

# Memory & Caches I

CSE 351 Autumn 2024

**Instructor:**

Ruth Anderson

**Teaching Assistants:**

Alexandra Michael

Connie Chen

Chloe Fong

Chendur Jayavelu

Joshua Tan

Nikolas McNamee

Nahush Shrivatsa

Naama Amiel

Neela Kausik

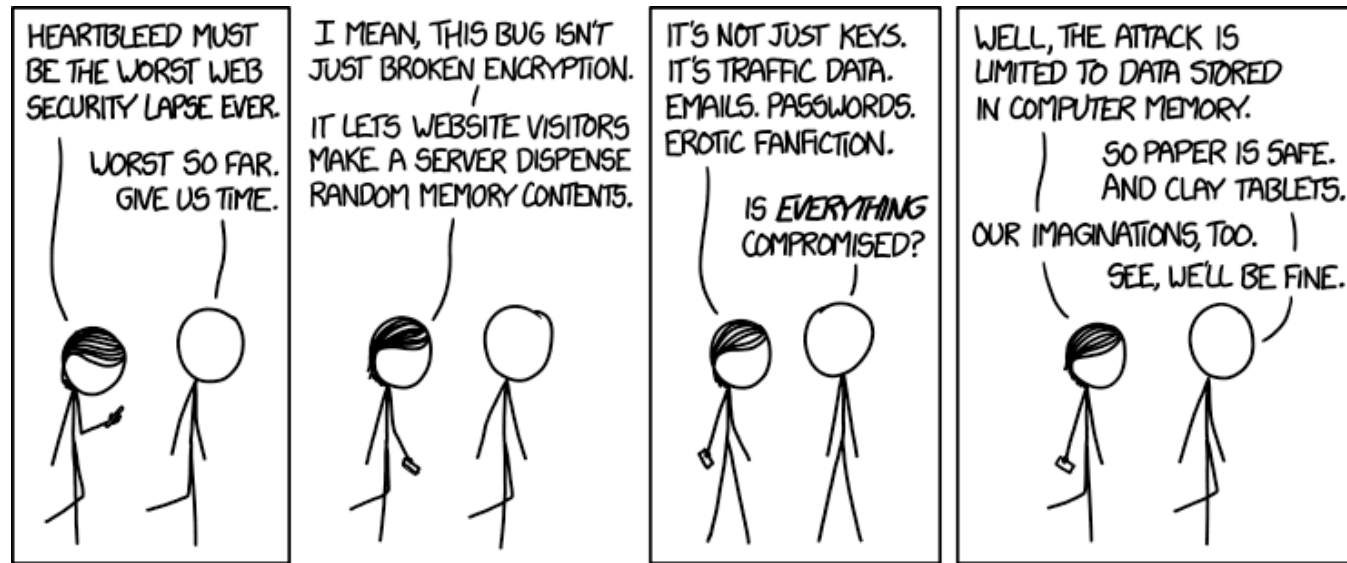
Renee Ruan

Rubee Zhao

Samantha Dreussi

Sean Siddens

Waleed Yagoub



**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/>

# Relevant Course Information

- ❖ HW14 due TONIGHT, Wednesday (10/30) @ 11:59 pm
- ❖ No Lecture on Fri 11/01 (No HW/Reading due)
  - HW15 not due until next Monday (11/04)
  - HW16 due<sup>MC</sup> next Wed (11/06)
- ❖ **Midterm Exam:** See [Ed Post](#) and [exams page](#)
  - Take home, on Gradescope
  - Open: Thursday 10/31 at 5pm; Due: Saturday 11/02 at 11:59pm
  - Review in section this week (10/31)
- ❖ Lab 3 due Mon 11/11
  - Encouraged to aim for Fri 11/08, actual deadline Mon 11/11
  - You have everything you need to do the lab as of last lecture!
  - Last part of HW15 (due Mon 11/04) is useful for Lab 3

# Aside: Units and Prefixes (Review)

- ❖ Here focusing on large numbers (exponents  $> 0$ )
- ❖ Note that  $10^3 \approx 2^{10}$   
*1000 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**eppelins, **Y**o
    - <https://xkcd.com/992/>
  - Post your best on Ed Discussion!

# Reading Review

## ❖ Terminology:

- Caches: cache blocks, cache hit, cache miss
- Principle of locality: temporal and spatial
- Average memory access time (AMAT): hit time, miss penalty, hit rate, miss rate

# Review Questions

- ❖ Convert the following to or from IEC:

$$\begin{aligned} 2^9 \cdot 512 \text{ Ki-books} &= 2^9 \cdot 2^{10} = 2^{19} \text{ books} \\ 2^7 \cdot 2^{20} \text{ cats} &= 128 \text{ Mi-cats} \end{aligned}$$

- ❖ Compute the average memory access time (AMAT) for the following system properties:

- HT ■ Hit time of 1 ns
- MR ■ Miss rate of 1%
- MP ■ Miss penalty of 100 ns

$$AMAT = HT + MR \cdot MP$$

$$= 1 \text{ ns} + 0.01 \cdot 100 \text{ ns}$$

$$= 1 \text{ ns} + 1 \text{ ns}$$

$$= 2 \text{ ns}$$

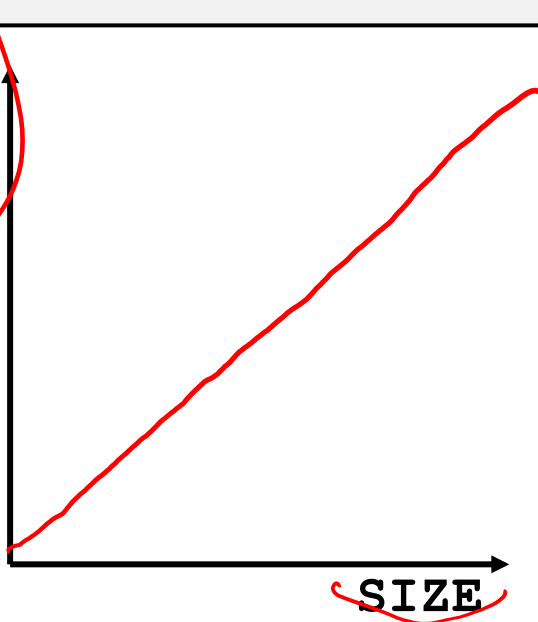
# How does execution time grow with SIZE?

*#define SIZE 1000*

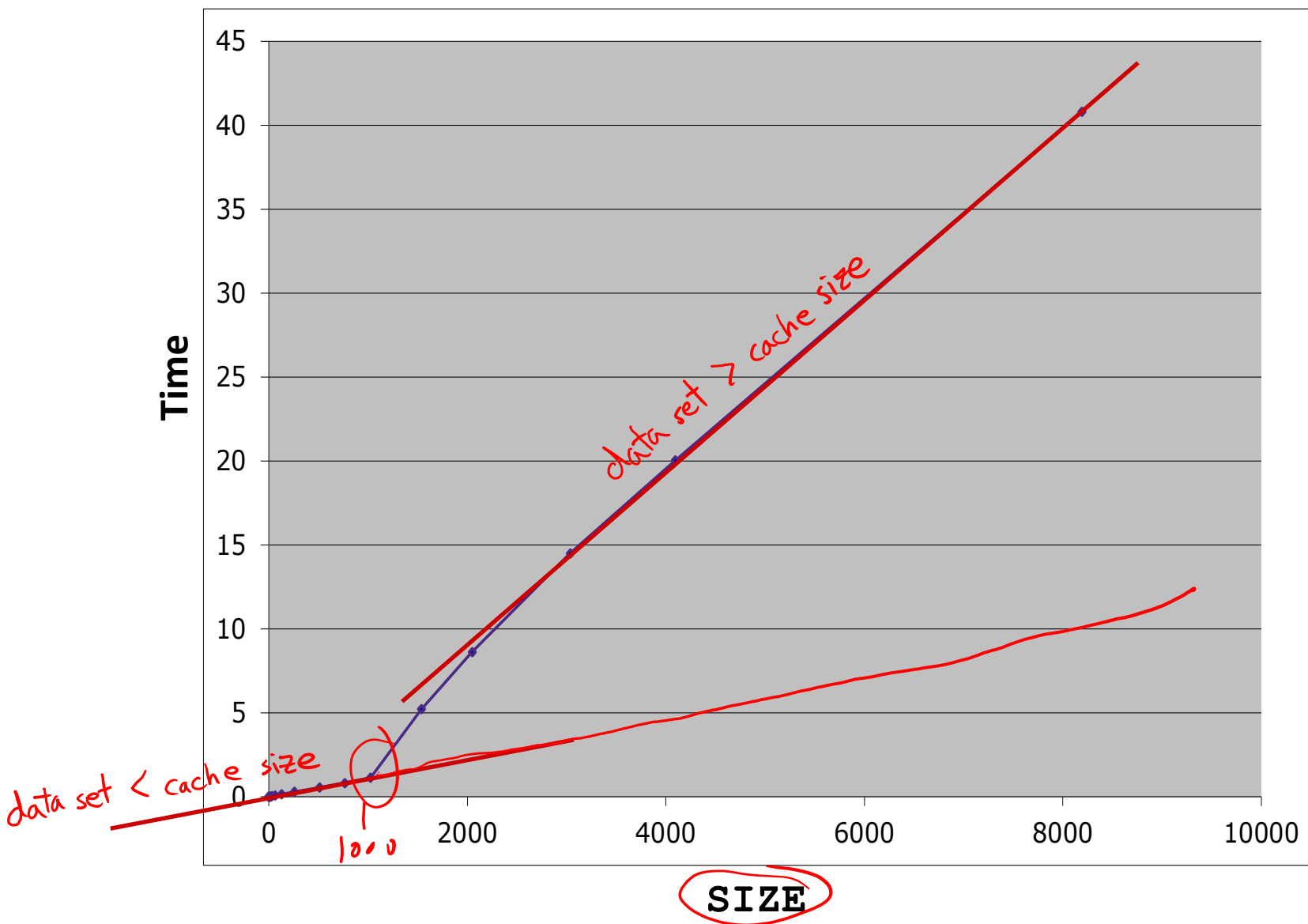
```
int array[SIZE];  
int sum = 0;  
  
for (int i = 0; i < 200000; i++) {  
    for (int j = 0; j < SIZE; j++) {  
        sum += array[j];  
    }  
}
```

Plot:

Execution Time



# Actual Data

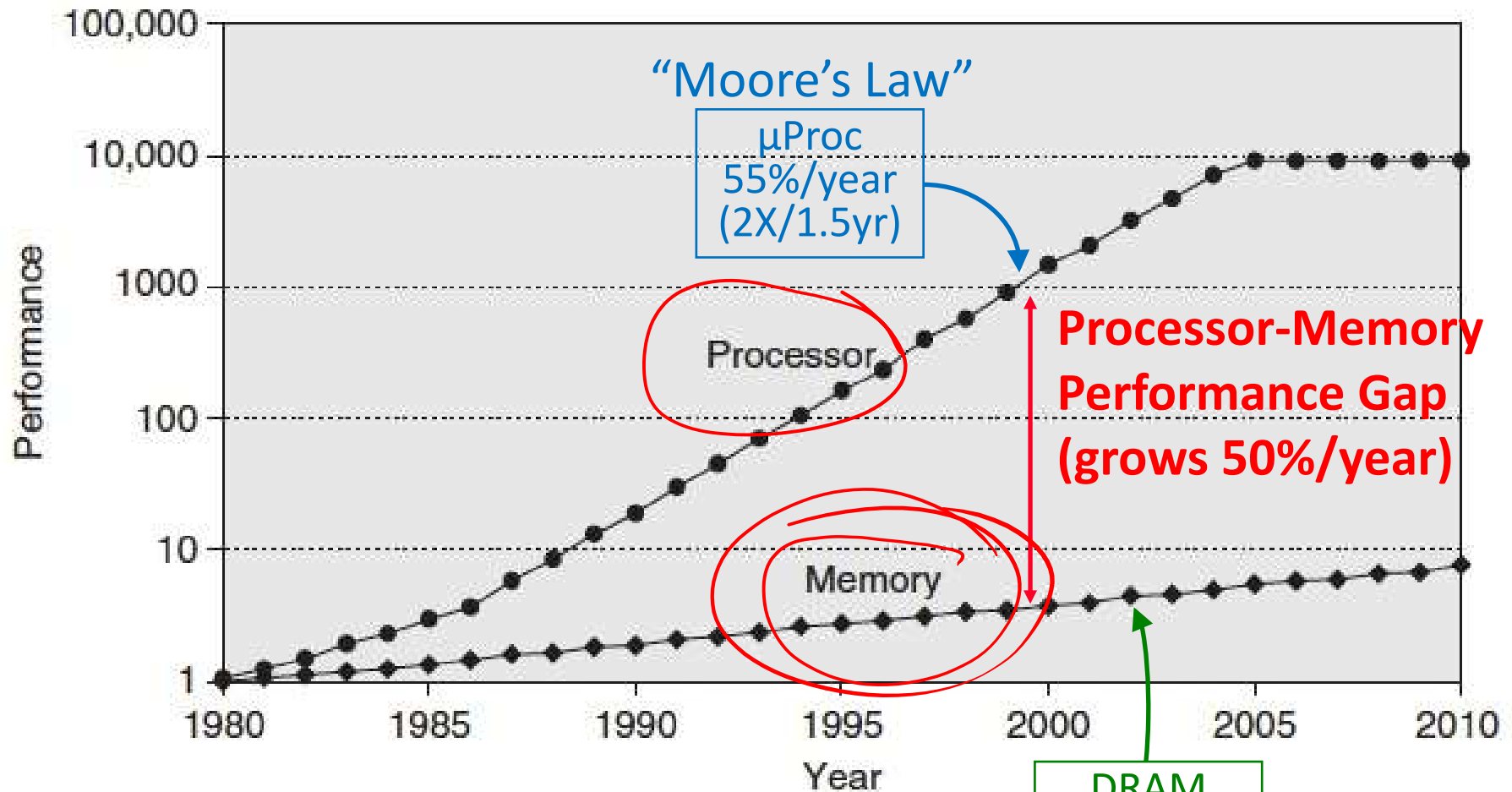


# Making memory accesses fast!

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

# Processor-Memory Gap

*add %rax, (%rbx)*



**1989** first Intel CPU with cache on chip

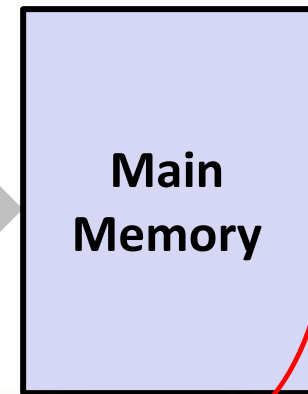
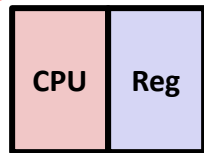
**1998** Pentium III has two cache levels on chip

# Problem: Processor-Memory Bottleneck

*add(%rax), %rbx*

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)



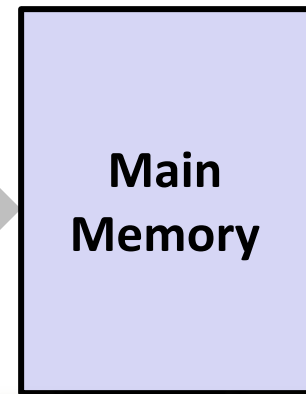
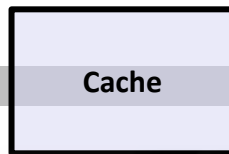
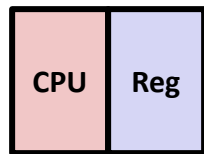
**Problem: lots of waiting on memory**

*cycle: single machine step (fixed-time)*

# Problem: Processor-Memory Bottleneck

Processor performance  
doubled about  
every 18 months

Bus latency / bandwidth  
evolved much slower



**Core 2 Duo:**

Can process at least  
256 Bytes/cycle



*fridge/  
pantry*

**Core 2 Duo:**

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



*grocery store*



*sandwich  
to mouth*

***Solution: caches***

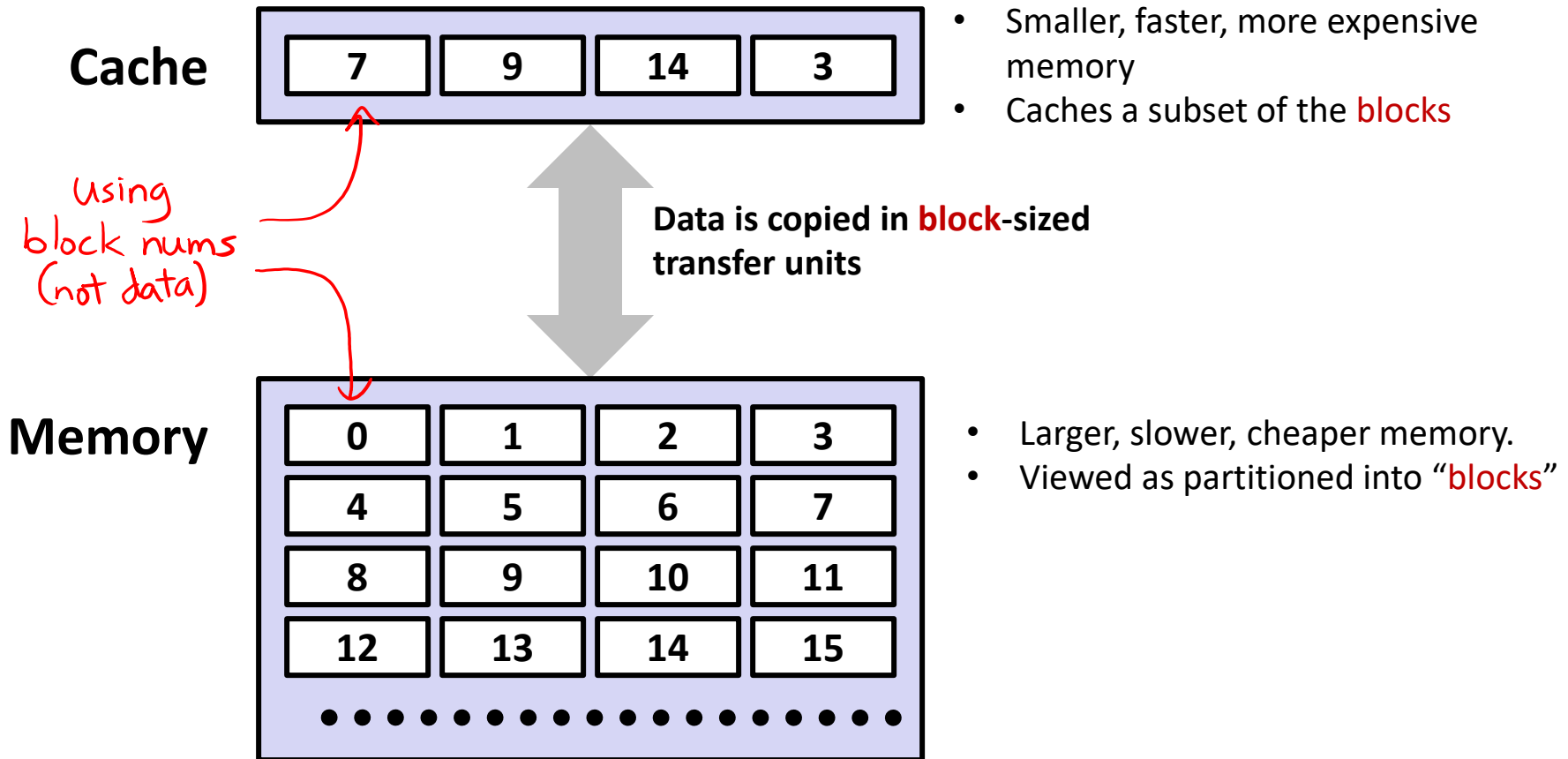
***cycle: single machine step (fixed-time)***

# Cache

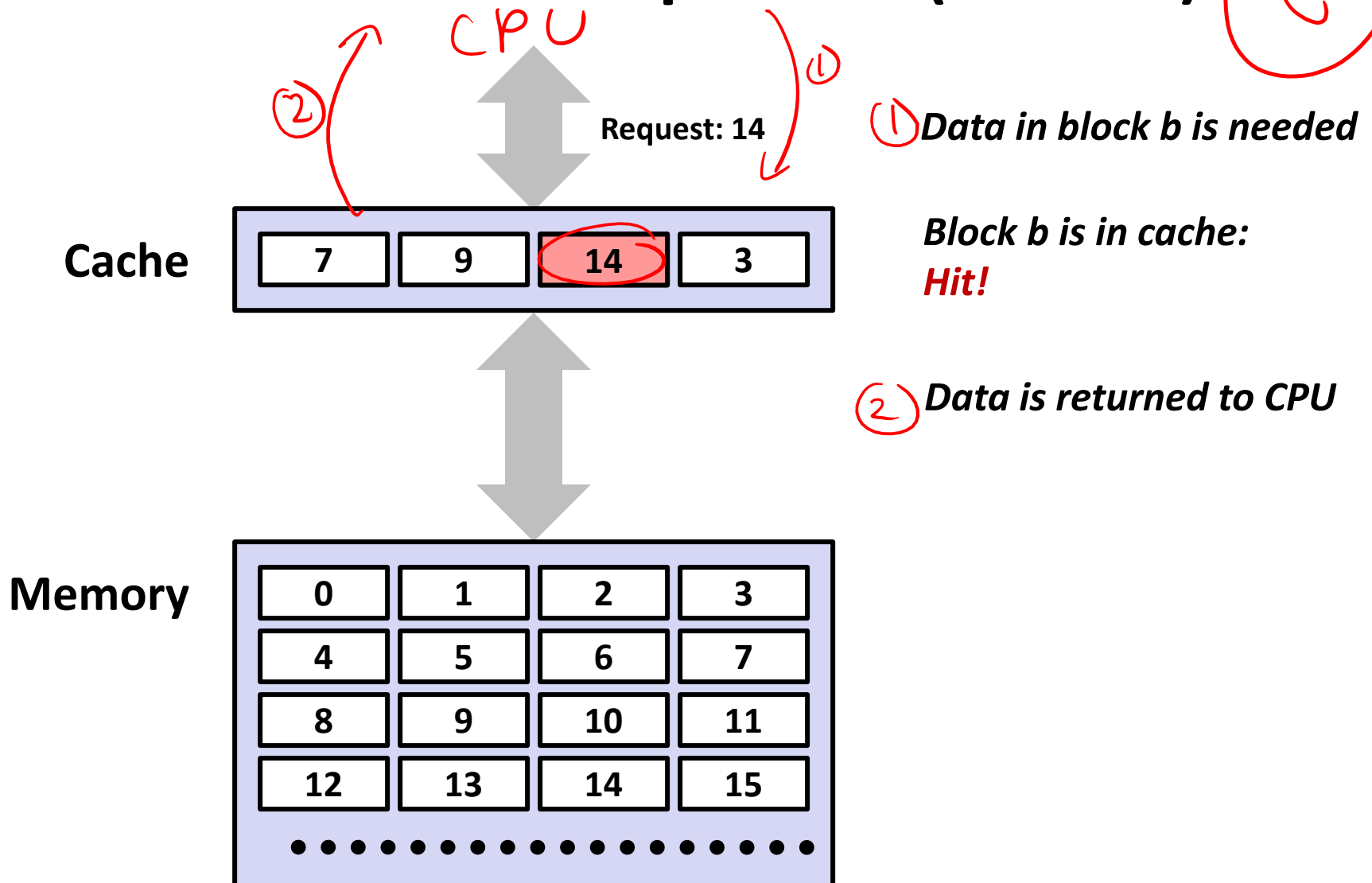
- ❖ Pronunciation: “cash”
  - We abbreviate this as “\$”
- ❖ English: A hidden storage space for provisions, weapons, and/or treasures
- ❖ Computer: Memory with short access time used for the storage of frequently or recently used instructions (i-cache/I\$) or data (d-cache/D\$)
  - *More generally*: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

# General Cache Mechanics (Review)

CPU

 $2 \ell_{\phi}(\gamma_{\max}), \gamma_{\max}$ 

# General Cache Concepts: **Hit** (Review)





# Why Caches Work (Review)

- ❖ **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

# Why Caches Work (Review)

- ❖ **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 (Review)

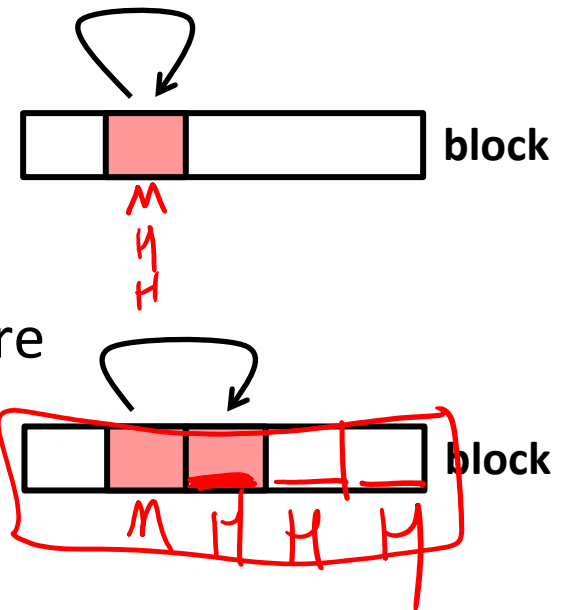
- ❖ **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- ❖ **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:

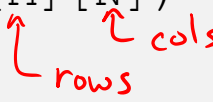
- Temporal: sum referenced in each iteration
- Spatial: consecutive elements of array a [ ] accessed

## ❖ Instructions:

- Temporal: cycle through loop repeatedly
- Spatial: reference instructions in sequence

# Locality Example #1 Code

```
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;

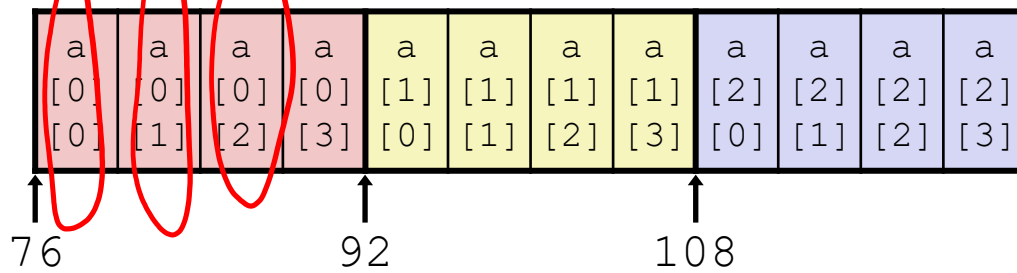
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}

```

*Handwritten:*  $a[0][0]$   
0 1  
0 2

## Layout in Memory



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

*Handwritten:* Rows Columns

**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 = ?

*Handwritten:* 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]

# Locality Example #2 Code

```
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;

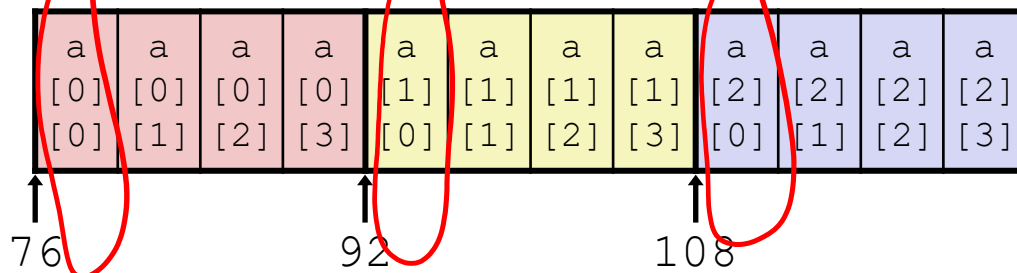
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}

```

Handwritten annotations: *rows* (pointing to *i*), *cols* (pointing to *j*).  
*a[0][0]*  
*a[1][0]*  
*a[2][0]*

## Layout in Memory



**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 = ?

Handwritten: *4*, *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]

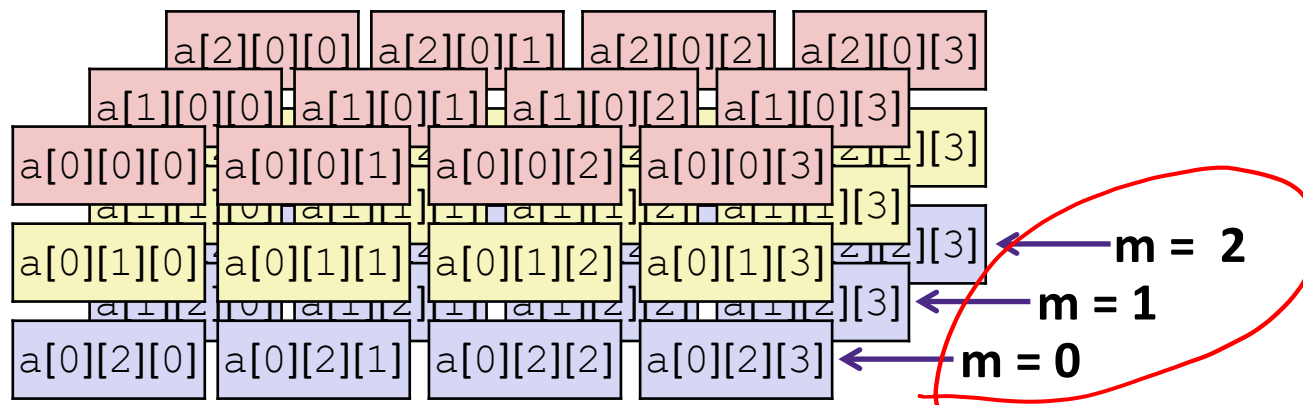
## Locality Example #3 Code

```
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;
}
```

- ❖ 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;
}

```

Handwritten red annotations on the code: A red arrow points from the `for (i = 0; i < N; i++)` loop to the `for (k = 0; k < M; k++)` loop. Another red arrow points from the `for (j = 0; j < L; j++)` loop to the `sum += a[k][i][j];` line. A red bracket is drawn around the `for (k = 0; k < M; k++)` loop. To the right of the code, the values 0, 1, 2 are written vertically, corresponding to the `k` index, with red circles around them.

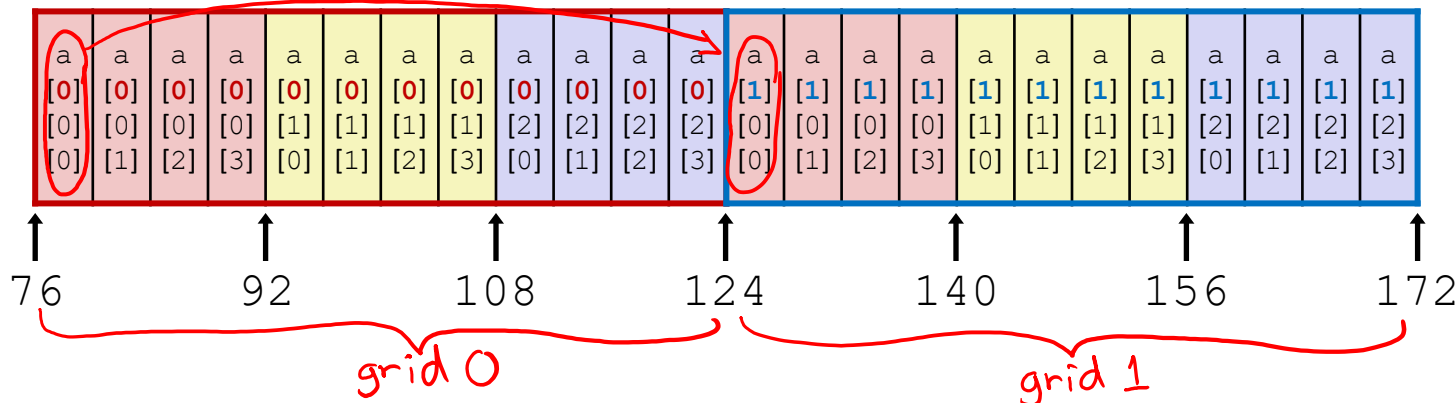
❖ What is wrong with this code?

*stride -  $N * L$*

❖ How can it be fixed?

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

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



# Cache Performance Metrics (Review)

- ❖ 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)

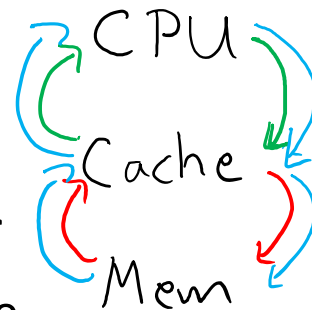
- Fraction of memory references not found in cache  
(misses / accesses) = 1 - Hit Rate

Hit takes HT

- ❖ Hit Time (HT)

Miss takes HT + MP

- 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 (Review)

- ❖ Two things hurt the performance of a cache:

- Miss rate and miss penalty

$$HT \cdot HR + MT \cdot MR$$

$$HT \cdot (1 - MR) + (HT + MP) \cdot MR$$

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

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

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

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

- Assume HT of 1 clock cycle and MP of 100 clock cycles

- 97%:  $\text{AMAT} = 1 + 0.03 \cdot 100 = 1 + 3 = 4$  clock cycles

- 99%:  $\text{AMAT} = 1 + 0.01 \cdot 100 = 1 + 1 = 2$  clock cycles

# Practice Question

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

$$\text{AMAT} = \text{HT} + \text{MR} * \text{MP} = 1 + 0.02 * 50 = 2 \text{ clock cycles} = 400 \text{ ps}$$

- ❖ Which improvement would be best?

A. **190 ps clock** (overclocking, faster CPU)

$$2 \text{ clock cycles} = 380 \text{ ps}$$

B. **Miss penalty of 40 clock cycles** (reduced Mem size)

$$1 + 0.02 * \boxed{40} = 1.8 \text{ clock cycles} = 360 \text{ ps}$$

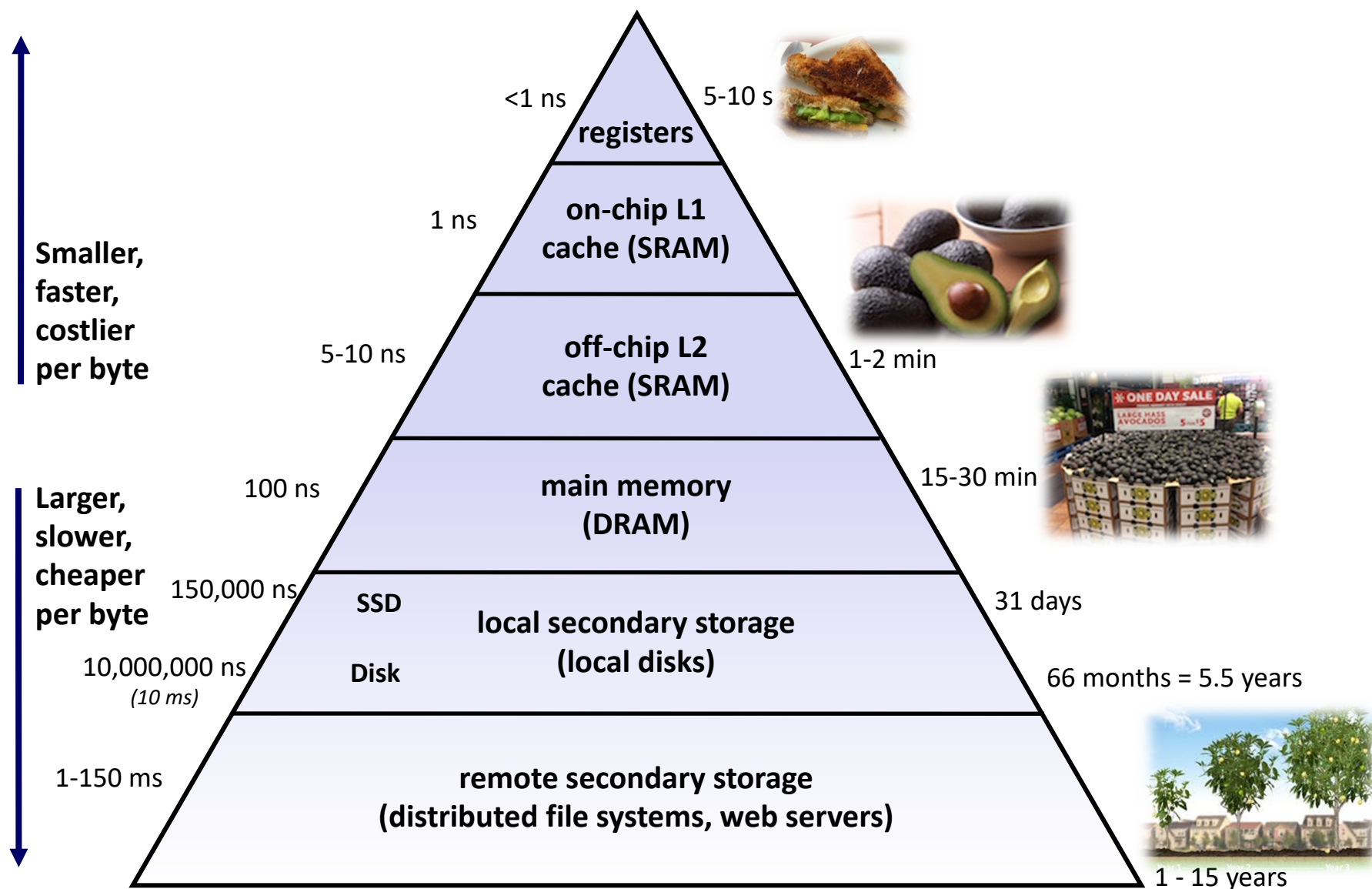
C. **MR of 0.015 misses/instruction** (write better code)

$$1 + \boxed{0.015} * 50 = 1.75 \text{ clock cycles} = 350 \text{ ps}$$

# 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!

# An Example Memory Hierarchy



# Summary

## ❖ Memory Hierarchy

- Successively higher levels contain “most used” data from lower levels
- Makes use of *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