Computer Architecture and Operating Systems
Lecture 8: Memory and Caches

Andrei Tatarnikov

atatarnikov@hse.ru
@andrewt0301

Processor-Memory Performance Gap

- Computer performance depends on:
  - Processor performance
  - Memory performance
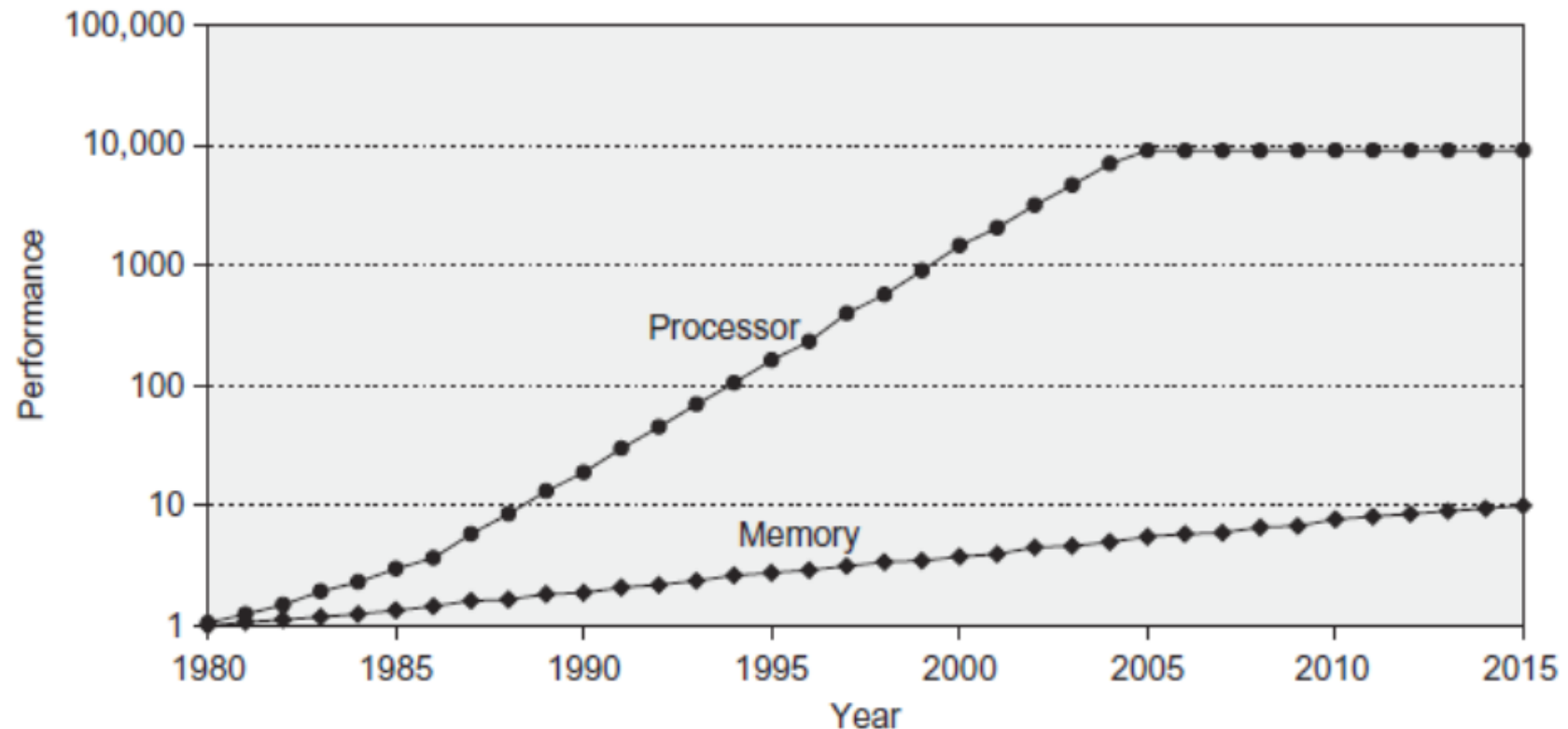
Memory Challenge

- Make memory appear as fast as processor

- Ideal memory:

  - Fast

  - Cheap (inexpensive)

  - Large (capacity)

  But can only choose two!

Memory Technology

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $20 – $75 per GB
- Magnetic disk
  - 5ms – 20ms, $0.20 – $2 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

Locality

**No need for large memory to access it fast**

Just exploit locality

- Temporal Locality:
  - Locality in time
  - If data used recently, likely to use it again soon
  - How to exploit: keep recently accessed data in higher levels of memory hierarchy

- Spatial Locality:
  - Locality in space
  - If data used recently, likely to use nearby data soon
  - How to exploit: when access data, bring nearby data into higher levels of memory hierarchy too
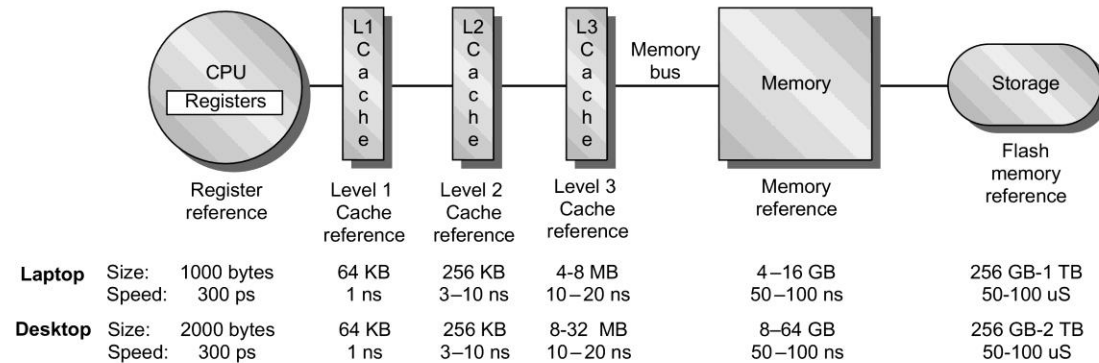
Taking Advantage of Locality

- Memory hierarchy

- Store everything on disk

- Copy recently accessed (and nearby) items from disk to smaller DRAM memory

  - Main memory

- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory

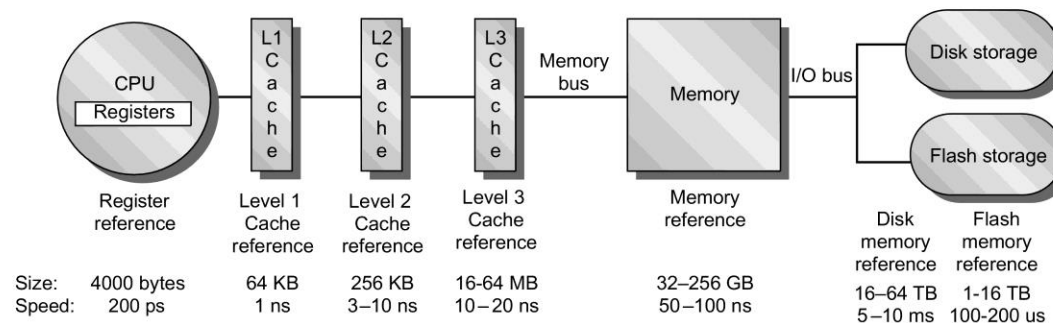 - Cache memory attached to CPU

Memory Hierarchy

- Personal mobile device
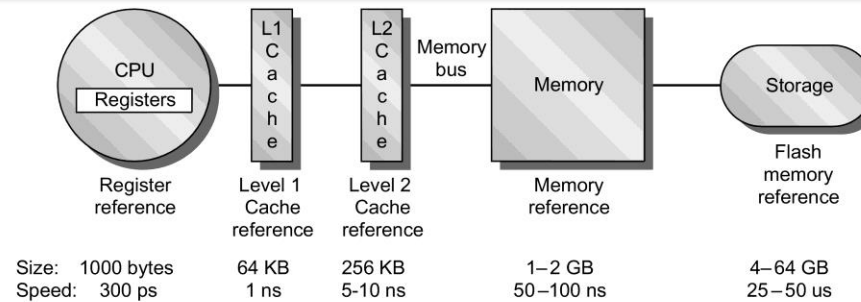
- Laptop or desktop

- Server

How It Works?

- **Block** (aka **line**): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - **Hit**: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - **Miss**: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 − hit ratio
  - Then accessed data supplied from upper level

Processor

L1

L2

Memory
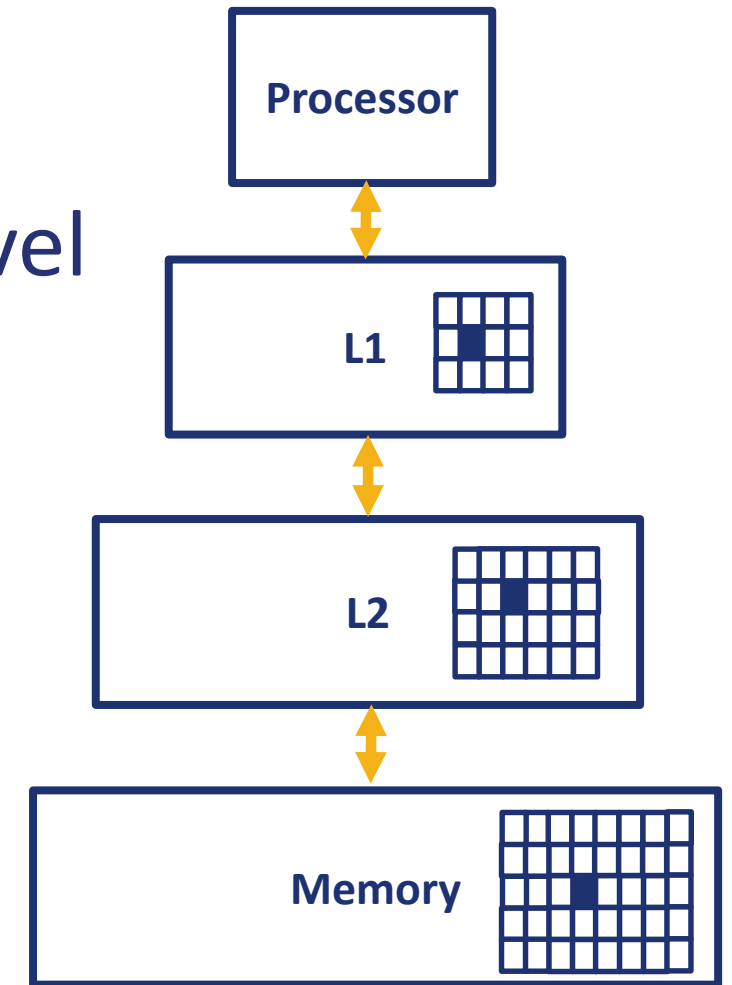
Hits and Misses

- On cache hit, CPU proceeds normally

- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Miss Types

- **Compulsory**: first time data accessed

- **Capacity**: cache too small to hold all data of interest

- **Conflict**: data of interest maps to a location in cache mapped to different data

Memory Performance

- **Hit**: data found in that level of memory hierarchy

- **Miss**: data not found (must go to next level)

  - **Hit Rate** = # hits / # memory accesses = 1 − Miss Rate

  - **Miss Rate** = # misses / # memory accesses = 1 − Hit Rate

- **Average memory access time (AMAT):** average time for processor to access data

  - **AMAT** = $t_{cache} + MR_{cache}[t_{MM} + MR_{MM}(t_{VM})]$

Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses X_1, ..., X_{n-1}, X_n

X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_3

X_4
X_1
X_{n-2}
X_{n-1}
X_2
X_n
X_3

a. Before the reference to X_n b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

Direct Mapped Cache

- Location determined by address

- Direct mapped: only one choice

  - (Block address) modulo (#Blocks in cache)

Cache

000 001 010 011 100 101 110 111

Memory

00001 00101 01001 01101 10001 10101 11001 11101
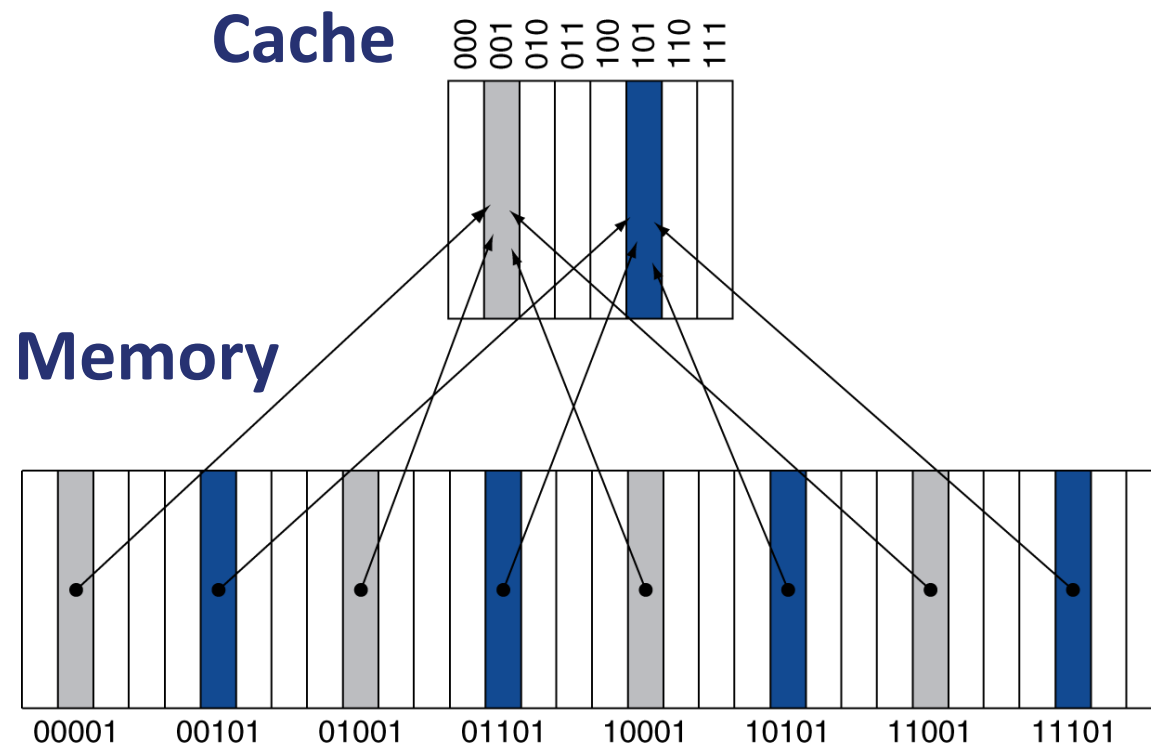
- #Blocks is a power of 2

- Use low-order address bits

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Direct Mapped Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

15

Direct Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Direct Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Direct Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Direct Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Associative Caches

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
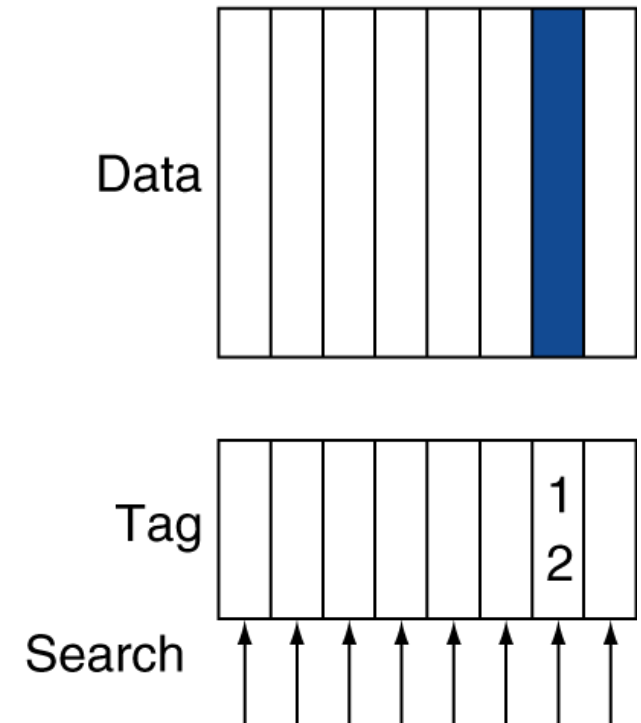 - *n* comparators (less expensive)

Associative Cache Examples
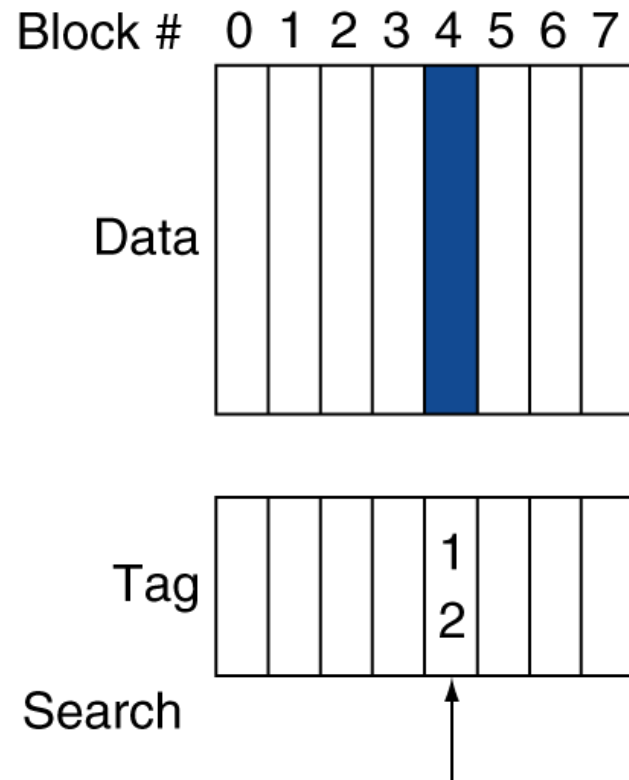
■ For a cache with 8 entries

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8

- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

- **2-way set associative**

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- **Fully associative**

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

25

How Much Associativity

- Increased associativity decreases miss rate

  - But with diminishing returns

- Simulation of a system with 64KB
  D-cache, 16-word blocks, SPEC2000

  - 1-way: 10.3%

  - 2-way: 8.6%

  - 4-way: 8.3%

  - 8-way: 8.1%

Replacement Policy

- Direct mapped
  - No choice

- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set

- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that

- Random
  - Gives approximately the same performance as LRU for high associativity

Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache

  - Keep track of whether each block is dirty

- When a dirty block is replaced

  - Write it back to memory

  - Can use a write buffer to allow replacing block to be read first

Write Allocation

- What should happen on a write miss?

- Alternatives for write-through

  - Allocate on miss: fetch the block

  - Write around: don't fetch the block

    - Since programs often write a whole block before reading it (e.g., initialization)

- For write-back

  - Usually fetch the block

Multilevel Caches

- Primary cache attached to CPU

  - Small, but fast

- Level-2 cache services misses from primary cache

  - Larger, slower, but still faster than main memory

- Main memory services L-2 cache misses

- Some high-end systems include L-3 cache

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\text{Memory Stall Cycles} = \frac{\text{Memory Accesses}}{\text{Program}} \times \text{Miss Rate} \times \text{Miss Penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instructions}} \times \text{Miss Penalty}$$

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: $0.02 \times 100 = 2$
  - D-cache: $0.36 \times 0.04 \times 100 = 1.44$
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster

Average Access Time

- Hit time is also important for performance

- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty

- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Overal Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can't neglect cache behavior when evaluating system performance

Matrix Multiplication

Loop order: i, j, k

```
for (int i= 0; i < n; i++) {
  for (int j= 0; j < n; j++) {
    for (int k= 0; k < n; k++) {
      C[i][j]+= A[i][k]*B[k][j];
    }
  }
}
```
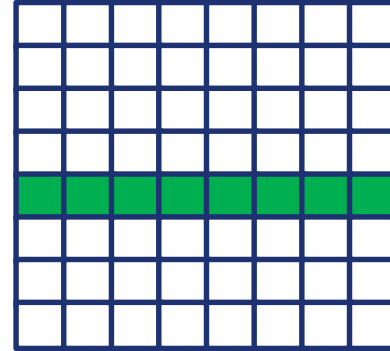
Running time:
13.714264 sec.
Performance:
~ 153 MFLOPS

Loop order: i, k, j

```
for (int i= 0; i < n; i++) {
  for (int k= 0; k < n; k++) {
    for (int j= 0; j < n; j++) {
      C[i][j]+= A[i][k]*B[k][j];
    }
  }
}
```

Running time:
2.739385 sec.
Performance:
~ 795 MFLOPS

Loop order: j, k, i

```
for (int j= 0; j < n; j++) {
  for (int k= 0; k < n; k++) {
    for (int i= 0; i < n; i++) {
      C[i][j]+= A[i][k]*B[k][j];
    }
  }
}
```

Running time:
19.074106 sec.
Performance:
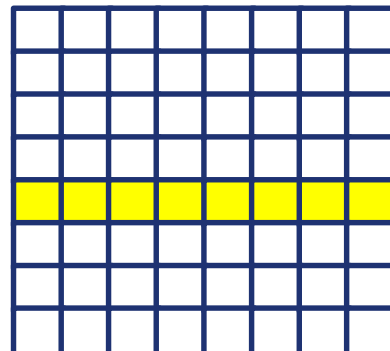~ 113 MFLOPS

Memory Access Patterns

Loop order: i, j, k
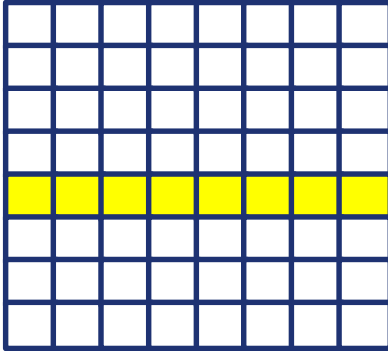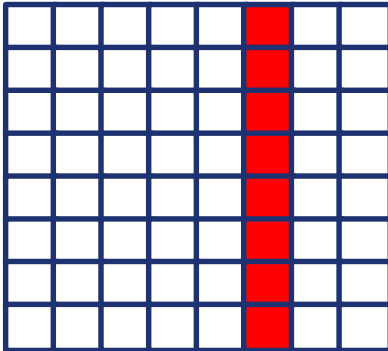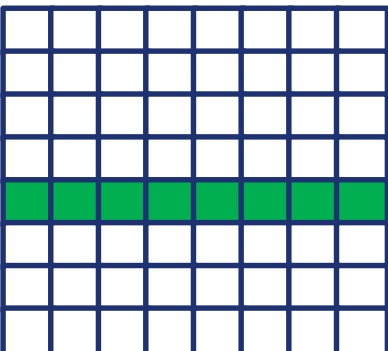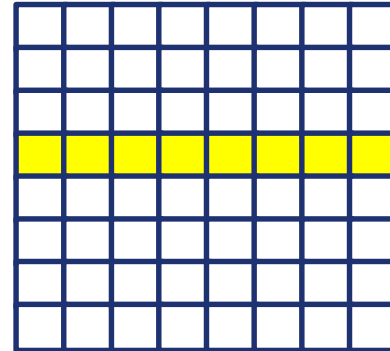
Loop order: i, k, j

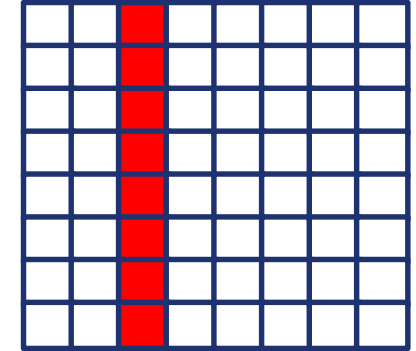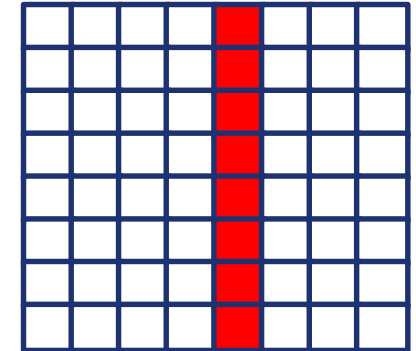Loop order: j, k, i

A

B

C

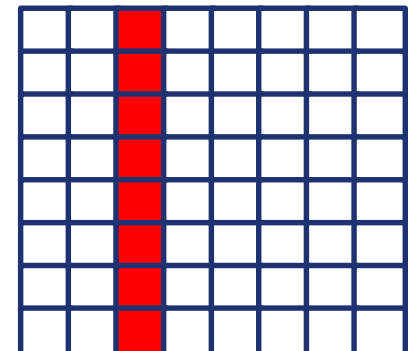# Any Questions?

```
               .text
__start:    addi t1, zero, 0x18
            addi t2, zero, 0x21
cycle:      beq t1, t2, done
            slt t0, t1, t2
            bne t0, zero, if_less
            nop
            sub t1, t1, t2
            j cycle
            nop
if_less:    sub t2, t2, t1
            j cycle
done:       add t3, t1, zero
```