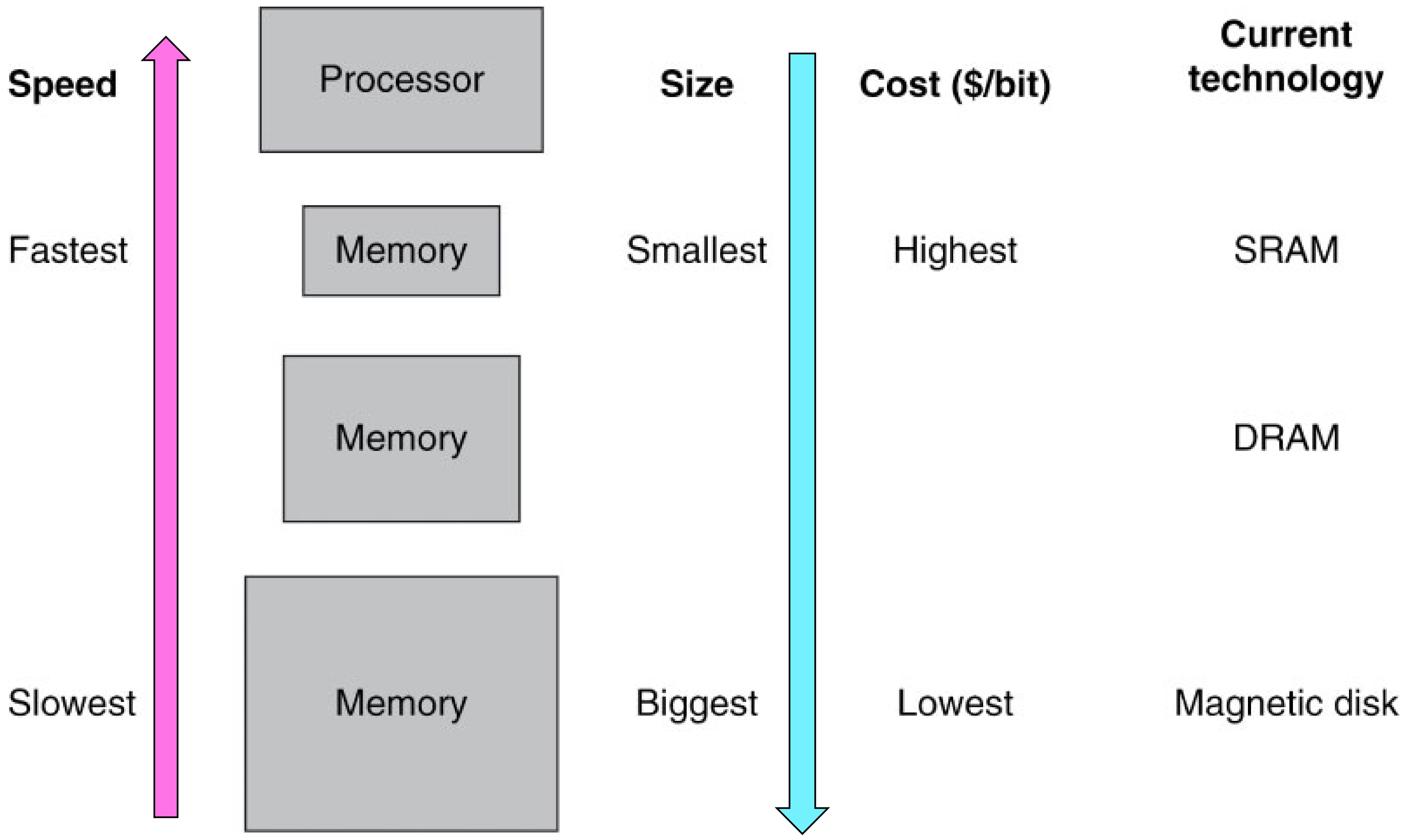
Chapter 5

Large and Fast: Exploiting Memory Hierarchy

Different Storage Memories



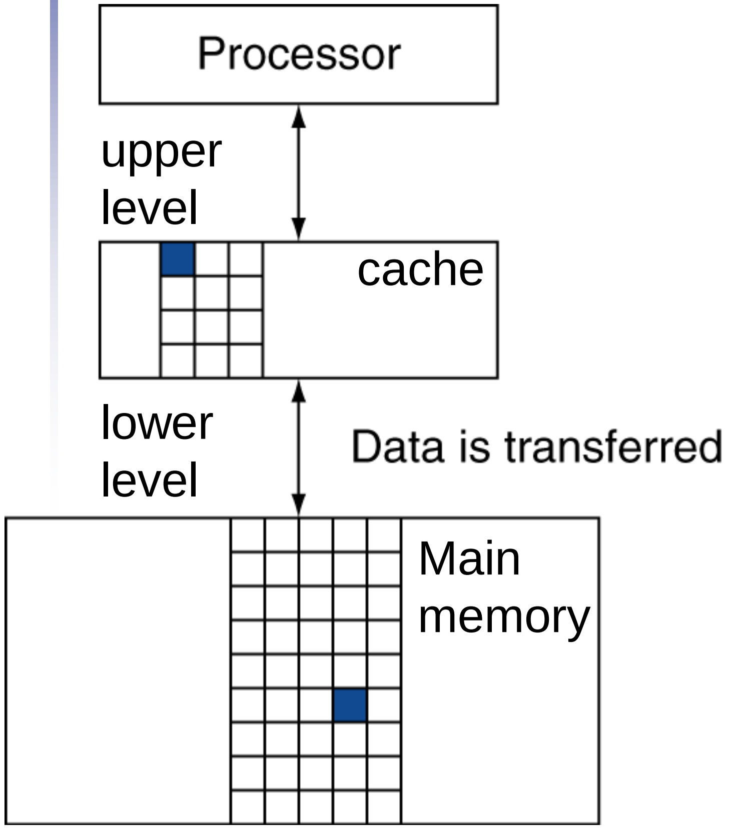
Principle of Locality

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

Taking Advantage of Locality

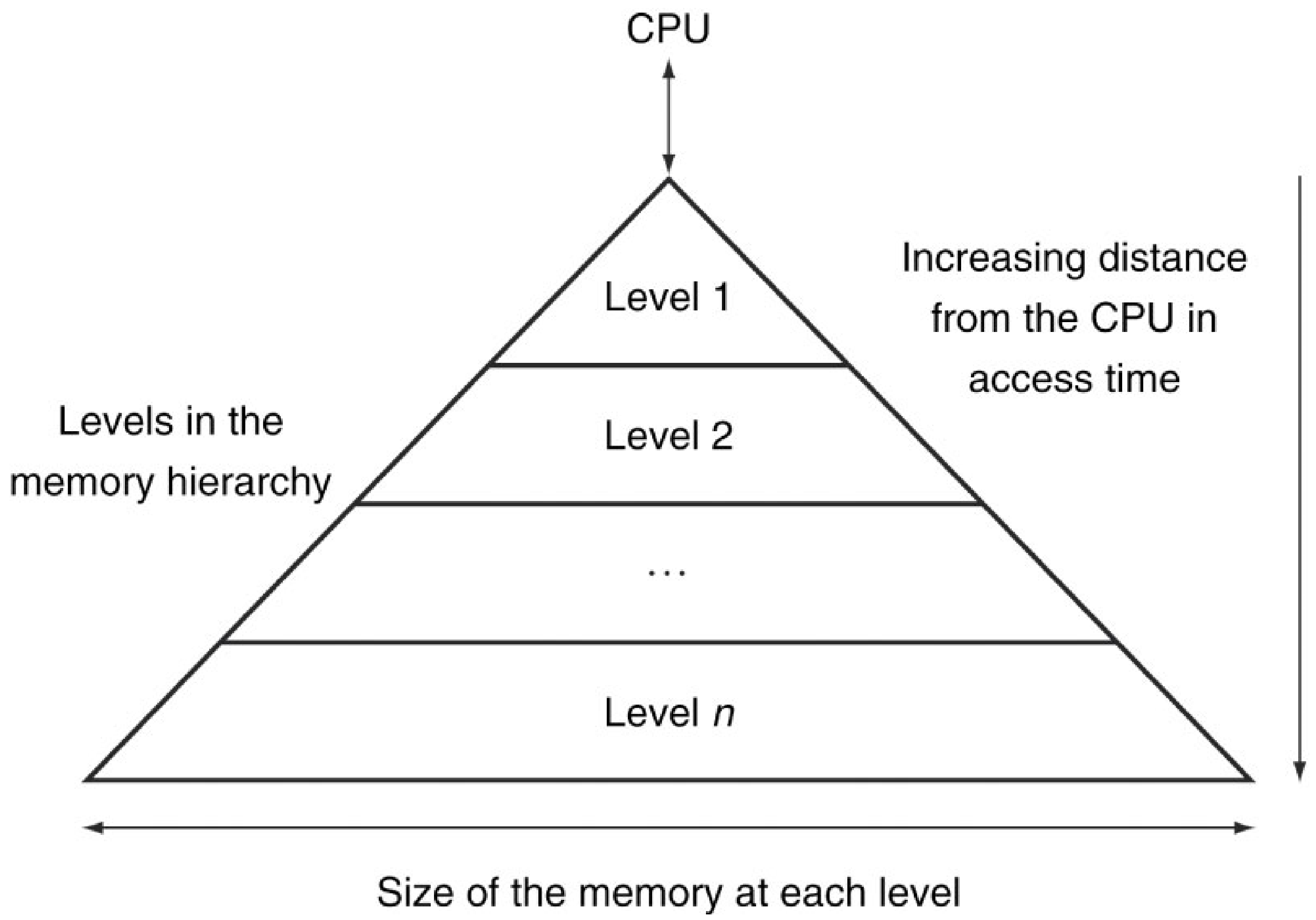
- Memory hierarchy
- Store everything on disk
  - Copy recently accessed (and nearby) items from disk to smaller DRAM memory
    - Main memory
    - Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
      - Cache memory attached to CPU

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

Structure of a Memory Hierarchy



CPU
Level 1
Level 2
. . .
Level *n*
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

Memory Technologies

- Static RAM (SRAM)
  - 0.5 – 2.5ns, $500 – $1000 per GB
- Dynamic RAM (DRAM)
  - 50 – 70ns, $10 – $20 per GB
- Flash memory
  - 5,000 – 50,000ns, $0.75 – $1.00 per GB
- Magnetic disk
  - 5,000,000 – 20,000,000ns, $0.05 – $0.10 per GB
- Ideal memory
  - Access time of SRAM (fast)
  - Capacity and cost/GB of disk (large)

Four Memory Technologies

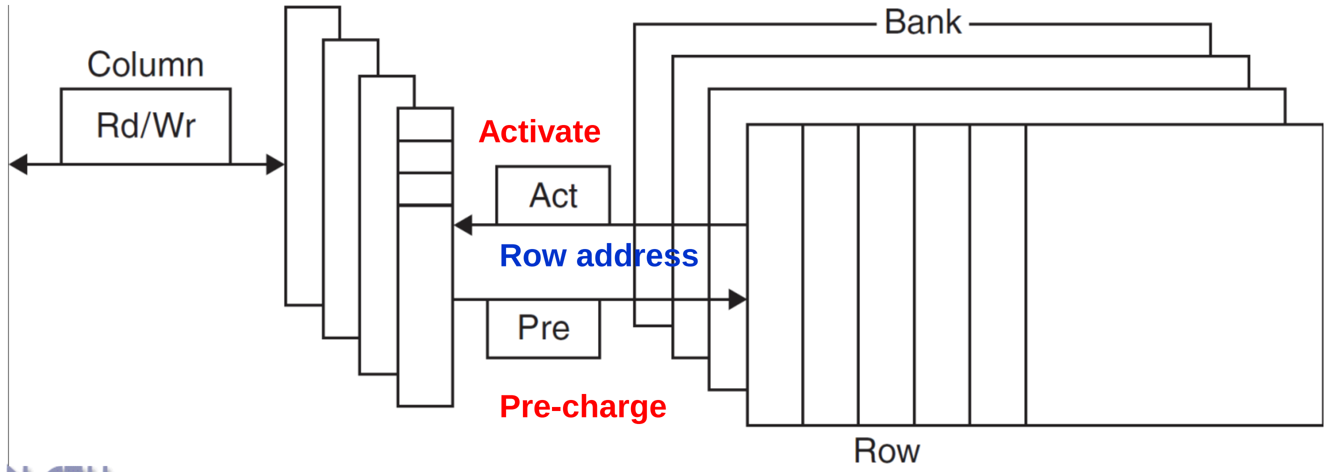
- DRAM (dynamic random access memory)
  - 1T per bit memory cell
  - Main memory
- SRAM (static random access memory)
  - ≥6T per bit memory cell
  - Cache levels closer to the processor
- Flash memory
  - Nonvolatile memory
  - Secondary memory in PMD
- Magnetic disk
  - The largest and slowest level in the memory hierarchy

SRAM Technology

- SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write.
  - Require minimal power to retain the charge in standby mode
    - 6-8 Transistors per bit
  - Do not need to refresh
  - The access time is almost the cycle time

DRAM Technology

- Data stored as a charge in a capacitor
  - A single transistor per bit
  - Must be periodically refreshed (i.e. read the content and write it back)
    - Performed on a DRAM "row"
    - Two-level decoding structure
      - Refresh an entire row (which shares a word line) with a read cycle followed by a write cycle



DRAM Performance Factors

- Row buffer
  - The buffer acts like an SRAM
  - Allows several words to be read and refreshed in parallel
  - Random bits can be accessed in the buffer until the next row access
- Synchronous DRAM
  - Allows for consecutive accesses in bursts without needing to send each address
  - Improves bandwidth
- DRAM banking
  - Allows simultaneous access to multiple DRAMs
  - Improves bandwidth

Advanced DRAM Organization

- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row 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
- Dual inline memory module (DIMM)
  - DIMMs typically contain 4-16 DRAMs

DRAM name based on Peak Chip Transfers / Sec
DIMM name based on Peak DIMM MBytes / Sec

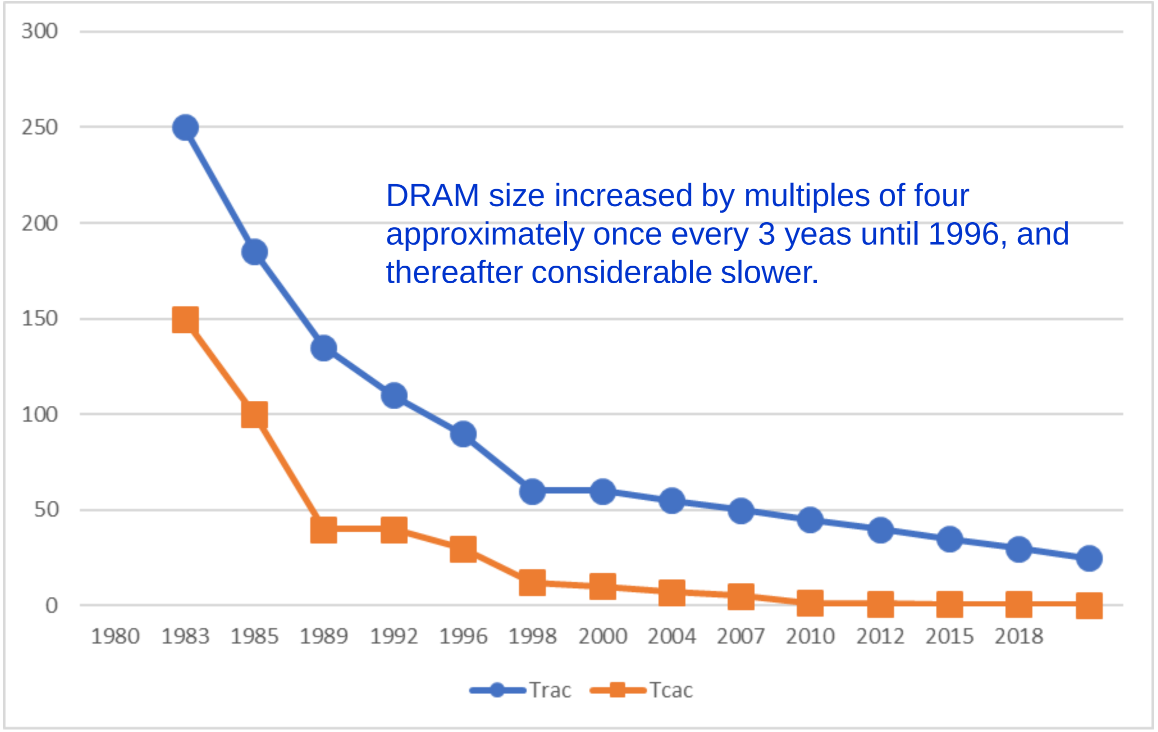
Standard	Clock Rate (MHz)	M transfers / second	DRAM Name	Mbytes/s/ DIMM	DIMM Name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400

x 2

x 8

DRAM Generations

Year	Capacity	$/GB
1980	64 Kibibit	$6,480,000
1983	256 Kibibit	$1,980,000
1985	1 Mebibit	$720,000
1989	4 Mebibit	$128,000
1992	16 Mebibit	$30,000
1996	64 Mebibit	$9,000
1998	128 Mebibit	$900
2000	256 Mebibit	$840
2004	512 Mebibit	$150
2007	1 Gibibit	$40
2010	2 Gibibit	$13
2012	4 Gibibit	$5
2015	8 Gibibit	$7
2018	16 Gibibit	$6

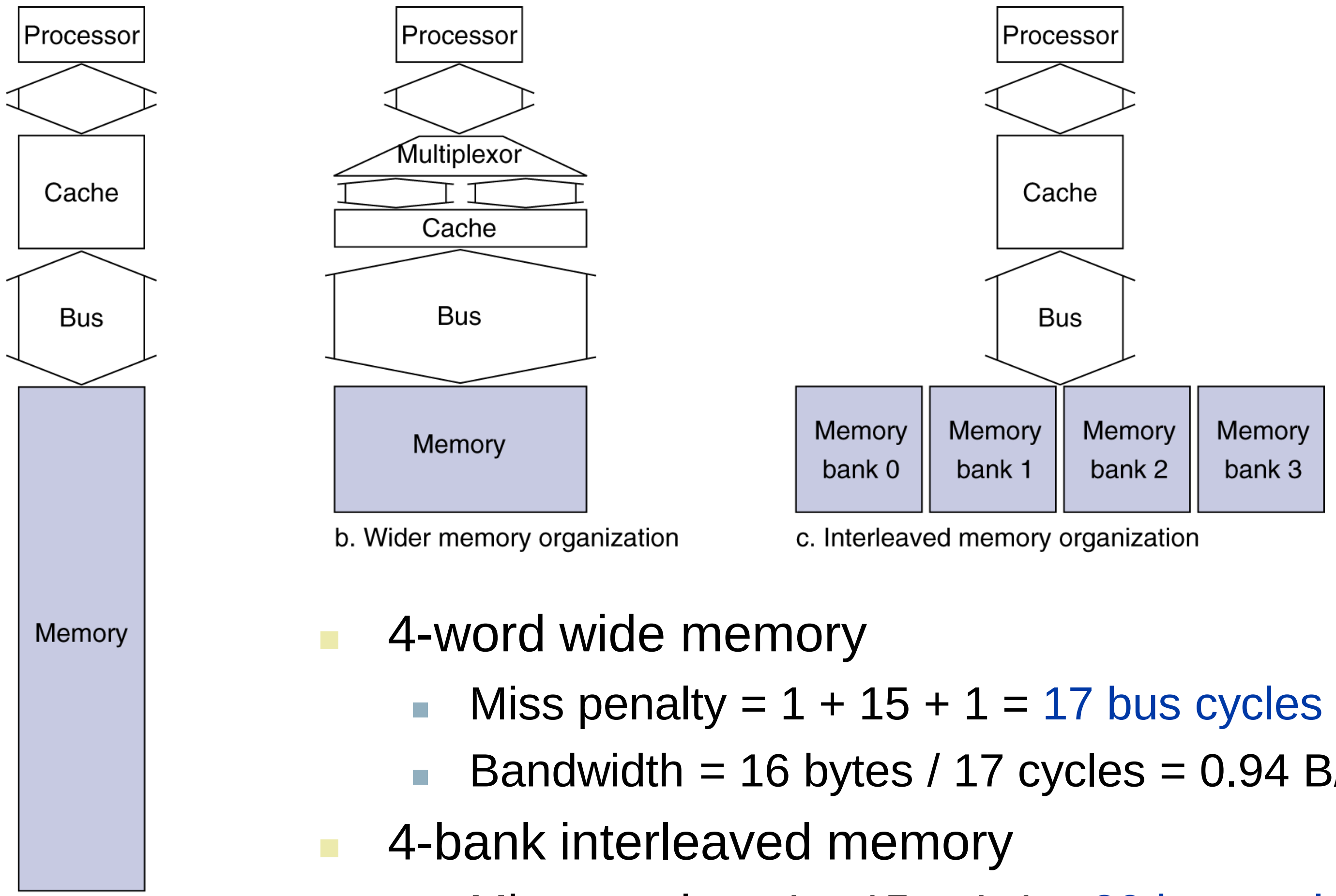


DRAM size increased by multiples of four approximately once every 3 yeas until 1996, and thereafter considerable slower.

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 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

Increasing Memory Bandwidth



a. One-word-wide memory organization

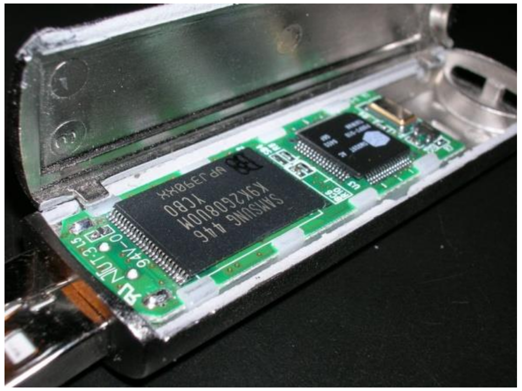
b. Wider memory organization

c. Interleaved memory organization

- 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

Flash Memory

- A type of electrically erasable programmable read-only memory (EEPROM)
- Non-volatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)

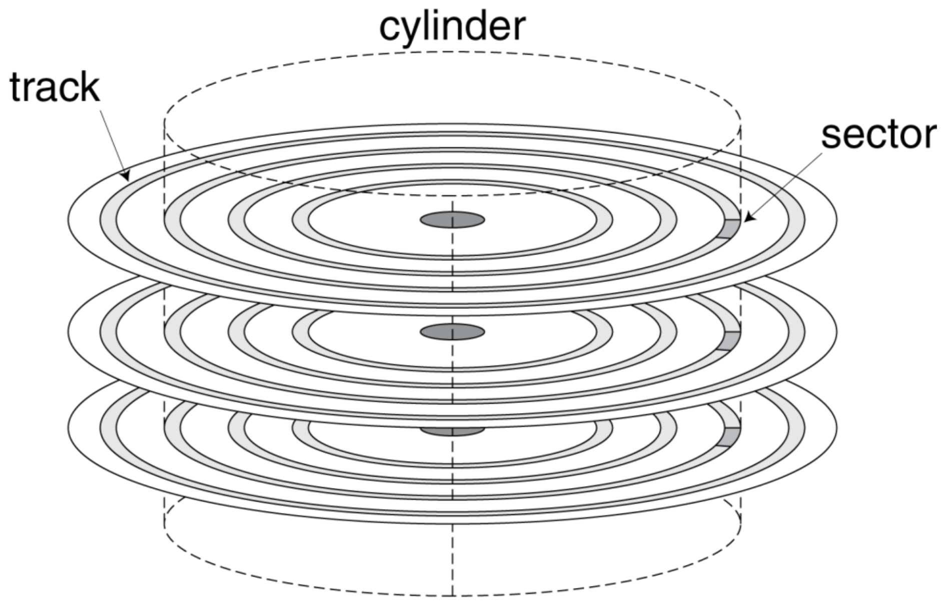


Flash Types

- NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …
- Flash bits wears out after (usually) 1000's of accesses
  - Not suitable for direct RAM or disk replacement
  - *Wear leveling*: remap data to less used blocks
- Error control code (ECC)
  - BCH, LDPC

Disk Memory

- Nonvolatile, rotating magnetic storage



Disk Sectors and Access

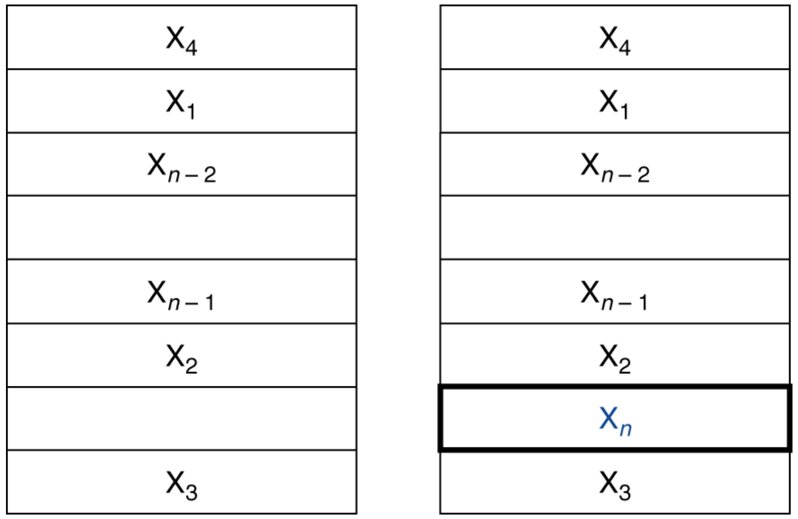
- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead

Disk Access Example

- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - 4ms seek time
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms
- If actual average seek time is 1ms
  - Average read time = 3.2ms

Cache Memory

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



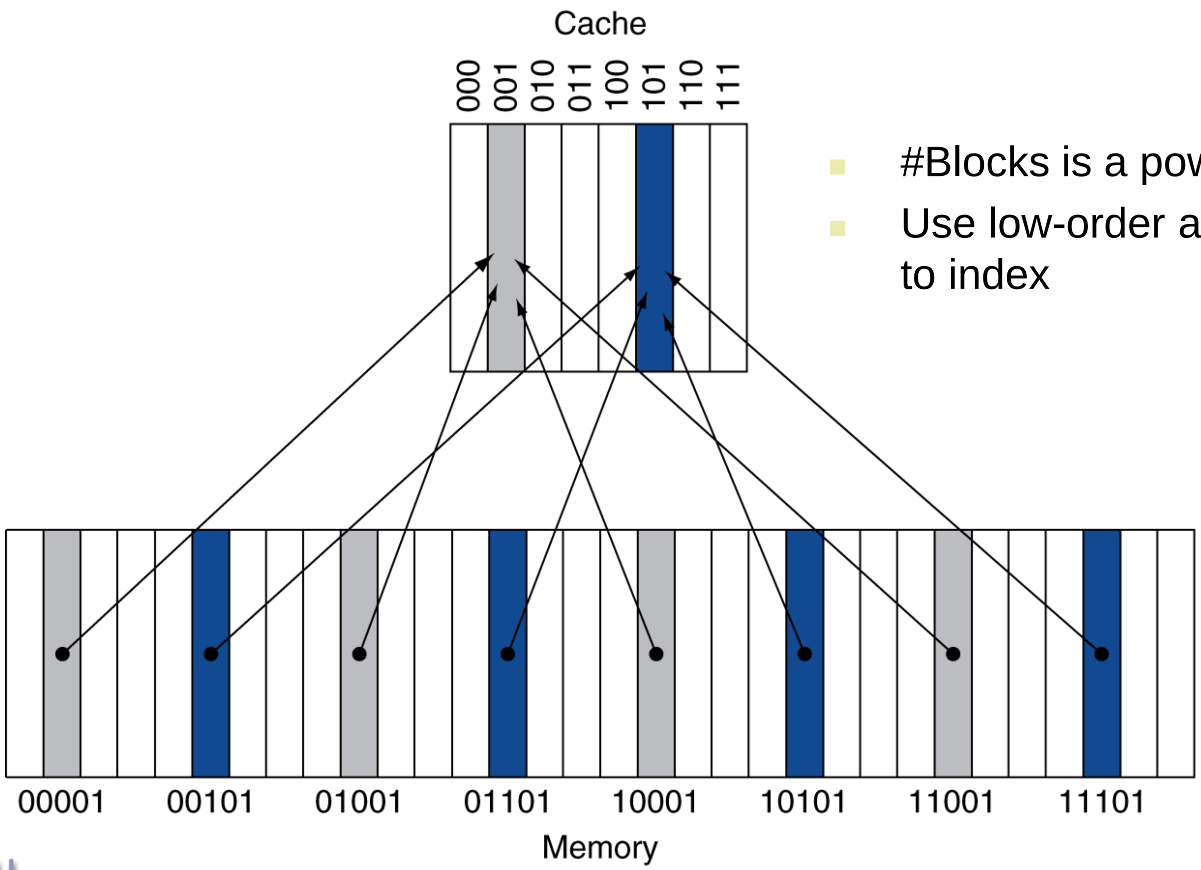
a. Before the reference to X_n b. After the reference to X_n

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

Location…. Search….

Direct Mapped Cache

- Location determined by address
- Direct mapped: one and only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits to index

Tags and Valid Bits

- Tags ?
  - 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*) to identify
- Valid ?
  - 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

Low-order bits

high-order bits

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

Cache Example (1/5)

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 (2/5)

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 (3/5)

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 (4/5)

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 (5/5)

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		

Total Number of Bits for a Cache

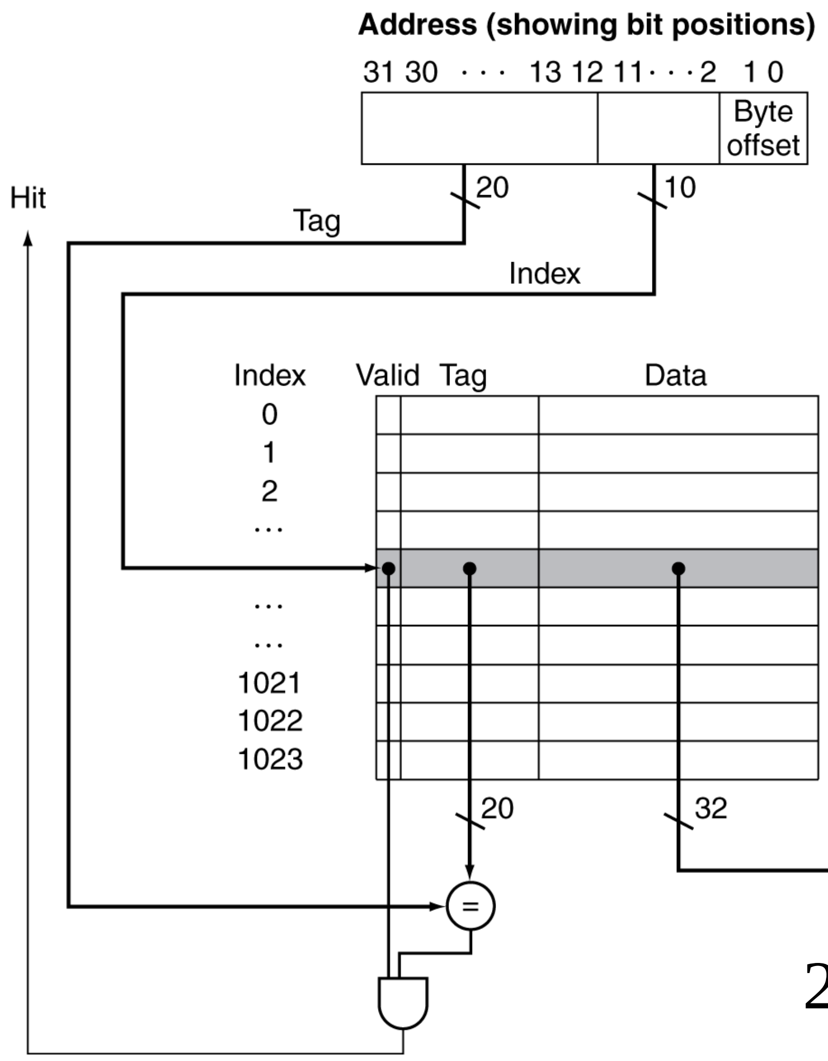
- A function of the cache size and the address size
- Direct-mapped cache example:
  - 32-bit addresses
  - 2^n block cache
  - Block size: 2^m words (or 2^{m+2} bytes or 2^{m+5} bits)

  ➔ Tag field requires 32–(n+m+2) bits

  ➔ Total number of bits for a direct-mapped cache is $2^n \times$ (block size + tag size + valid bit)

  ➔ $2^n \times (2^{m+5} + (32 - n - m + 2) + 1)$

Address Subdivision



Block size: 1 word

Total cache size=?

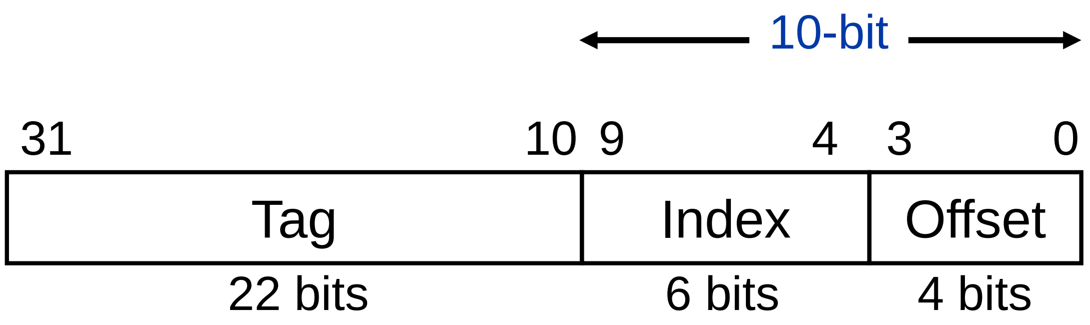
$$2^{10} \times (32 + 20 + 1) \text{ bits}$$

Larger Block Size Example

- 64 blocks, 16 bytes/block

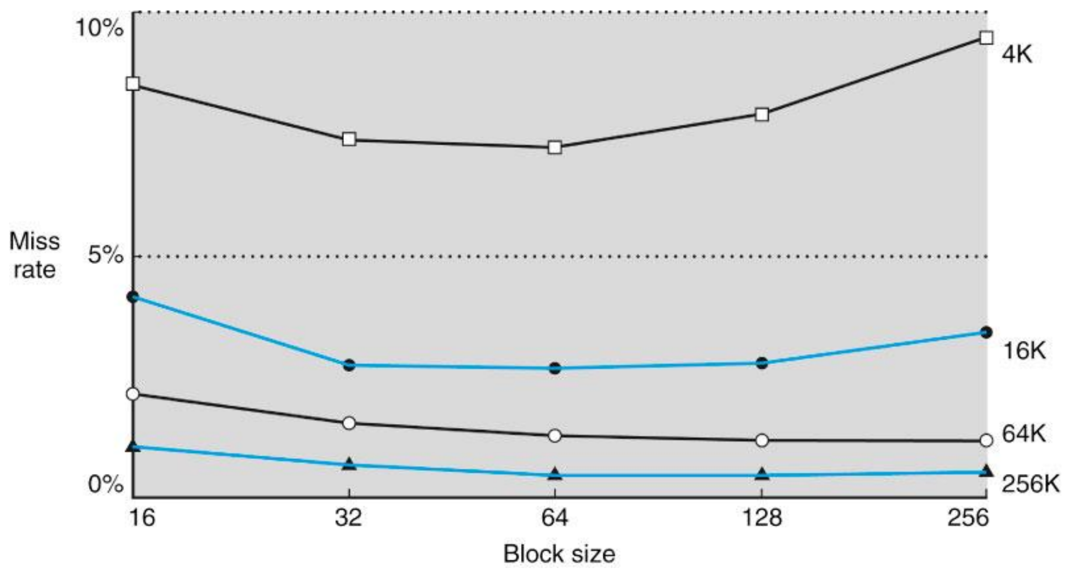
  $64\times16=2^{10}$

  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor$ = 75
- Block number = 75 modulo 64 = 11



$$2^{10}\times(4\times32+(32-10)+1)\ \text{bits}$$

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
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

Handling 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
- Instruction cache miss example
  - Send the original PC (i.e. current PC–4) to the memory
  - Instruct main memory to perform a read and wait for the memory to complete its access
  - Write the cache entry (putting the data from memory, writing the tag field, and turning the valid bit on)
  - Restart the instruction fetch

Write-Through Cache

- 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 Cache

- 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

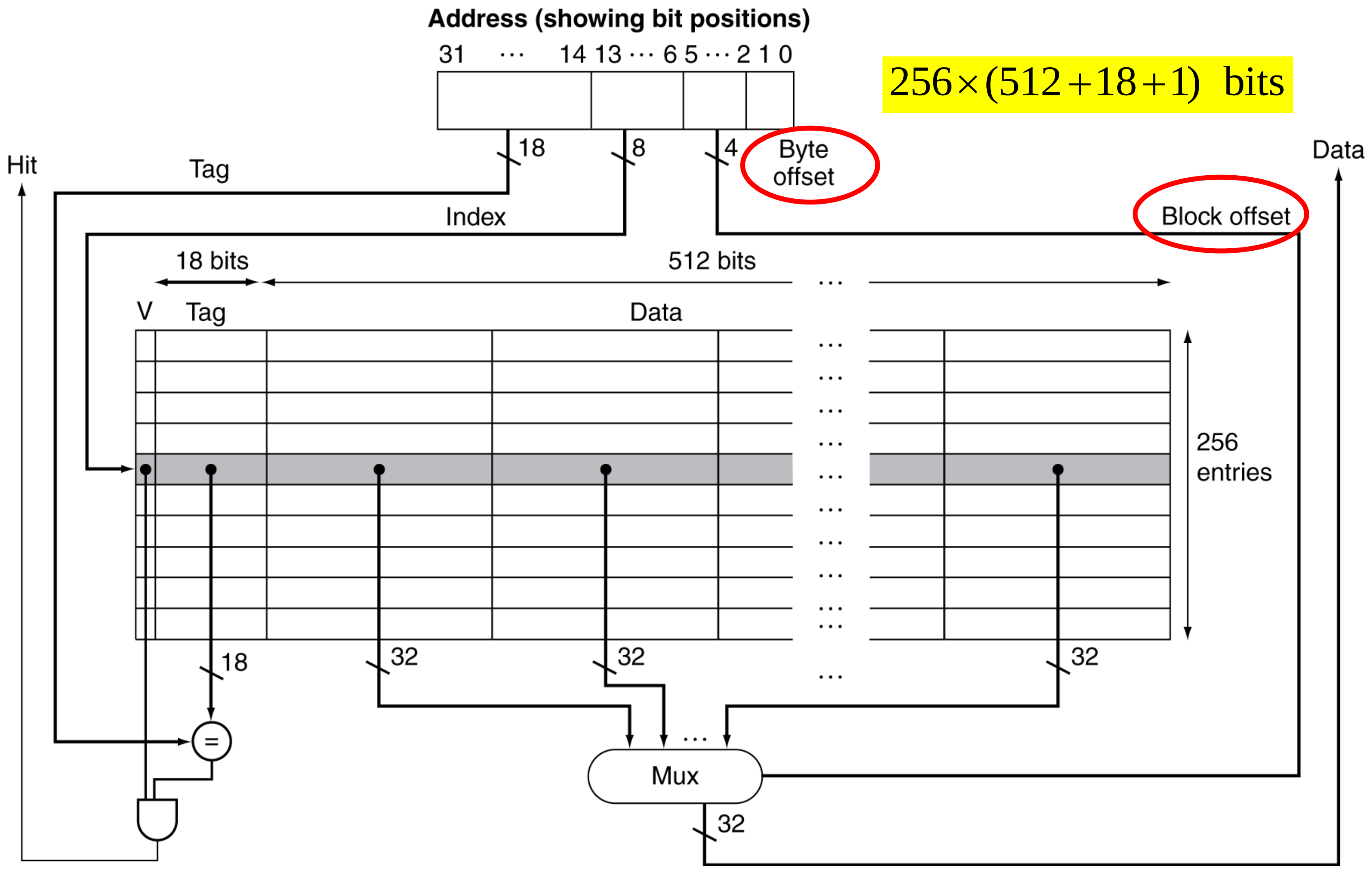
- *What should happen on a write miss?*
- Alternatives for write-through
  - Allocate on miss: fetch the block (then overwrite it)
  - Write around: don’t fetch the block (but update the portion of the block in memory)
    - Since programs often write a whole block before reading it (e.g., initialization)
- 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
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH

$256 \times (512+18+1)$ bits



Measuring Cache Performance

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

$$\boxed{\begin{aligned}&\text{Memory stall cycles}\\&= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}\\&= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\end{aligned}}$$

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

Performance Summary

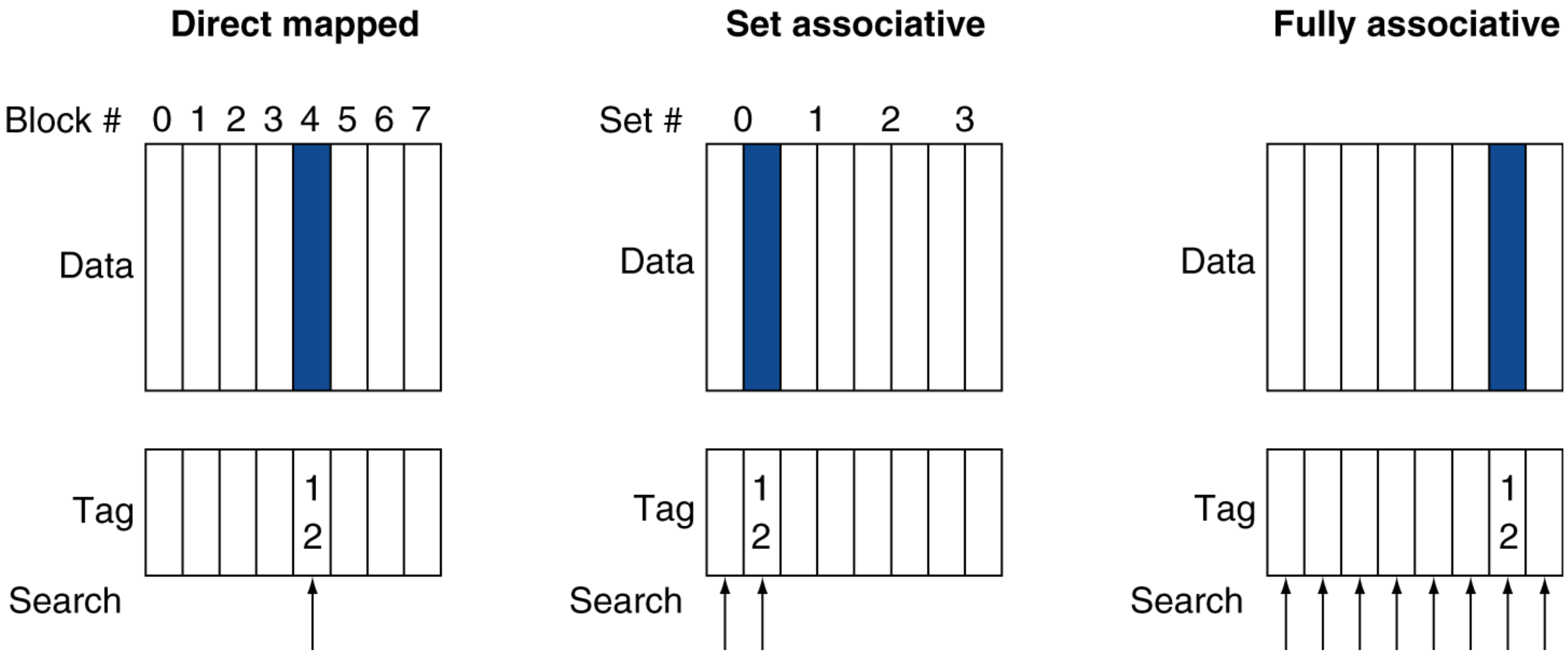
- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can’t neglect cache behavior when evaluating system performance

Associative Caches

- Fully associative caches
  - 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 caches
  - 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)
- 1-way set associative cache
  - Direct-mapped cache

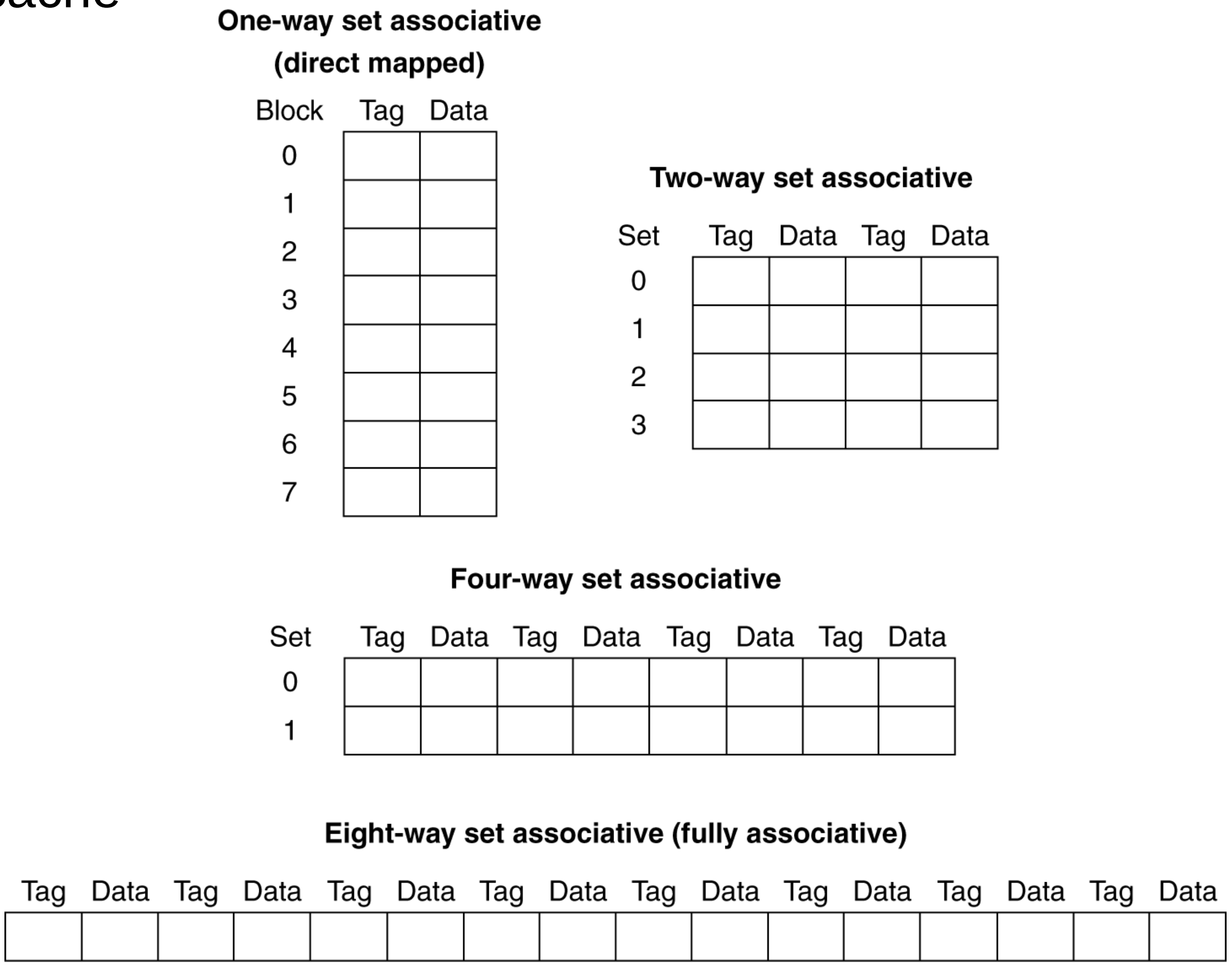
Associative Cache Example

- The location of a memory block whose block address is 12 in a cache with 8 blocks varies for direct-mapped, set-associative, and fully associative placement



Spectrum of Associativity

- 8-block cache



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

Example: Misses and Associativity in a Cache (1/2)

- Compare three 4-block caches: direct mapped, 2-way set associative, and fully associative caches
  - 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]	

Example: Misses and Associativity in a Cache (2/2)

- 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]		

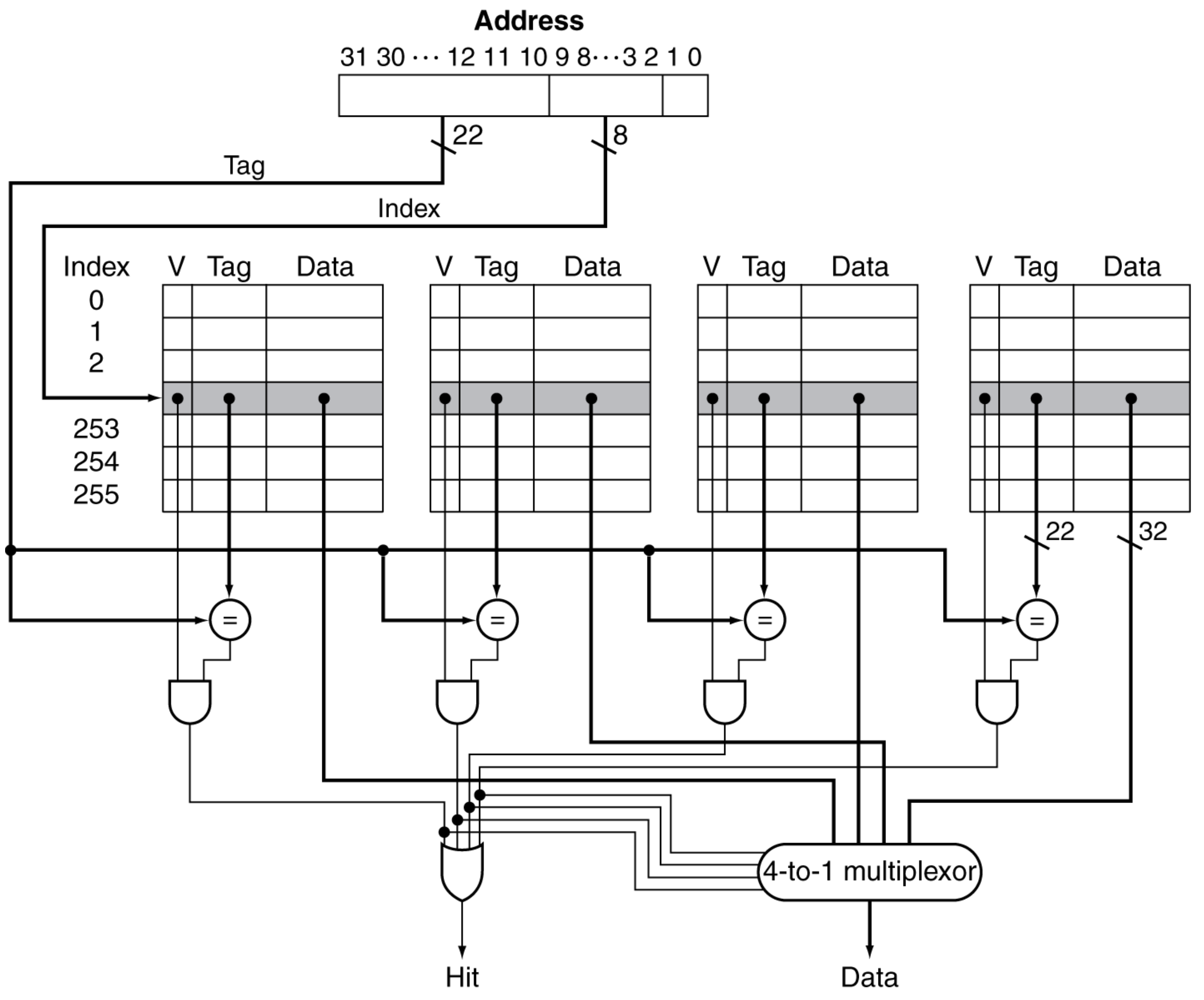
- Fully associative

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 serves misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory serves (L1-cache miss) && (L-2 cache miss) (or called global miss)

- 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 $\times$ 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 $\times$ 20 + 0.005 $\times$ 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

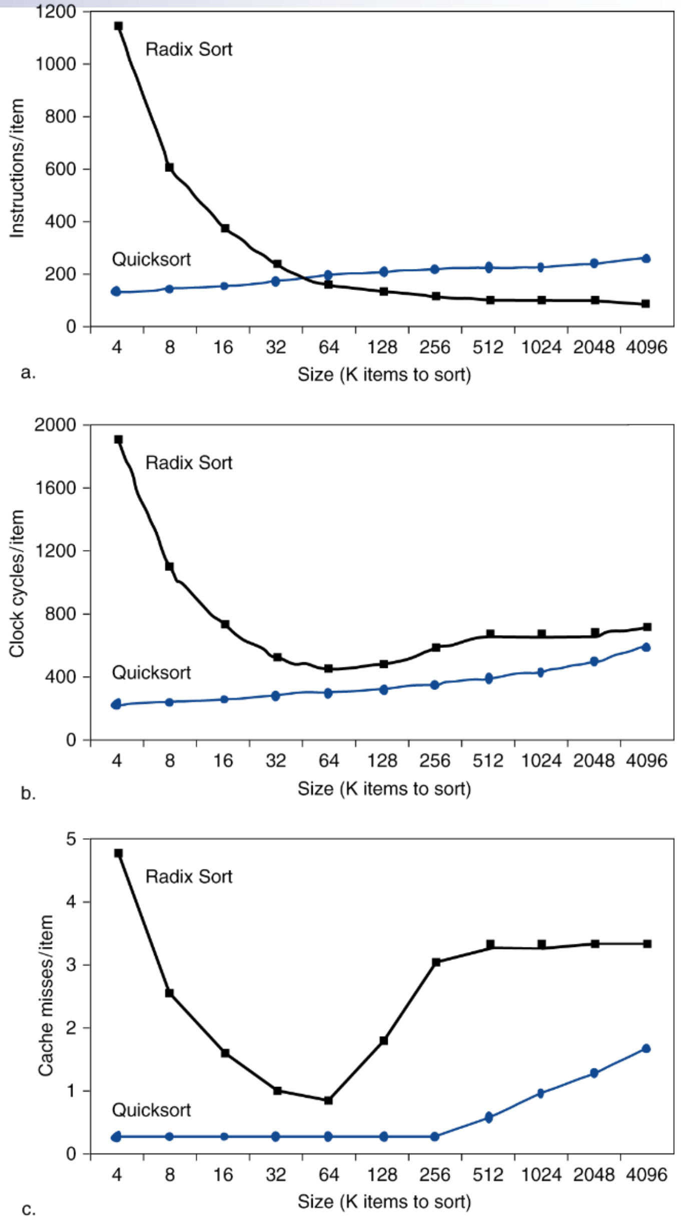
- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

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? Quicksort?
  - For larger arrays, radix-sort has an algorithmic advantage over quicksort in terms of number of operations
  - Quicksort has fewer misses per item to be sorted
- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access



Software Optimization via Blocking

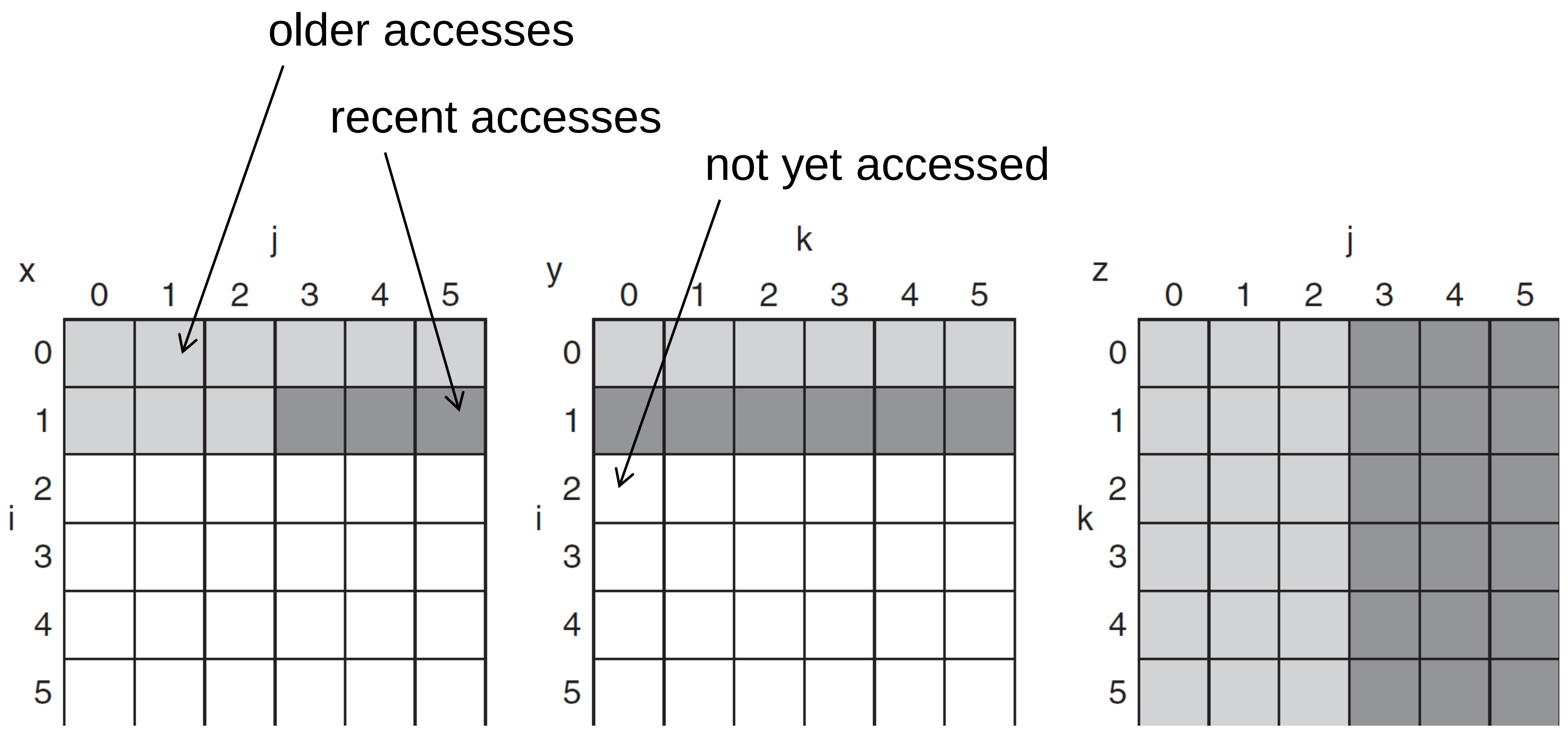
- Goal: maximize accesses to data before it is replaced
- Reuse the data with the cache to lower miss rate
- Consider inner loops of DGEMM:

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

It reads all n-by-n elements of B, reads the same n elements in what corresponds to one row of A repeatedly, and writes what corresponds to one row of n elements of C

DGEMM Access Pattern

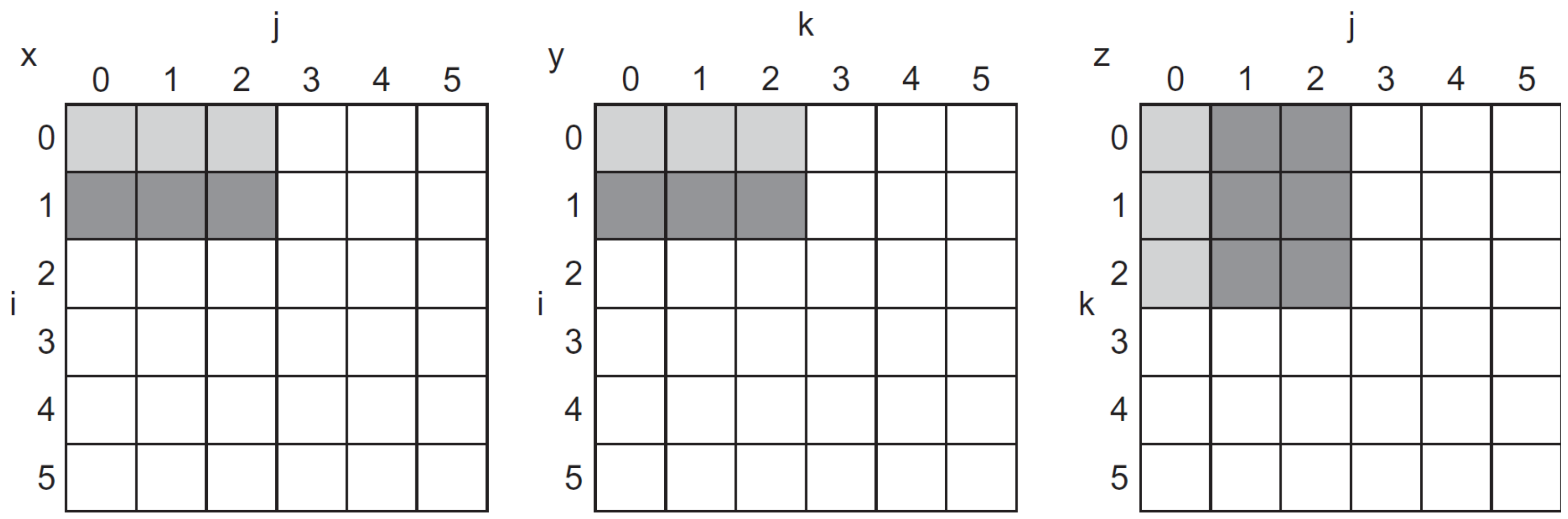
- A snapshot of the three arrays C, A, and B when n=6 and i=1

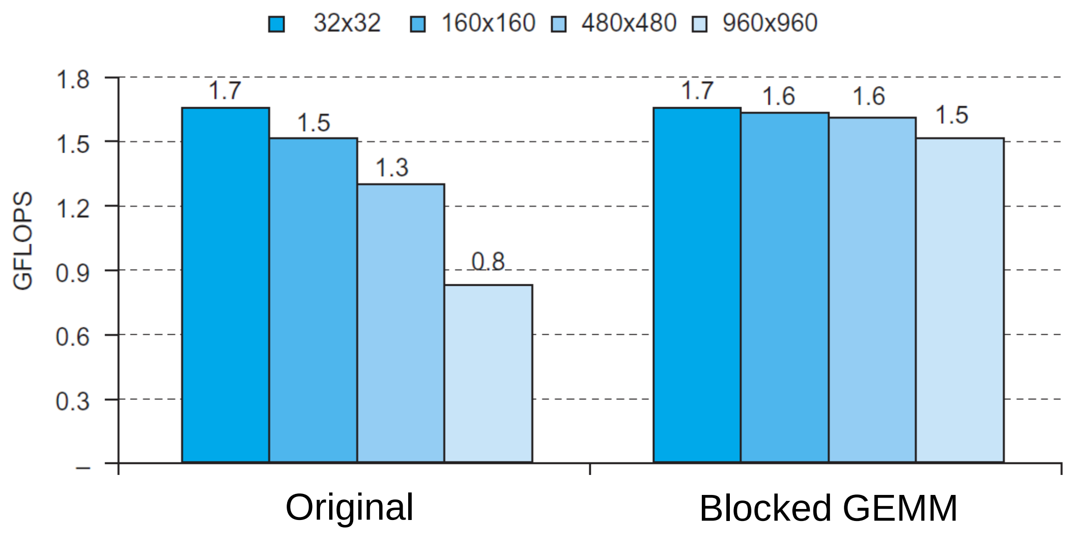


Cache Blocked DGEMM

```
#define BLOCKSIZE 32
void do_block (int n, int si, int sj, int sk, double *A, double
*B, double *C)
{
 for (int i = si; i < si+BLOCKSIZE; ++i)
  for (int j = sj; j < sj+BLOCKSIZE; ++j)
  {
   double cij = C[i+j*n];/* cij = C[i][j] */
   for( int k = sk; k < sk+BLOCKSIZE; k++ )
    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
   C[i+j*n] = cij;/* C[i][j] = cij */
  }
 }
 void dgemm (int n, double* A, double* B, double* C)
 {
  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
   for ( int si = 0; si < n; si += BLOCKSIZE )
    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
     do_block(n, si, sj, sk, A, B, C);
 }
```

Blocked DGEMM Access Pattern





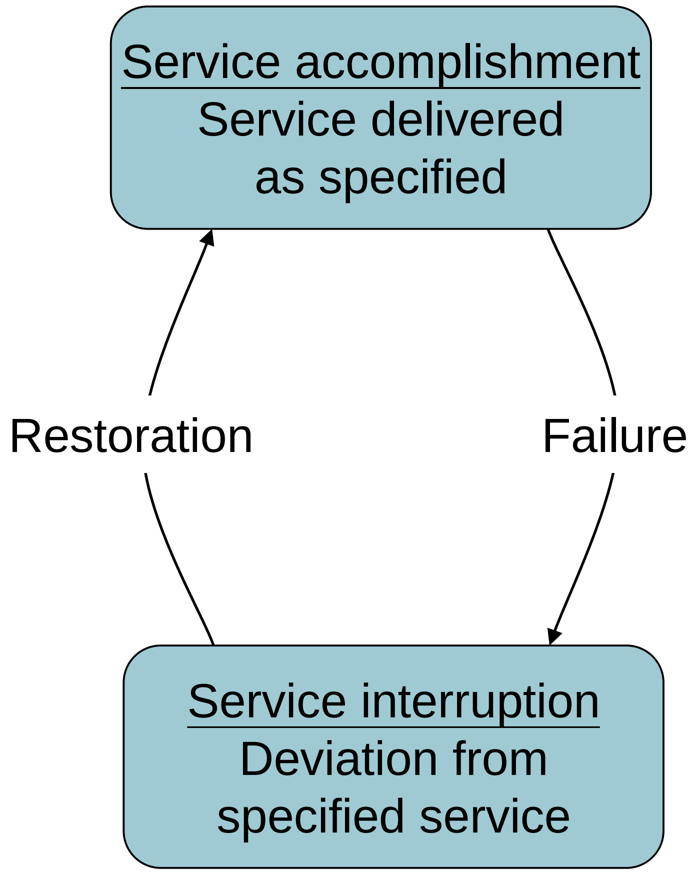
Summary for Memory Hierarchy

- The number of memory-stall cycles depends on both the miss rate and the miss penalty
  - Using associativity to reduce miss rates
    - Allow more flexible placement of blocks within the cache
  - Using multilevel cache hierarchies to reduce miss penalties
    - Allow a larger secondary cache to handle misses to the primary cache.
  - Using software optimization to improve effectiveness of caches
    - Change algorithm (e.g. with blocking technique to deal with a large array) to improve cache behavior

Dependable Memory Hierarchy

- Two-state system model:

Dependability is redundancy !!


Service accomplishment
Service delivered as specified
Restoration
Failure
Service interruption
Deviation from specified service

- Fault: failure of a component
    - May or may not lead to system failure

Dependability Measures

- Reliability: mean time to failure (MTTF)
  - A measure of the continuous service accomplishment from a reference point
- Service interruption: mean time to repair (MTTR)
- Mean time between failures (MTBF)
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

The Hamming SEC Code

- Hamming distance
  - Number of bits that are different between two bit patterns
- Minimum distance = 2 provides single bit error detection
  - E.g. parity code
- Minimum distance = 3 provides single error correction, 2 bit error detection

Hamming SEC Encoding

- To calculate Hamming code:
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks certain data bits:

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

  - Set parity bits to create even parity for each group

Decoding SEC

- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
  - E.g.
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped

SEC/DED Code

- Add an additional parity bit for the whole word (p_n)
- Make Hamming distance = 4
- Decoding:
  - Let H = SEC parity bits
    - H even, p_n even, no error
    - H odd, p_n odd, correctable single bit error
    - H even, p_n odd, error in p_n bit
    - H odd, p_n even, double error occurred
- Note: ECC DRAM uses SEC/DED with 8 bits protecting each 64 bits

Virtual Machines (VMs)

- 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 with modern high-performance computers
- Examples
  - IBM VM/370 (1970s technology!)
  - VMWare
  - Microsoft Virtual PC

Virtual Machine Monitor (VMM)

- The software that supports VMs is called a VMM
  - The VMM is much smaller than a traditional OS
- Maps virtual resources to physical resources
  - Memory, I/O devices, CPUs
- Guest OS may be different from host OS
- Guest code runs on native machine in user mode
  - Traps to VMM on privileged instructions and access to protected resources
- 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

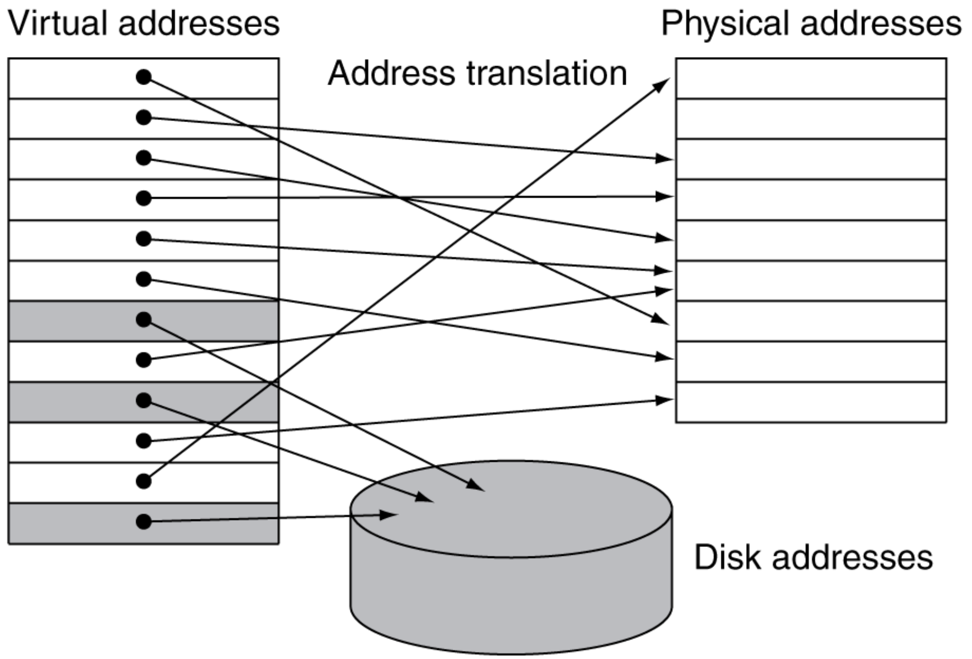
Instruction Set Support for VMM

- Requirements for a VMM
  - It presents a software interface to guest software
  - It isolates the state of guests from each other
  - It protect itself from guest software
- At least two processor modes in ISA:
  - User mode and System mode
- 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

Virtual Memory

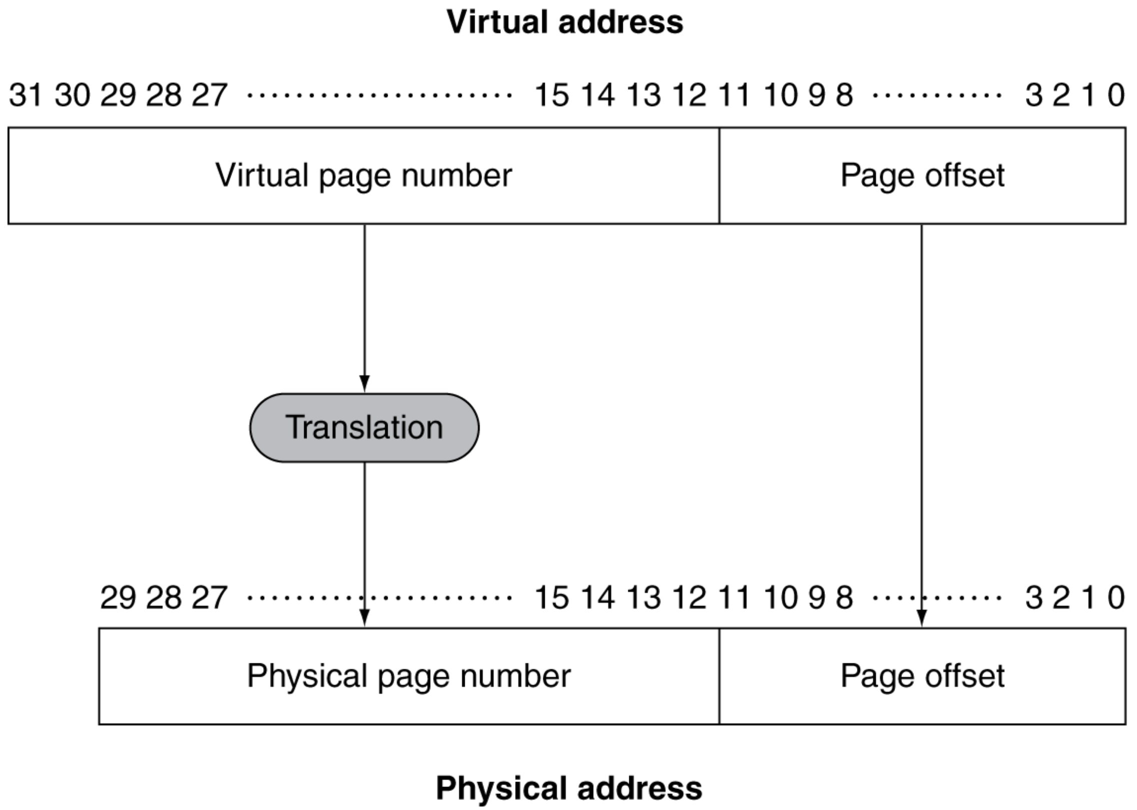
- Use main memory as a "cache" for secondary (disk) storage (➔ main memory is not enough…)
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM "block" is called a page
  - VM translation "miss" is called a page fault

Virtual Address vs Physical Address



- The processor generates virtual addresses, while the memory is accessed using physical addresses.
- Both the virtual memory and physical memory are broken into pages.
  - A virtual page is mapped to a physical page
  - It is possible for a virtual page to be absent from main memory (i.e. not be mapped to a physical addresses). In this case, the page resides on disk or flash memory
  - A physical page can be shared by having two virtual addresses point to the same physical address

Address Translation Example



- The page size is 2^{12} = 4 KB
- The number of physical pages is 218
- Hence, main memory can have at most 1GB, where the virtual address space is 4GB

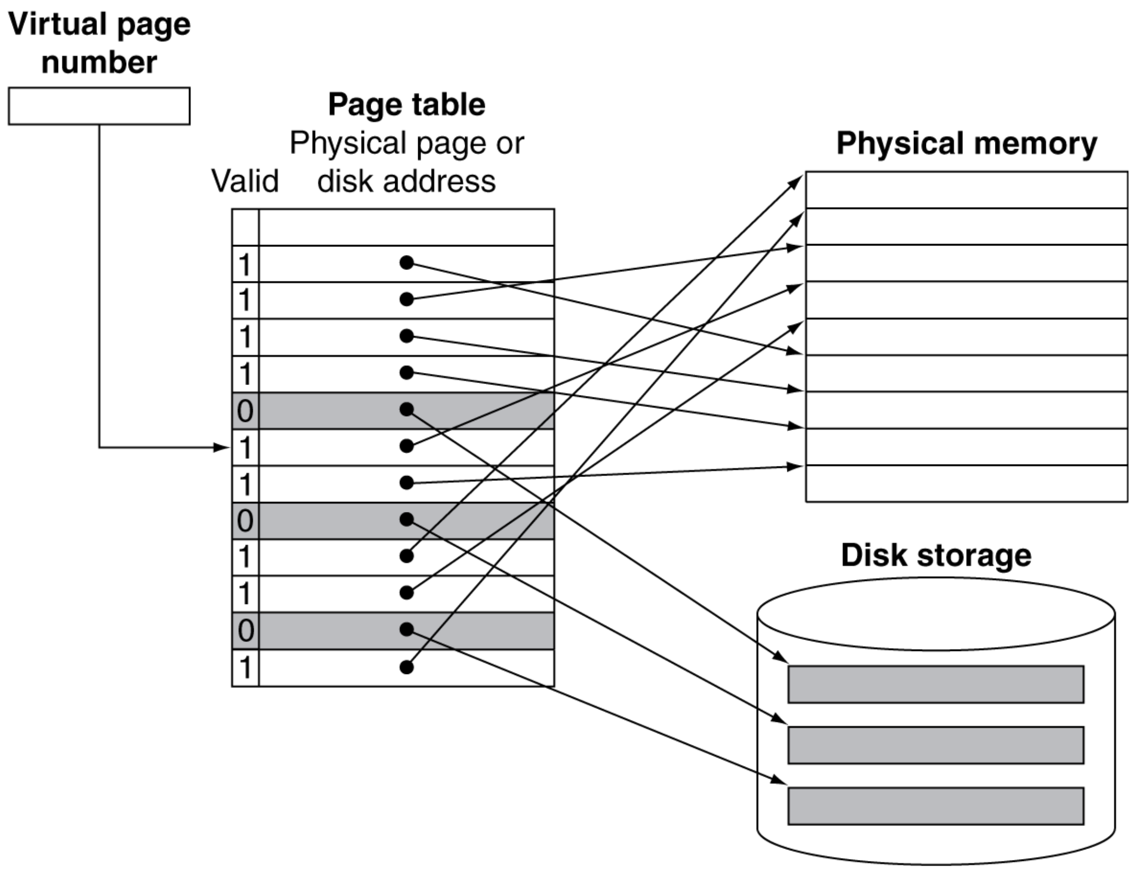
Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Page should be large enough to try to amortize the high access time
    - 4KB – 16KB per page are typical today
  - Takes millions of clock cycles
  - Handled by OS code
- Try to minimize page fault rate
  - Fully associative placement
  - Smart LRU replacement algorithms
- Write through will not work for virtual memory

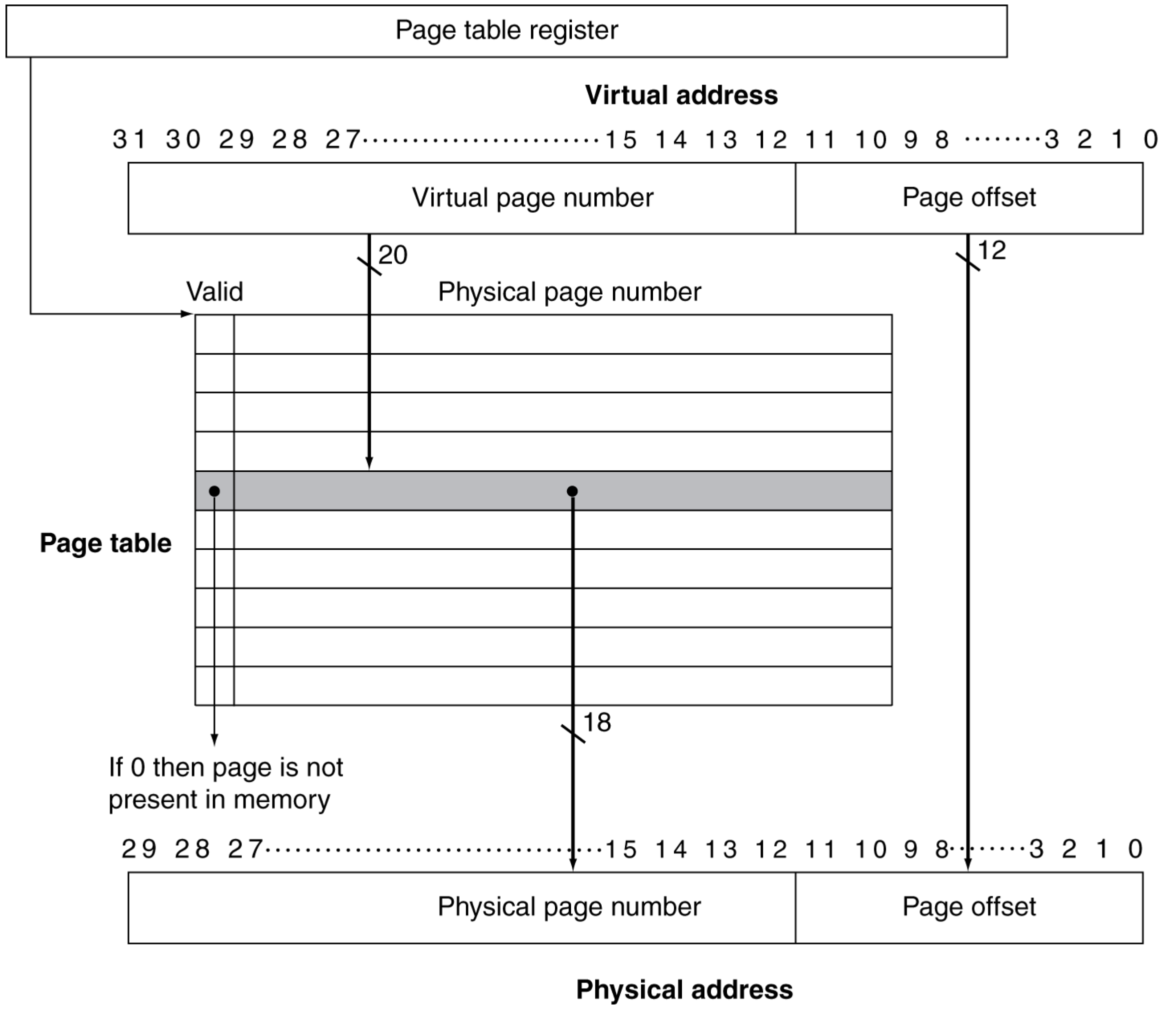
Page Table

- Full search for fully associative placements is impractical in VM. Instead, it locate pages by using a page table.
- Page table stores placement information
  - Array of page table entries (PTEs) is indexed by virtual page number
- Page table resides in main memory
  - Page table register is used in CPU pointing to the page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (valid, referenced, dirty, …)
- If page is not present
  - PTE can refer to location in swap space on disk

Mapping Pages to Storage



Translation Using a Page Table



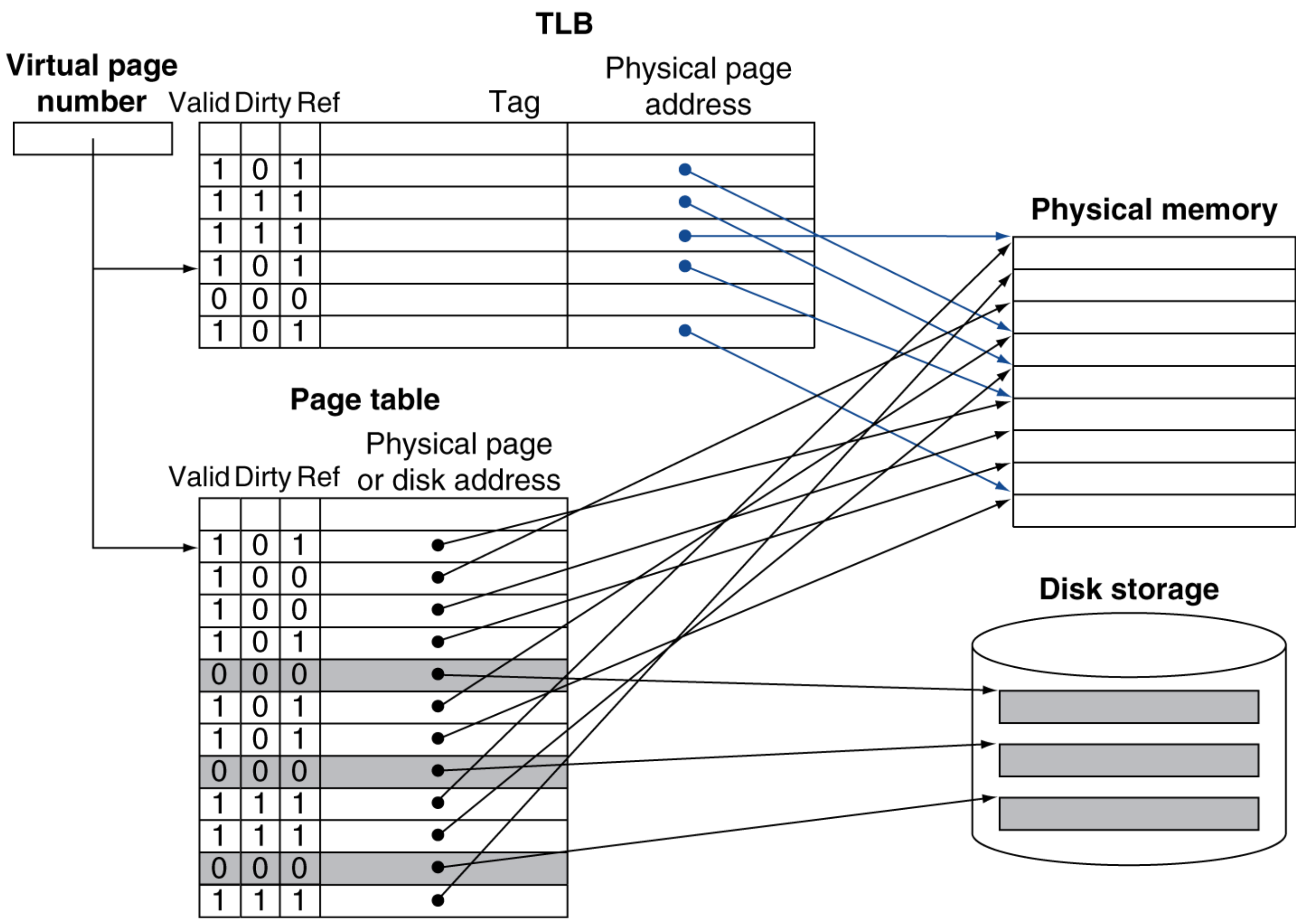
Replacement and Writes

- 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
  - 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 (at least twice) memory references
  - One to access the PTE
  - Then the actual memory access
- But access to page tables has temporal and spatial 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 a miss in the TLB occurs, we must determine whether it is a page fault or merely a TLB miss.
- If the 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 the page is not in memory (a true page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction
- TLB misses will be much more frequent than true page faults.

TLB Miss Handler

- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not present
- 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

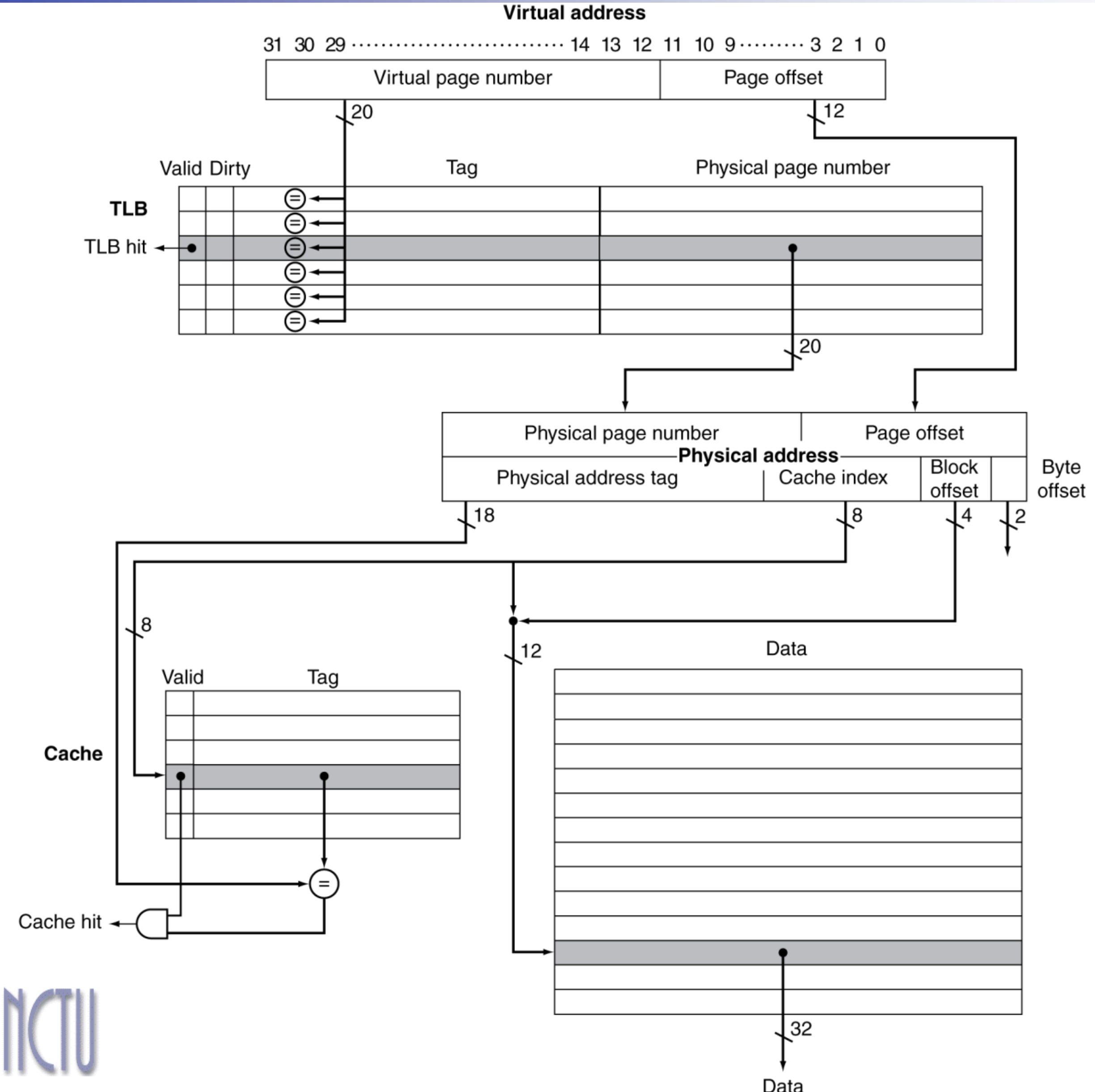
Page Fault Handler

- 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

Why Virtual Address Cache?

- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Faster, but complications due to aliasing
    - Different virtual addresses for shared physical address

TLB in Intrinsity FastMATH



Integrating TLB, VM, and Cache

- Data cannot be in the cache unless it is present in main memory
  - A virtual address is translated by the TLB and sent to the cache where the appropriate data is found, retrieved, and sent back to the processor
  - All possible combinations:

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

  - Worst case: miss in all three components of the memory hierarchy

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)

Summary: The Memory Hierarchy

The BIG Picture

- Caches, TLBs, and VMs may initially look very different, but common principles apply at all levels of the memory hierarchy
- Four questions at each level in the memory 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
  - But, 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

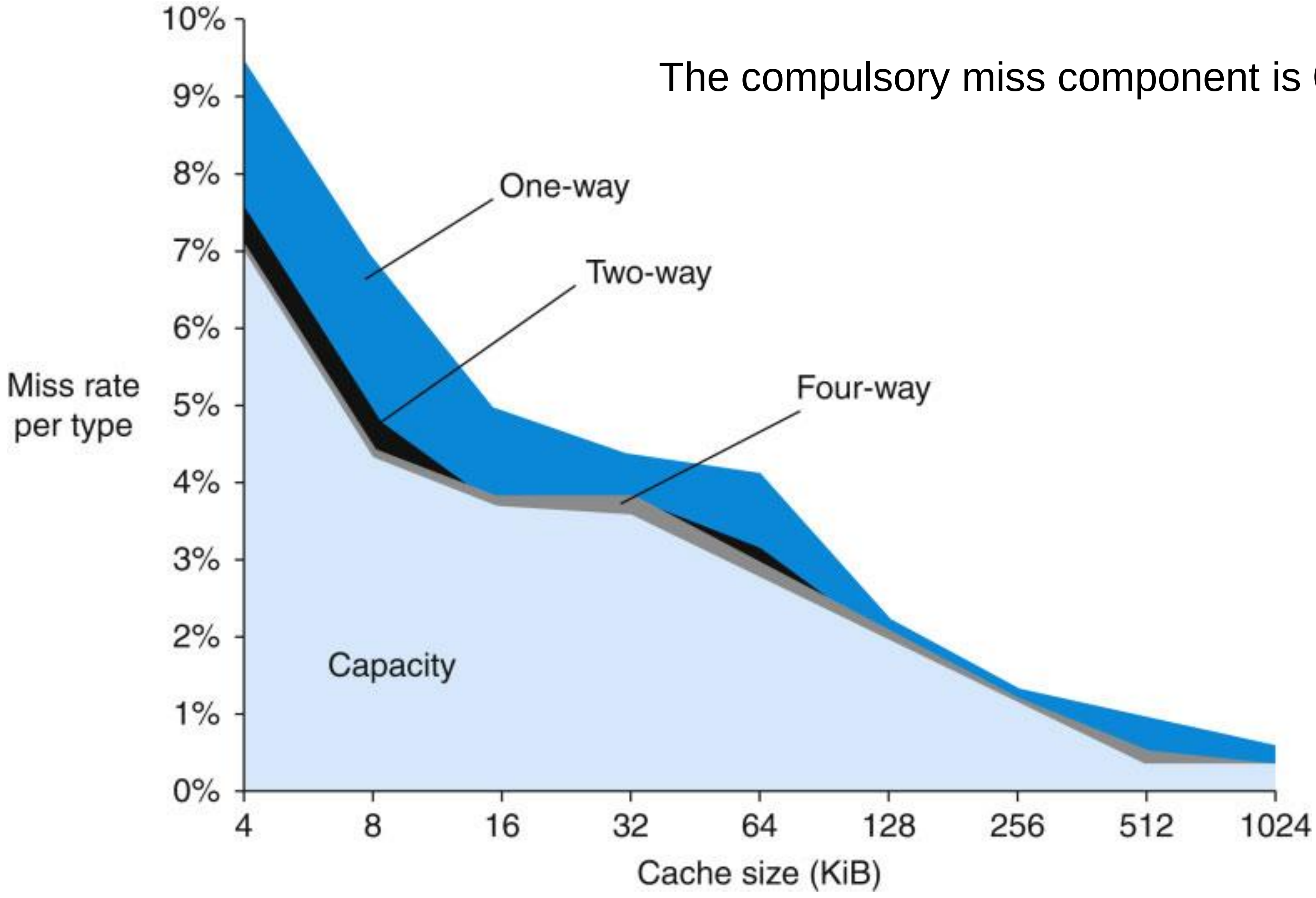
Write Policy

- 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 (or dirty)
  - Need to keep more state (e.g. dirty bit)
- Virtual memory
  - Only write-back is feasible

Three Cs: Sources of Misses

- 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)
  - Occur in a non-fully associative cache, due to the competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

3Cs in Miss Rate Example

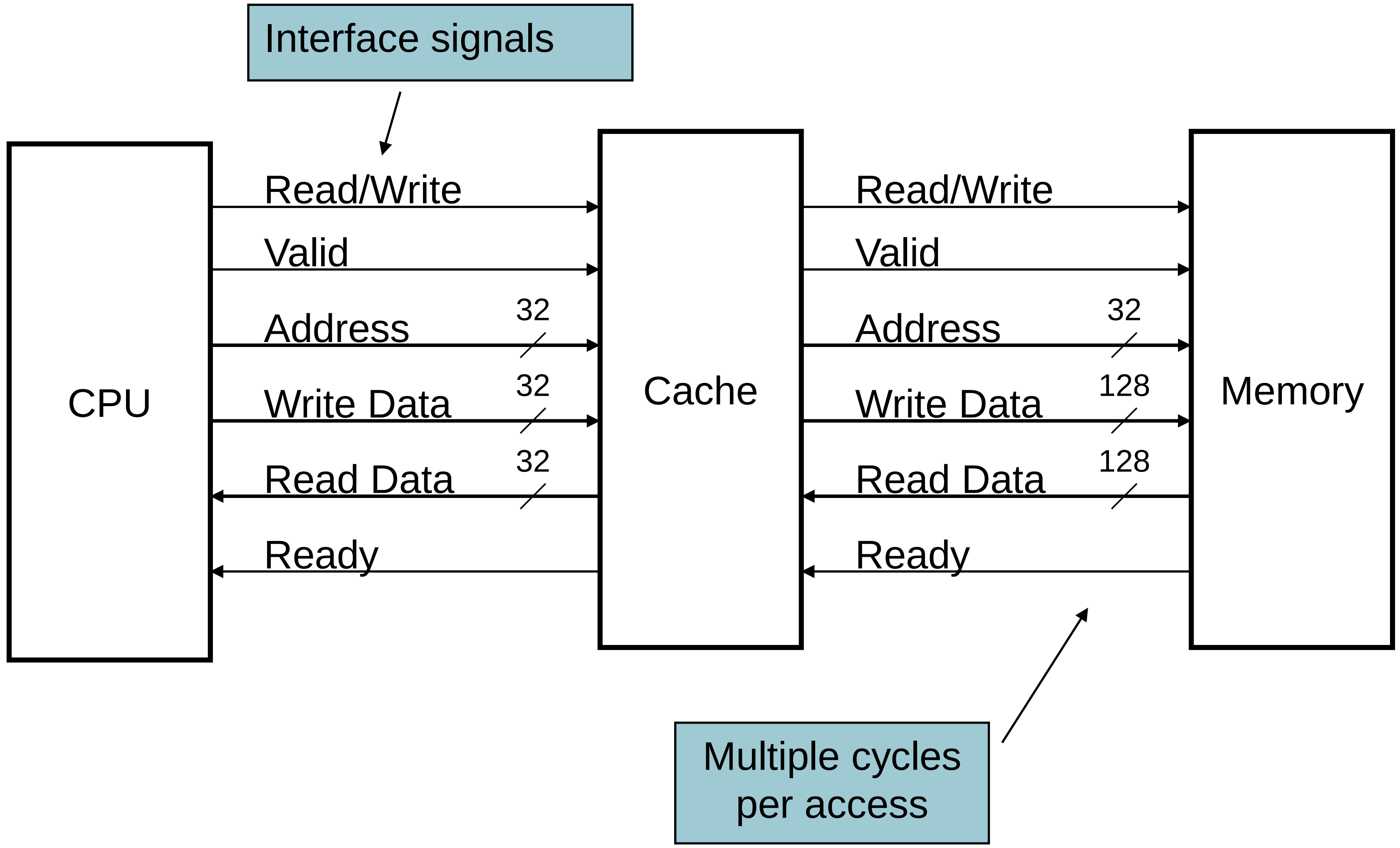


The compulsory miss component is 0.006%

Cache Design Trade-offs

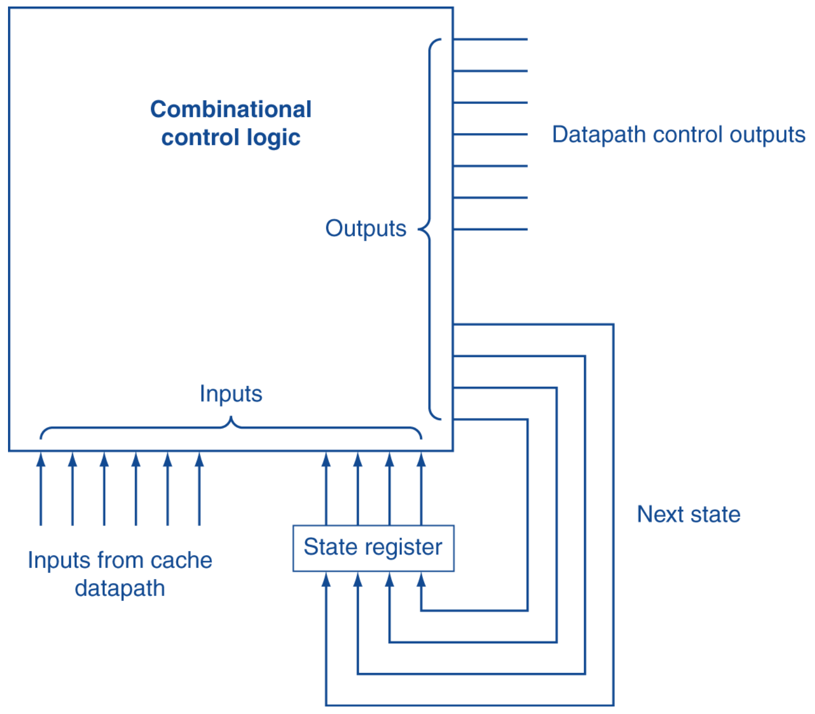
Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size,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

Cache Control

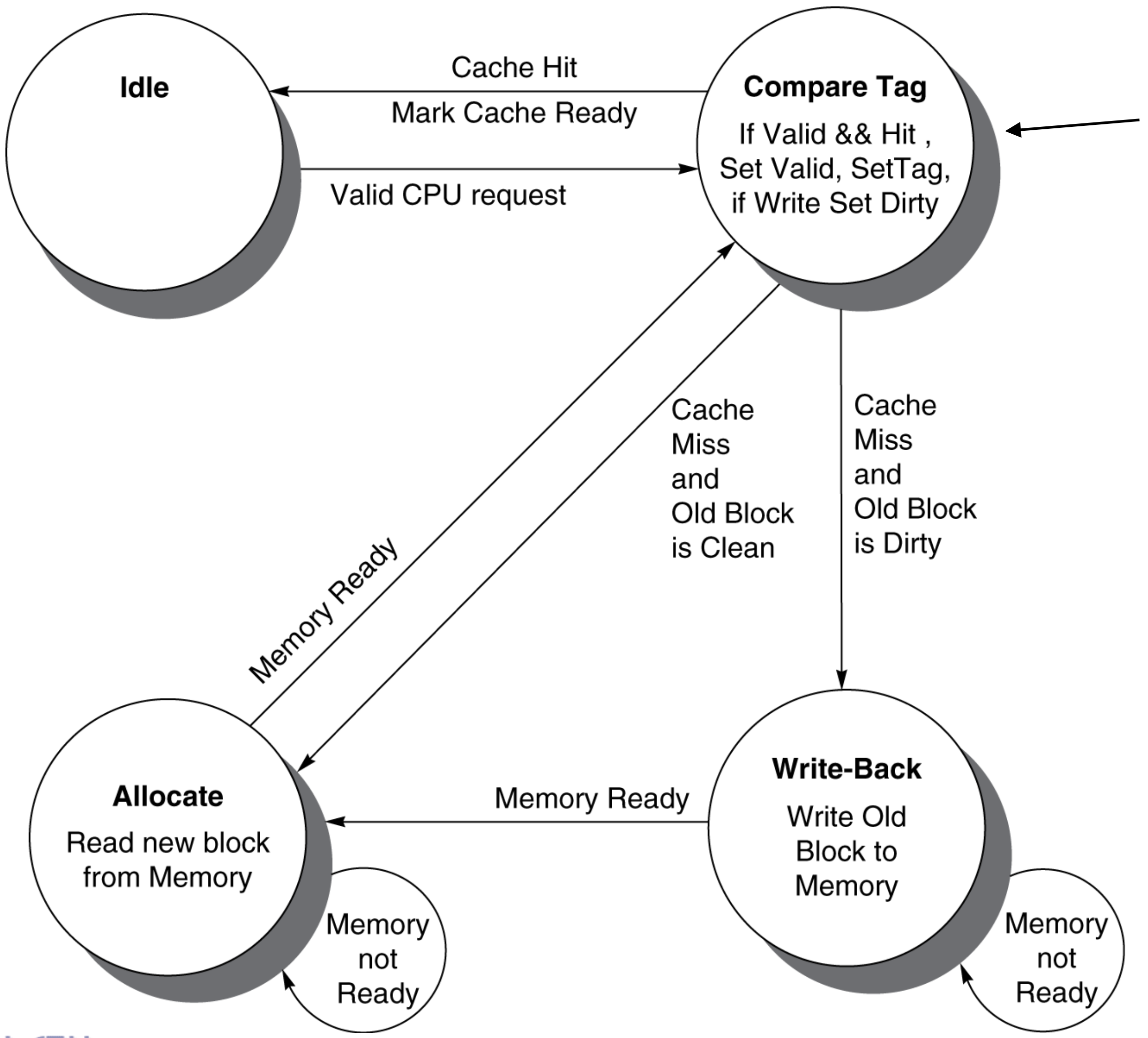


Finite State Machines

- 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

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
  - 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 Caches in ARM A53 vs. Intel i7

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	_	Approximated LRU
L3 block size	_	64 bytes
L3 write policy	_	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Address Translation and TLB Hardware

Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

DGEMM Performance

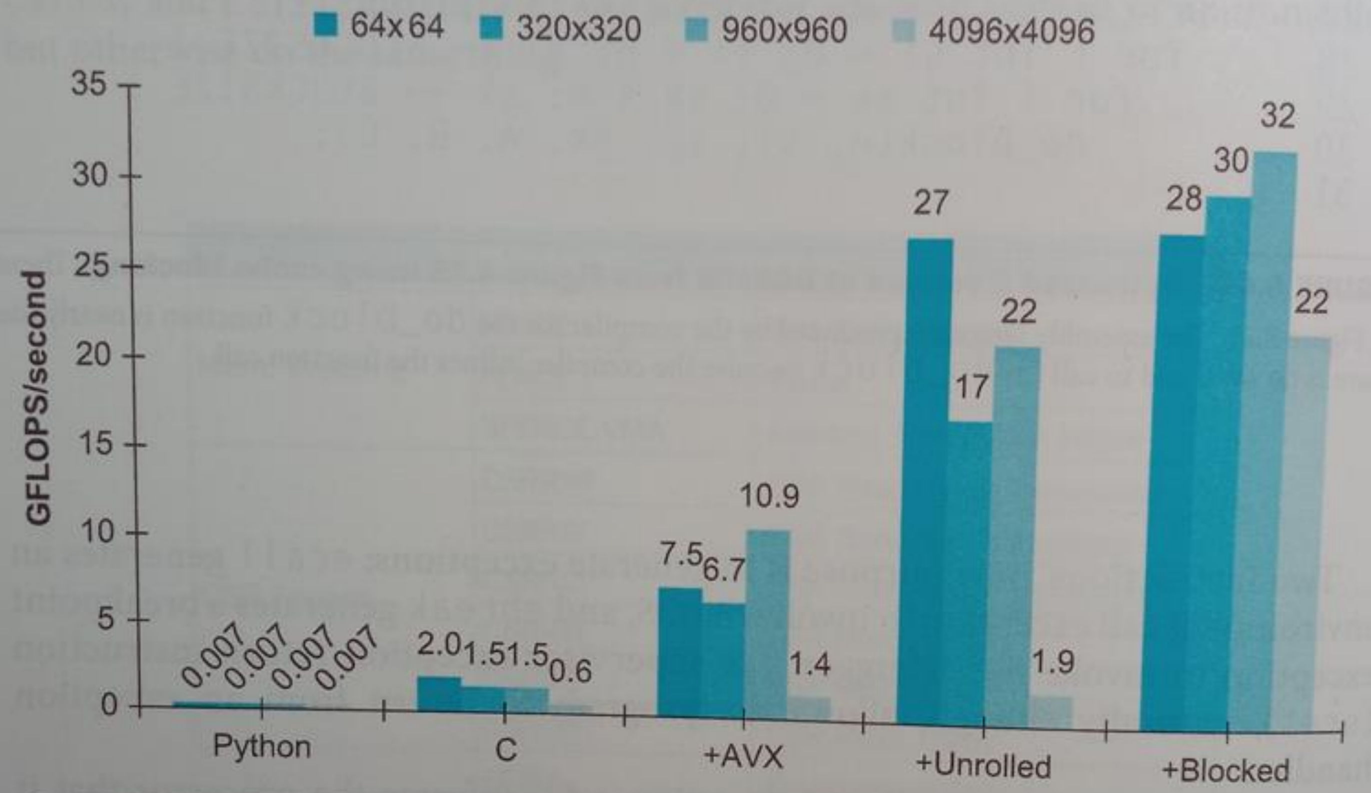


FIGURE 5.49 Performance of multiple versions of DGEMM as change matrix size measured in billion floating point operations per second (GFLOPS/second). The fully optimized code is 14–32 times faster than the C version in Chapter 2. Python runs at 0.007 GFLOPS/second for all matrix sizes. The Intel i7 hardware speculates by prefetching from the L3 to L1 and L2 caches, which is why the benefits of blocking are not as high as on some microprocessors.

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 behaviour when writing or generating code in a computer
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality

Pitfalls

- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores $\Rightarrow$ need to increase associativity
- Using AMAT to evaluate performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation

Pitfalls

- Extending address range using segments
  - E.g., Intel 80286
  - But a segment is not always big enough
  - Makes address arithmetic complicated
- Implementing a VMM on an ISA not designed for virtualization
  - E.g., non-privileged instructions accessing hardware resources
  - Either extend ISA, or require guest OS not to use problematic instructions

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