

## Introduction to Computer Systems

15-213/18-243, fall 2009  
23<sup>rd</sup> Lecture, Nov. 24<sup>th</sup>

### Instructors:

Roger Dannenberg and Greg Ganger

## Today

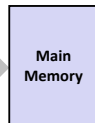
- Memory hierarchy, caches, locality
- Cache organization

## 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  
(1 SSE two operand add and mult)

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

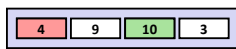
**Solution: Caches**

## Cache

- **Definition:** Computer memory with short access time used for the storage of frequently or recently used instructions or data

## General Cache Mechanics

Cache

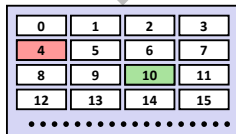


Smaller, faster, more expensive  
memory caches a subset of  
the blocks



Data is copied in block-sized  
transfer units

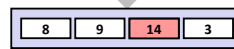
Memory



Larger, slower, cheaper memory  
viewed as partitioned into "blocks"

## General Cache Concepts: Hit

Cache

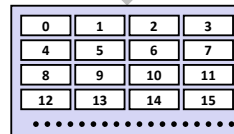


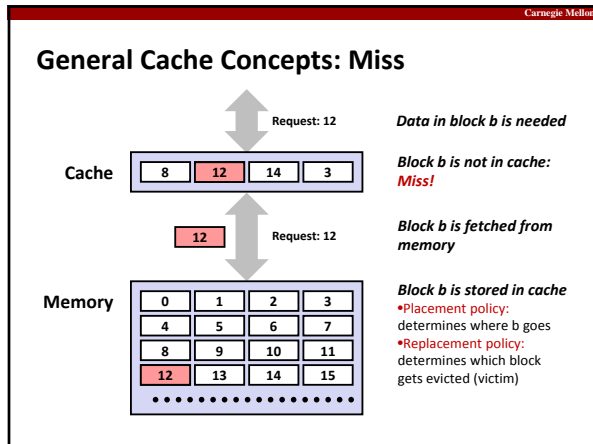
Request: 14

Data in block b is needed

Block b is in cache:  
**Hit!**

Memory





- Carnegie Mellon
- ## 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 numbers:
      - 1-2 clock cycle for L1
      - 5-20 clock cycles for L2
  - **Miss Penalty**
    - Additional time required because of a miss
      - typically 50-200 cycles for main memory (Trend: increasing!)

- Carnegie Mellon
- ## Lets think about those numbers
- **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 cycle + 0.03 \* 100 cycles = **4 cycles**
      - 99% hits: 1 cycle + 0.01 \* 100 cycles = **2 cycles**
  - **This is why "miss rate" is used instead of "hit rate"**

- Carnegie Mellon
- ## Types of Cache Misses
- **Cold (compulsory) miss**
    - Occurs on first access to a block
  - **Conflict miss**
    - Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
      - e.g., block i must be placed in slot (i mod 4)
    - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
      - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
  - **Capacity miss**
    - Occurs when the set of active cache blocks (working set) is larger than the cache

- Carnegie Mellon
- ## 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

- Carnegie Mellon
- ## 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;
}
```

## Locality Example #2

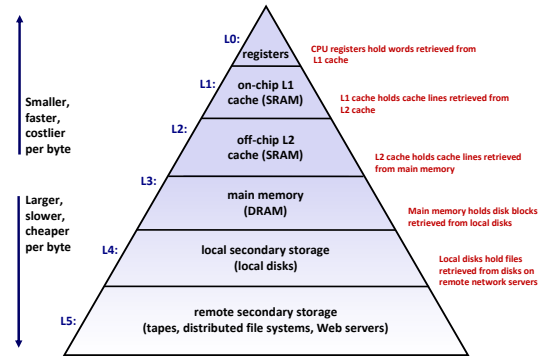
```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

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

## 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 of registers  $\leftrightarrow$  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

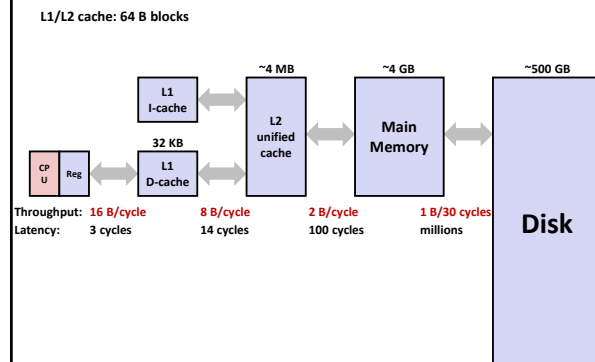


## Examples of Caching in the Hierarchy

| Cache Type           | What is Cached?      | Where is it Cached? | Latency (cycles) | Managed By       |
|----------------------|----------------------|---------------------|------------------|------------------|
| Registers            | 4-byte words         | CPU core            | 0                | Compiler         |
| TLB                  | Address translations | On-Chip TLB         | 0                | Hardware         |
| L1 cache             | 64-bytes block       | On-Chip L1          | 1                | Hardware         |
| L2 cache             | 64-bytes block       | Off-Chip L2         | 10               | Hardware         |
| Virtual Memory       | 4-KB page            | Main memory         | 100              | Hardware+OS      |
| Buffer cache         | Parts of files       | Main memory         | 100              | OS               |
| Network buffer cache | Parts of files       | Local disk          | 10,000,000       | AFS client       |
| Browser cache        | Web pages            | Local disk          | 10,000,000       | Web browser      |
| Web cache            | Web pages            | Remote server disks | 1,000,000,000    | Web proxy server |

## Memory Hierarchy: Core 2 Duo

*Not drawn to scale*



Carnegie Mellon

## Today

- Memory hierarchy, caches, locality
- Cache organization

Carnegie Mellon

## General Cache Organization (S, E, B)

**Cache size:**  
 $S \times E \times B$  data bytes

$B = 2^b$  bytes per cache block (the data)

Carnegie Mellon

## Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

**Address of word:**  
tag    set index    block offset

$B = 2^b$  bytes per cache block (the data)

Carnegie Mellon

## Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set  
Assume: cache block size 8 bytes

**Address of int:**  
t bits    0...01    100

find set

Carnegie Mellon

## Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set  
Assume: cache block size 8 bytes

**Address of int:**  
t bits    0...01    100

valid? + match: assume yes = hit

block offset

Carnegie Mellon

## Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set  
Assume: cache block size 8 bytes

**Address of int:**  
t bits    0...01    100

valid? + match: assume yes = hit

block offset

int (4 Bytes) is here

**No match: old line is evicted and replaced**

Carnegie Mellon

### Example

```

int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}

```

Ignore the variables *sum, i, j*  
 assume: cold (empty) cache,  
 a[0][0] goes here

32 B = 4 doubles

blackboard

Carnegie Mellon

### E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set  
 Assume: cache block size 8 bytes

Address of short int:  
 t bits    0...01    100

find set

Carnegie Mellon

### E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set  
 Assume: cache block size 8 bytes

Address of short int:  
 t bits    0...01    100

valid? + match: yes = hit

compare both

block offset

Carnegie Mellon

### E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set  
 Assume: cache block size 8 bytes

Address of short int:  
 t bits    0...01    100

valid? + match: yes = hit

match both

short int (2 Bytes) is here

block offset

**No match:**

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

Carnegie Mellon

### Example

```

int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}

```

Ignore the variables *sum, i, j*  
 assume: cold (empty) cache,  
 a[0][0] goes here

32 B = 4 doubles

blackboard

Carnegie Mellon

### What about writes?

- Multiple copies of data exist:
  - L1, L2, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

## Software Caches are More Flexible

### Examples

- File system buffer caches, web browser caches, etc.

### Some design differences

- Almost always fully associative
  - so, no placement restrictions
  - index structures like hash tables are common
- Often use complex replacement policies
  - misses are very expensive when disk or network involved
  - worth thousands of cycles to avoid them
- Not necessarily constrained to single "block" transfers
  - may fetch or write-back in larger units, opportunistically

## Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimization (bonus... not on final exam):
  - Cache optimizations

## Optimizations for the Memory Hierarchy

### Write code that has locality

- Spatial: access data contiguously
- Temporal: make sure access to the same data is not too far apart in time

### How to achieve?

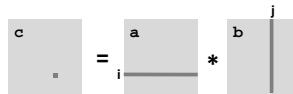
- Proper choice of algorithm
- Loop transformations

### Cache versus register level optimization:

- In both cases locality desirable
- Register space much smaller + requires scalar replacement to exploit temporal locality
- Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)

## Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n+j];
}
```



## Cache Miss Analysis

### Assume:

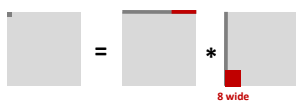
- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size  $C \ll n$  (much smaller than  $n$ )

### First iteration:

- $n/8 + n = 9n/8$  misses



- Afterwards in cache: (schematic)



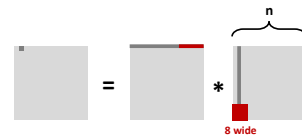
## Cache Miss Analysis

### Assume:

- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size  $C \ll n$  (much smaller than  $n$ )

### Second iteration:

- Again:  $n/8 + n = 9n/8$  misses



### Total misses:

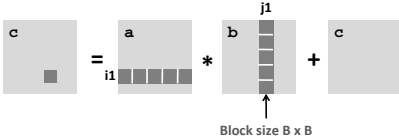
- $9n/8 * n^2 = (9/8) * n^3$

## Blocked Matrix Multiplication

```

c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (il = i; il < i+B; il++)
                    for (jl = j; jl < j+B; jl++)
                        for (kl = k; kl < k+B; kl++)
                            c[il*n+jl] += a[il*n + kl]*b[kl*n + jl];
}

```

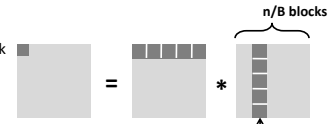


## Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size  $C \ll n$  (much smaller than  $n$ )
  - Three blocks fit into cache:  $3B^2 < C$

### First (block) iteration:

- $B^2/8$  misses for each block
- $2n/B * B^2/8 = nB/4$  (omitting matrix  $c$ )



- Afterwards in cache (schematic)

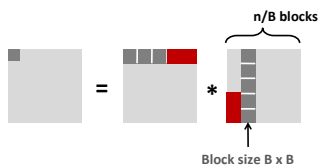


## Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size  $C \ll n$  (much smaller than  $n$ )
  - Three blocks fit into cache:  $3B^2 < C$

### Second (block) iteration:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$



### Total misses:

- $nB/4 * (n/B)^2 = n^3/(4B)$

## Summary

- No blocking:  $(9/8) * n^3$
- Blocking:  $1/(4B) * n^3$
- Suggest largest possible block size  $B$ , but limit  $3B^2 < C!$  (can possibly be relaxed a bit, but there is a limit for  $B$ )
- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data:  $3n^2$ , computation  $2n^3$
    - Every array elements used  $O(n)$  times!
  - But program has to be written properly

## Locality Example #3

```

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

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

    return sum;
}

```

- How can it be fixed?