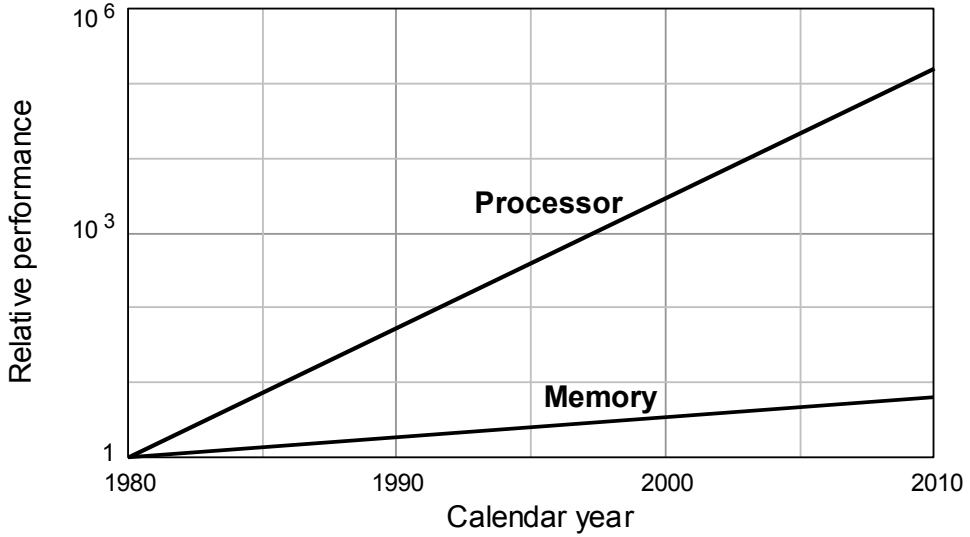
CS/ECE 3330
Computer Architecture

Chapter 5
Memory

Last Chapter

- **Focused exclusively on processor itself**
- **Made a lot of simplifying assumptions**

- **Reality: "The Memory Wall"**



Memory Technology

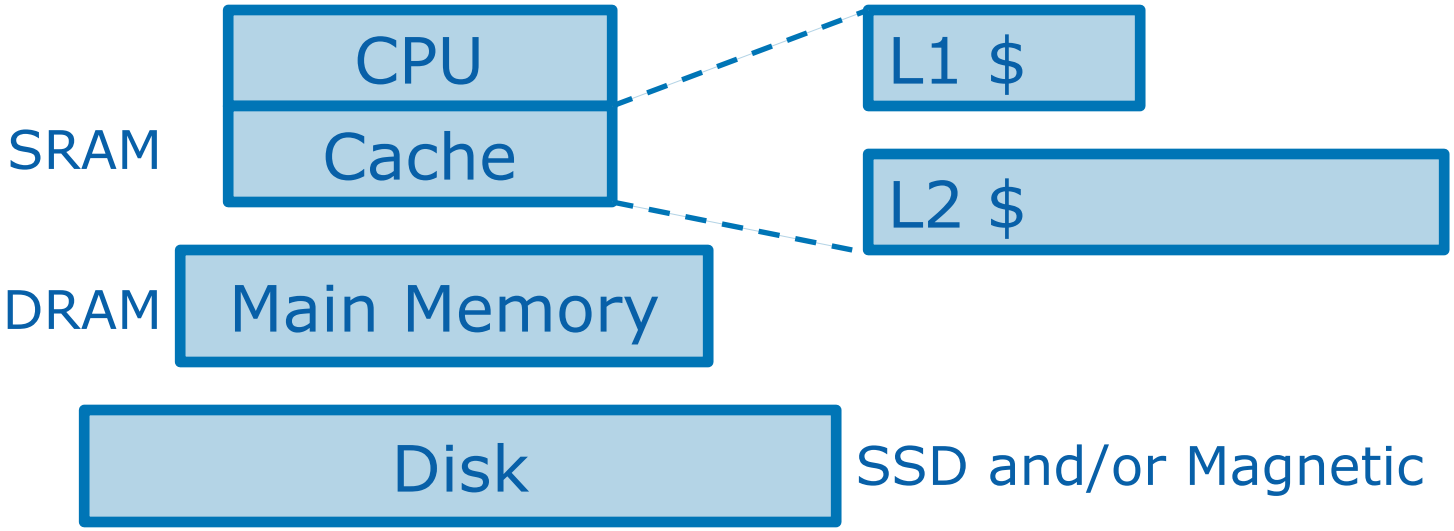
- **Static RAM (SRAM)**
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- **Dynamic RAM (DRAM)**
  - 50ns – 70ns, $20 – $75 per GB
- **Magnetic disk**
  - 5ms – 20ms, $0.20 – $2 per GB
- **Ideal memory**
  - Access time of SRAM
  - Capacity and cost/GB of disk

Principle of Locality

- **Programs access a small proportion of their address space at any time**
- **Temporal locality**
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- **Spatial locality**
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

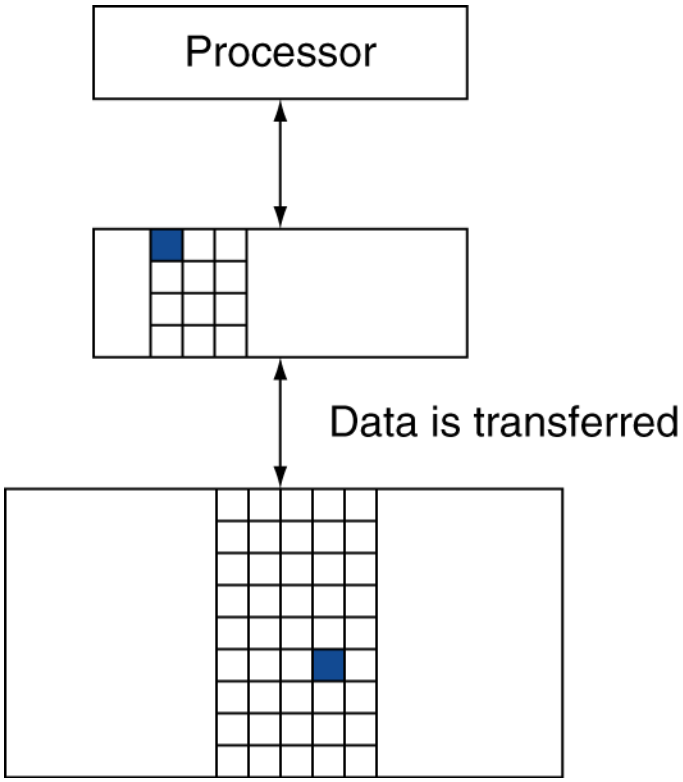
Taking Advantage of Locality

- **Memory hierarchy**



- **Store everything on disk**
- **Copy recently accessed (and nearby) items from disk to smaller DRAM memory**
- **Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory**

Memory Hierarchy Levels



Block (aka line): unit of copying

- May be multiple words

If accessed data is present in upper level

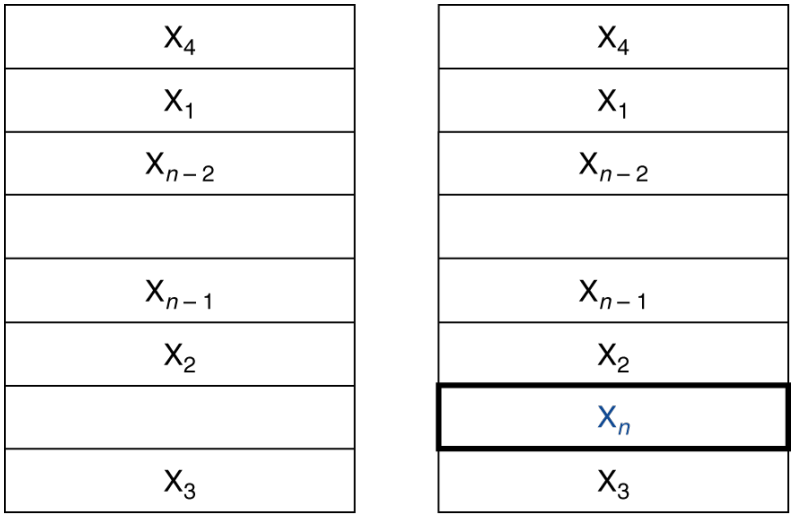
- Hit: access satisfied by upper level
  - Hit ratio: hits/accesses

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

Cache Memory

- The level of the memory hierarchy closest to the CPU
- Given accesses X1, …, Xn–1, Xn

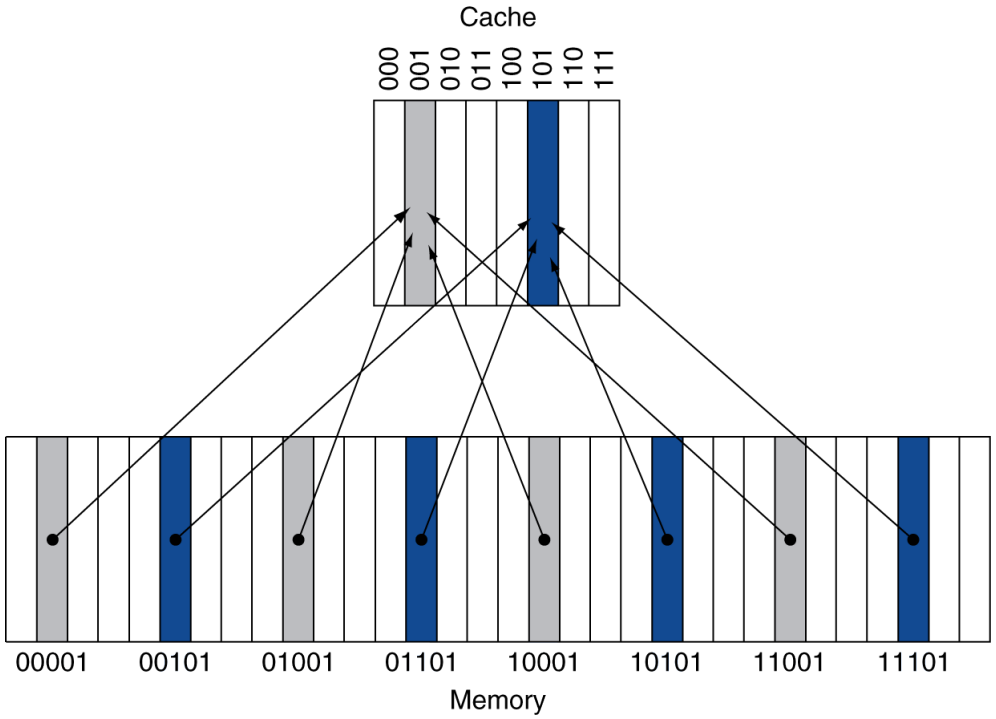


a. Before the reference to X_n b. After the reference to X_n

- 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)



- **#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

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		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10110		

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

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		

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		

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		

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

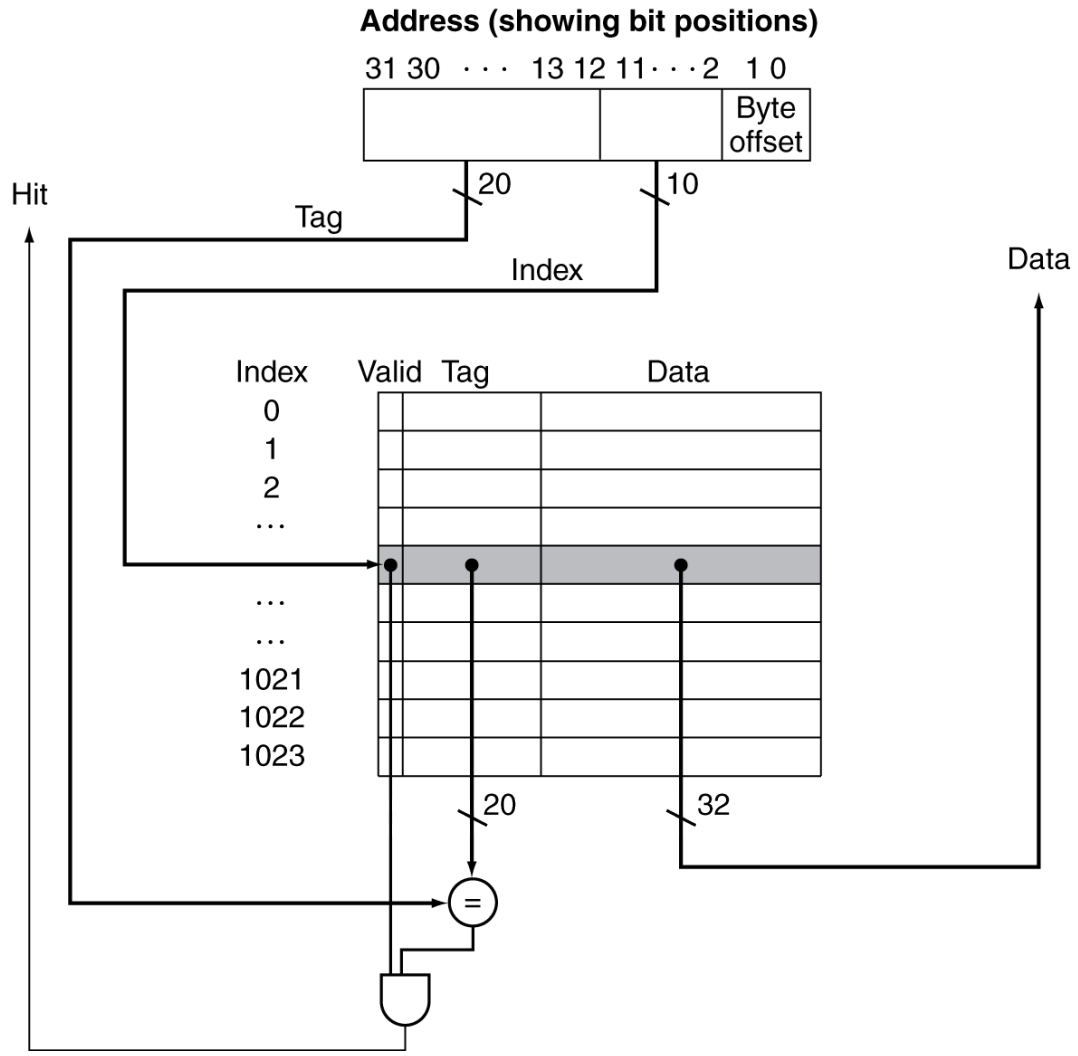
Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010
16	10 000	Hit	000

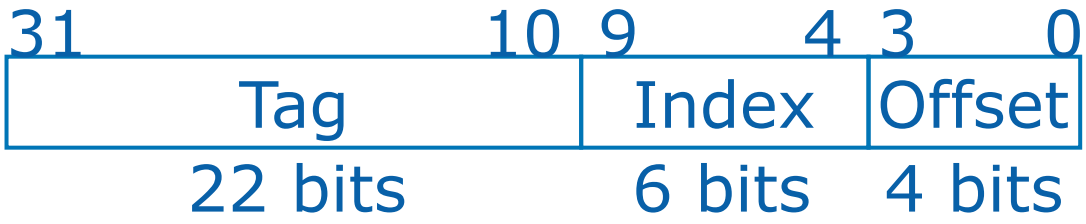
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision



Example: Larger Block Size

- **64 blocks, 16 bytes/block**
  - To what block number does address 1200 map?
- **Block address = ⌊1200/16⌋ = 75**
- **Block number = 75 modulo 64 = 11**



1024KB Cache

Block	Data Addr
0	0-15
1	16-31
…	…
10	160-175
11	**176-191**
12	192-207
…	…
62	992-1007
63	1008-1024

Block Size Considerations

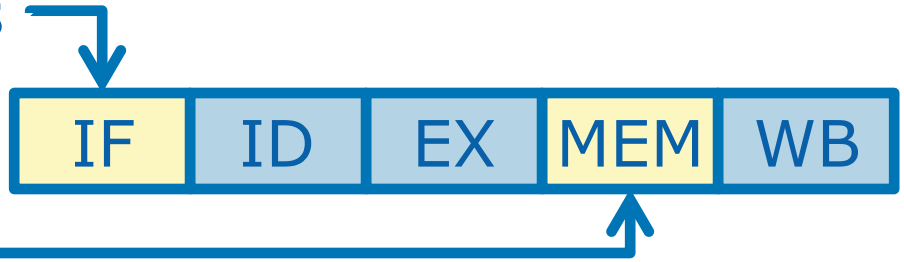
- **Larger blocks should reduce miss rate**
  - Due to spatial locality
- **But in a fixed-sized cache**
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution (i.e., loading data that won't be used by the computer)
- **Larger miss penalty**
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

Resume execution when the needed word is returned

Transfer requested word first, then remaining words

Cache 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



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

Measuring Cache Performance

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

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

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

Cache Performance Example

- **Given**
  - I-cache miss rate = 3%
  - D-cache miss rate = 7%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- **Miss cycles per instruction**
  - I-cache: _____
  - D-cache: _____
- **Actual CPI =** _____
  - Ideal CPU is _____ 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 = _____ns
    - _ cycles per instruction

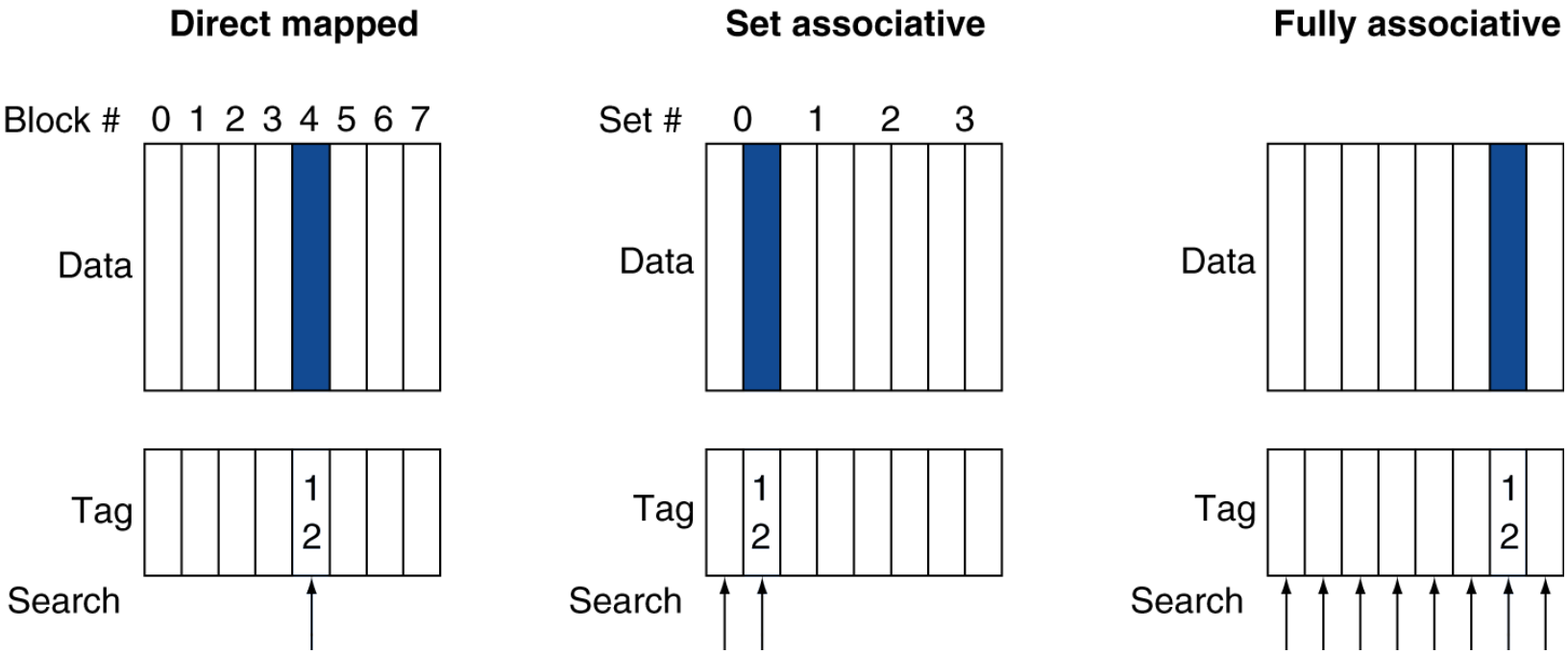
Performance Observations

- **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**

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 Example



Spectrum of Associativity

- 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]		

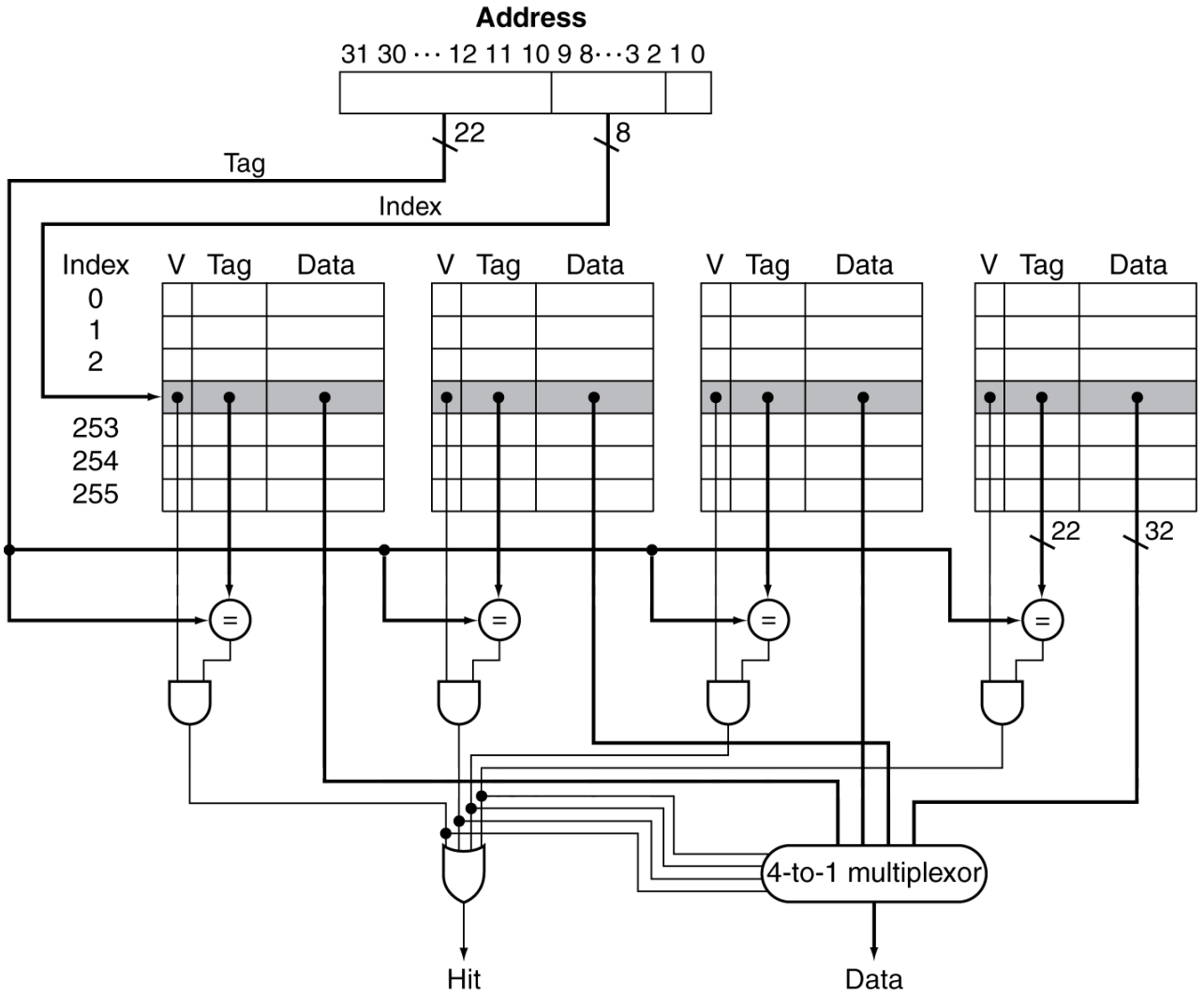
- **Fully associative**

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]	

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%

Set Associative Cache Organization



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

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**

Multilevel Cache Example

- **Given**
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- **With just primary cache**
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- **Now add L-2 cache**
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- **Primary miss with L-2 hit**
  - Penalty = 5ns/0.25ns = 20 cycles
- **Primary miss with L-2 miss**
  - Extra penalty = 400 cycles
- **CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4**
- **Performance ratio = 9/3.4 = 2.6**

Multilevel Cache Considerations

- **Primary cache**
  - Focus on minimal hit time
- **L-2 cache**
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- **Results**
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

Summary

- **Many options for cache configurations**
  - Words per block (cache line sizes)
  - Direct mapped, set associative, fully associative
  - Number of cache levels (L1, L2, etc.)
  - Replacement policies: LRU, FIFO, etc.
- **Can have significant performance impact**
  - (Know how to calculate it!)

Example Test Problems

- For the 16-bit memory address 0xcafe, indicate the tag, index, and block offset (in binary), given the assumptions shown. Assume a direct-mapped cache, and that 1 word = 4 bytes.
- (0xcafe = 1100 1010 1111 1110)

Assumptions	Tag	Index	Offset
32 cache entries 8 byte block			
256 cache entries 8 word block			

Example Test Problems

(0xcafe = 1100 1010 1111 1110)

Assumptions	Tag	Index	Offset
32 cache entries 8 byte block			

Example Test Problems

(0xcafe = 1100 1010 1111 1110)

Assumptions	Tag	Index	Offset
256 cache entries 3 word block			

Example Test Problems

- How many bits are required for a direct-mapped cache with 16KB of data and 4-word blocks, assuming a 32-bit address?
- 16KB = 4K (2^{12}) words.
- Block size of 4 (2^2) words, there are 1024 (2^{10}) blocks. Each block has 4x32 =128 bits of data (16 bytes), plus a tag, which is 32 – 10 (bits for block) – 4 (bits for offset), plus a valid bit.
- Total size: 2^{10} x (4 x 32 + (32 – 10 – 4) + 1) =
- 2^{10} x 147 = 147Kbits = 18.4KB (1.15x storage amount for the data)

Clarification on Caches

- **The "offset" for a cache entry seems to be a bit confusing.**
- **On page 462 in the textbook, the following cache is described:**
  - 32-bit byte addresses
  - Direct-mapped
  - The cache size is 2^n blocks, so n bits are used for the index
  - The block size is 2^m words (2^{m+2} bytes), so m bits are used for the **word** within the block, and two bits are used for the byte part of the address.

Clarification on Caches

- **(Repeated)**
  - The cache size is 2^n blocks, so n bits are used for the index
  - The block size is 2^m words (2^{m+2} bytes), so m bits are used for the **word** within the block, and two bits are used for the byte part of the address.
- **The book continues and says**
  - The size of the tag field is $32 - (n + m + 2)$
- **The key is noticing that the book's definition for *m* refers to a word (4 bytes), not a byte (and thus, that 2 extra bits are necessary for te byte offset)**

Clarification on Caches

- **(Repeated)**
  - The cache size is 2^n blocks, so n bits are used for the index
  - The block size is 2^m words (2^{m+2} bytes), so m bits are used for the **word** within the block, and two bits are used for the byte part of the address.
- **I find it easier to refer to the offset as a byte reference (in red above). E.g.:**
  - If the block size is given in words, change that to bytes by multiplying by 4. Then use that power of two to calculate the offset: e.g., 4-word blocks = 16 byte blocks, so the offset needs $\log_2(16)$=**4** bits.

Clarification on Caches

- **Example:**
  - Assume a direct-mapped cache that uses 32-bit addresses, has 512 blocks, and each block is 8 words:

  - Offset: $\log_2$(8 words * 4 bytes/words) =
    =$\log_2$(32) = 5 bits
  - Index: $\log_2$(512) = 9 bits
  - Tag: Remaining bits: 32 – 5 – 9 = 18 bits:

```
1001 0010 1111 1101 1111 0011 1010 0000
--------Tag----------------|---Index----| -Offset-
```

Homework Example (5.3.1-3, data (b))

- 32-bit memory addresses, given as word addresses:

Fill in the table, Direct Mapped cache with 16 1-*word* blocks.

Word Addr	Binary Address	Tag	Index	Hit/ Miss
6	00000110	0000	0110	M
214	11010110	1101	0110	M
175	10101111	1010	1111	M
214	11010110	1101	0110	H
6	00000110	0000	0110	M
84	01010100	0101	0100	M
65	01000001	0100	0001	M
174	10101110	1010	1110	M
64	01000000	0100	0000	M
105	01101001	0110	1001	M
85	01010101	0101	0101	M
215	11010111	1101	0111	M

Homework Example (5.3.1-3, data (b))

- 32-bit memory addresses, given as word addresses:

Fill in the table, Direct Mapped cache with 8 2-*word* blocks.

Word Addr	Binary Address	Tag	Index	Hit/ Miss
6	00000110	0000	011	M
214	11010110	1101	011	M
175	10101111	1010	111	M
214	11010110	1101	011	H
6	00000110	0000	011	M
84	01010100	0101	010	M
65	01000001	0100	000	M
174	10101110	1010	111	H
64	01000000	0100	000	H
105	01101001	0110	100	M
85	01010101	0101	010	H
215	11010111	1101	011	M

Virtual Memory

Consider a typical Netbook

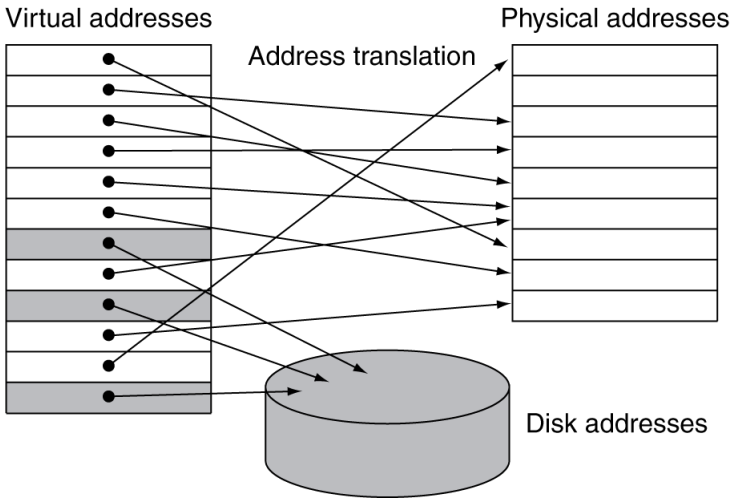
- **I have 1 GB RAM**
  - What happens if my application needs 2 GB?
  - What happens if I have three applications running, each requiring 1 GB?
  - What if I get a new netbook with 2 GB RAM, do I need to rewrite my applications?
- **I usually run multiple applications at once**
  - Do they need to access disjoint memory locations?
  - Can I write a virus that overwrites random memory locations to mess up other programs?

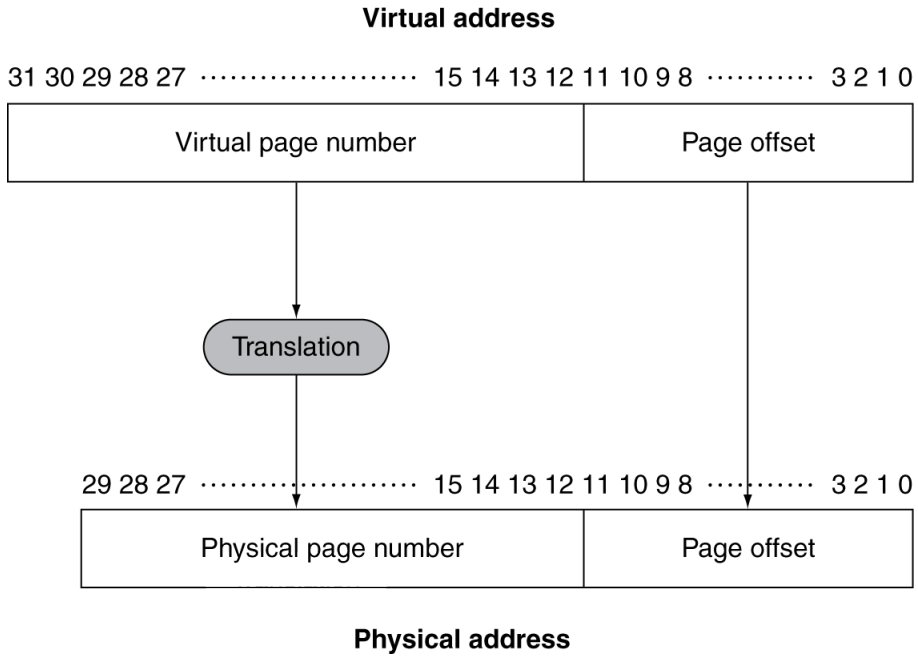
Virtual Memory

- **Use main memory as a "cache" for secondary (disk) storage**
  - Managed jointly by CPU hardware and the operating system (OS)
- **Programs share main memory**
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- **CPU and OS translate virtual addresses to physical addresses**
  - VM "block" is called a page
  - VM translation "miss" is called a page fault

Address Translation

- **Fixed-size pages (e.g., 4K)**





Page Fault Penalty

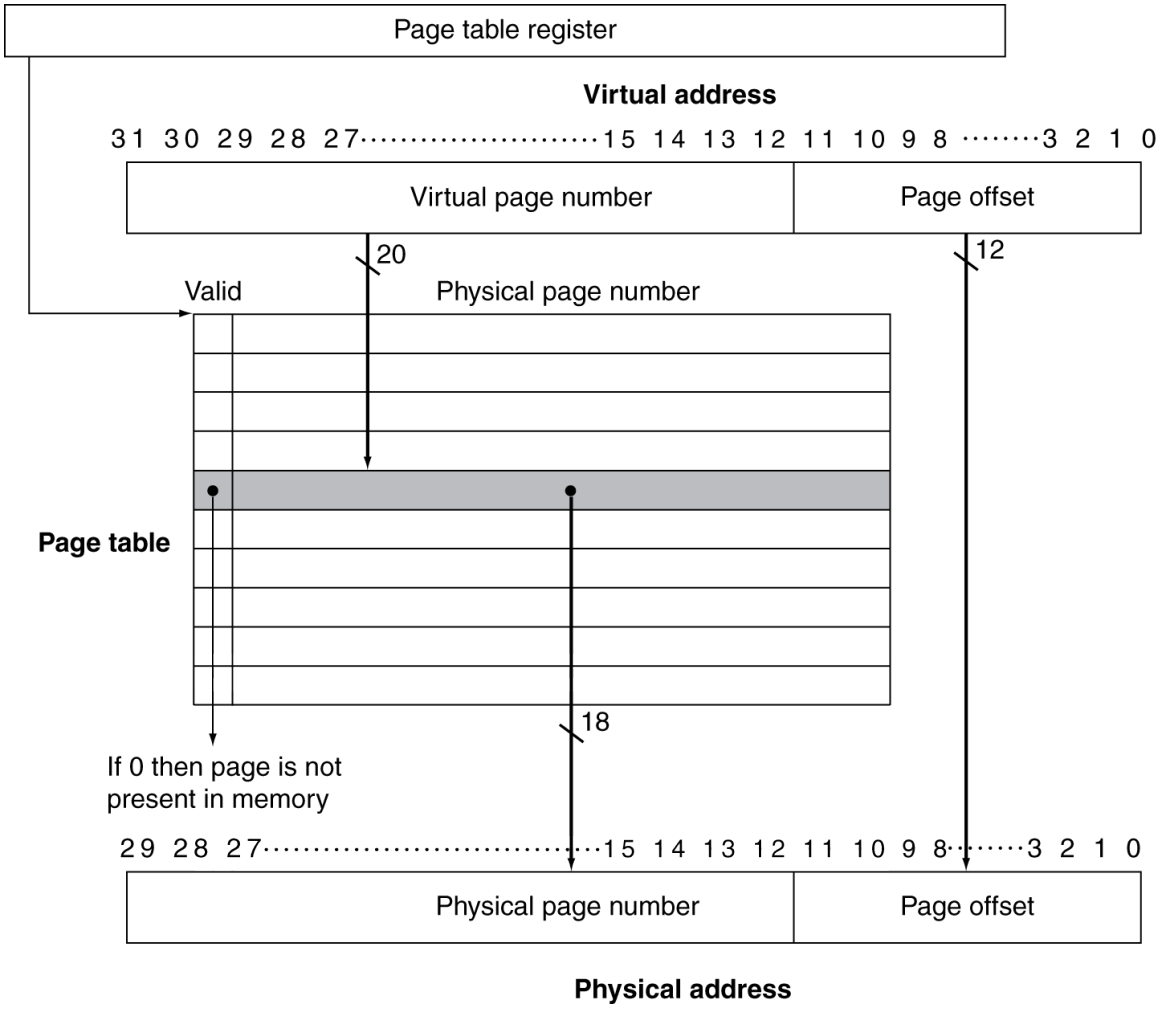
- **On page fault, the page must be fetched from disk**
  - Takes millions of clock cycles
  - Handled by OS code
- **Try to minimize page fault rate**
  - Fully associative placement
  - Smart replacement algorithms

Page Tables

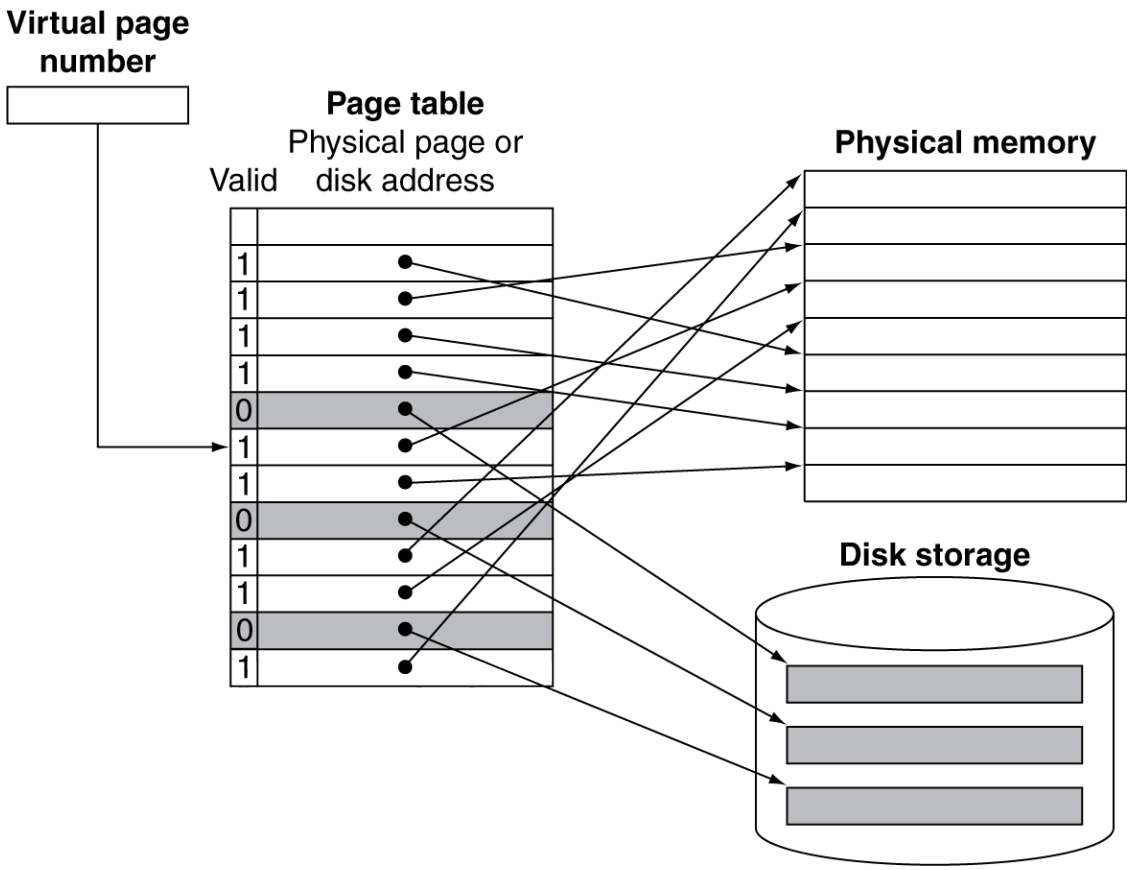
PTE: page table

- **Stores placement information**
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- **If page is present in memory**
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- **If page is not present**
  - PTE can refer to location in swap space on disk

Translation Using a Page Table



Mapping Pages to Storage



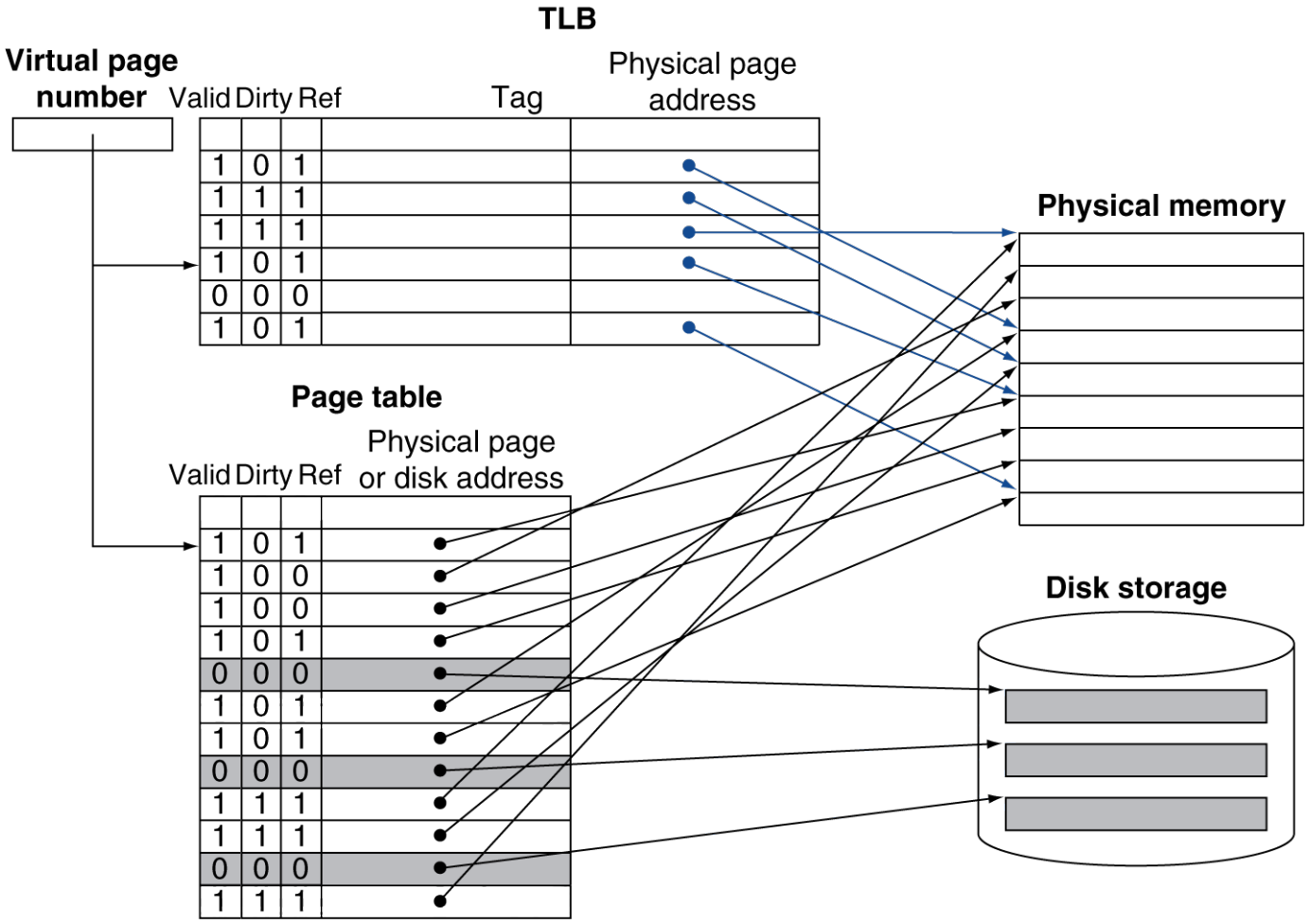
Replacement and Writes

- **To reduce page fault rate, prefer least-recently used (LRU) replacement**
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- **Disk writes take millions of cycles**
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

Fast Translation Using a TLB

- **Address translation would appear to require extra memory references**
  - One to access the PTE
  - Then the actual memory access
- **But access to page tables has good locality**
  - So use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

Fast Translation Using a TLB



TLB Misses

- **If page is in memory**
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- **If page is not in memory (page fault)**
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

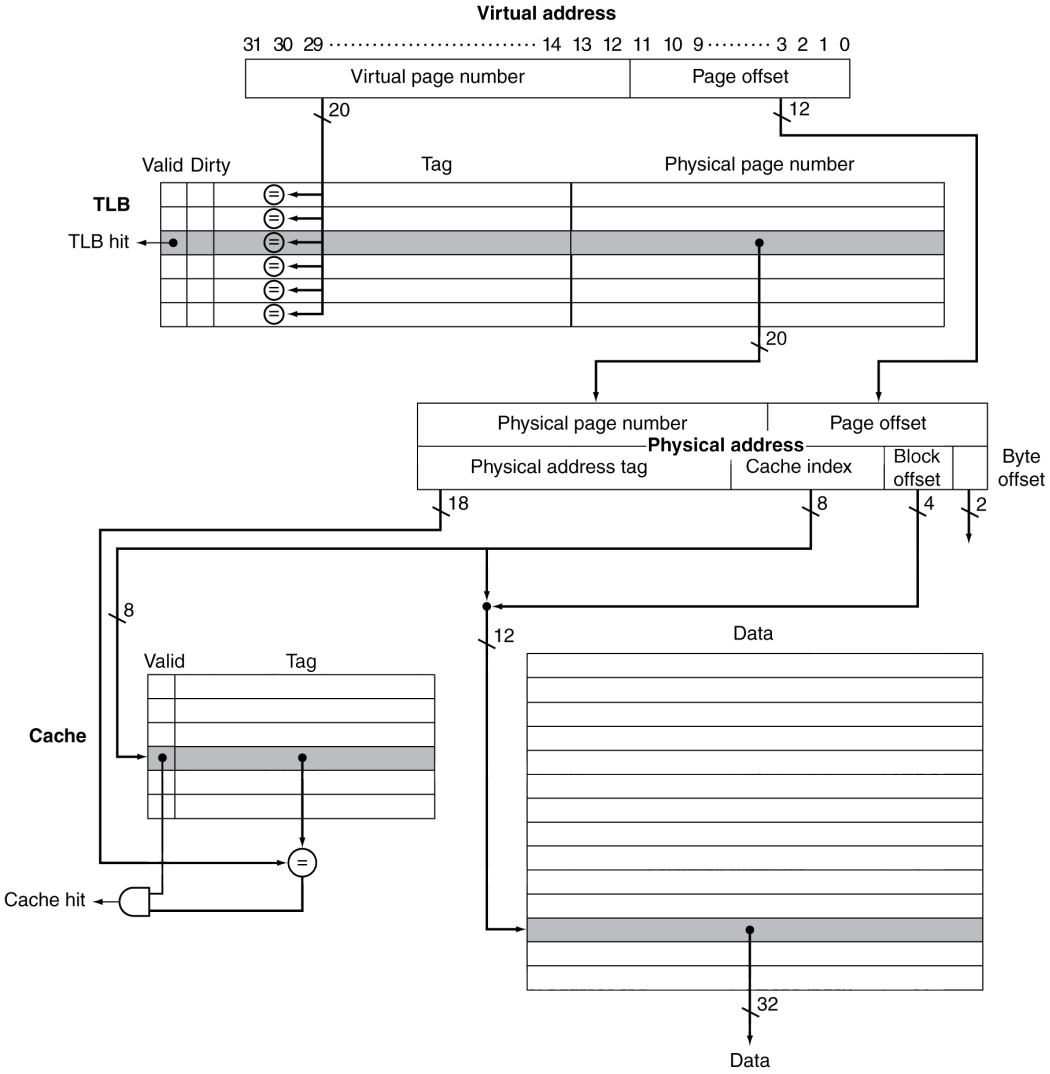
TLB Miss Handler

- **TLB miss indicates**
  - Page present, but PTE not in TLB
  - Page not preset
- **Must recognize TLB miss before destination register overwritten**
  - Raise exception
- **Handler copies PTE from memory to TLB**
  - Then restarts instruction
  - If page not present, page fault will occur

Page Fault Handler

- **Use faulting virtual address to find PTE**
- **Locate page on disk**
- **Choose page to replace**
  - If dirty, write to disk first
- **Read page into memory and update page table**
- **Make process runnable again**
  - Restart from faulting instruction

TLB and Cache Interaction



If cache tag uses physical address

- Need to translate before cache lookup

Alternative: use virtual address tag

- Complications due to aliasing
  - Different virtual addresses for shared physical address

Memory Protection

- **Different tasks can share parts of their virtual address spaces**
  - But need to protect against errant access
  - Requires OS assistance
- **Hardware support for OS protection**
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS)

Practice Test-like Question

- **For the virtual address 0xFA28, calculate the page number/offset, assuming the page sizes shown.**

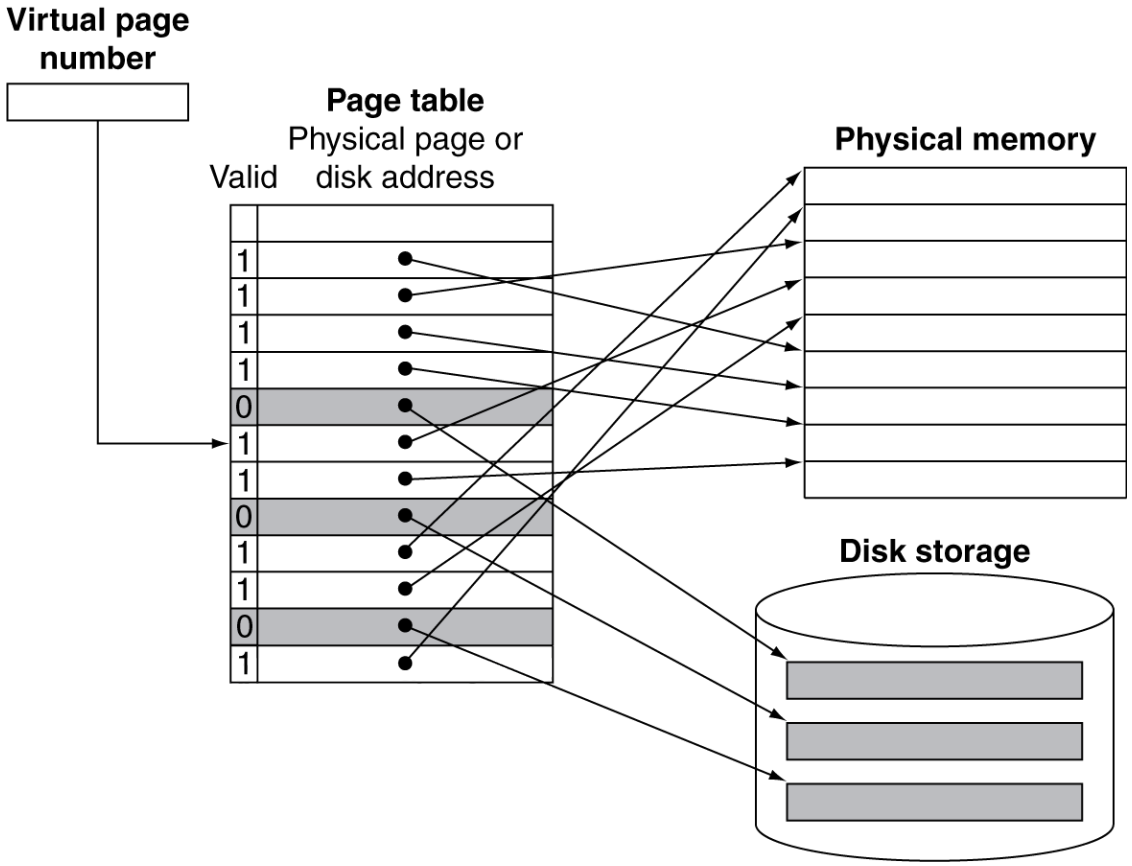
Page Size	Page Number	Offset
1 KB		
4 KB		
8 KB		

Memory Systems

CS/ECE 3330

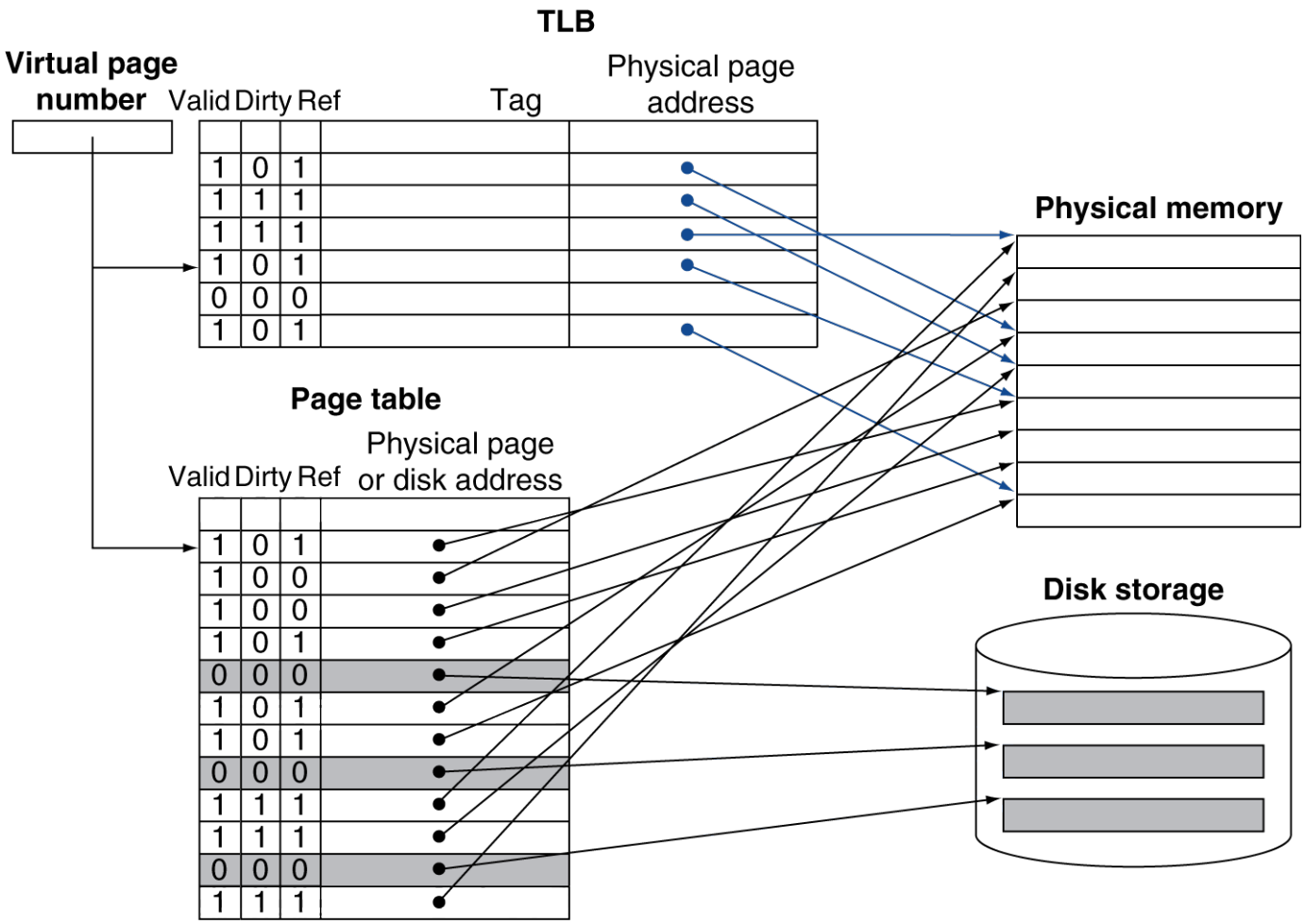
Last Time: Virtual Memory

- Illusion of unbounded memory
- Protection and isolation



Last Time: The TLB

- Hardware-based "cache" of the page table



The Memory Hierarchy

The BIG Picture

- **Common principles apply at all levels of the memory hierarchy**
  - Based on notions of caching
- **At each level in the hierarchy**
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

Block Placement

- **Determined by associativity**
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- **Higher associativity reduces miss rate**
  - Increases complexity, cost, and access time

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- **Hardware caches**
  - Reduce comparisons to reduce cost
- **Virtual memory**
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Replacement

- **Choice of entry to replace on a miss**
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- **Virtual memory**
  - LRU approximation with hardware support

Write Policy

- **Write-through**
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- **Write-back**
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- **Virtual memory**
  - Only write-back is feasible, given disk write latency

Sources of Misses (the "3 Cs")

- **Compulsory misses (aka cold start misses)**
  - First access to a block
- **Capacity misses**
  - Due to finite cache size
  - A replaced block is later accessed again
- **Conflict misses (aka collision misses)**
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Cache Design Trade-offs

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Miss Penalty Reduction

- **Return requested word first**
  - Then back-fill rest of block
- **Non-blocking miss processing**
  - Hit under miss: allow hits to proceed
  - Miss under miss: allow multiple outstanding misses
- **Hardware prefetch: instructions and data**
- **Opteron X4: bank interleaved L1 D-cache**
  - Two concurrent accesses per cycle

Can the programmer improve a program's cache hit rate?

- Memory access patterns can be influenced by the programmer.
- Example: In C, arrays are stored in "row-major" order, and in Fortran arrays are stored in "column-major" order:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- In C (row-order): In memory: 1,2,3,4,5,6
- In Fortran (column-order): In memory:

  1,4,2,5,3,6

Access memory such that the cache is filled with data you will use next

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- In C: 1,2,3,4,5,6
- A C-program that accesses rows and then columns will have more cache hits (for a bigger data set):

```
for (i = 0; i < 2; i++)
    for (j = 0; j < 3; j++)
        printf("%d\n", A[i][j]);
```

- The first row will be loaded into the cache on the first access, and the next accesses will be cache hits.

Access memory such that the cache is filled with data you will use next

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- In C: 1,2,3,4,5,6
- A C-program that accesses columns and then rows will have more cache *misses* (for a bigger data set):

```
for (j = 0; j < 3; j++)
    for (i = 0; i < 2; i++)
        printf("%d\n", A[i][j]);
```

- The first row will be loaded into the cache on the first access, but the second row will be needed immediately. The cache blocks will be replaced frequently, leading to more misses.

Example: Matrix Multiply (MATSIZE=768)

$$AB = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

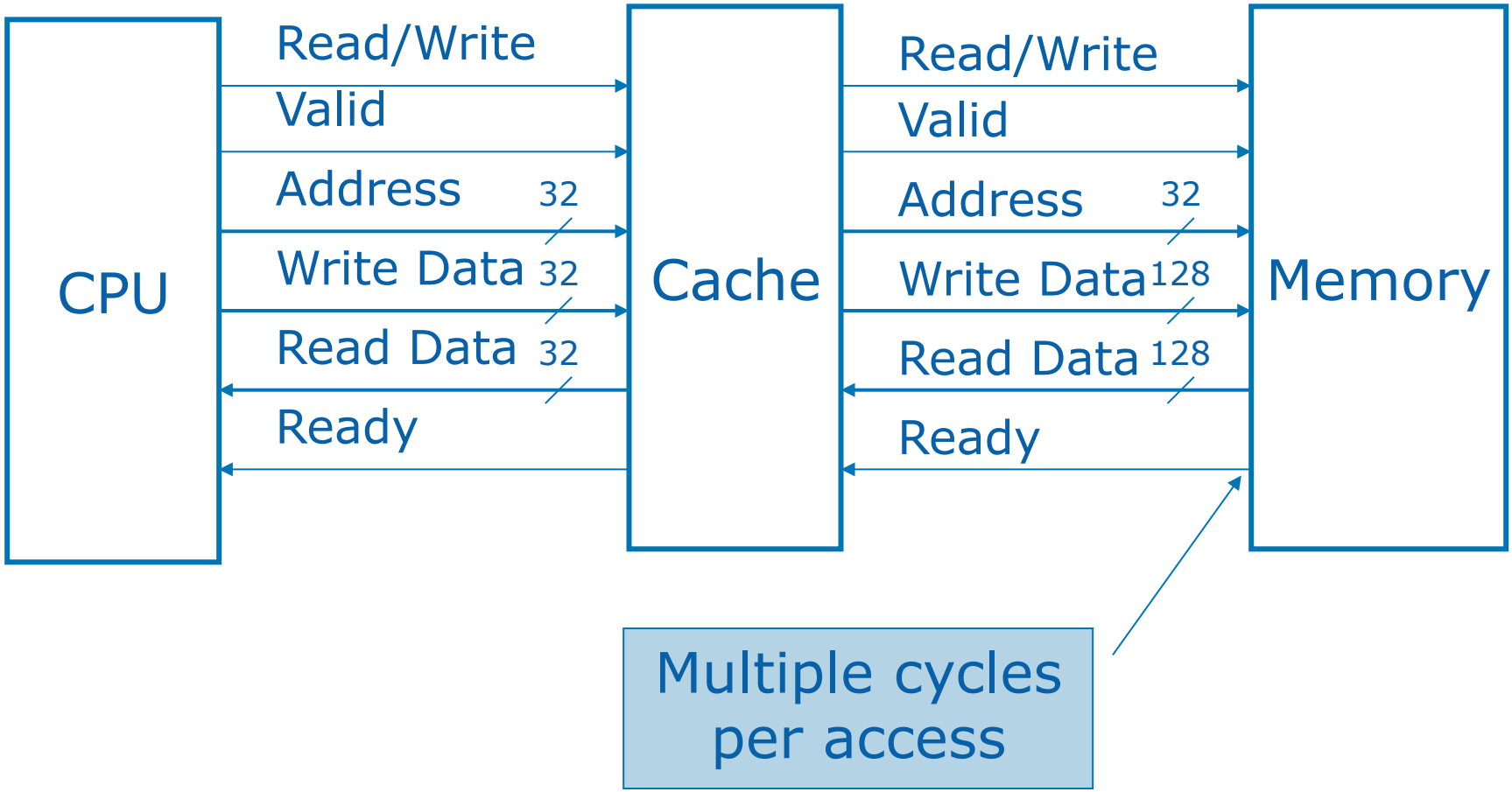
- **Program 1: Time =**

```
for(i=0;i<MATSIZE;i++)
    for(j=0;j<MATSIZE;j++)
        for(k=0;k<MATSIZE;k++)
            C[i][j] += A[i][k] * B[k][j];
```

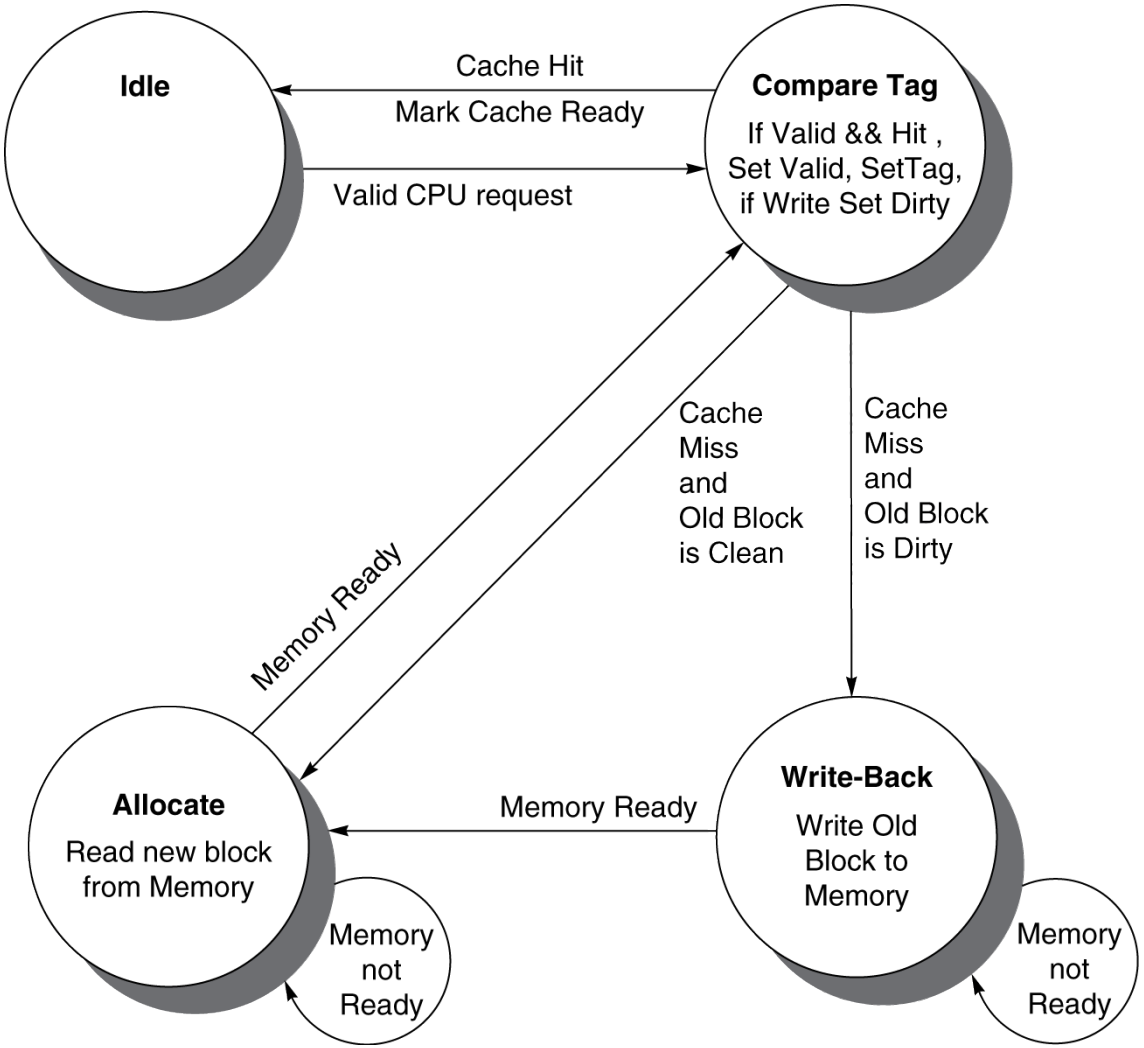
- **Program 2: Time =**

```
for(j=0;j<MATSIZE;j++)
    for(j=0;j<MATSIZE;j++)
        for(i=0;i<MATSIZE;i++)
            C[i][j] += A[i][k] * B[k][j];
```

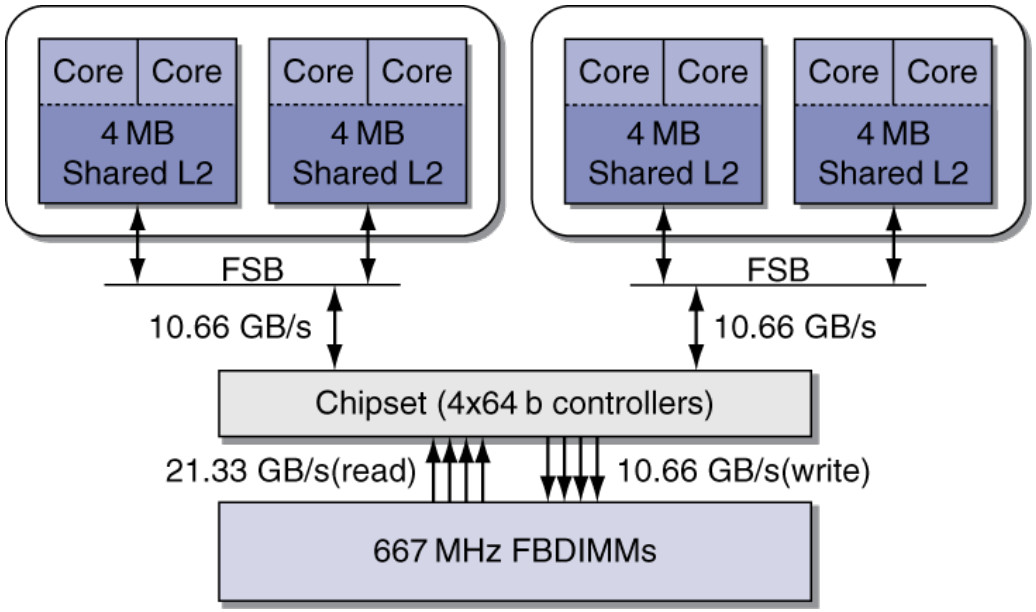
Cache Control: Interface Signals



Cache Controller FSM



Consider a Multicore Processor



Cache Coherence Problem

- **Suppose two CPU cores share a physical address space**
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Cache Coherence Protocols

- **Operations performed by caches in multiprocessors to ensure coherence**
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- **Snooping protocols**
  - Each cache monitors bus reads/writes
- **Directory-based protocols**
  - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- **Cache gets exclusive access to a block when it is to be written**
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

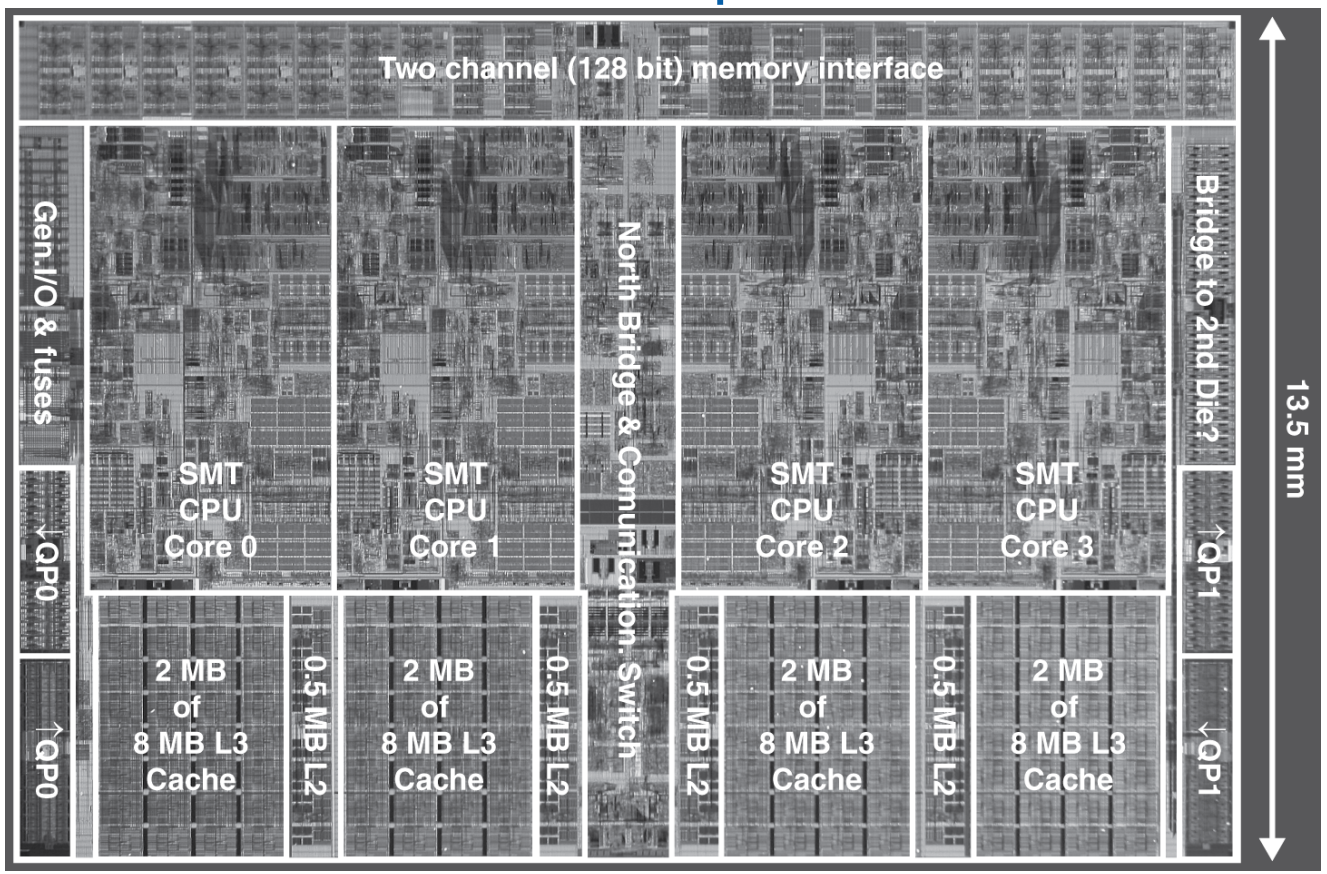
CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Memory Consistency

- **When are writes seen by other processors?**
  - "Seen" means a read returns the written value
  - Can't be instantaneous
- **Assumptions**
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- **Consequence**
  - P writes X then writes Y
    ⇒ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

Multilevel On-Chip Caches

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache.
8MB of L3 is shared between all four cores.

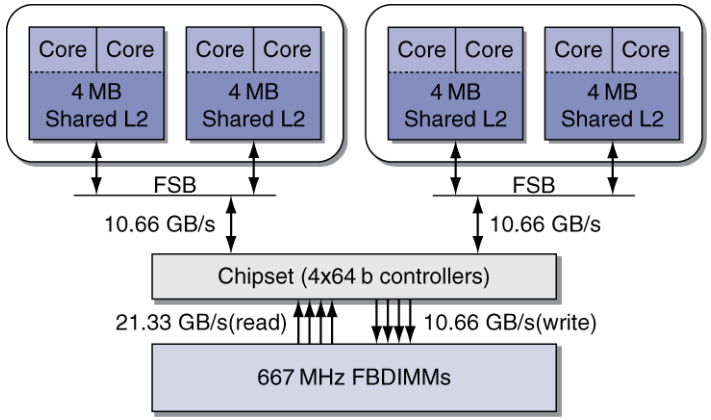
2-Level TLB Organization

	Intel Nehalem	AMD Opteron X4
Virtual addr	48 bits	48 bits
Physical addr	44 bits	48 bits
Page size	4KB, 2/4MB	4KB, 2/4MB
L1 TLB (per core)	L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both 4-way, LRU replacement	L1 I-TLB: 48 entries L1 D-TLB: 48 entries Both fully associative, LRU replacement
L2 TLB (per core)	Single L2 TLB: 512 entries 4-way, LRU replacement	L2 I-TLB: 512 entries L2 D-TLB: 512 entries Both 4-way, round-robin LRU
TLB misses	Handled in hardware	Handled in hardware

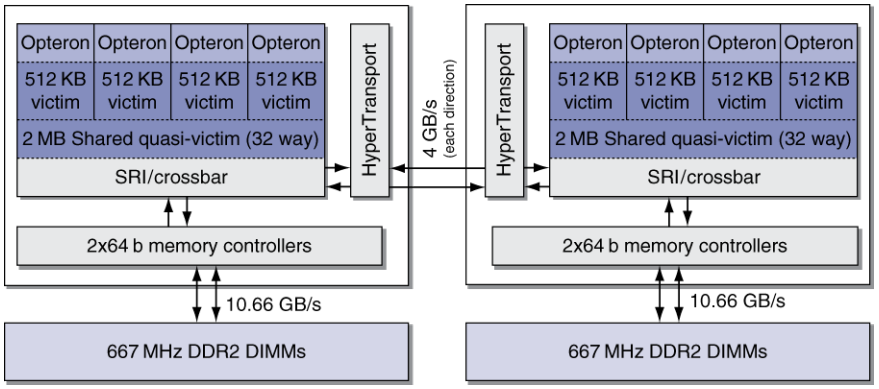
3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/ allocate	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/ allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/ allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

More Multicore Architecture Examples

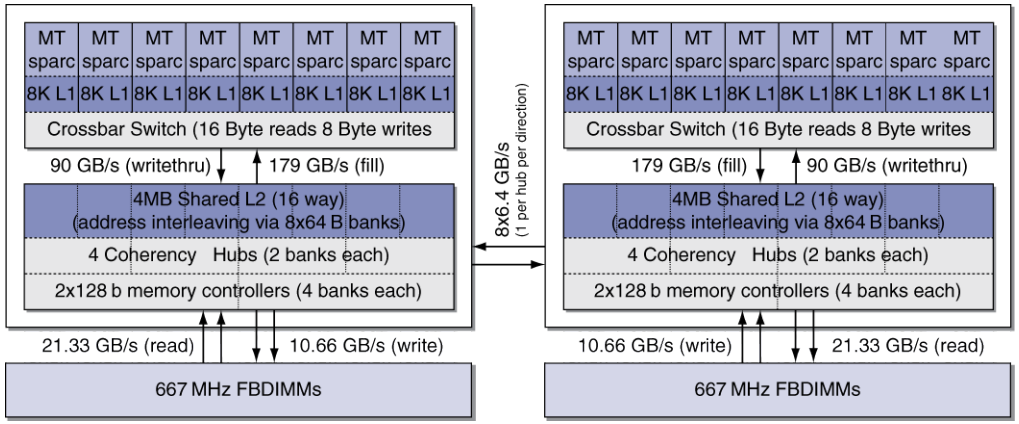


2 × quad-core
Intel Xeon e5345
(Clovertown)

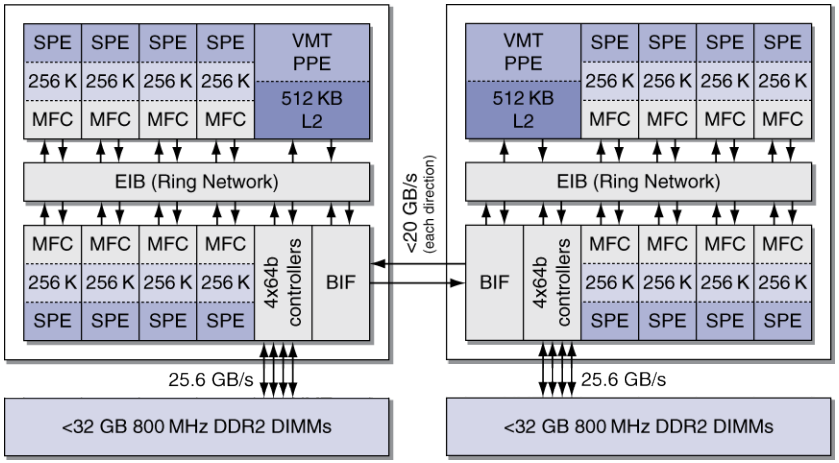


2 × quad-core
AMD Opteron X4 2356
(Barcelona)

More Multicore Architecture Examples



2 × oct-core
Sun UltraSPARC
T2 5140 (Niagara 2)



2 × oct-core
IBM Cell QS20

Pitfalls

- **Byte vs. word addressing**
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4
- **Ignoring memory system effects when writing or generating code**
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality

Concluding Remarks

- **Fast memories are small, large memories are slow**
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺
- **Principle of locality**
  - Programs use a small part of their memory space frequently
- **Memory hierarchy**
  - L1 cache ⇔ L2 cache ⇔ … ⇔ DRAM memory ⇔ disk
- **Memory system design is critical for multiprocessors**

The "Other" VM: Virtual Machines

Virtual Machines

- **Host computer emulates guest operating system and machine resources**
  - Improved isolation of multiple guests
  - Avoids security and reliability problems
  - Aids sharing of resources
- **Virtualization has some performance impact**
  - Feasible w/ modern high-performance computers
- **Examples**
  - IBM VM/370 (1970s technology!)
  - VMware
  - Microsoft Virtual PC

Virtual Machine Monitor

- **Maps virtual resources to physical resources**
  - Memory, I/O devices, CPUs
- **Guest code runs on native machine in user mode**
  - Traps to VMM on privileged instructions and access to protected resources
- **Guest OS may be different from host OS**
- **VMM handles real I/O devices**
  - Emulates generic virtual I/O devices for guest

Example: Timer Virtualization

- **In native machine, on timer interrupt**
  - OS suspends current process, handles interrupt, selects and resumes next process
- **With Virtual Machine Monitor**
  - VMM suspends current VM, handles interrupt, selects and resumes next VM
- **If a VM requires timer interrupts**
  - VMM emulates a virtual timer
  - Emulates interrupt for VM when physical timer interrupt occurs

Instruction Set Support

- **User and System modes**
- **Privileged instructions only available in system mode**
  - Trap to system if executed in user mode
- **All physical resources only accessible using privileged instructions**
  - Including page tables, interrupt controls, I/O registers
- **Renaissance of virtualization support**
  - Current ISAs (e.g., x86) adapting