

The Hardware/Software Interface

CSE351 Spring 2013

Memory and Caches I

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();
```

Assembly
language:

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

Machine
code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer
system:



Data & addressing
Integers & floats
Machine code & C
x86 assembly
programming
Procedures &
stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:



Themes of CSE 351

■ Interfaces and abstractions

- So far: data type abstractions in C; x86 instruction set architecture (interface to hardware)
- Today: abstractions of *memory*
 - Soon: process and virtual memory abstractions

■ Representation

- Integers, floats, addresses, arrays, structs

■ Translation

- Understand the assembly code that will be generated from C code

■ Control flow

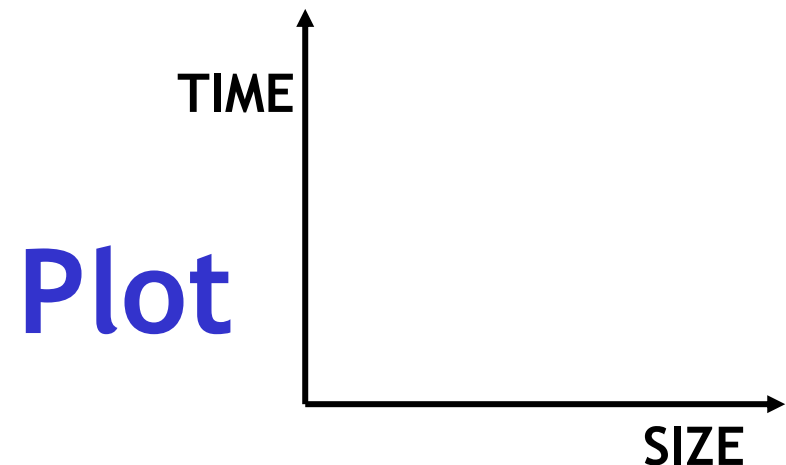
- Procedures and stacks; buffer overflows

Making memory accesses fast!

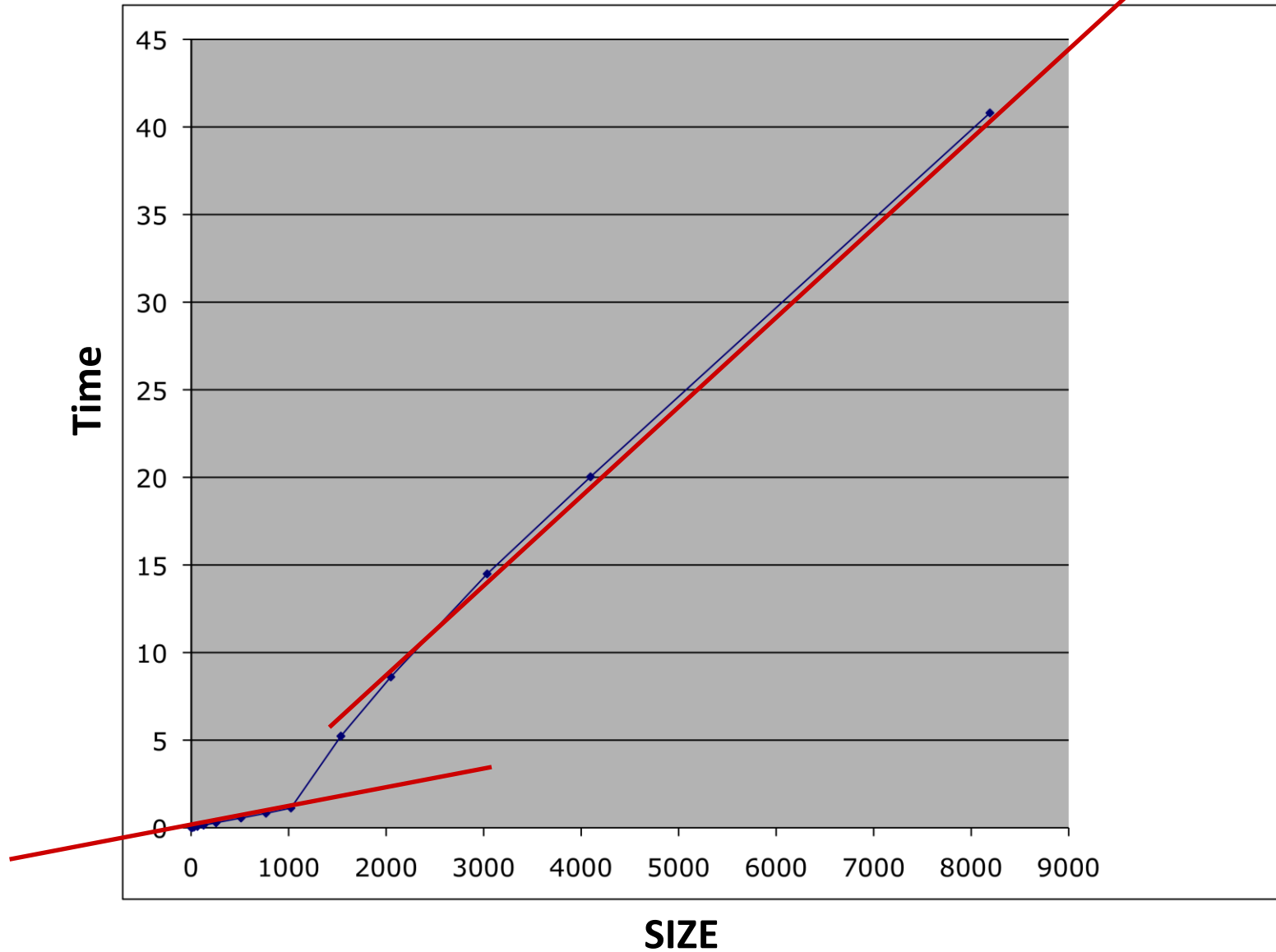
- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches

How does execution time grow with SIZE?

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

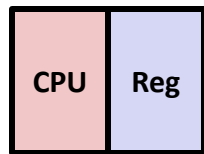


Actual Data

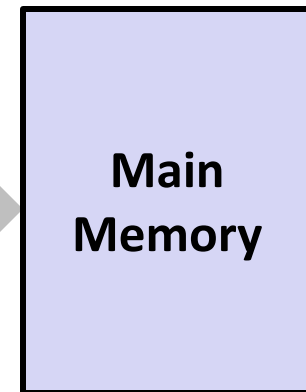


Problem: Processor-Memory Bottleneck

Processor performance
doubled about
every 18 months



Bus bandwidth
evolved much slower



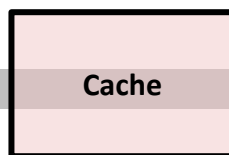
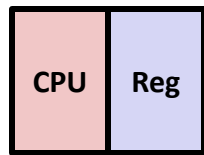
Core 2 Duo:
Can process at least
256 Bytes/cycle

Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100 cycles

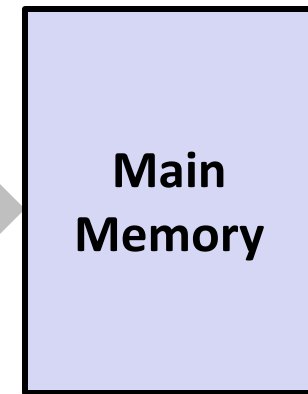
Problem: lots of waiting on memory

Problem: Processor-Memory Bottleneck

Processor performance
doubled about
every 18 months



Bus bandwidth
evolved much slower



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

Core 2 Duo:
Bandwidth
2 Bytes/cycle
Latency
100 cycles

Solution: caches

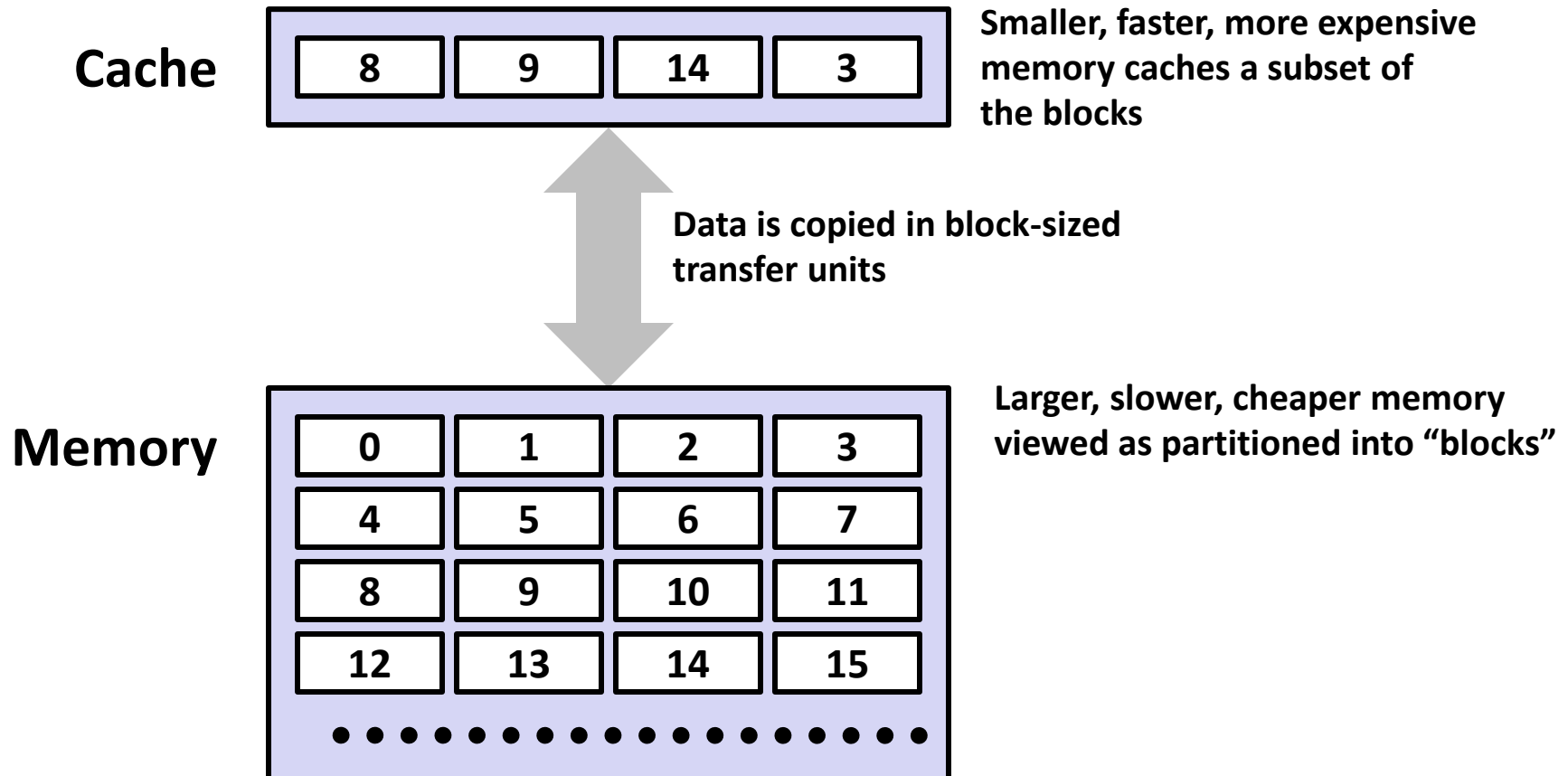
Cache

- **English definition:** a hidden storage space for provisions, weapons, and/or treasures
- **CSE definition:** computer memory with short access time used for the storage of frequently or recently used instructions or data (i-cache and d-cache)

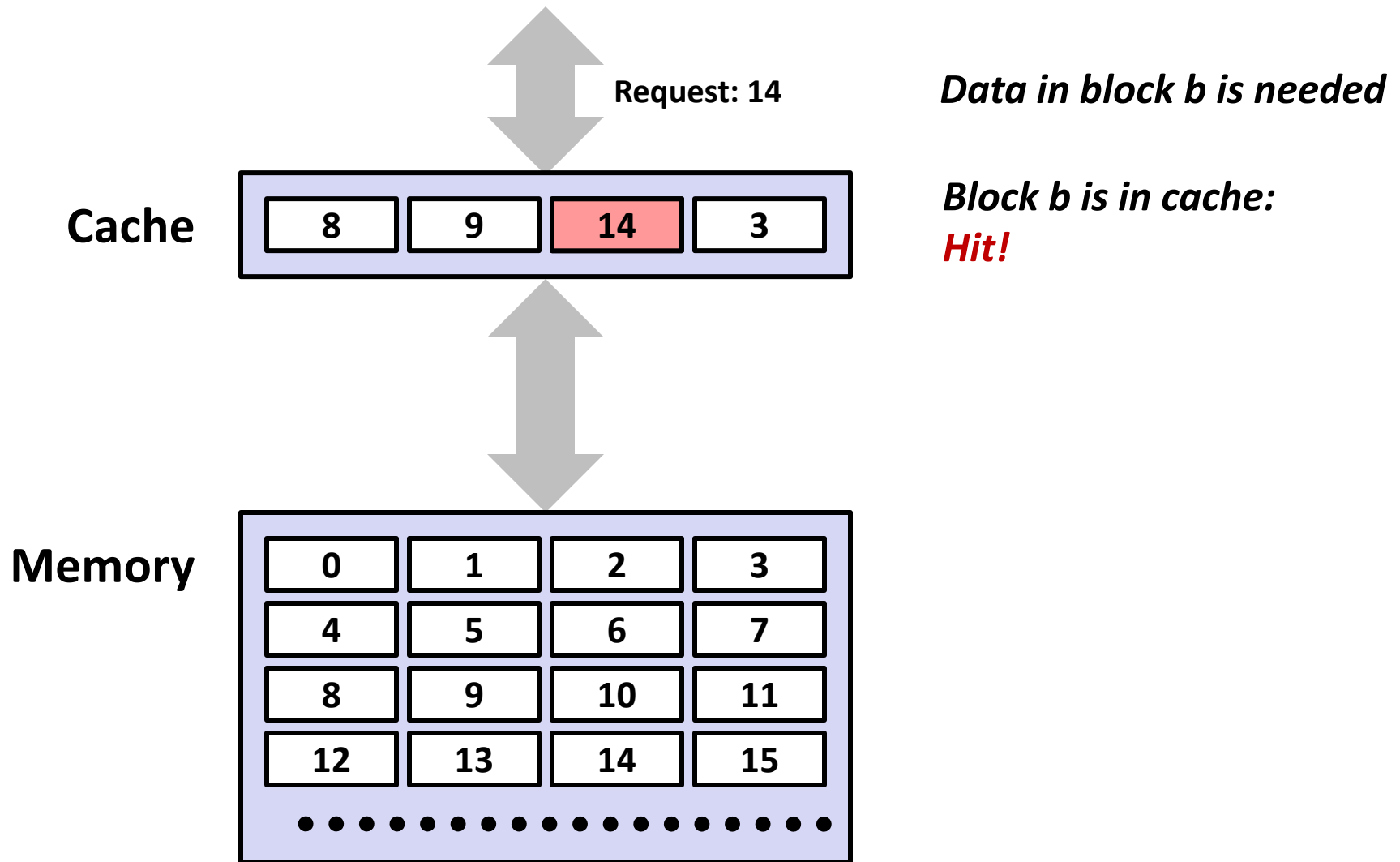
more generally,

used to optimize data transfers between system elements with different characteristics (network interface cache, I/O cache, etc.)

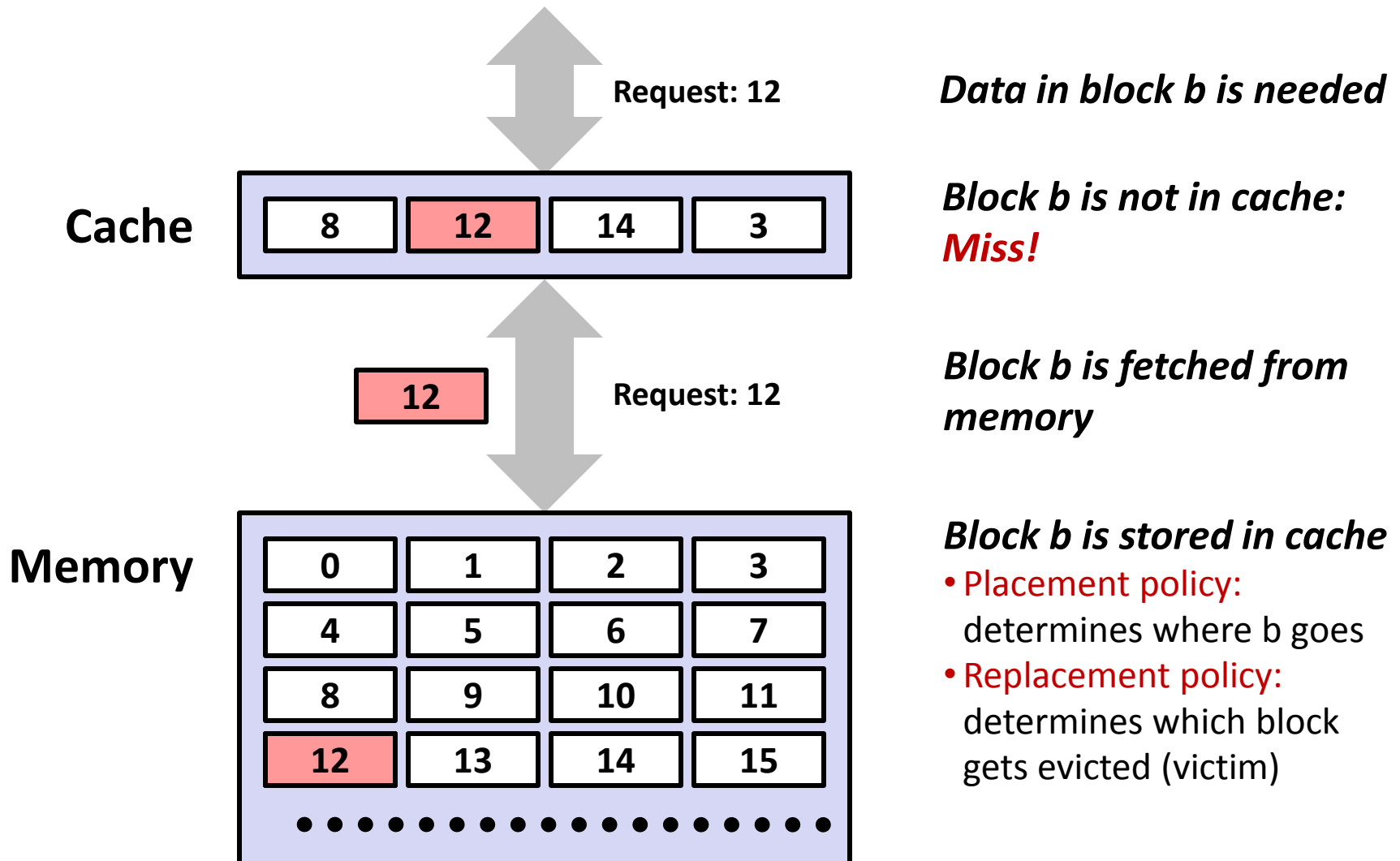
General Cache Mechanics



General Cache Concepts: **Hit**



General Cache Concepts: **Miss**



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 is this important?



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?**

Why Caches Work

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

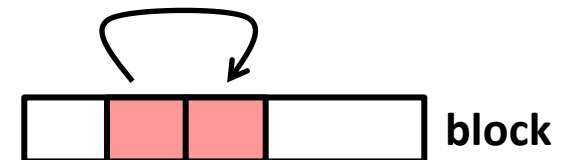
- **Temporal locality:**

- 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: Locality?

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

Example: Locality?

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

■ Data:

- Temporal: **sum** referenced in each iteration
- Spatial: array **a []** accessed in stride-1 pattern

Example: Locality?

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

■ Data:

- Temporal: **sum** referenced in each iteration
- Spatial: array **a []** accessed in stride-1 pattern

■ Instructions:

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

Example: Locality?

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

■ Data:

- Temporal: **sum** referenced in each iteration
- Spatial: array **a []** accessed in stride-1 pattern

■ Instructions:

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

- **Being able to assess the locality of code is a crucial skill for a programmer**

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;
}
```

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]

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;
}
```

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]

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

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;
}
```

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]

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;
}
```

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]

- 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-N

Locality Example #3

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; 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?

Cost of Cache Misses

- **Huge difference between a hit and a miss**
 - Could be 100x, if just L1 and main memory
- **Would you believe 99% hits is twice as good as 97%?**
 - Consider:
 - Cache hit time of 1 cycle
 - Miss penalty of 100 cycles

Cost of Cache Misses

- **Huge difference between a hit and a miss**
 - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
 - Consider:
 - Cache hit time of 1 cycle
 - Miss penalty of 100 cycles

 - Average access time:
 - 97% hits: $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles}$
 - 99% hits: $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$

- **This is why “miss rate” is used instead of “hit rate”**

Cache Performance Metrics

■ Miss Rate

- Fraction of memory references not found in cache (misses / accesses)
= 1 - hit rate
- Typical numbers (in percentages):
 - 3% - 10% for L1
 - Can be quite small (e.g., < 1%) for L2, depending on size, etc.

■ Hit Time

- Time to deliver a line in the cache to the processor
 - Includes time to determine whether the line is in the cache
- Typical hit times: 1 - 2 clock cycles for L1; 5 - 20 clock cycles for L2

■ Miss Penalty

- Additional time required because of a miss
- Typically 50 - 200 cycles for L2 (trend: increasing!)

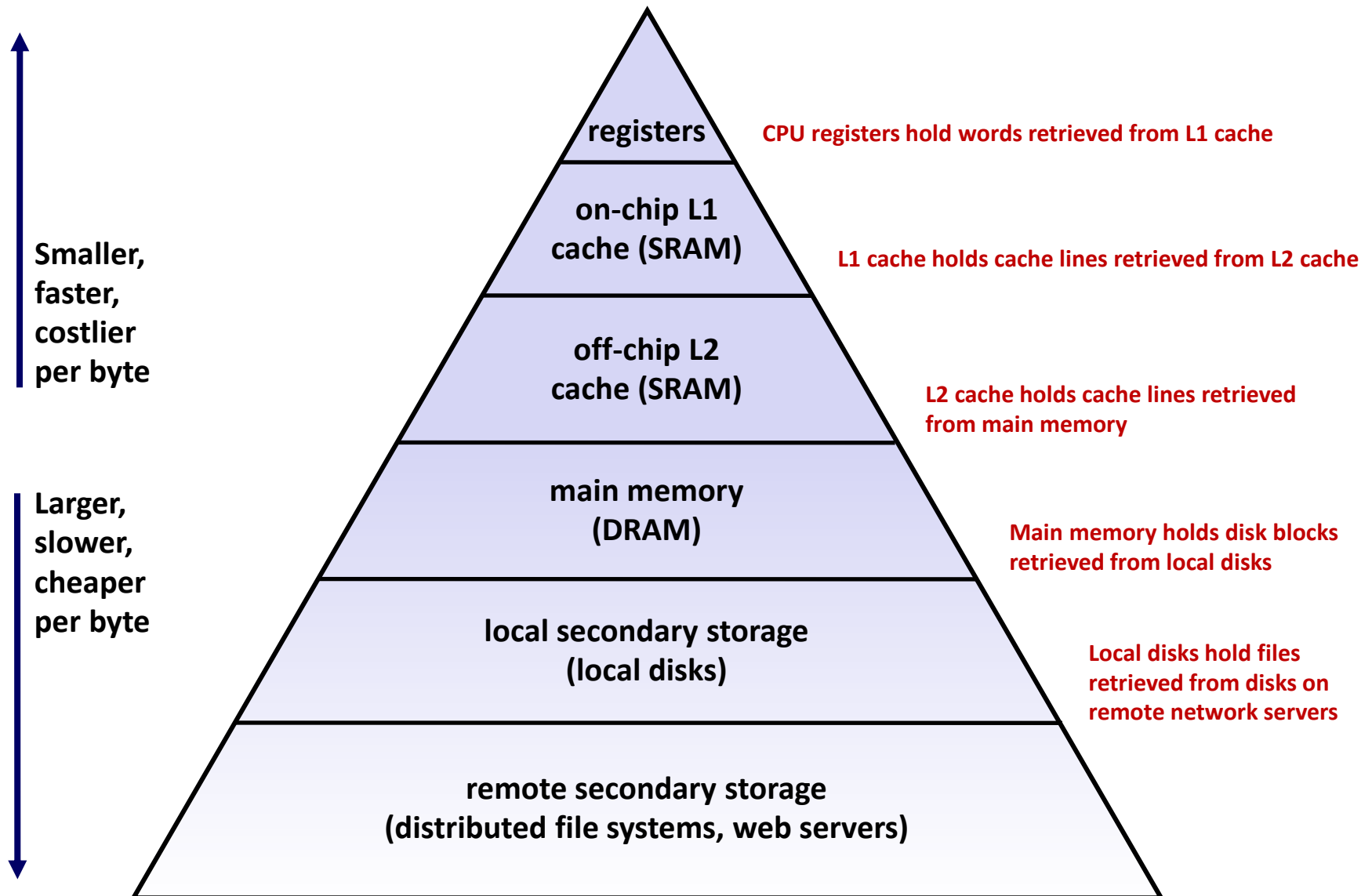
Can we have more than one cache?

- Why would we want to do that?

Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software systems:**
 - Faster storage technologies almost always cost more per byte and have lower capacity
 - The gaps between memory technology speeds are widening
 - True for: registers \leftrightarrow cache, cache \leftrightarrow DRAM, DRAM \leftrightarrow disk, etc.
 - Well-written programs tend to exhibit good locality
- **These properties complement each other beautifully**
- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy**

An Example Memory Hierarchy



Memory Hierarchies

■ Fundamental idea of a memory hierarchy:

- For each k , the faster, smaller device at level k serves as a cache for the larger, slower device at level $k+1$.

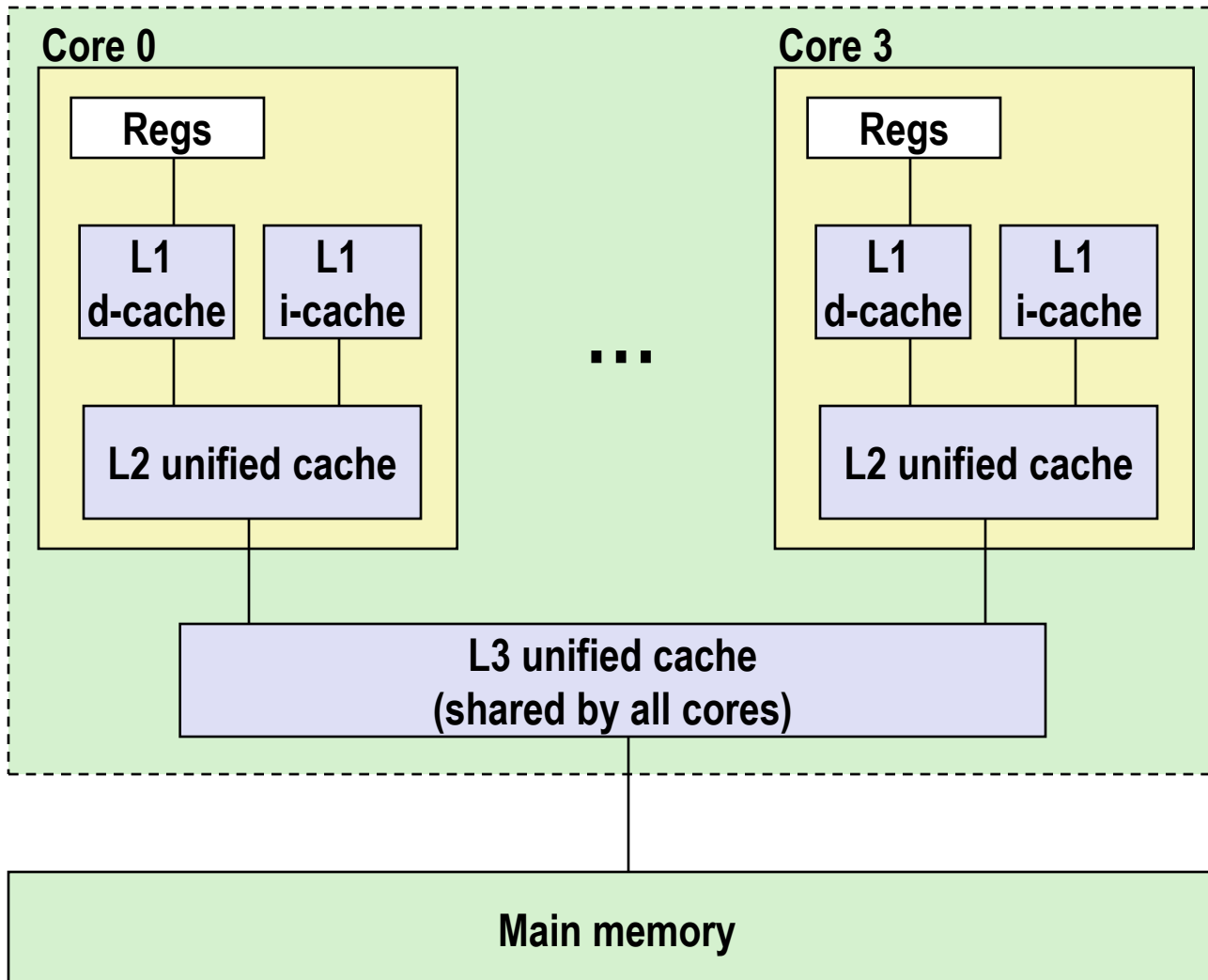
■ Why do memory hierarchies work?

- Because of locality, programs tend to access the data at level k more often than they access the data at level $k+1$.
- Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.

- ***Big Idea:*** The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Intel Core i7 Cache Hierarchy

Processor package



L1 i-cache and d-cache:

32 KB, 8-way,
Access: 4 cycles

L2 unified cache:

256 KB, 8-way,
Access: 11 cycles

L3 unified cache:

8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for
all caches.