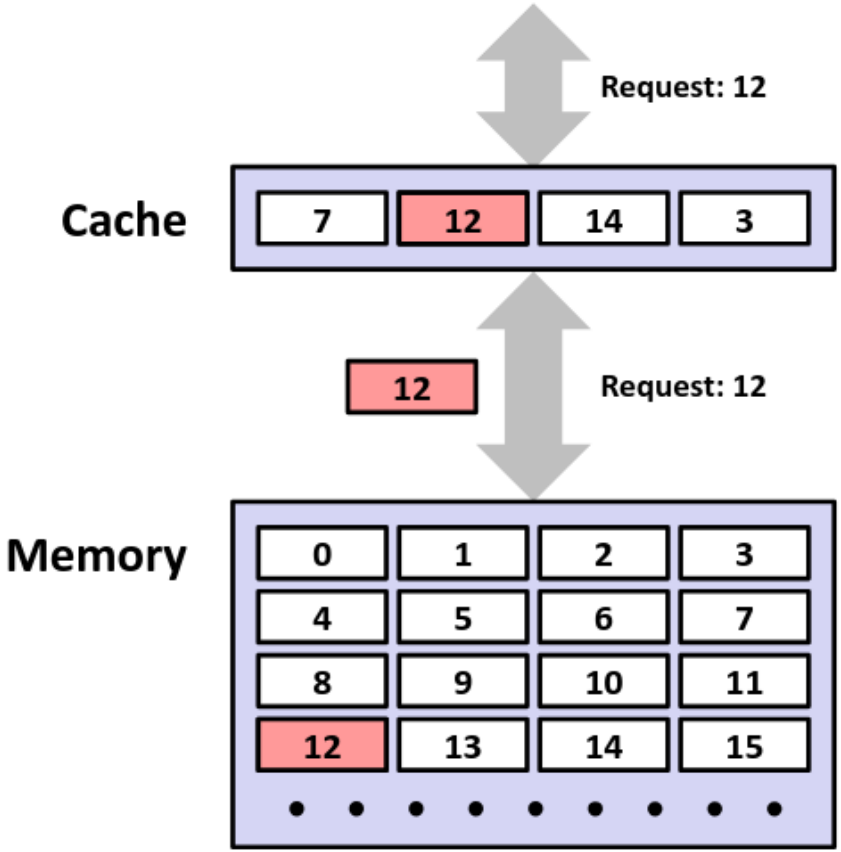
Cache Mechanics: Miss

- 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



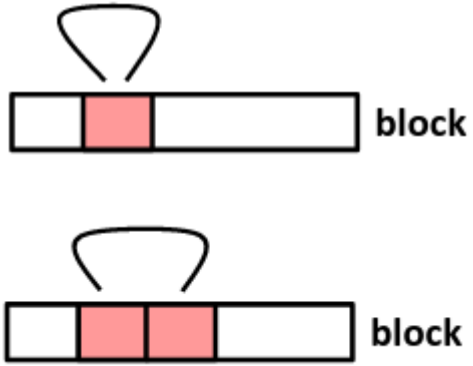
Cache Mechanics: Miss (pt 2)

- 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;
}
```

Layout in Memory

a [0] [0]	a [0] [1]	a [0] [2]	a [0] [3]	a [1] [0]	a [1] [1]	a [1] [2]	a [1] [3]	a [2] [0]	a [2] [1]	a [2] [2]	a [2] [3]

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?

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;
}
```

Layout in Memory

a [0] [0]	a [0] [1]	a [0] [2]	a [0] [3]	a [1] [0]	a [1] [1]	a [1] [2]	a [1] [3]	a [2] [0]	a [2] [1]	a [2] [2]	a [2] [3]

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?

Locality Example 3

```
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:

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**)

Practice Questions

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)?
2. Which of these improvements would result in the lowest AMAT?
   A) 190 ps clock
   B) Miss penalty of 40 clock cycles
   C) Miss rate of 0.015 misses/instruction

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

If HR = 99%, AMAT = _____

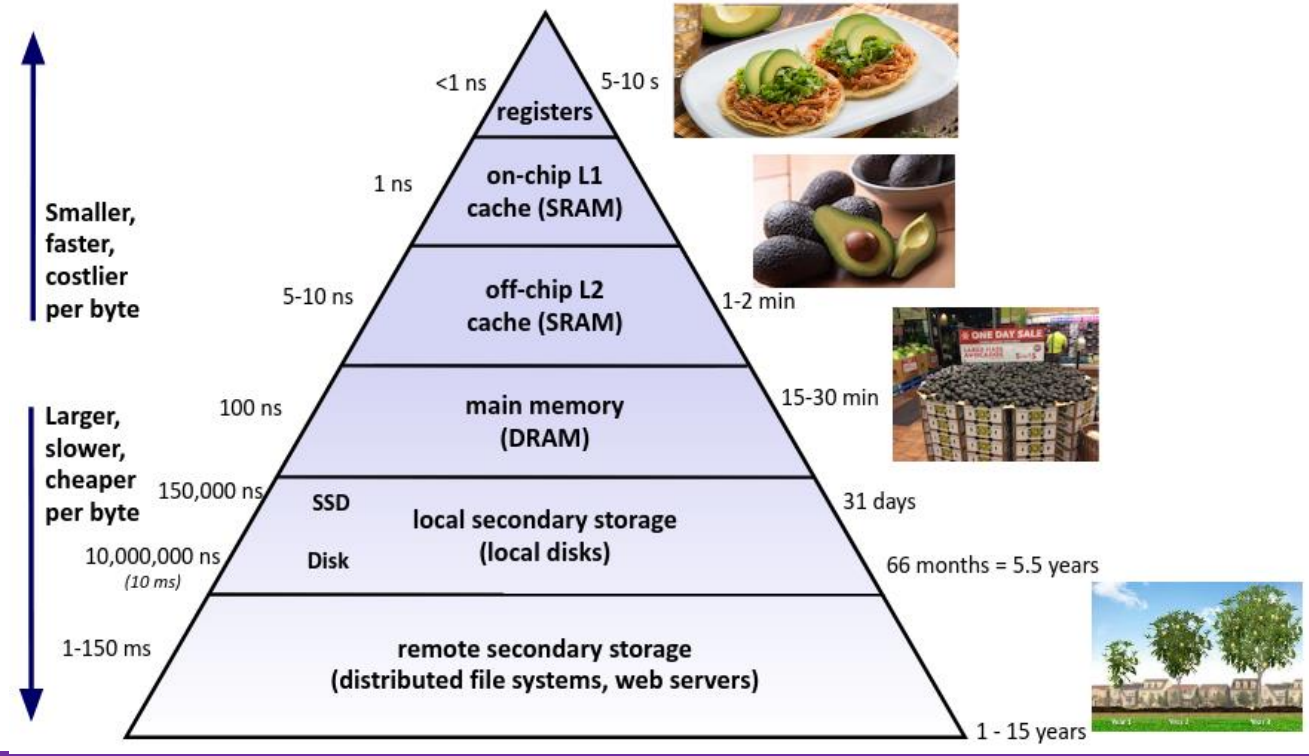
If HR = 97%, AMAT = _____

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!

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**