Computer Organization and Design (SPring 2023)

Large and Fast: Exploiting Memory Hierarchy

Yujuan Tan

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
- Flash Memory
  - 0.1 – 2 ms, $5 – 50 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

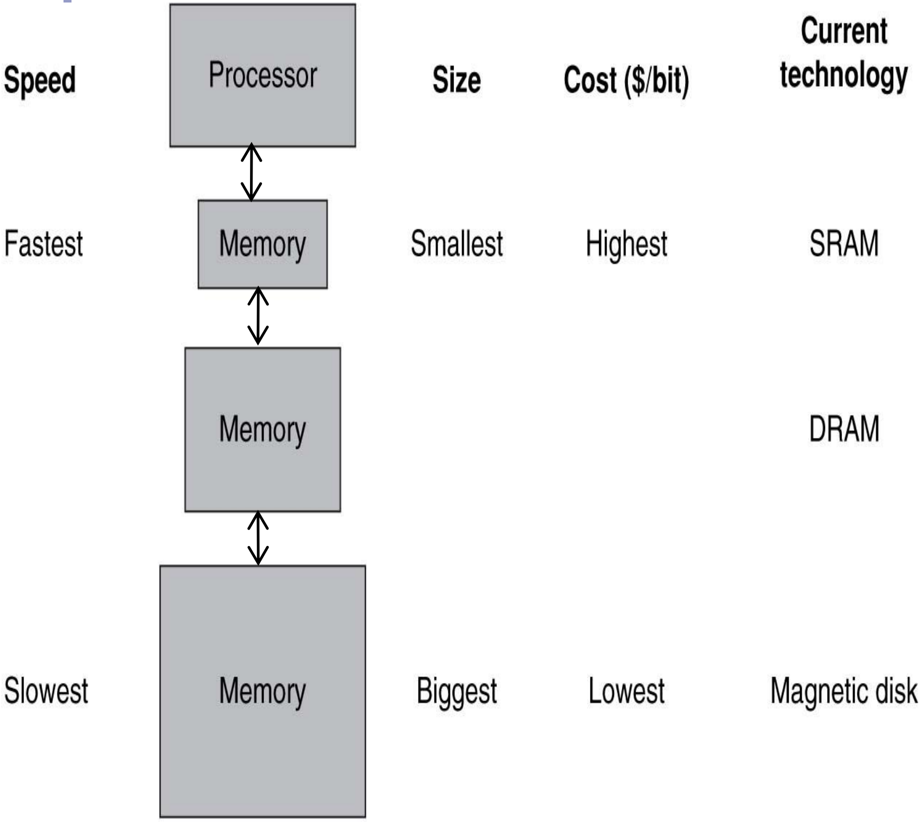
Principle of Locality

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

Memory Hierarchy Levels

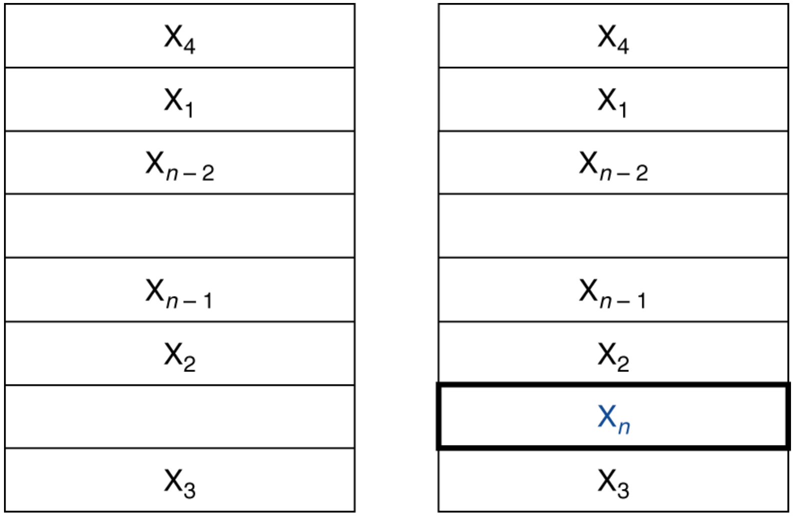


- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses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

§5.2 The Basics of Caches

Cache Memory

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

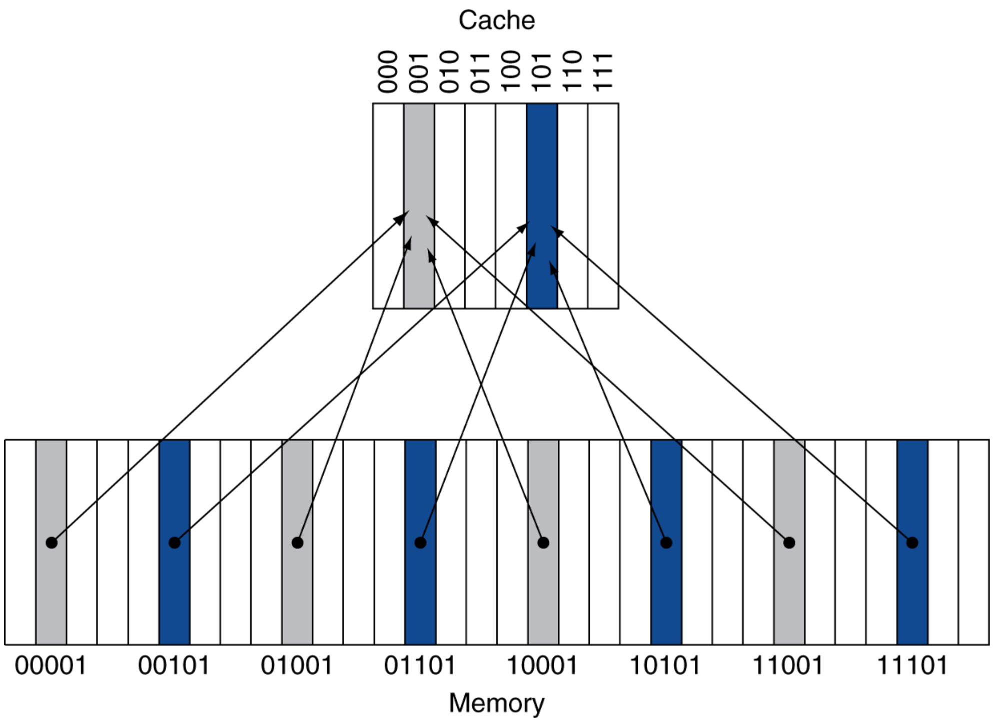


a. Before the reference to X_n b. After the reference to X_n

- 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 for block addr

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

Block address (lower order address bits)

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

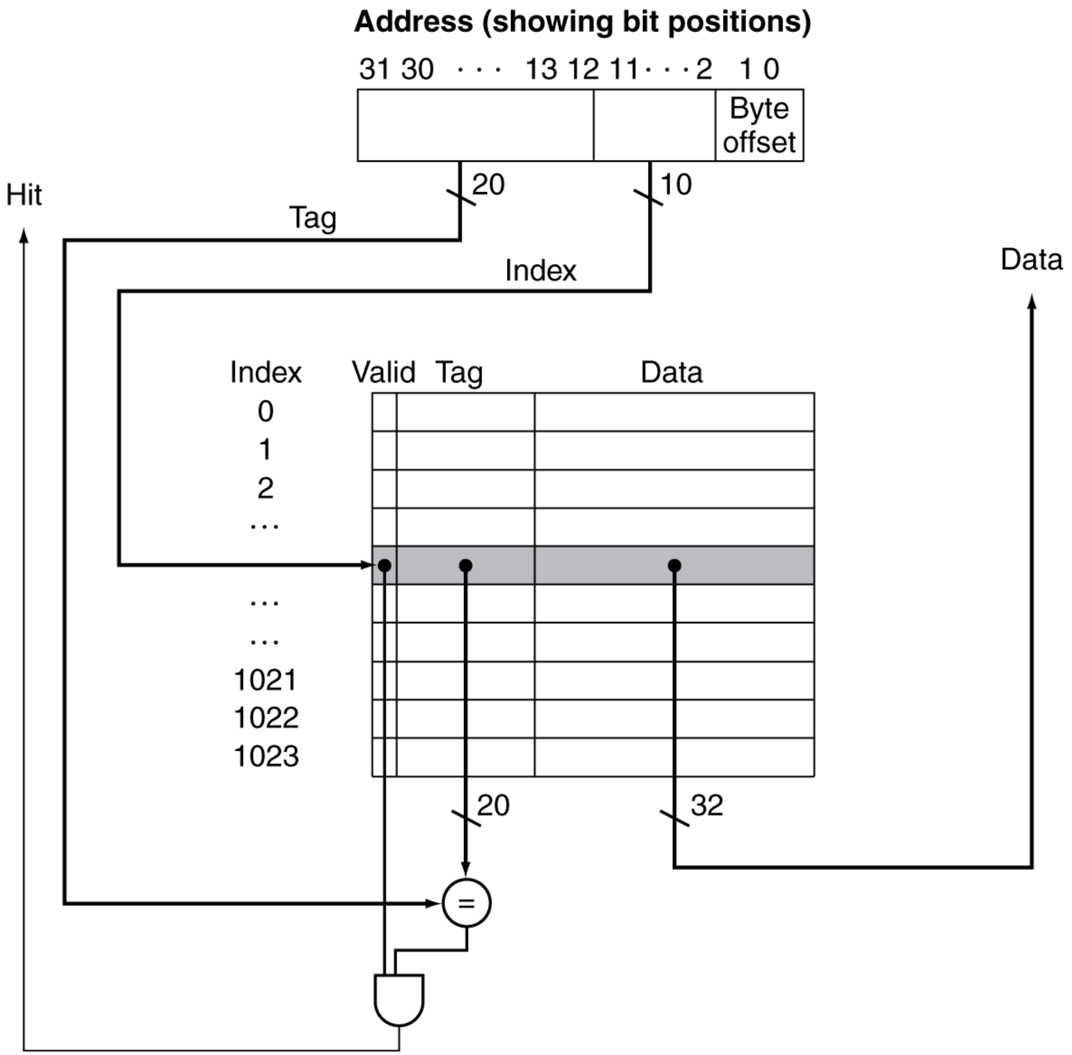
Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

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

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



Cache Field Sizes

- The number of bits in a cache includes both the storage for data and for the tags
  - 32-bit byte address
  - For a direct mapped cache with 2^n blocks, *n* bits are used for the index
  - For a block size of 2^m words (2^{m+2} bytes), *m* bits are used to address the word within the block and 2 bits are used to address the byte within the word
- What is the size of the tag field?
  - 32 – (n+m+2)
- The total number of bits in a direct-mapped cache is then

$$2^n \text{ x (block size + tag field size + valid field size)}$$

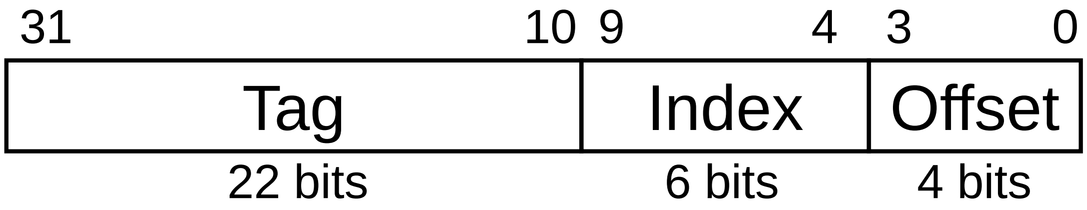
Cache Field Sizes: Example

- How many total bits are required for a direct mapped cache with 16KB of data and 4-word blocks assuming a 32-bit address?
  - 16KB is 4K (2^{12}) words
  - With a block size of 4 words, there are 1024 (2^{10}) blocks
  - Each block has 4x32 or 128 bits of data plus a tag which is 32 – (10 + 2 + 2) = 18 bits, plus a valid bit = 19 bits
  - So the total cache size is

    2^{10} x (4x32 +18 + 1) = 2^{10} x 147 = 147Kbits
  - or about 1.15x as many as needed just for storage of the data

Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
  - Block no. = (Block addr) modulo (#Blocks in cache)
- Block address = Byte addr/Bytes per block $= \lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11



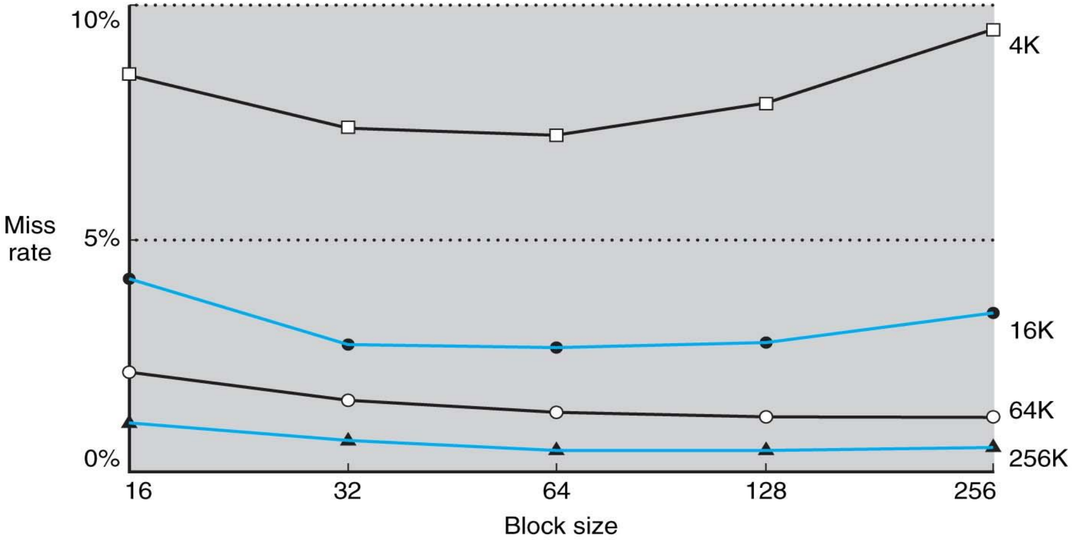
31 10 9 4 3 0

Tag Index Offset

22 bits 6 bits 4 bits

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
- Larger miss penalty
  - Larger blocks ⇒ more time required to fetch
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help
    - Early restart: resume execution as soon as requested word in block is returned
      - Rather than wait for entire block to be returned
    - Critical word first: organize memory so that requested word is transferred first
      - Remainder of block transferred afterwards
      - Wrap address around to beginning of block



Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
    - Stall the CPU pipeline
        - Instead of interrupts which require saving context
    - Fetch block from next level of hierarchy
    - Instruction cache miss
        - Restart instruction fetch
    - Data cache miss
        - Complete data access

Handling Cache Writes

- 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 Solution: 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
    - without waiting for the block to be written into memory first

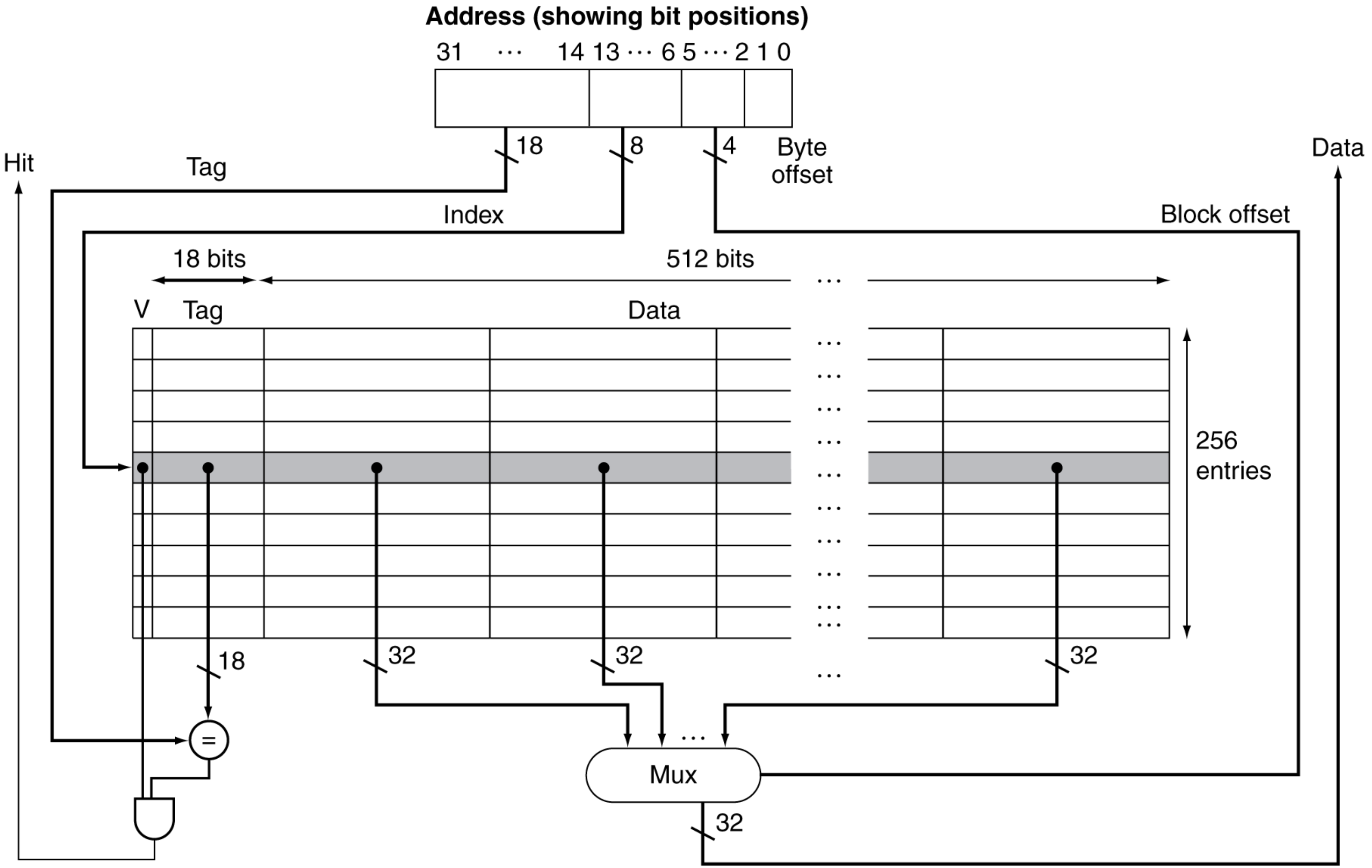
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block, overwrite appropriate portion of block with word(s)
  - Write around: don't fetch the block
    - Aka no write allocate
    - Since programs often write whole blocks of data without needing to read, or access data immediately (e.g., when OS zeros a memory page)
- Alternatives for write-back
  - Usually fetch the block

Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
    - OS decides which to use for an application
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

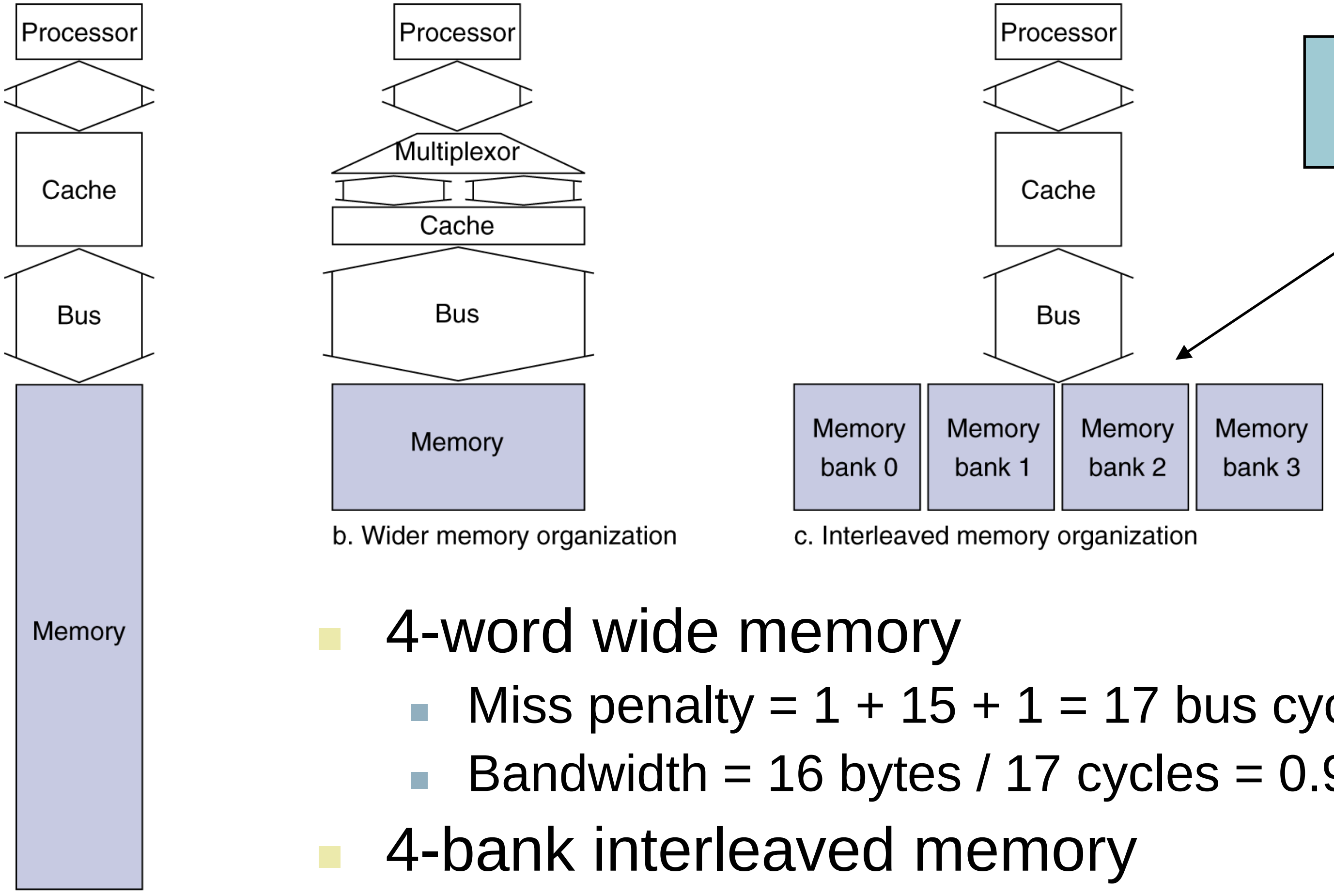
Example: Intrinsity FastMATH



Main Memory Supporting Caches

- Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- Example cache block read
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data word transfer
- For 4-word block, 1-word-wide DRAM
  - Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle
    - 1 GHz clocked DRAM would have 250 MB/sec bandwidth

Increasing Memory Bandwidth



a. One-word-wide memory organization

b. Wider memory organization

c. Interleaved memory organization

Independent writes/bank

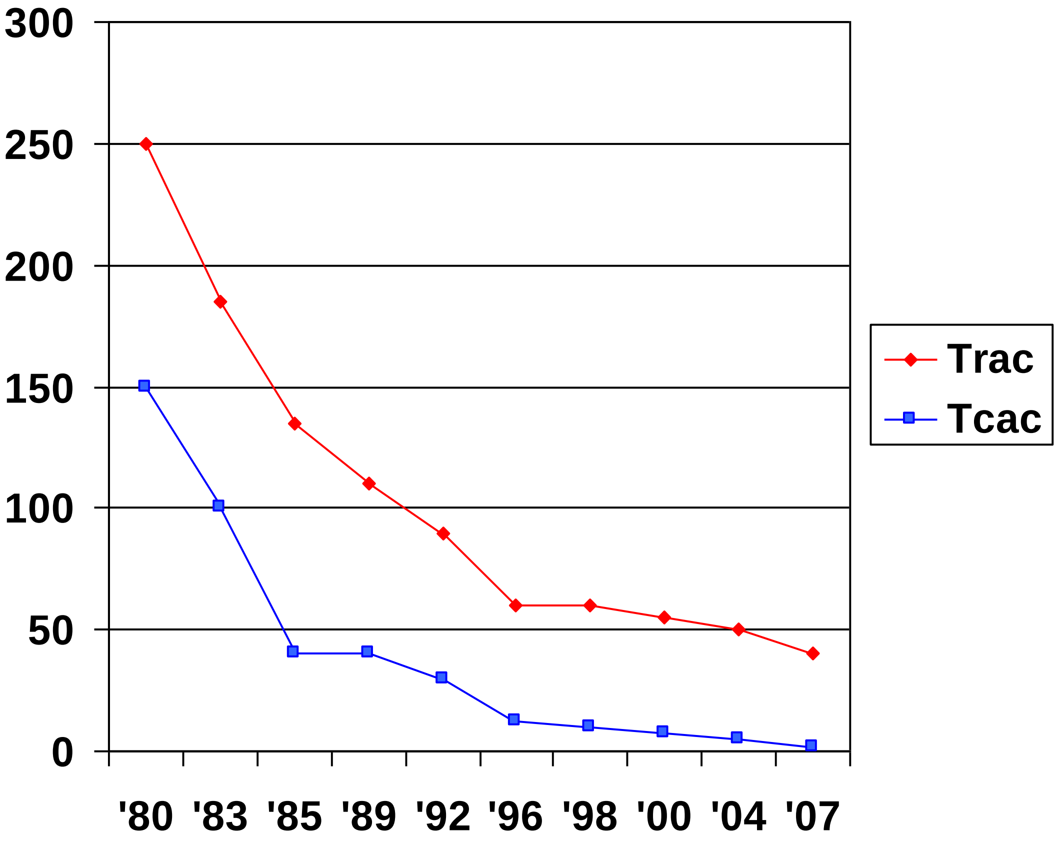
- 4-word wide memory
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- 4-bank interleaved memory
  - Miss penalty = 1 + 15 + 4×1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

Advanced DRAM Organization

- DRAMs include clocks to eliminate memory-processor synchronization time
  - Synchronized DRAM (SDRAM)
- DRAM bits are organized as a rectangular array
  - DRAM accesses (and buffers) an entire row
  - Burst mode: supply successive words from a row (buffer) with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs

DRAM Generations

Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



Trac: Total access time to new row/column
Tcac: Column access time to existing row

主观题 10分

设置

计算机的字长为32位且采用32位地址，假设主存的最大容量为 8MB，分为512K个主存块，Cache中数据容量为64KB，请问若按照采用直接映射方式。请问：

1) Cache划分为多少块？每个块中包含多少个字

2) Cache每块中地址为多少位?

3) Cache的存放内存块的Tag有多少位?

4)请计算Cache总的容量有多大（需要考虑有效位和标记位)

正常使用主观题需2.0以上版本雨课堂

作答

Measuring Cache Performance

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

$$\text{Memory stall cycles}$$

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

$$= \frac{\text{Instructio ns}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instructio n}} \times \text{Miss penalty}$$

§5.3 Measuring and Improving Cache Performance

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (without memory stalls) = 2
  - Load & stores are 36% of instructions
- How much faster would a processor run with a perfect cache that never missed?
- Miss cycles per instruction
  - I-cache: 0.02 × 100 = 2
  - D-cache: 0.36 × 0.04 × 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster

Cache Performance Example

- What if we speed-up the computer by reducing the $\text{CPI}_{\text{ideal}}$ to 1, without changing clock rate?
    - Actual CPI = 1 + 3.44 = 4.44
    - Ideal CPU is 4.44/1 =4.44 times faster
    - Amount of execution time spent on memory stalls rises from 3.44/5.44 = 63% to 3.44/4.44 = 77%
- What if we increased clock rate?
    - Similar increase in performance lost due to cache misses
- Remember Amdahl's law!

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, miss rate/instruction = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

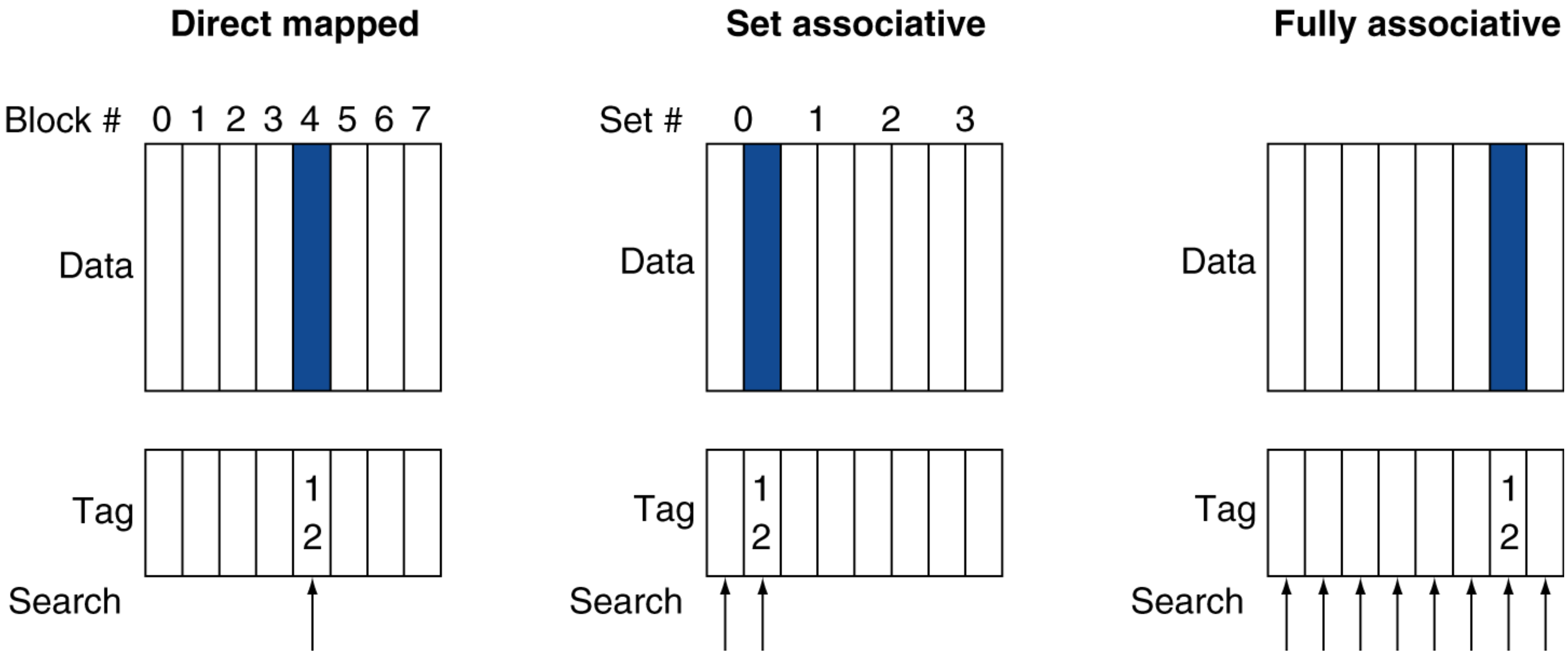
Performance Summary

- 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

Associative Caches

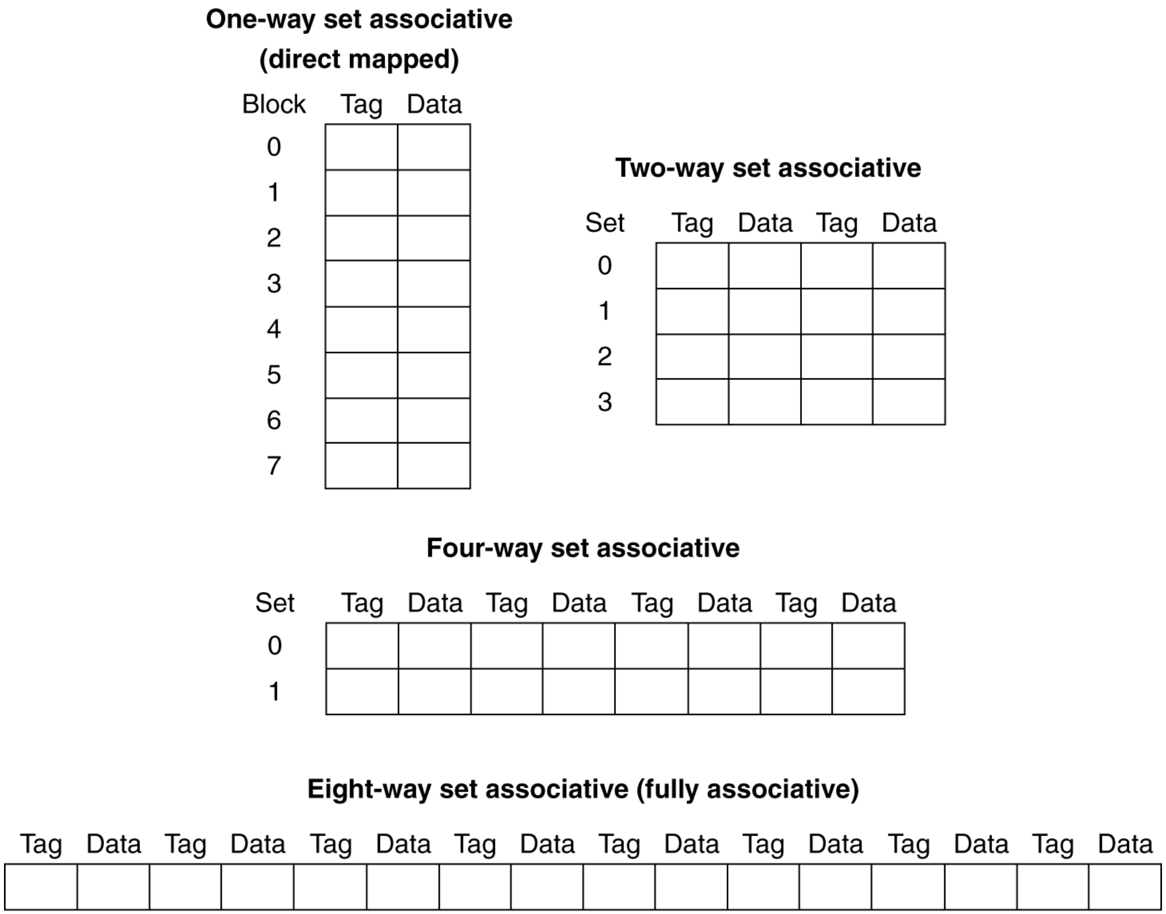
- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - One comparator per cache entry (expensive!)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in 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, 8,6
- Direct mapped
  - Block address to cache block mapping
    - 0 modulo 4 = 0; 6 modulo 4 = 2; 8 modulo 4 = 0;

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
  - 0 modulo 2 = 0; 6 modulo 2 = 0; 8 modulo 2 = 0;

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]		

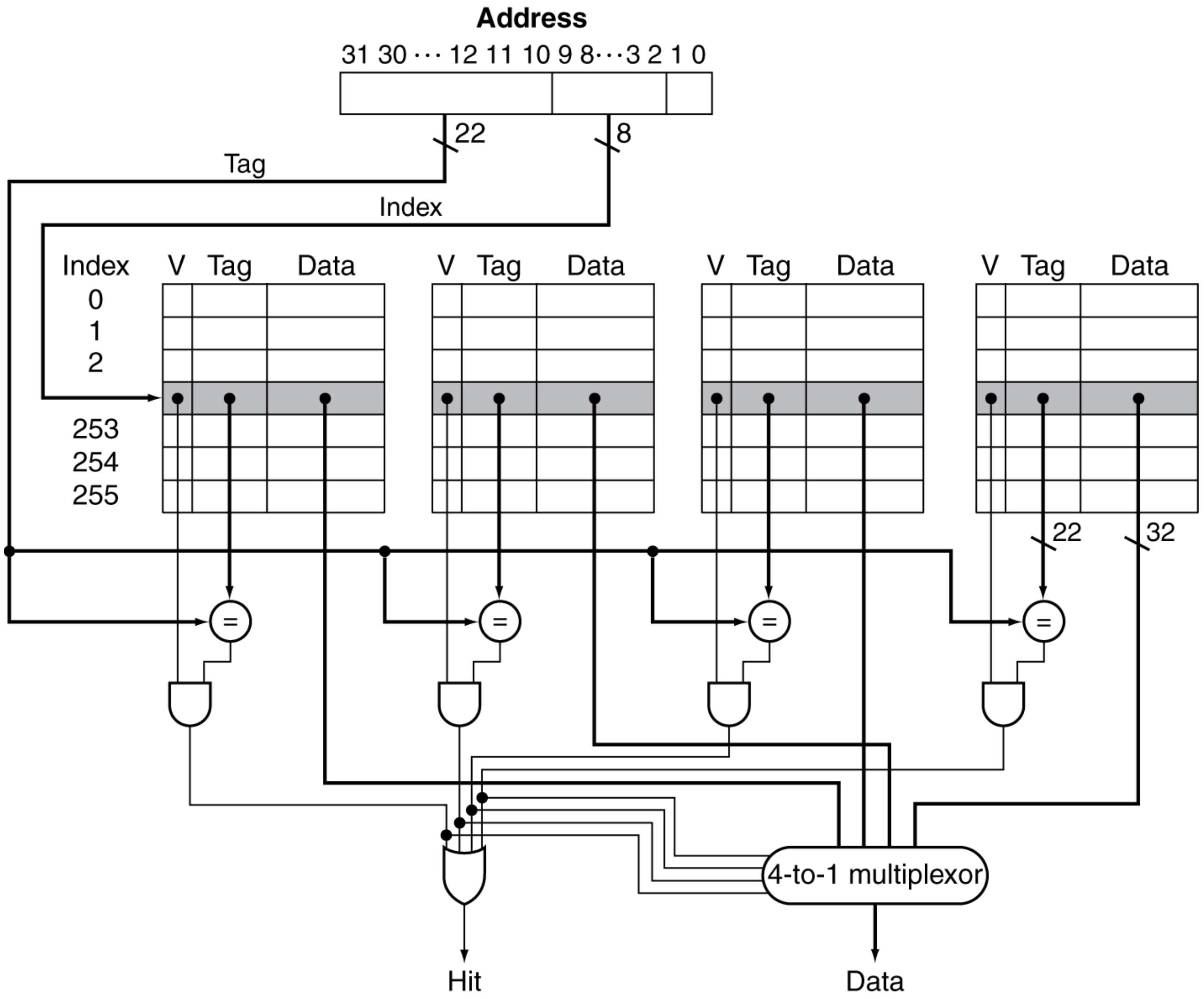
- Fully associative

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

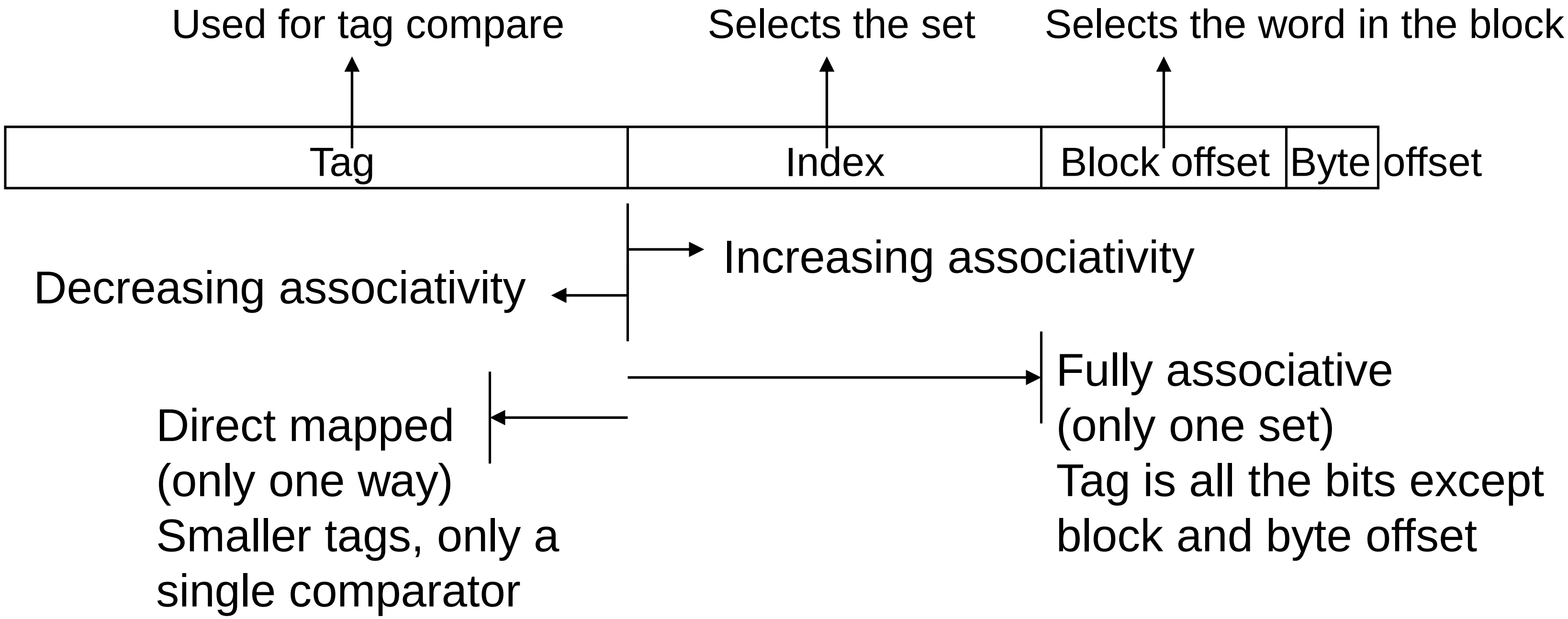
- Increased associativity decreases miss rate
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000; Miss Rates:
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Set Associative Cache Organization



Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



Used for tag compare

Selects the set

Selects the word in the block

Tag	Index	Block offset	Byte offset

Decreasing associativity

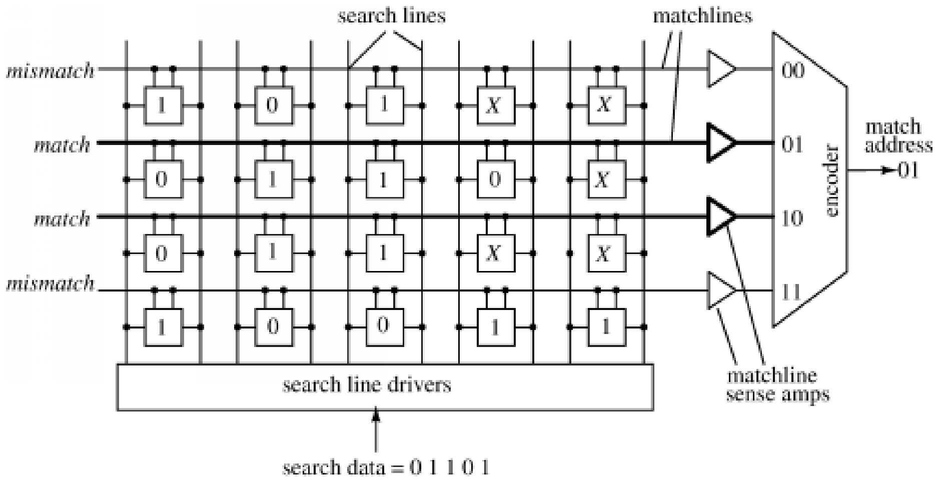
Increasing associativity

Direct mapped
(only one way)
Smaller tags, only a single comparator

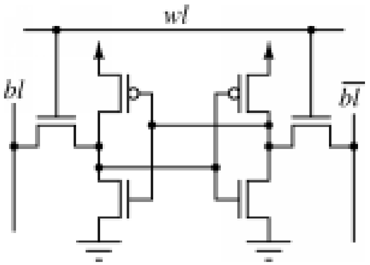
Fully associative
(only one set)
Tag is all the bits except block and byte offset

Content Addressable Memory

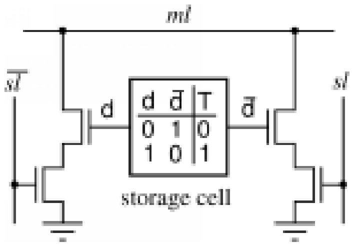
- CAM
  - Circuit that combines comparison and storage in a single device
  - Supply data, CAM looks for a copy, returning index of matching rows
  - High speed table lookup



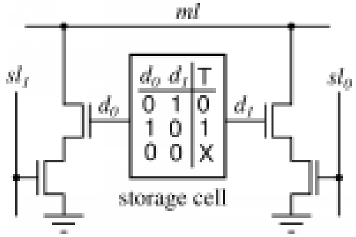
NOR-based CAM implementation



6 transistor SRAM cell



Binary CAM cell



Ternary CAM cell

- Recent trend (~2011)
  - 2, 4 way set associativity built using SRAMs, comparators
  - 8 way and above built using CAMs

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

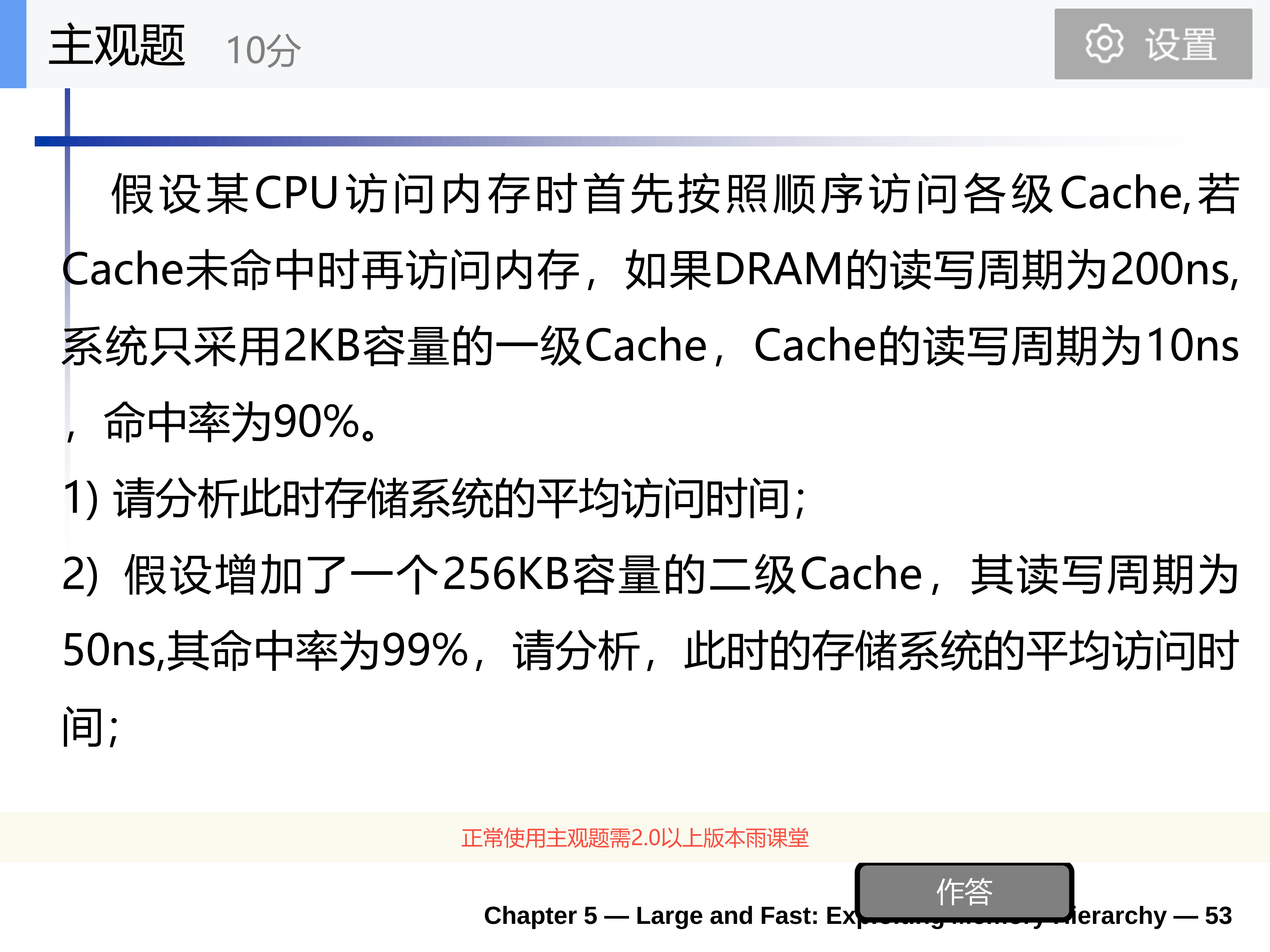
- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- How much faster will the processor be with a secondary cache with 5 ns access time, and is large enough to reduce miss rate to 0.5%?
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 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



主观题 10分

⚙ 设置

假设某CPU访问内存时首先按照顺序访问各级Cache,若Cache未命中时再访问内存，如果DRAM的读写周期为200ns,系统只采用2KB容量的一级Cache，Cache的读写周期为10ns，命中率为90%。

1) 请分析此时存储系统的平均访问时间;

2) 假设增加了一个256KB容量的二级Cache，其读写周期为50ns,其命中率为99%，请分析，此时的存储系统的平均访问时间;

正常使用主观题需2.0以上版本雨课堂

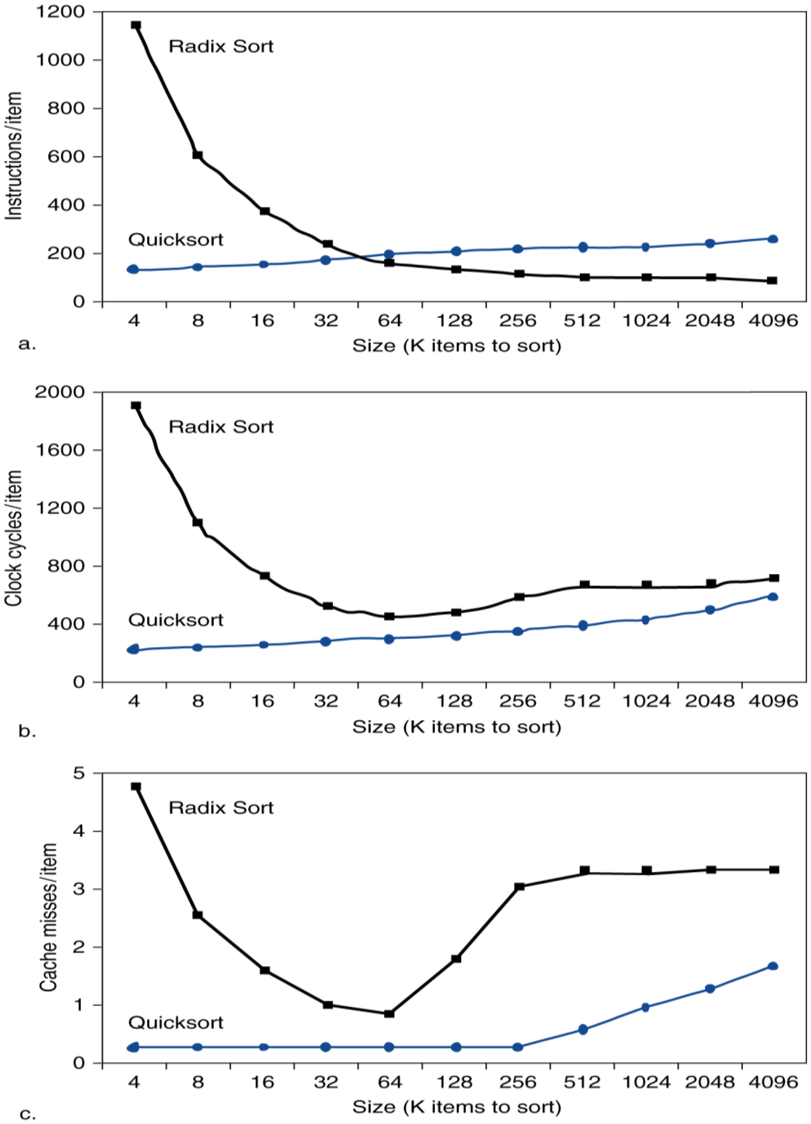
作答

Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation

Interactions with Software

- Radix sort vs. Quicksort
  - Radix sort has algorithmic advantage in terms on no. of operations per item
  - But time/item for radix sort increases as shown in Fig b. => Why??
    - *See Fig. c for answer*
- Misses depend on memory access patterns
  - Algorithm behavior, compiler optimization for memory access
- Autotuning helps performance
  - Parameterize algorithms and find best parameter combination at runtime for given architecture
    - e.g. write back vs. write through, etc.



Virtual Memory

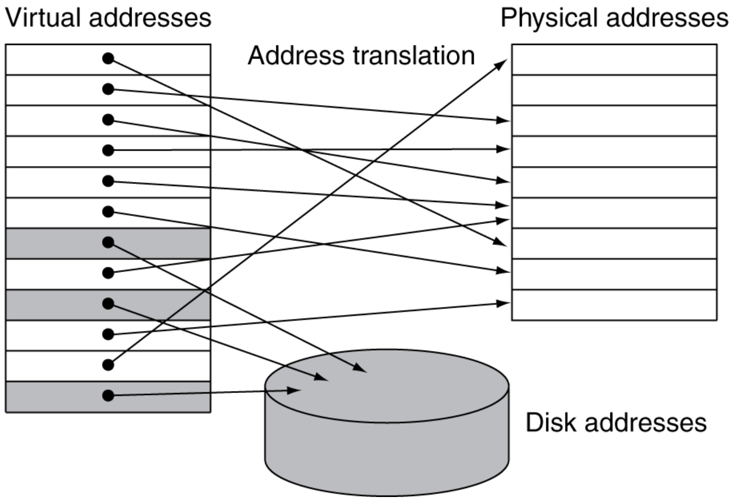
- Use main memory as a "cache" for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual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

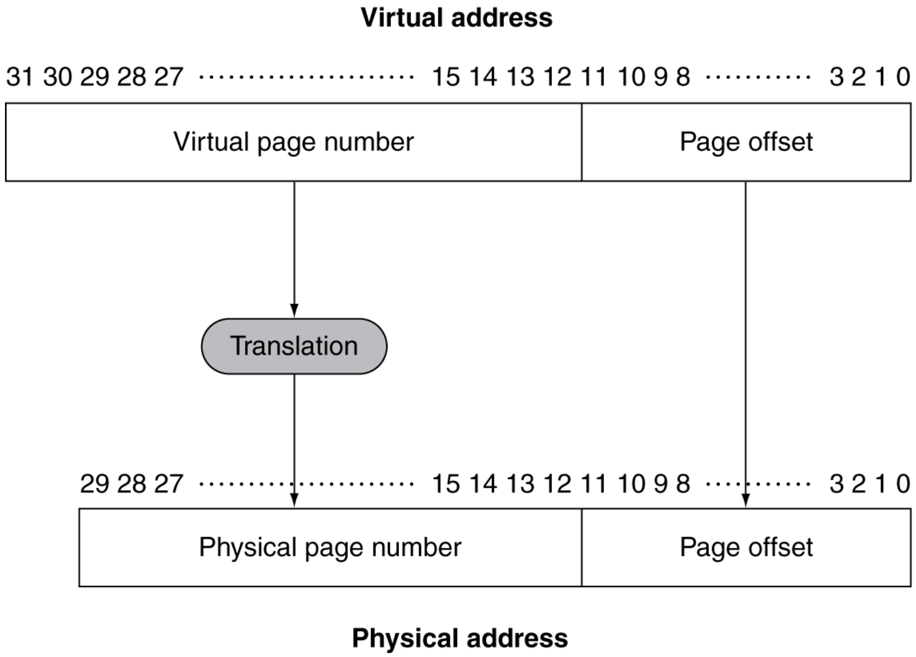
Segmentation vs. Paging

- Paging: fixed size blocks
- Segmentation: variable size blocks
  - Address consists of two parts: segment address (mapped to physical address) and a segment offset
  - Bounds check required to ensure offset within segment
  - (+) Enables more powerful methods of protection, sharing
  - (-) Splits address space into logically separate pieces that must be manipulated as a 2-part address
    - Paging makes boundary between page number and offset invisible to programmers and compilers
  - Will focus on paging => more efficient

Address Translation

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





- Main memory can have 1GB while virtual address space is 4 GB

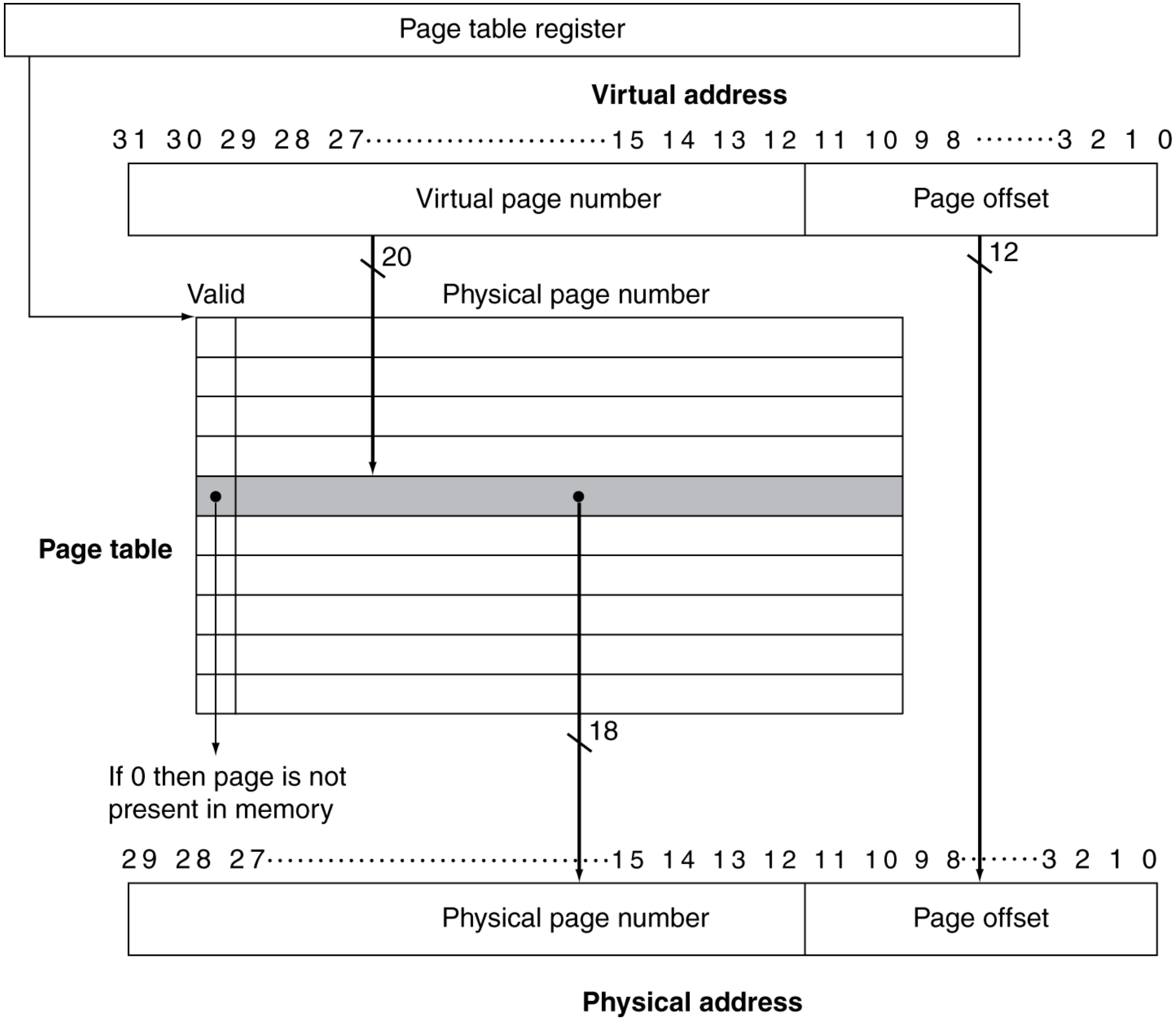
Page Tables

- Stores placement information
  - Array of page table entries (PTEs), 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

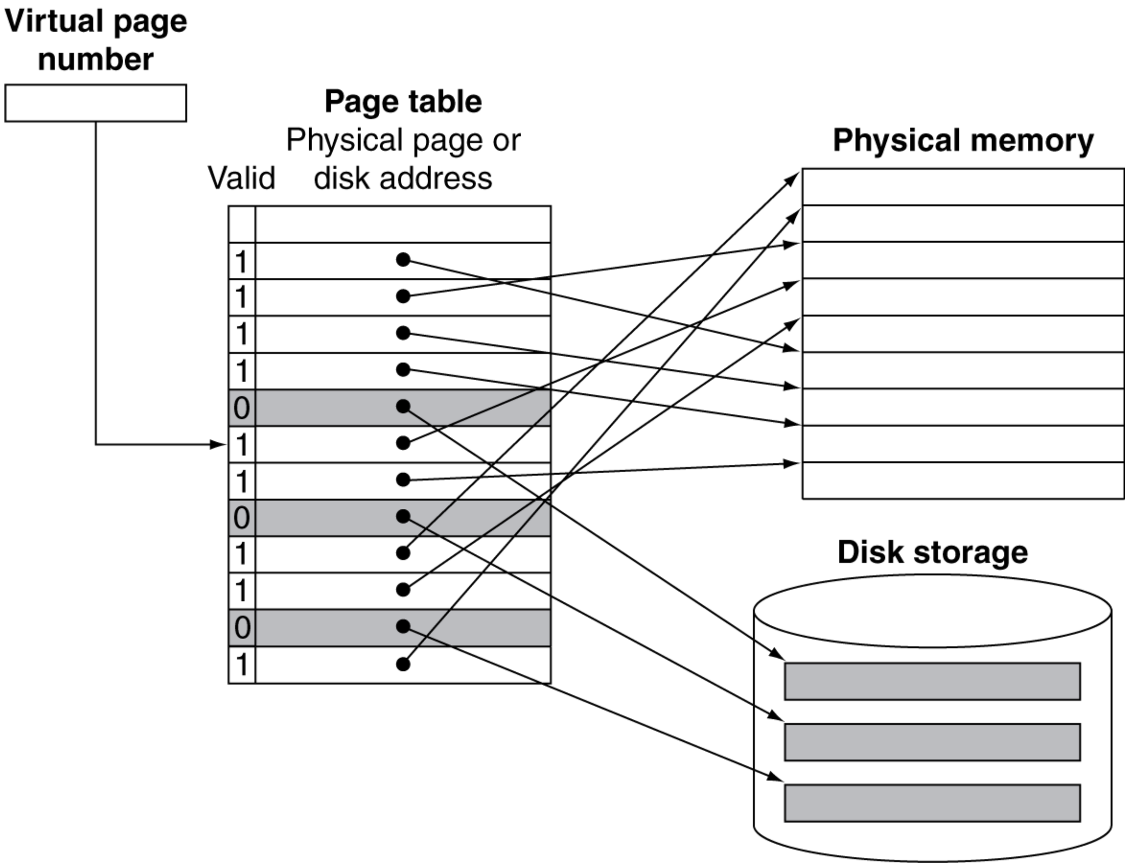
Page Fault Penalty

- 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

Translation Using a Page Table



Mapping Pages to Storage



Page Table Size Calculation

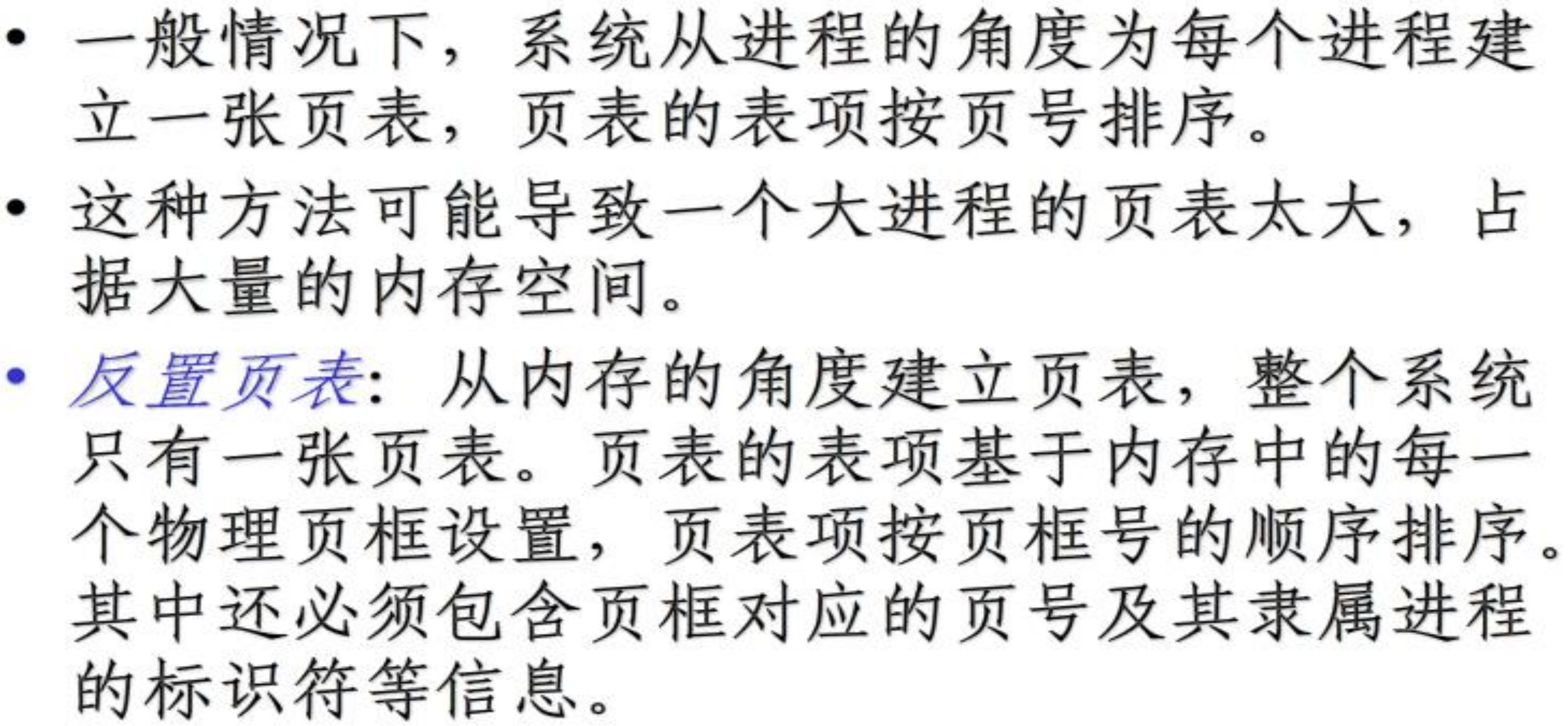
- 32-bit virtual address space
- 4 KB pages, 4 bytes/PTE
- What is the total page table size?

- No. of page table entries = $2^{32}/2^{12} = 2^{20}$
- Size of page table = 2^{20} PTE x 2^{2} bytes/PTE = 4 MB per program in execution at any given time
- What if there are hundreds of programs in execution?
- How can we handle 64-bit addresses that would need 2^{52} PTEs?

Techniques to Minimize Page Table Size

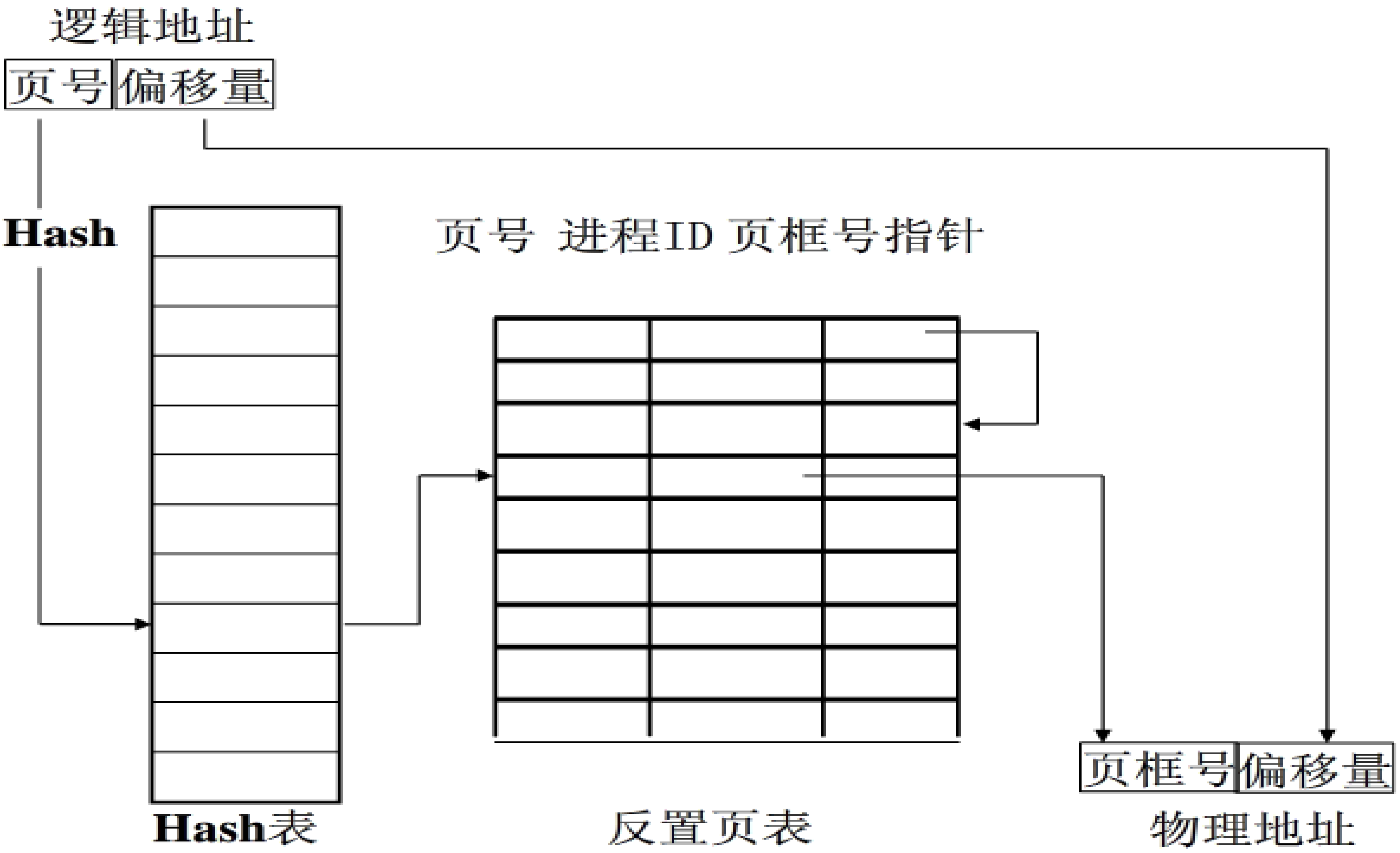
- Limit register: restricts size of page table for a process
  - If virtual page number becomes larger than contents of limit register, add entries to page table (i.e., page tables grows)
- Use hashing function for virtual address to limit page table size to number of physical pages in main memory
  - Inverted page table; more complex lookup procedure
- Multiple levels of page tables (typically 2-level)
  - First level maps large fixed size blocks of virtual address space 64 to 256 pages in total (segment table); Second level contains page tables; More complex address translation
- Paging the page tables（改变存储方式，不减小页表数量）
  - Page tables reside in virtual address space
  - Most modern systems allow this to reduce the actual main memory tied up in page tables

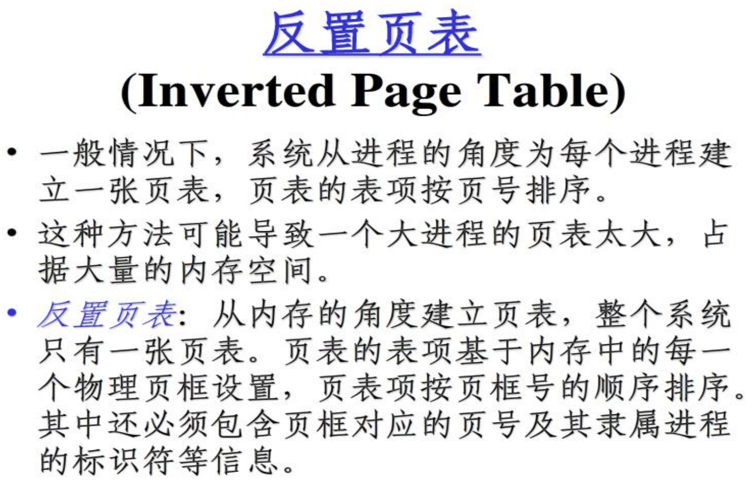
Inverted Page Table



- 一般情况下，系统从进程的角度为每个进程建立一张页表，页表的表项按页号排序。
- 这种方法可能导致一个大进程的页表太大，占据大量的内存空间。
- *反置页表*：从内存的角度建立页表，整个系统只有一张页表。页表的表项基于内存中的每一个物理页框设置，页表项按页框号的顺序排序。其中还必须包含页框对应的页号及其隶属进程的标识符等信息。

Inverted Page Table： Address Translation

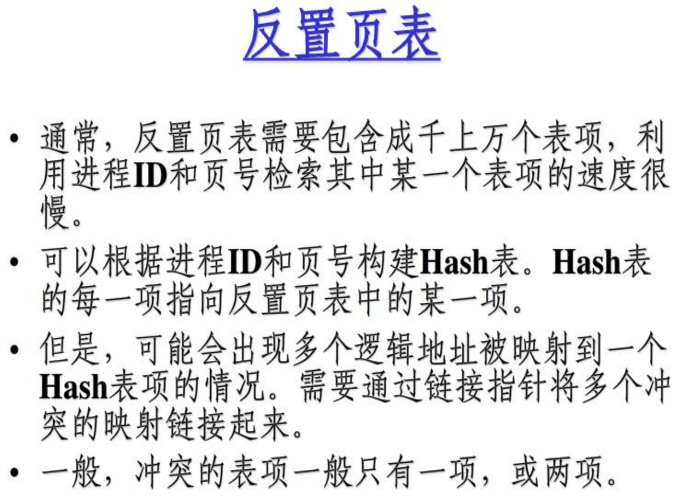




反置页表

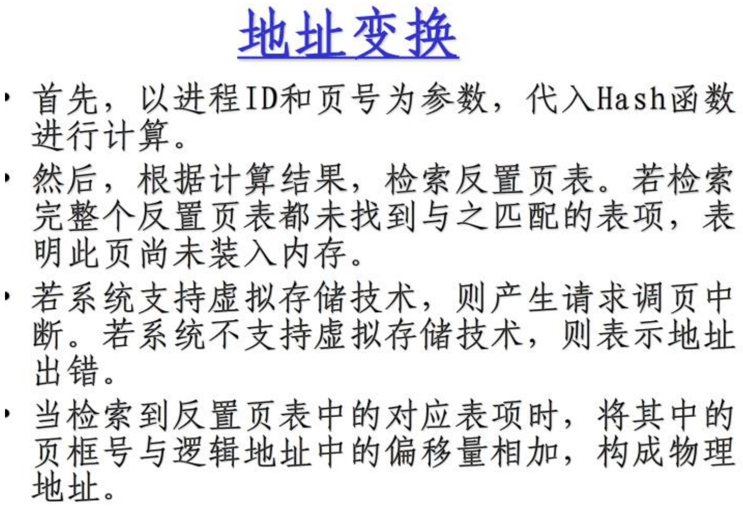
(Inverted Page Table)

- 一般情况下，系统从进程的角度为每个进程建立一张页表，页表的表项按页号排序。
- 这种方法可能导致一个大进程的页表太大，占据大量的内存空间。
- *反置页表*：从内存的角度建立页表，整个系统只有一张页表。页表的表项基于内存中的每一个物理页框设置，页表项按页框号的顺序排序。其中还必须包含页框对应的页号及其隶属进程的标识符等信息。



反置页表

- 通常，反置页表需要包含成千上万个表项，利用进程ID和页号检索其中某一个表项的速度很慢。
- 可以根据进程ID和页号构建Hash表。Hash表的每一项指向反置页表中的某一项。
- 但是，可能会出现多个逻辑地址被映射到一个Hash表项的情况。需要通过链接指针将多个冲突的映射链接起来。
- 一般，冲突的表项一般只有一项，或两项。



地址变换

- 首先，以进程ID和页号为参数，代入Hash函数进行计算。
- 然后，根据计算结果，检索反置页表。若检索完整个反置页表都未找到与之匹配的表项，表明此页尚未装入内存。
- 若系统支持虚拟存储技术，则产生请求调页中断。若系统不支持虚拟存储技术，则表示地址出错。
- 当检索到反置页表中的对应表项时，将其中的页框号与逻辑地址中的偏移量相加，构成物理地址。

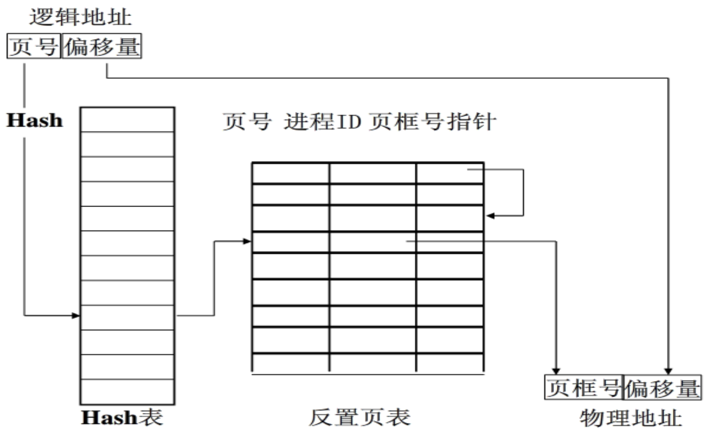


图3.20 利用反置页表进行地址变换

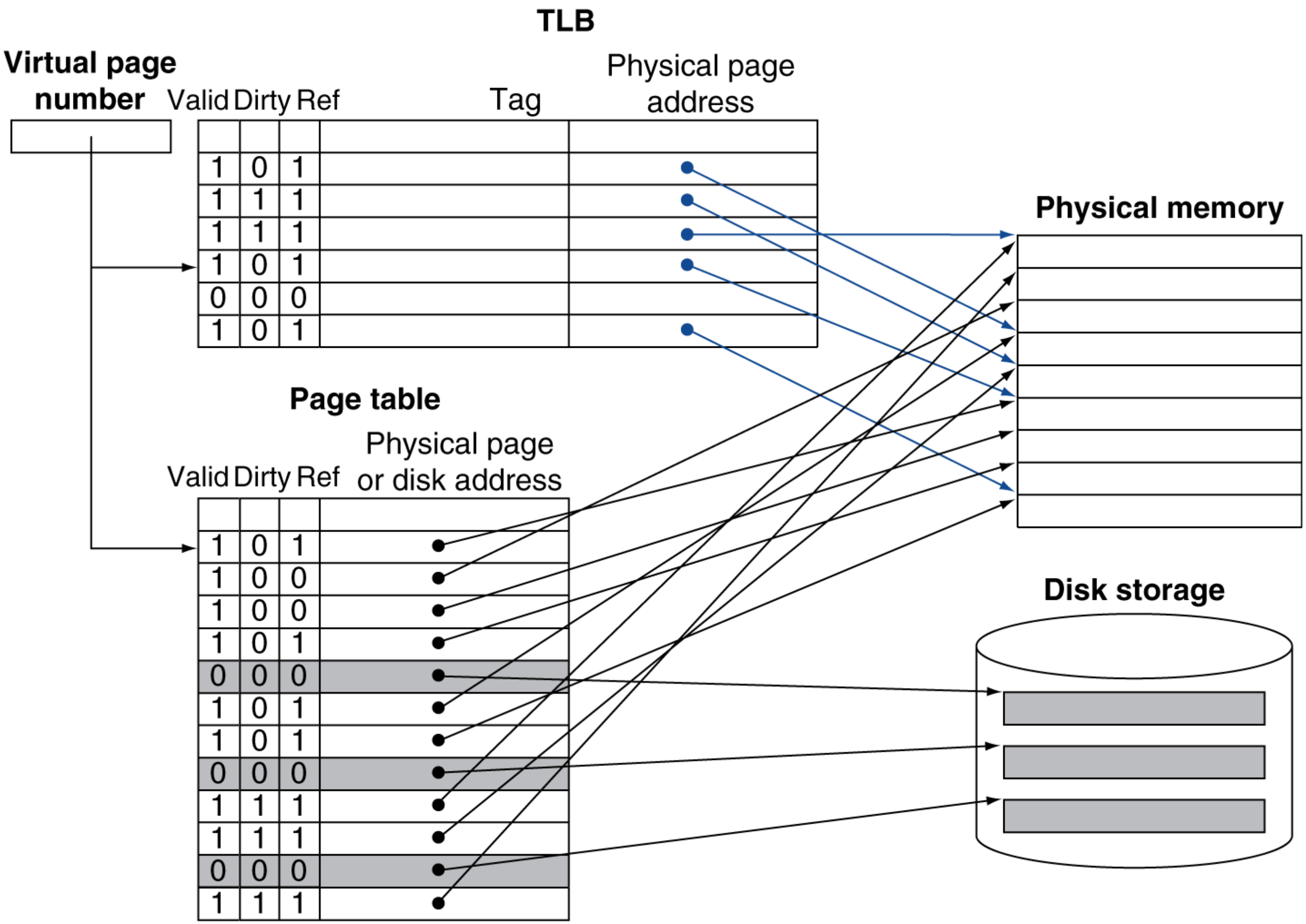
Replacement and Writes

- 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
  - Small TLBs are typically fully associative, large TLBs have small associativity (due to cost issues)
  - Misses could be handled by hardware or software
    - Write back scheme for writing reference, dirty bits to PTE

Fast Translation Using a TLB



TLB

Valid	Dirty	Ref	Tag	Physical page address
1	0	1		●
1	1	1		●
1	1	1		●
1	0	1		●
0	0	0		
1	0	1		●

Page table

Valid	Dirty	Ref	Physical page or disk address
1	0	1	●
1	0	0	●
1	0	0	●
1	0	1	●
0	0	0	●
1	0	1	●
1	0	1	●
0	0	0	●
1	1	1	●
1	1	1	●
0	0	0	●
1	1	1	●

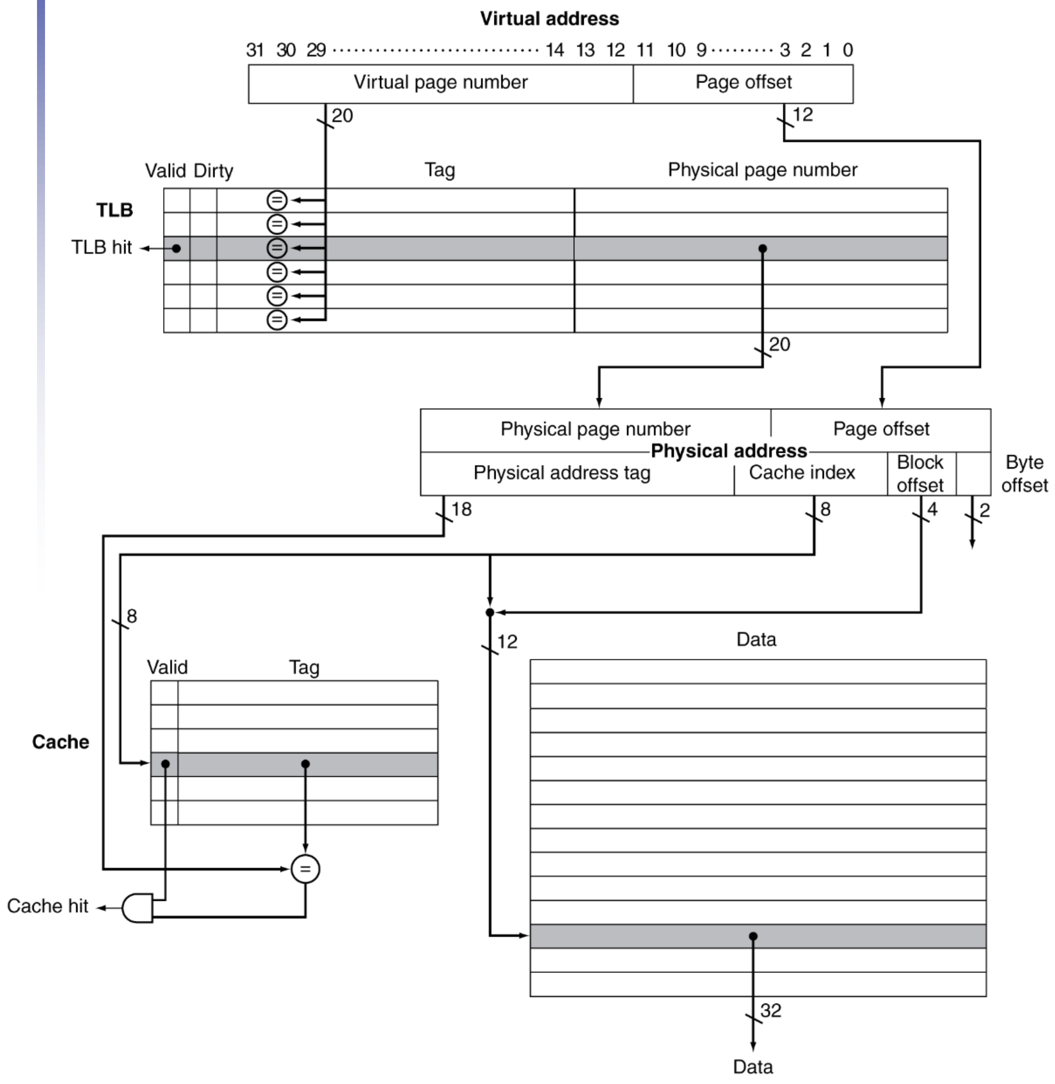
TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - 10’s of cycles
  - Could be handled in hardware
    - Can get complex for more complex 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 page table
  - Then restart the faulting instruction
  - 1,000,000’s of cycles
- TLB misses more frequent than page faults

Page Fault Handler

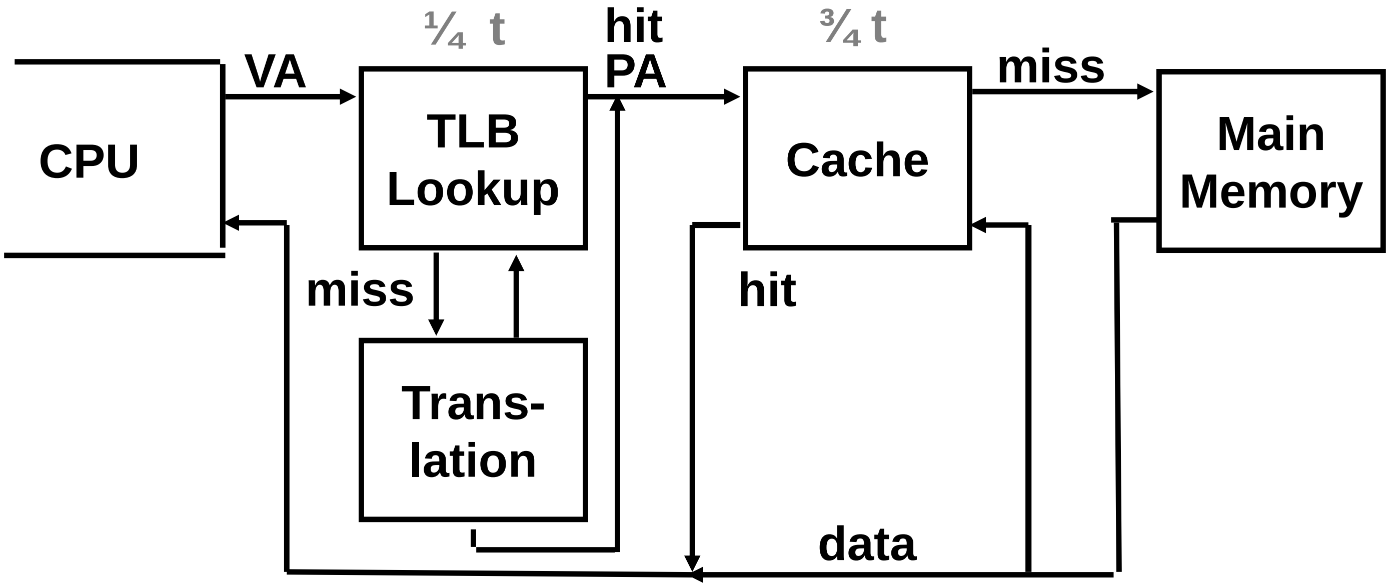
- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction

TLB and Cache Interaction Example



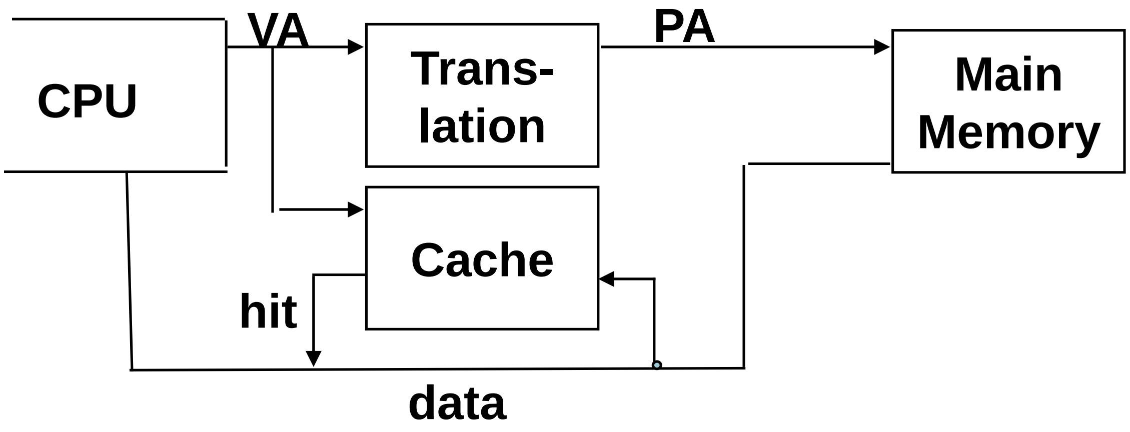
- Intrinsity FastMATH
- Fully associative TLB
- Concatenation avoids 16:1 multiplexor
- Direct mapped cache
  - Physically tagged and physically indexed

Cache Addressing



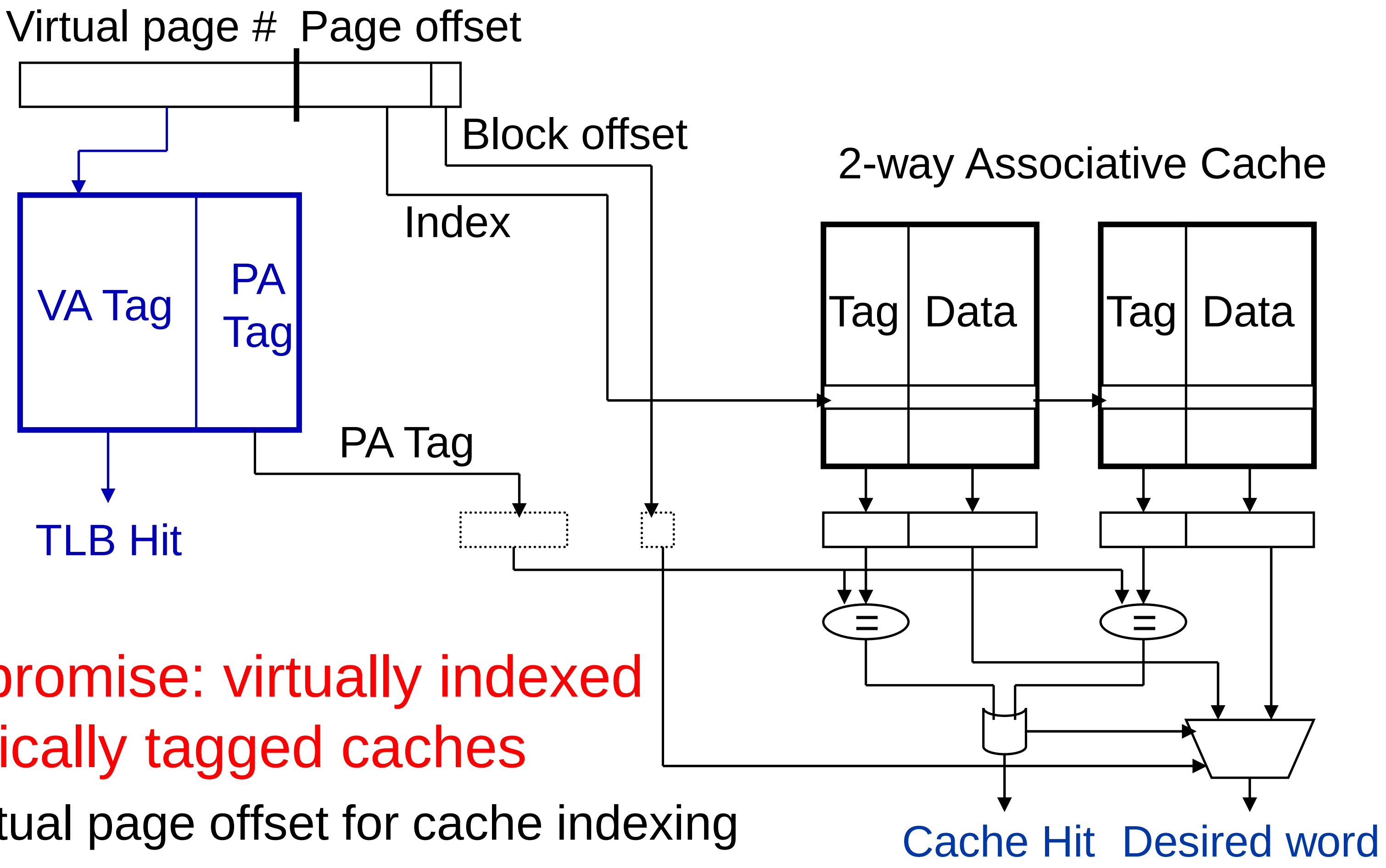
- Physically tagged and physically indexed caches
    - Need to translate address before cache lookup

Cache Addressing



- Virtually tagged and virtually indexed caches
  - TLB not in critical path and accessed only on cache miss to translate virtual address to physical main memory address
  - Complications due to aliasing
    - Different virtual addresses for shared physical address
      - stored in separate locations in cache
    - One program can write data without other program being aware that the data has changed – ambiguous!

Cache Addressing



- Compromise: virtually indexed physically tagged caches
  - Virtual page offset for cache indexing
    - Really a physical address as it is not translated
  - No aliasing in this case; better performance due to overlap of cache access with TLB access

TLB, VM, Cache Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – TLB miss, PA in page table
Miss	Hit	Miss	Yes – TLB miss, PA in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

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
  - Supervisor mode (aka kernel mode)
    - Runs privileged instructions not available in user mode
  - Page tables, TLBs and other state information only writeable in supervisor mode (read-only in user mode)
  - System call exception (e.g., syscall in MIPS) to go from user mode to supervisor mode

TLB Miss Handling in MIPS

- Consider a TLB miss for a page that is present in memory (i.e., the Valid bit in page table is set)
  - MIPS handles TLB misses in software
  - A TLB miss (or a page fault exception) must be asserted by the end of the same clock cycle that the memory access occurs so that the next clock cycle will begin exception processing

Register	CP0 Reg #	Description
EPC	14	Where to restart after exception
Cause	13	Cause of exception
BadVAddr	8	Address that caused exception
Index	0	Location in TLB to be read/written
Random	1	Pseudorandom location in TLB
EntryLo	2	Physical page address and flags
EntryHi	10	Virtual page address
Context	4	Page table address & page number

MIPS Software TLB Miss Handler

- When a TLB miss occurs, the hardware saves the address that caused the miss in `BadVAddr` and transfers control to 8000 0000_{hex}, the location of the TLB miss handler

```
TLBmiss:
 mfc0   $k1, Context       #copy addr of PTE into $k1
 lw     $k1, 0($k1)        #put PTE into $k1
 mtc0   $k1, EntryLo       #put PTE into EntryLo
 tlbwr                     #put EntryLo into TLB
                           #     at Random
 eret                      #return from exception
```

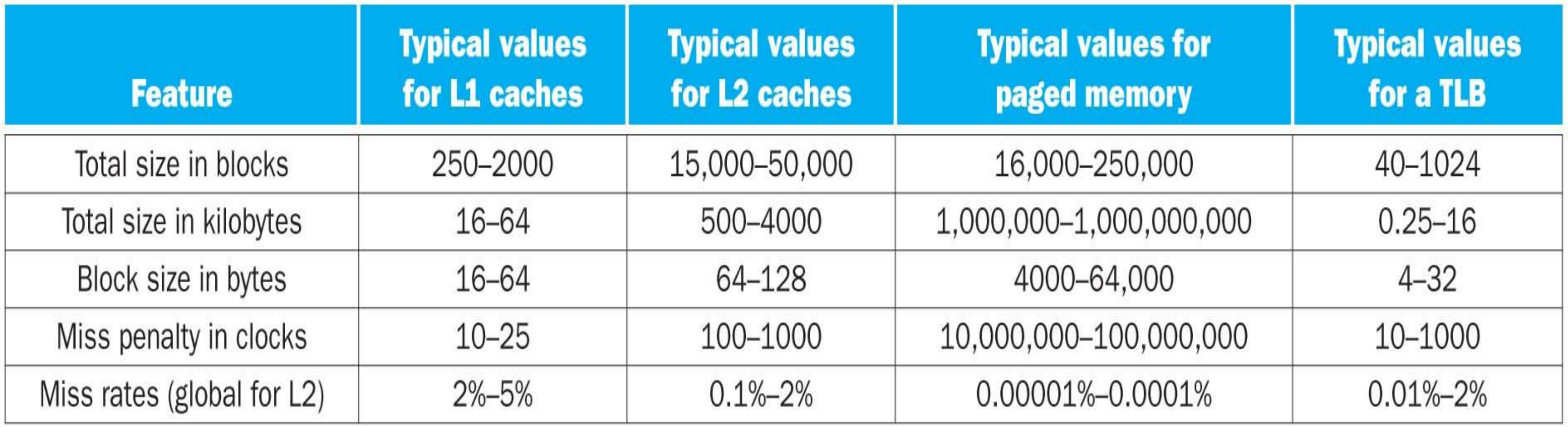
- MIPS hardware places address of missing page in `Context`
- `tlbwr` copies from `EntryLo` into the TLB entry selected by the control register `Random`
- A TLB miss takes about a dozen clock cycles to handle

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

Characteristics of Memory Hierarchy

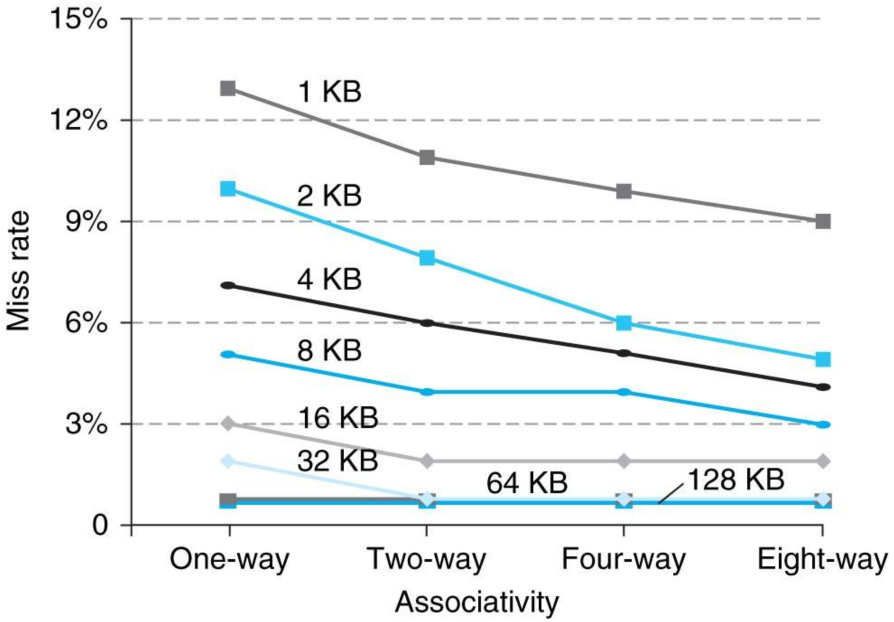


Feature	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Total size in blocks	250–2000	15,000–50,000	16,000–250,000	40–1024
Total size in kilobytes	16–64	500–4000	1,000,000–1,000,000,000	0.25–16
Block size in bytes	16–64	64–128	4000–64,000	4–32
Miss penalty in clocks	10–25	100–1000	10,000,000–100,000,000	10–1000
Miss rates (global for L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

- Four Questions for Memory Hierarchy
  - Q1: Where can a block be placed in the upper level?
  - Q2: How is a block found if it is in the upper level?
  - Q3: Which block should be replaced on a miss?
  - Q4: What happens on a write?

Q1. 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

Q2. 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

- Hardware caches
  - Reduce comparisons to reduce cost
  - Usually set associative or (less commonly) direct mapped
- Virtual memory systems (with page tables)
  - Almost always fully associative, as misses are very expensive
  - Software can use sophisticated replacement schemes to further reduce miss rate

Q3. Replacement on Miss?

- 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

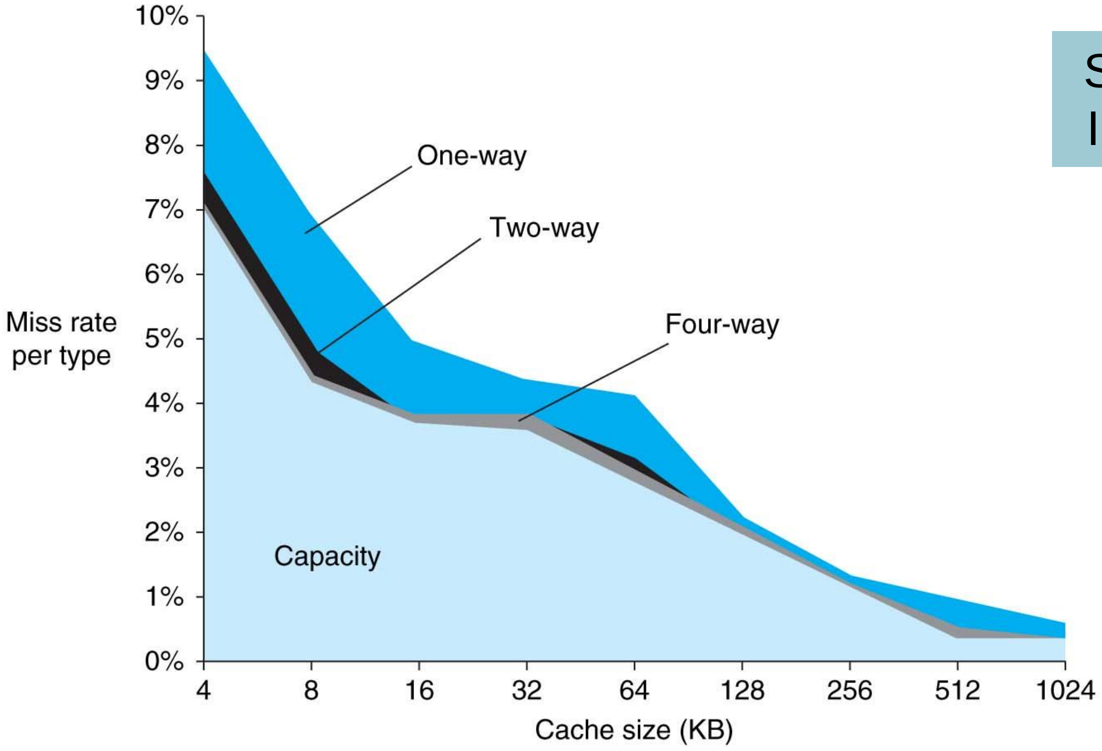
Q4. Write Policy?

- 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 Cache Misses

- Compulsory (cold start or process migration, first reference):
  - First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size (may increase access time)
  - Solution 2: increase associativity (may increase access time)

Sources of Cache Misses



SPEC2000
INT and FP

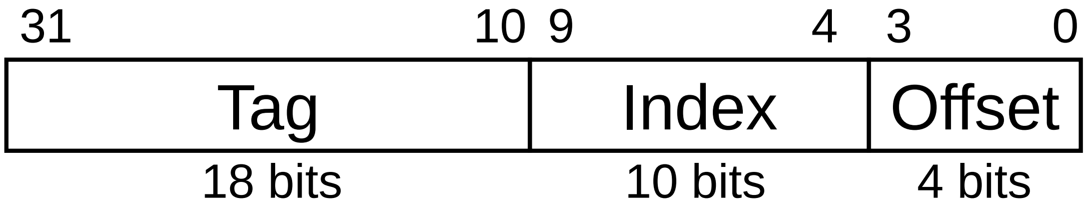
- Compulsory misses 0.006% (not shown in graph)
- Capacity misses depend on cache size
- Conflict misses shown change with associativity

Cache Design Trade-offs: Summary

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.

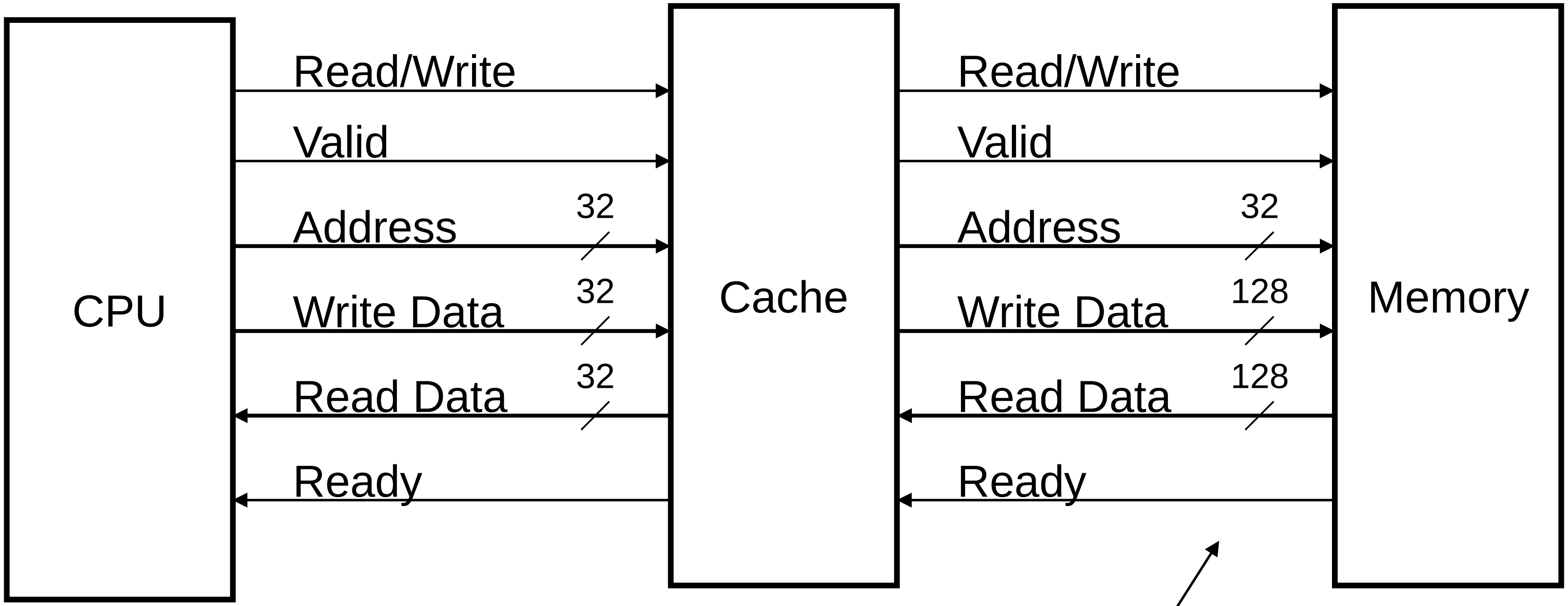
Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete



31 … 10	9 … 4	3 … 0
Tag	Index	Offset
18 bits	10 bits	4 bits

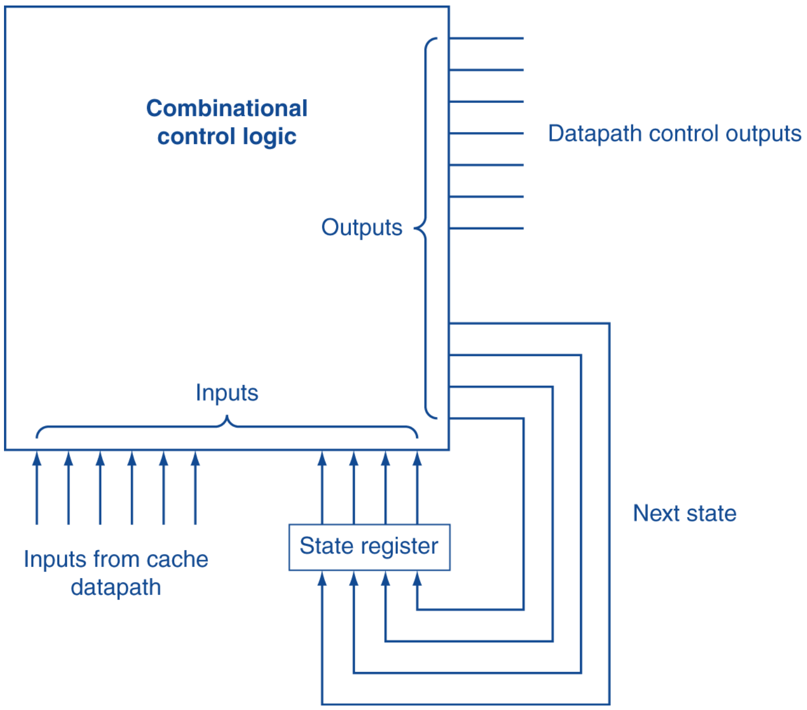
Interface Signals



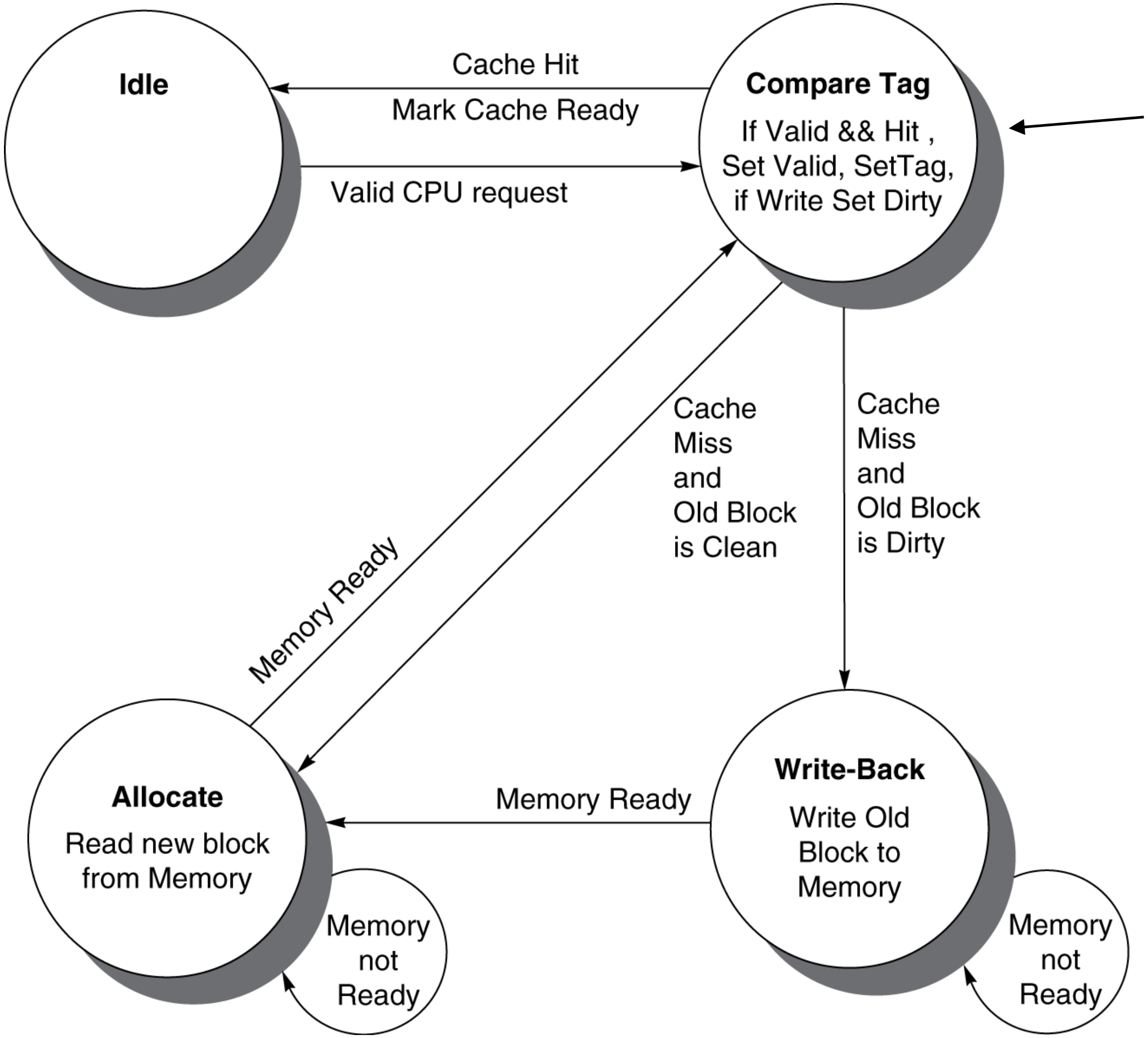
Multiple cycles per access

Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state
    = f_n (current state,
    current inputs)
- Control output signals
  = f_o (current state)



Cache Controller FSM



Could partition into separate states to reduce clock cycle time

§5.8 Parallelism and Memory Hierarchies: Cache Coherence

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

Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)
    $\Rightarrow$ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X
    $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X

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

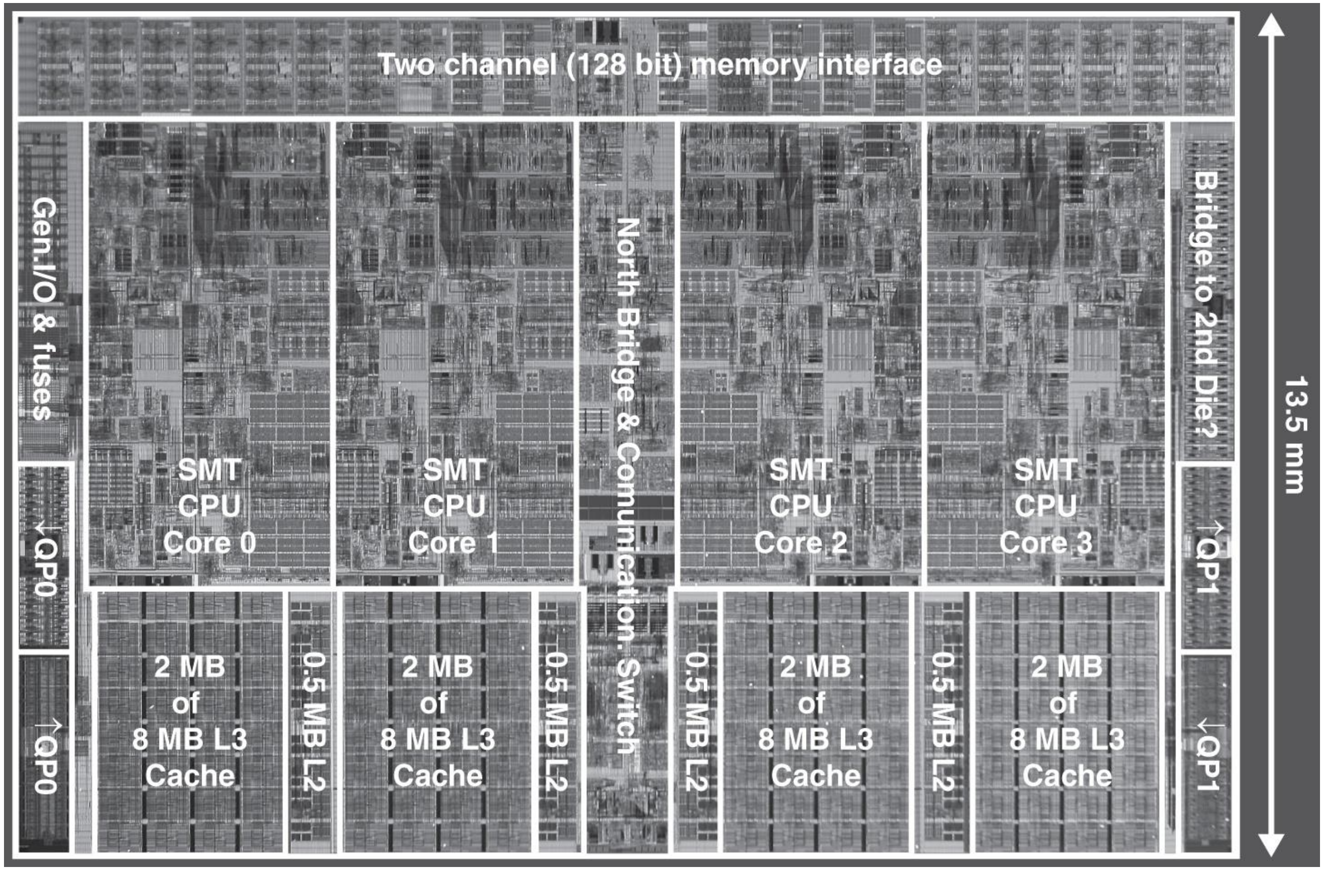
Memory Consistency

- When are writes seen by other processors
  - “Seen” means a read returns the written value
  - Can’t be instantaneously
- Assumptions for maintaining consistency
  - 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
    $\Rightarrow$ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

Multilevel On-Chip Caches

§5.10 Real Stuff: The AMD Opteron X4 and Intel Nehalem

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache
Shared 8MB L3 cache; 13.5x19.6mm die, 731 million transistors

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

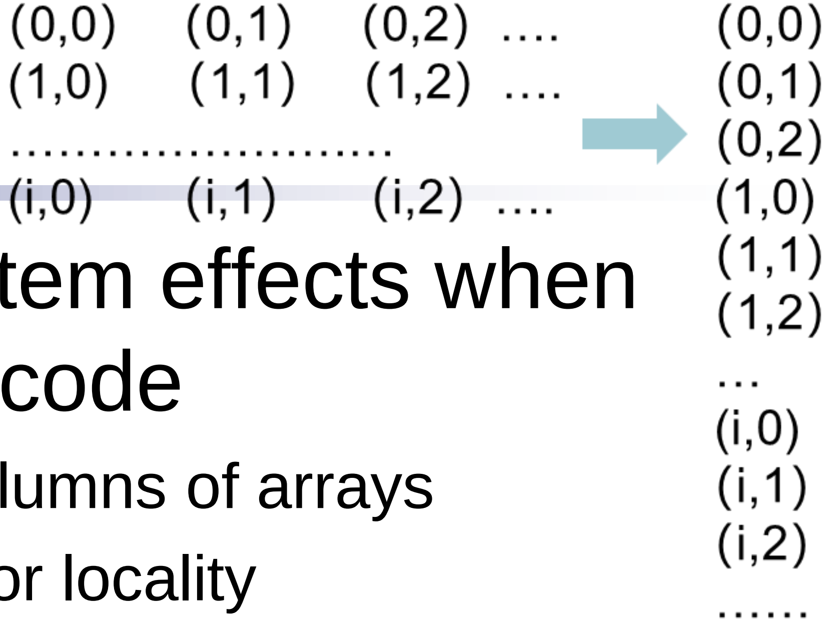
	Intel Nehalem	**AMD Opteron X4**
L1 caches (per core)	L1 I-cache: 32KB, **64-byte** blocks**, 4-way**, **approx LRU** replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, **8-way**, **approx LRU** replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, **64-byte** blocks, **2-way**, LRU 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)	512KB, **64-byt**e 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

n/a: data not available

Miss Penalty Reduction

- Return requested word first
  - Then back-fill rest of block
- Non-blocking miss processing
  - Allow processor to make references to cache while it is handling an earlier miss
  - 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

Pitfalls



- Ignoring memory system effects when writing or generating code
  - Iterating over rows vs. columns of arrays
  - Large strides result in poor locality
  - MIPS CPU runtime for code is half that of the case when loop order is changed to k, j, i

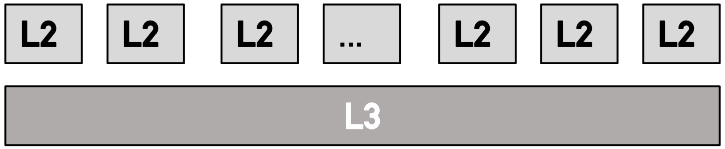
```
for (i = 0; i! = 500; i = i + 1)
   for (j = 0; j! = 500; j = j + 1)
    for (k = 0; k! = 500; k = k + 1)
     x[i][j] = x[i][j]
          + y[i][k] * z[k][j];
```

Pitfalls

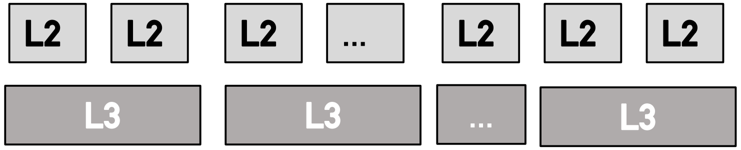
- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores ⇒ need to increase associativity!
- Using AMAT to evaluate memory hierarchy performance of out-of-order processors
  - Processor continues to execute instructions on cache miss, and may even have more cache misses on a cache miss ⇒ AMAT not accurate!
  - Instead, evaluate performance by simulation

鲲鹏920系列芯片架构——Cache模式

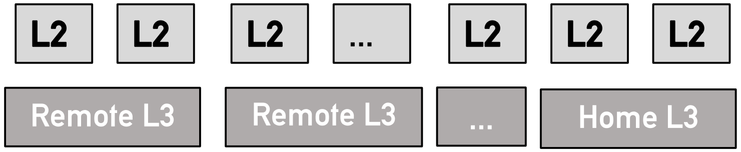
- Share Cache：对所有的L2来说L3 cache是共享的，一个进程可以使用整个L3的容量



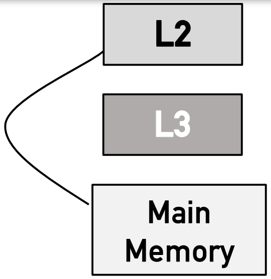
- Private Cache：有N个Private的L3，每个Private L3只缓存对应的L2的数据。即一个进程只能使用对应的部分L3的容量，无法使用全部L3的容量，L3和L3之间不通信



- Partitioned Cache：与Private相同的是，一个进程只能使用对应的部分L3容量；与Private不同的是，L3细分为一个Home的L3和N个Remote的L3，Home的L3类似L4，所以L3和L3之间会通信，由Home的L3来维护多个Partitioned L3之间的一致性



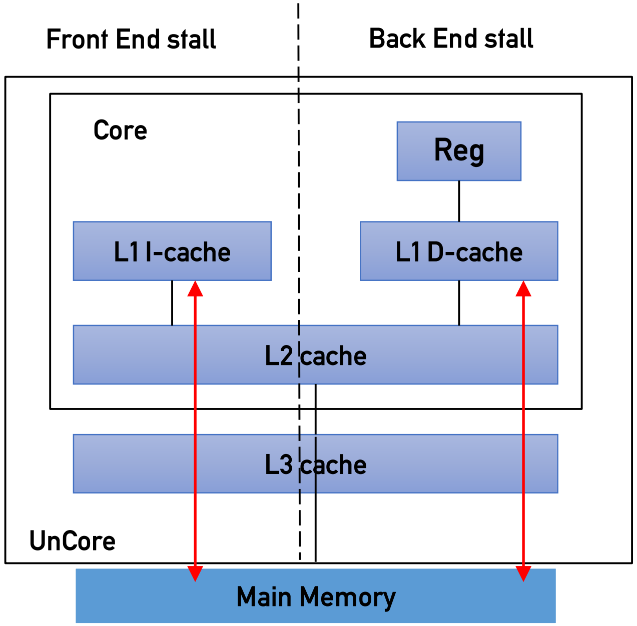
- Non-inclusive L3：支持Non-inclusive模式，Memory和L2间直接数据访问



芯片架构– Cache 时延

- 前端限制：主要是iCache Miss，iTLB Miss。指令解码能力不足也是原因之一，但可能性较低。
- 后端限制：主要是各级cache 包括dTLB miss。执行单元资源不足也是原因之一，但可能性较低。
- 错误分支预测：与Cache无关，但好的编码习惯能减少分支预测错误。

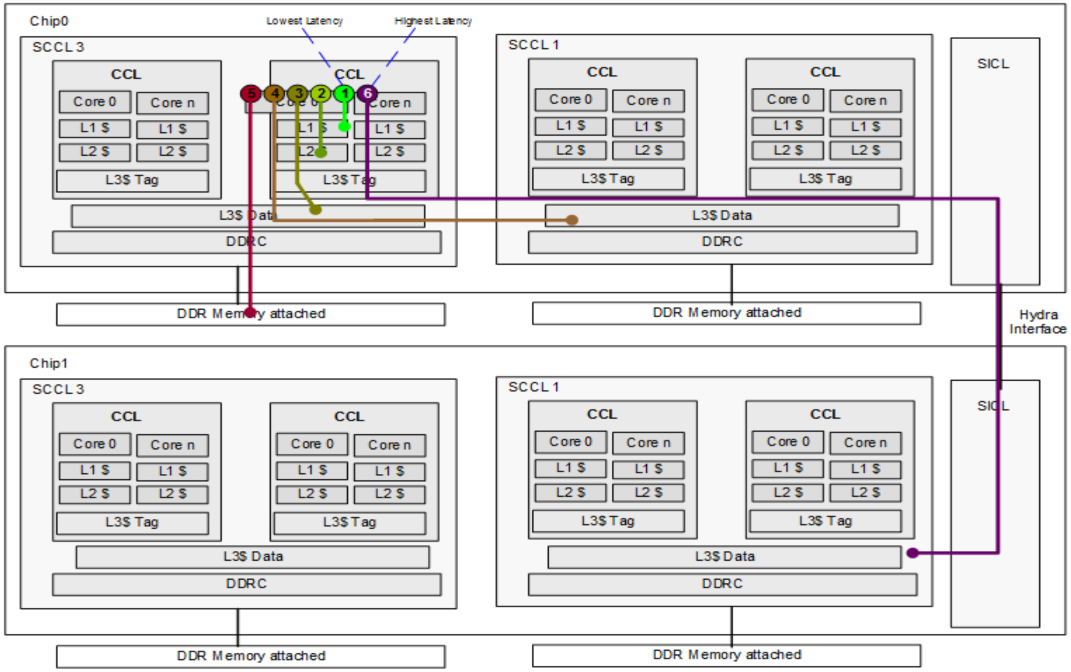
- 三类 Cache Misses
- Compulsory Misses
- 第一次读取数据时的 Cache miss。
- Capacity Misses
- 没有足够Cache空间存储所有热数据
- 在两次连续使用某一Cache数据之间，出现了太多的其它数据冲刷。
- Conflict Misses
- 太多（超过Associativity Way）不同数据映射到同一Cache Set中造成Cache碰撞。



鲲鹏920系列芯片架构——内存子系统

- **SMMU(System Memory Management System)**，为设备提供地址转换和访问保护功能。
- **CCL**访问的内存空间的属性由**MMU(memory management unit**，内存管理单元)中的页表控制
- **ICL**访问的内存空间的属性由**SMMU**中的页表或源设备控制
- 内存访问延迟受数据所在位置影响。如果目标数据位置在物理上接近内存访问发起者，则时延较低。

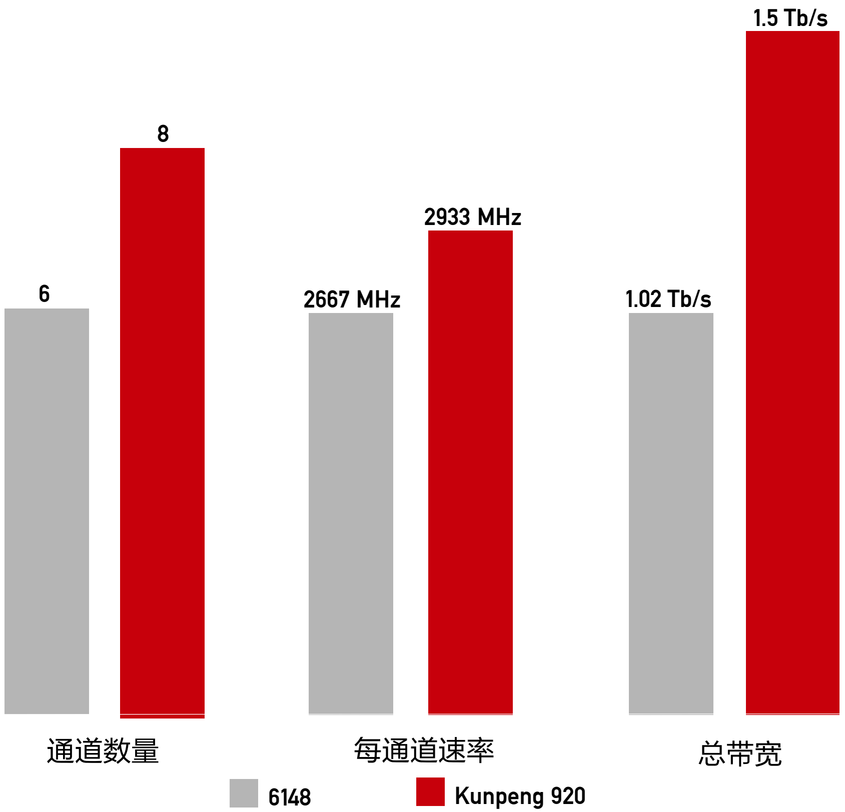
鲲鹏920系列芯片架构—内存子系统



- 一个CPU Die包含4个 DDR Channel
- 一个Socket包含2个CPU Die，8个DDR Channel
- 每个控制器支持2DPC 2933
- 本地内存访问均在本地进行，不走片间互联总线，因此访存时延最小，总体性能最好。

时延	ARM CPU Cycles	Skylake Cycles
Register	1	1
L1 cache	4	4
L2 cache	8	14
L3 cache	40	55
DRAM	71 -221(ns)	83-143(ns)

鲲鹏920系列芯片架构——内存子系统 (2)


内存带宽

- 8通道DDR4 带来46% 带宽提升，同时容量也可按需提升
- 延迟优化，和业界主流水平相当/更优
- 完整的Cache & Memory QoS 方案（类似于RDT），为用户的不同业务部署带来方便

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

Assignment 5

- Homework assignment

5.1, 5. 2, 5.3, 5.5, 5.6,5.7, 5.8

- To be submitted in the next week class