

Memory & Caches I

CSE 351 Summer 2020

Instructor:

Porter Jones

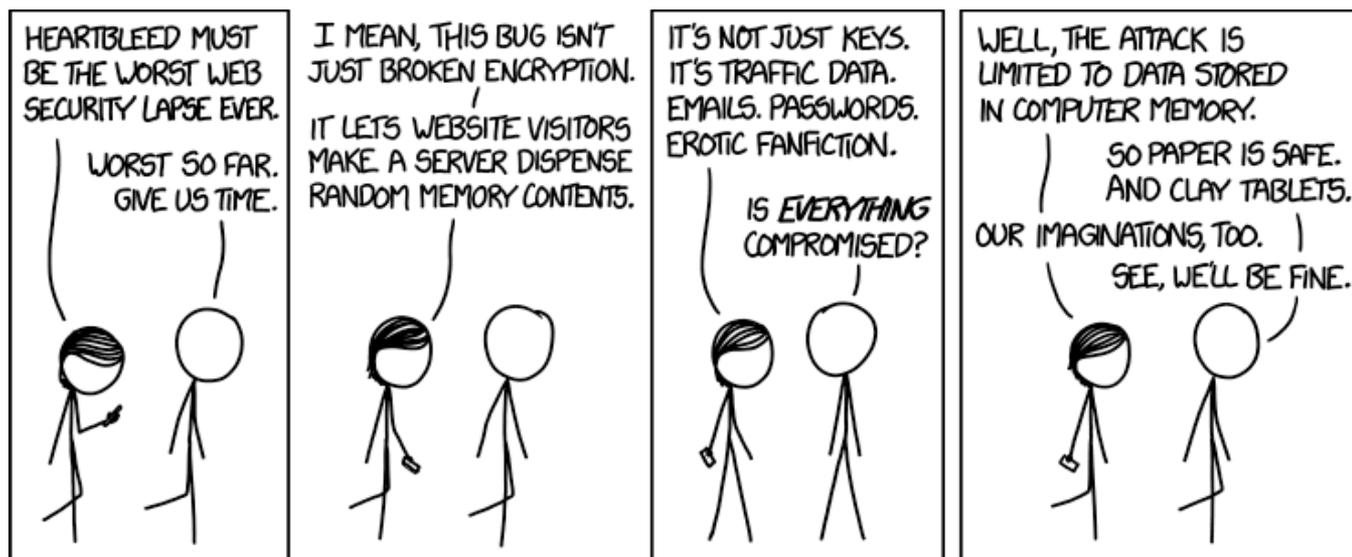
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



Alt text: I looked at some of the data dumps from vulnerable sites, and it was ... bad. I saw emails, passwords, password hints. SSL keys and session cookies. Important servers brimming with visitor IPs. Attack ships on fire off the shoulder of Orion, c-beams glittering in the dark near the Tannhäuser Gate. I should probably patch OpenSSL.

<http://xkcd.com/1353/>

Administrivia

- ❖ Questions doc: <https://tinyurl.com/CSE351-7-27>
- ❖ hw14 due Wednesday (7/29) – 10:30am
 - This one is especially long, please start early
- ❖ hw15 due Friday (7/31) – 10:30am
- ❖ Lab 3 due Friday (7/31) – 11:59pm
 - You get to write some buffer overflow exploits!
- ❖ Unit Summary 2 Due next Wednesday (8/5) – 11:59pm

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();
```

- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches**
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Assembly language:

```
get_mpg:
    pushq    %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

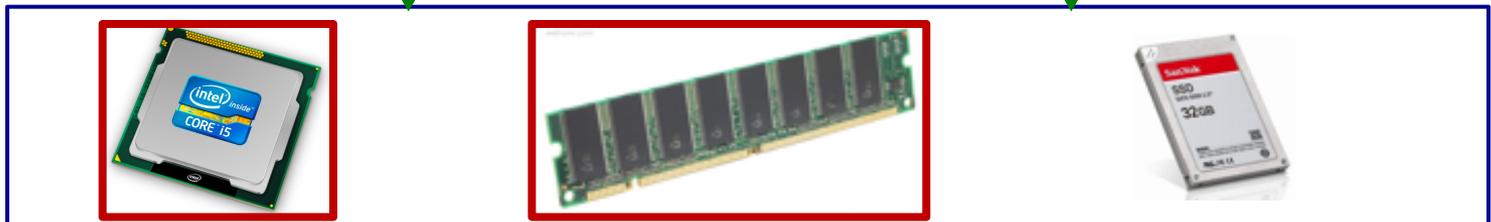
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



Aside: Units and Prefixes

- ❖ Here focusing on large numbers (exponents > 0)
- ❖ Note that $10^3 \approx 2^{10}$
1000 \approx 1024
- ❖ SI prefixes are *ambiguous* if base 10 or 2
- ❖ IEC prefixes are *unambiguously* base 2

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

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

How to Remember?

- ❖ Will be given to you on Final reference sheet

- ❖ Mnemonics
 - There unfortunately isn't one well-accepted mnemonic
 - But that shouldn't stop you from trying to come with one!
 - **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
 - xkcd: **K**arl **M**arx **G**ave **T**he **P**roletariat **E**leven **Z**eppelin's, **Y**o
 - <https://xkcd.com/992/>
 - Post your best on Piazza!

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++) {
```

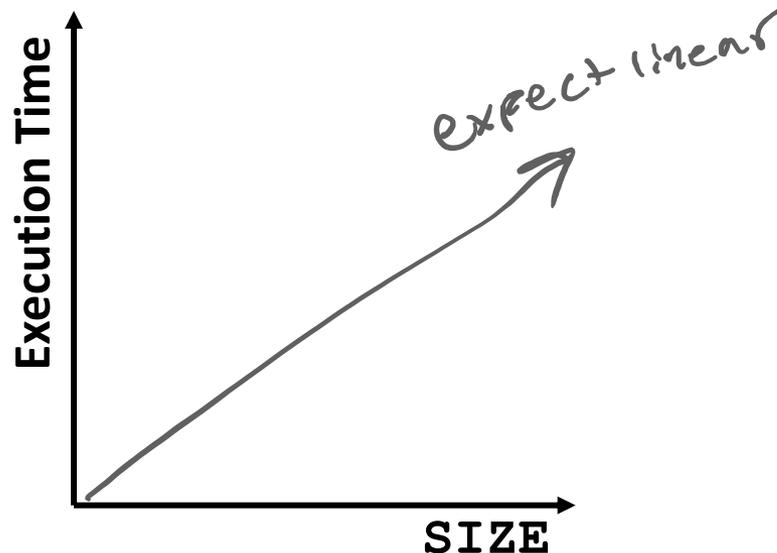
```
    sum += array[j];
```

```
  }  
}
```

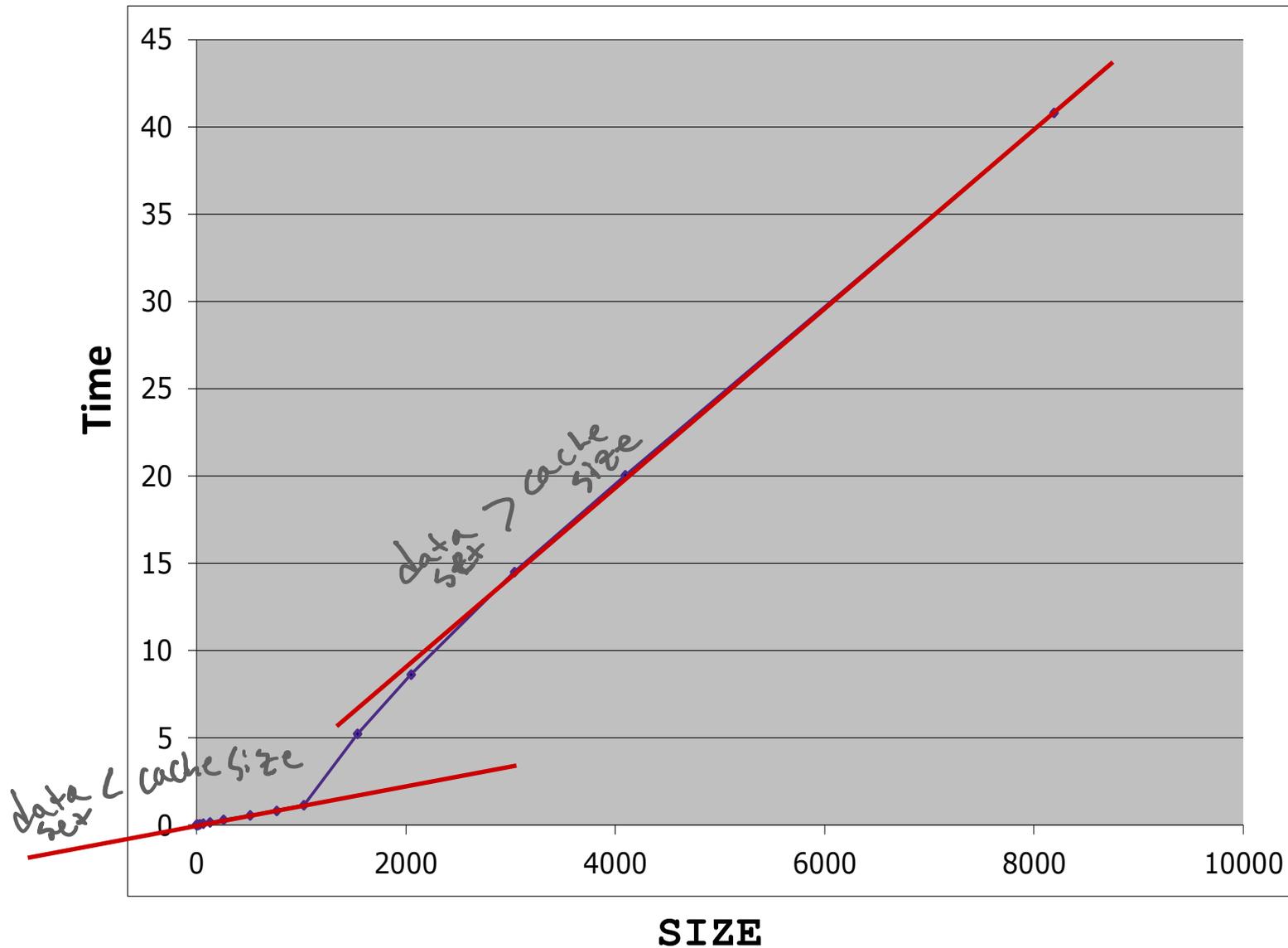
*sum array,
repeat 200,000
times*

*execute SIZE * 200,000 times*

Plot:



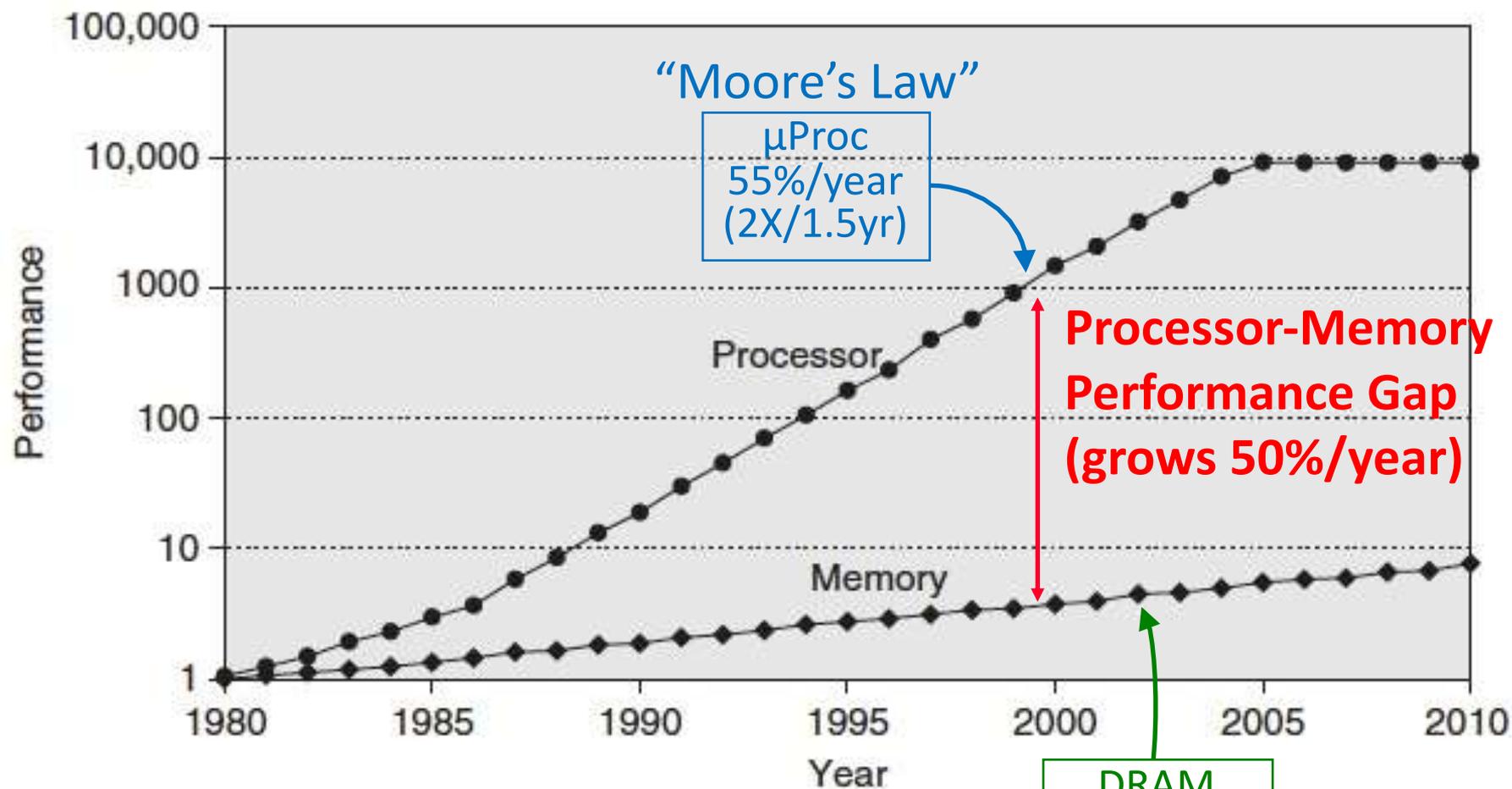
Actual Data



Making memory accesses fast!

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

Processor-Memory Gap

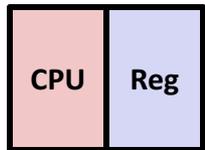


1989 first Intel CPU with cache on chip

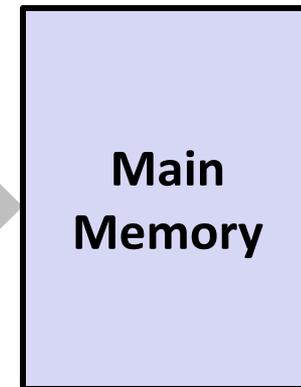
1998 Pentium III has two cache levels on chip

Problem: Processor-Memory Bottleneck

Processor performance
doubled about
every 18 months



Bus latency / bandwidth
evolved much slower



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

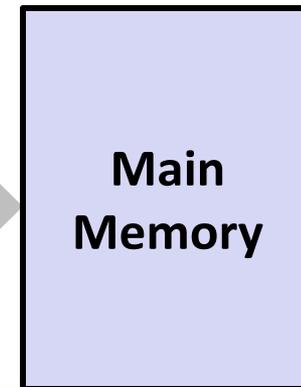
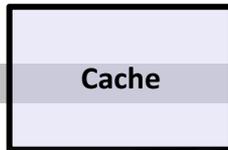
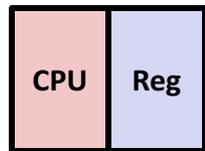


cycle: single machine step (fixed-time)

Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower



fridge/pantry



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

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



grocery store



plate to mouth

Solution: caches

cycle: single machine step (fixed-time)

Cache

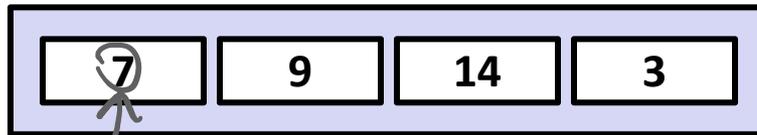
- ❖ Pronunciation: “cash”
 - We abbreviate this as “\$”
- ❖ English: A hidden storage space for provisions, weapons, and/or treasures
- ❖ Computer: Memory with short access time used for the storage of frequently or recently used instructions (i-cache/I\$) or data (d-cache/D\$)
 - *More generally*: Used to optimize data transfers between any system elements with different characteristics (network interface cache, I/O cache, etc.)

General Cache Mechanics

CPU



Cache



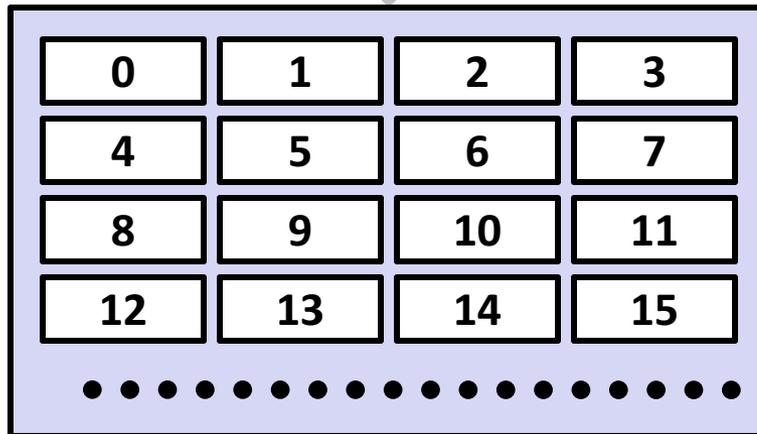
using
block ids
(not data)

- Smaller, faster, more expensive memory
- Caches a subset of the **blocks**

Data is copied in **block-sized** transfer units example: 32 bytes

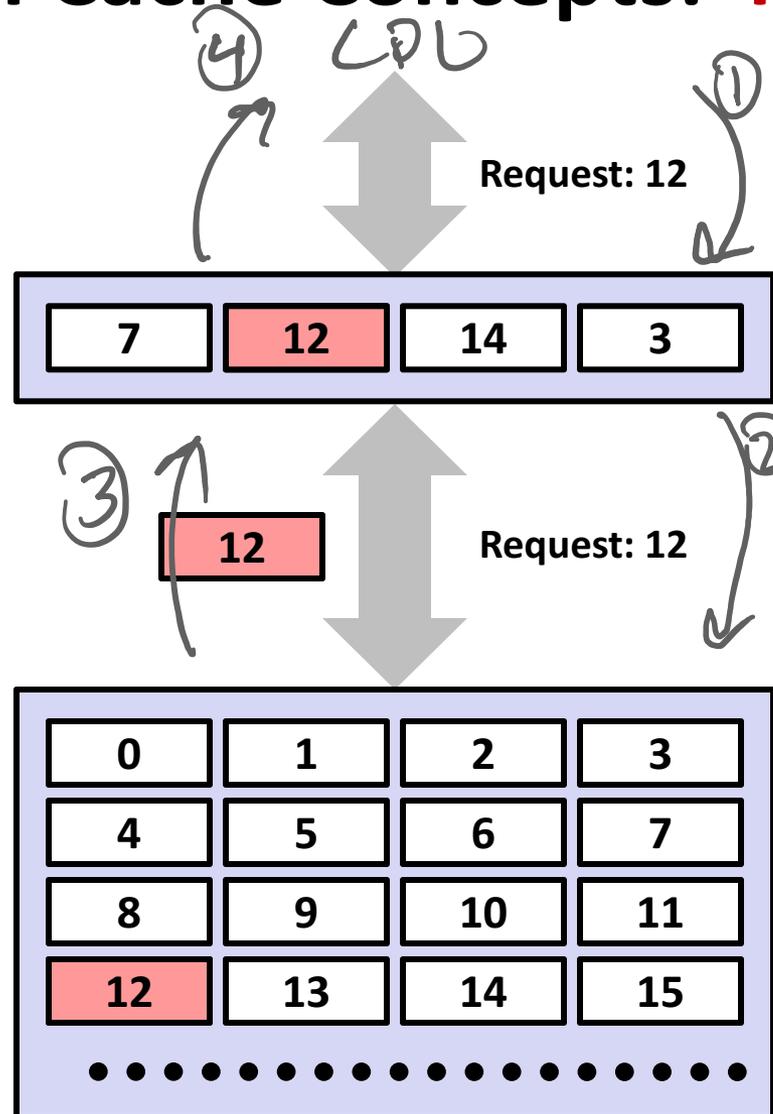


Memory



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

General Cache Concepts: Miss



① Data in block *b* is needed

Block *b* is not in cache:
Miss!

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 is 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

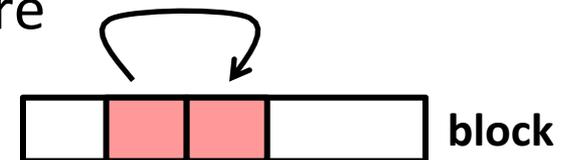
- ❖ **Temporal locality:**

- 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?

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

❖ Data:

- Temporal: sum referenced in each iteration
- Spatial: consecutive elements of array a [] accessed

❖ Instructions:

- Temporal: cycle through loop repeatedly
- Spatial: reference instructions 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;
}
```

Locality Example #1



row col

```

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;
}
    
```

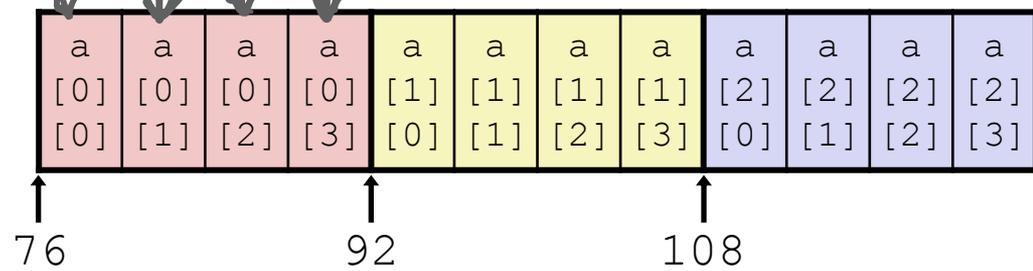
*0 0
0 1
0 2
...*

M = 3, N = 4

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:**
- stride = ?
"stride = 1"
- 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]

Layout in Memory --



Note: 76 is just one possible starting address of array a

Locality Example #2

```
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



```
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;
}
```

0 3
1 2
2 0

M = 3, N = 4

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:

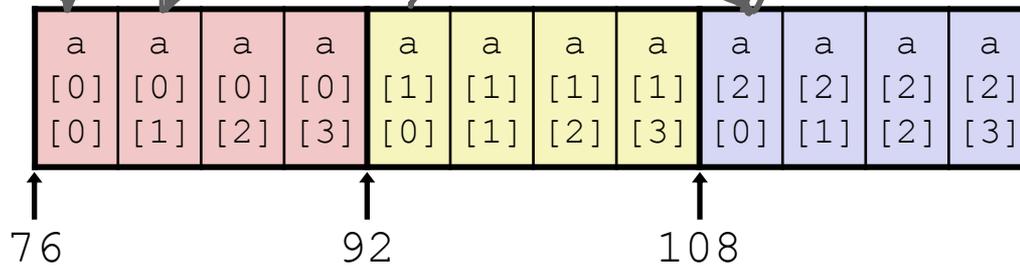
stride = ?

stride-4

stride-N

- 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



Locality Example #3

```

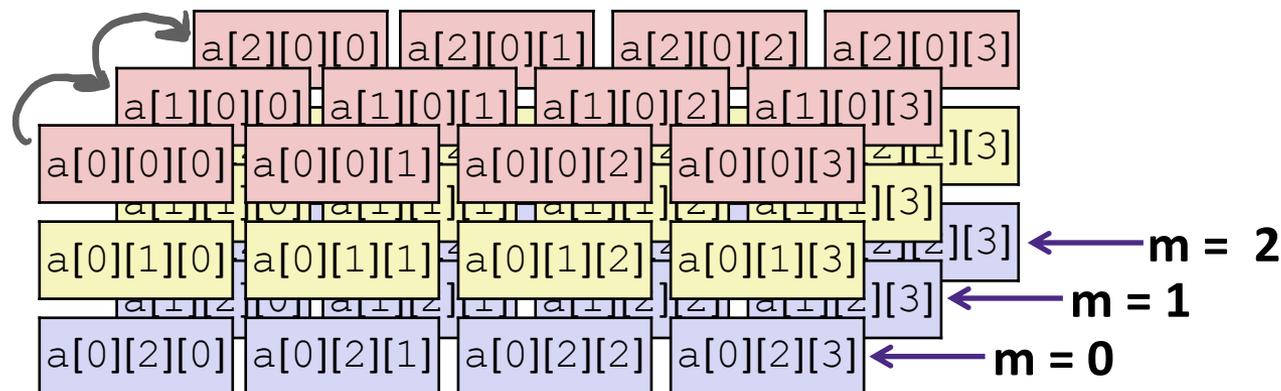
int sum_array_3D(int a[M][N][L])
{
    int i, j, k, sum = 0;

    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?



Locality Example #3

```
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;
}
```

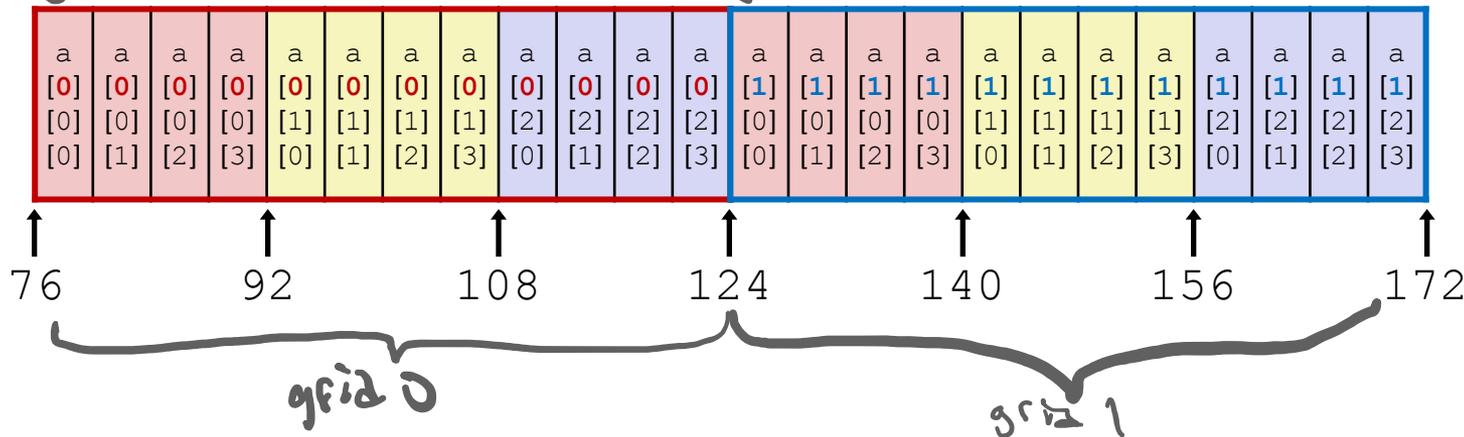
❖ What is wrong with this code?

Stride $N \times L$

❖ How can it be fixed?

*inner loop: $i \rightarrow \text{Stride} - L$
 $j \rightarrow \text{Stride} - 1$
 $k \rightarrow \text{Stride}$*

Layout in Memory (M = ?, N = 3, L = 4)



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)

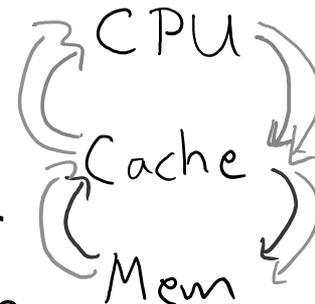
- Fraction of memory references not found in cache (misses / accesses) = $1 - \text{Hit Rate}$

❖ 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

Hit takes HT

Miss takes HT+MP



❖ Miss Penalty (MP) ! = Miss Time (MT)

- 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

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

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

- ❖ 99% hit rate twice as good as 97% hit rate!
 - Assume HT of 1 clock cycle and MP of 100 clock cycles
 - 97%: $\text{AMAT} = 1 + .03 * 100 = 4 \text{ clock cycles}$
 - 99%: $\text{AMAT} = 1 + .01 * 100 = 2 \text{ clock cycles}$

Polling Question [Cache I]

*clock speed
per cycle*

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

$$\text{AMAT} = \text{HT} + \text{MR} * \text{MP} = 1 + .02 * 50 = 2 \text{ cycles}$$

$$\text{Total time} = 2 \text{ cycles} * 200 \text{ ps} = 400 \text{ ps}$$

- ❖ Which improvement would be best?

- Vote at <http://PollEv.com/pbjones>

A. 190 ps clock (*over clocking, faster CPU*)

$$2 \text{ cycles} * 190 = 380 \text{ ps}$$

B. Miss penalty of 40 clock cycles (*change memory*)

$$(1 + .02 * 40) * 200 = 1.8 * 200 = 360 \text{ ps}$$

C. MR of 0.015 misses/instruction (*write better code*)

$$(1 + .015 * 50) * 200 = 1.75 * 200 = 350 \text{ ps}$$

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!

example:

CPU



L1



L2



memory

① optimize L1 for fast HT
② optimize L2 for low MR

Summary

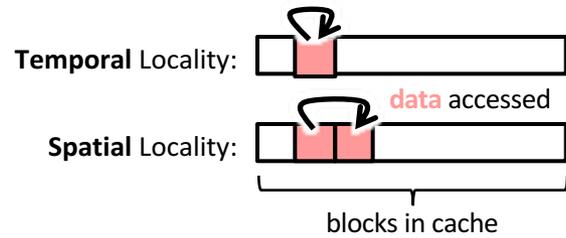
❖ Memory Hierarchy

- 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) = $HT + MR \times MP$
 - Hurt by Miss Rate and Miss Penalty

Notes Diagrams



Handout: Any Locality?

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

❖ Data:

- Temporal:
- Spatial:

❖ Instructions:

- Temporal:
- Spatial: