

Large and Fast: Exploiting Memory Hierarchy



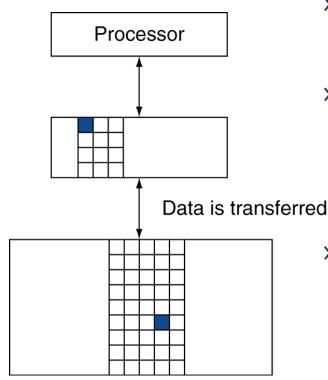
Principle of Locality

- 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

Memory Technology

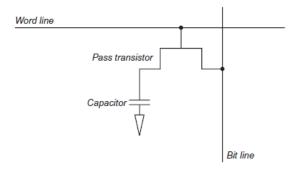
- > Static RAM (SRAM)
 - 0.5ns 2.5ns, \$500 \$1000 per GB
- > Dynamic RAM (DRAM)
 - 50ns 70ns, \$10 \$20 per GB
- > Magnetic disk
 - 5ms 20ms, \$0.05 \$0.10 per GB
- > Ideal memory
 - Access time of SRAM
 - Capacity and cost/GB of disk

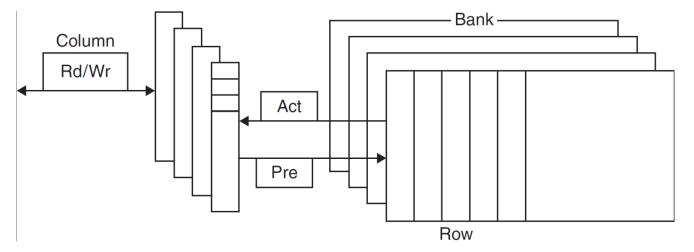
Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5-2.5 ns	\$500-\$1000
DRAM semiconductor memory	50-70ns	\$10-\$20
Flash semiconductor memory	5,000-50,000ns	\$0.75-\$1.00
Magnetic disk	5,000,000-20,000,000ns	\$0.05-\$0.10

DRAM Technology

> Data stored as a charge in a capacitor

- Single transistor used to access the charge
- Must periodically be refreshed
 - > Read contents and write back
 - > Performed on a DRAM "row"



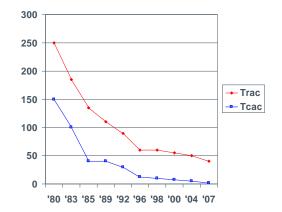


Advanced DRAM Organization

- > Bits in a DRAM organized as a rectangular array
 - DRAM accesses an entire row
 - Burst mode: supply successive words from a row with reduced latency
- > Row buffer
 - Allows several words to be read and refreshed in parallel
- > DRAM banking
 - Allows simultaneous access to multiple DRAMs
 - Improves bandwidth

DRAM Generations

Year introduced	Chip size	\$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	\$1,500,000	250 ns	150 ns
1983	256 Kibibit	\$500,000	185 ns	100 ns
1985	1 Mebibit	\$200,000	135 ns	40 ns
1989	4 Mebibit	\$50,000	110 ns	40 ns
1992	16 Mebibit	\$15,000	90 ns	30 ns
1996	64 Mebibit	\$10,000	60 ns	12 ns
1998	128 Mebibit	\$4,000	60 ns	10 ns
2000	256 Mebibit	\$1,000	55 ns	7 ns
2004	512 Mebibit	\$250	50 ns	5 ns
2007	1 Gibibit	\$50	45 ns	1.25 ns
2010	2 Gibibit	\$30	40 ns	1 ns
2012	4 Gibibit	\$1	35 ns	0.8 ns



DRAM Performance Factors

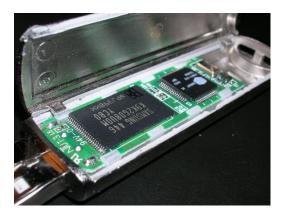
- > Synchronous DRAM
 - Clocked signal allows for consecutive accesses in bursts without needing to send each address
 - Improves bandwidth
- > Double data rate (DDR) SDRAM
 - Transfer on rising and falling clock edges (2n)
- > Current generation: DDR4
 - 8n prefetch buffer: burst of 8 datawords x each access
 - E.g., DDR4-3200 (PC4-25600) with DIMMs up to 64 GiB
 - > Memory clock: 400 MHz; Data rate: 3200 MT/s; Transfer rate: 25.6 GB/s

Flash Storage

> Nonvolatile semiconductor storage

- 100× 1000× faster than disk
- Smaller, lower power, more robust
- But more \$/GB (between disk and DRAM)





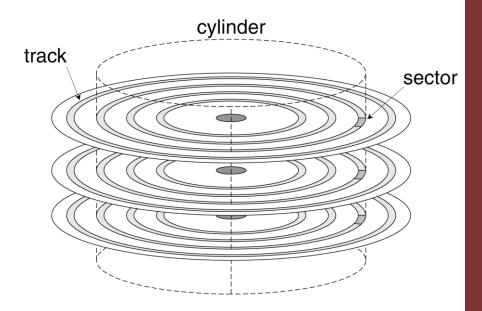
Flash Types

- > 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 1000's of accesses
 - Not suitable for direct RAM or disk replacement
 - Wear leveling: remap data to less used blocks

Disk Storage

> Nonvolatile, rotating magnetic storage





Disk Sectors and Access

- > Each sector records
 - Sector ID
 - Data (from 512 to 4096 bytes)
 - 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

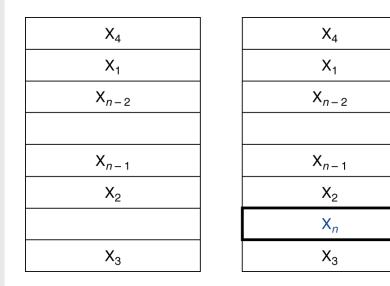
Disk Performance Issues

- > Manufacturers quote average seek time
 - Based on all possible seeks
 - Locality and OS scheduling lead to smaller actual average seek times
- Smart disk controller allocate physical sectors on disk
 - Present logical sector interface to host
 - SCSI, ATA, SATA
- > Disk drives include caches
 - Prefetch sectors in anticipation of access
 - Avoid seek and rotational delay

Cache Memory

> Cache memory

- The level of the memory hierarchy closest to the CPU
- > Given accesses X₁, ..., X_{n-1}, X_n



How do we know if the data is present?

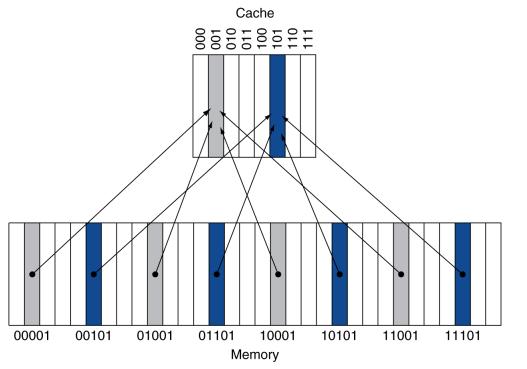
Where do we look?

a. Before the reference to X_n

b. After the reference to X_n



- > Location determined by address
- > Granularity of a cache block (a.k.a. cache line)
- > Direct mapped: only one choice
 - (Block address) modulo (#Blocks in cache)



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

Tags and Valid Bits

> How do we know which particular block is stored in a cache location?

- Store block address as well as the data
- Actually, only need the high-order bits
- Called the tag
- > What if there is no data in a location?
 - Valid bit: 1 = present, 0 = not present
 - Initially 0

> 8-blocks, 1 word/block, direct mapped

> Initial state

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

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	Υ	10	Mem[10110]
111	N		

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

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

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		

H

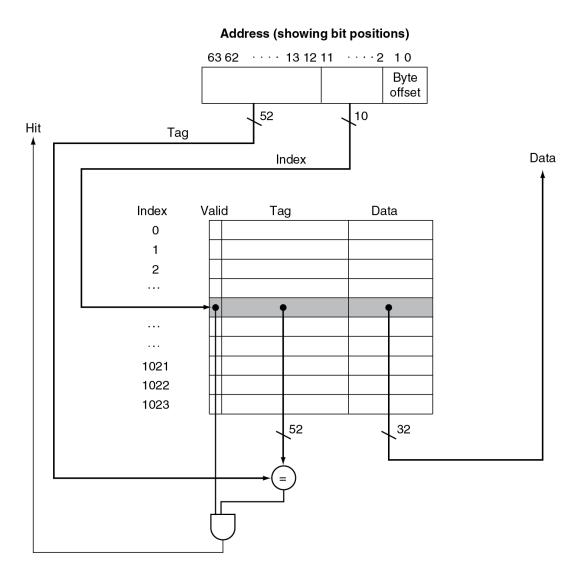
Word a	ddr	Binary ad	dr	Hit/miss	Cache block
16		10 000		Miss	000
3		00 011		Miss	011
16		10 000		Hit	000
l					
Index	V	Tag	Dat	а	
000	Υ	10	Ме	m [10000]	
001	N				
010	Υ	11	Mei	m[11010]	
011	Υ	00	Ме	m [00011]	
100	N				
101	N				
110	Y	10	Mei	m[10110]	
111	N				

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	Υ	10	Mem[10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision

H



Example: Larger Block Size

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



Block Size Considerations

- > Larger blocks should reduce miss rate
 - Due to spatial locality
- > But in a fixed-sized cache
 - Larger blocks \Rightarrow fewer of them
 - Larger blocks \Rightarrow pollution
- > Larger miss penalty
 - Can override benefit of reduced miss rate
 - Early restart and critical-word-first can help

Cache Misses

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

Write-Through

> On data-write hit, could just update the block in cache

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

Write-Back

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

Keep track of whether each block is dirty

- > When a dirty block is replaced
 - Write it back to memory
 - Can use a write buffer to allow replacing block to be read first

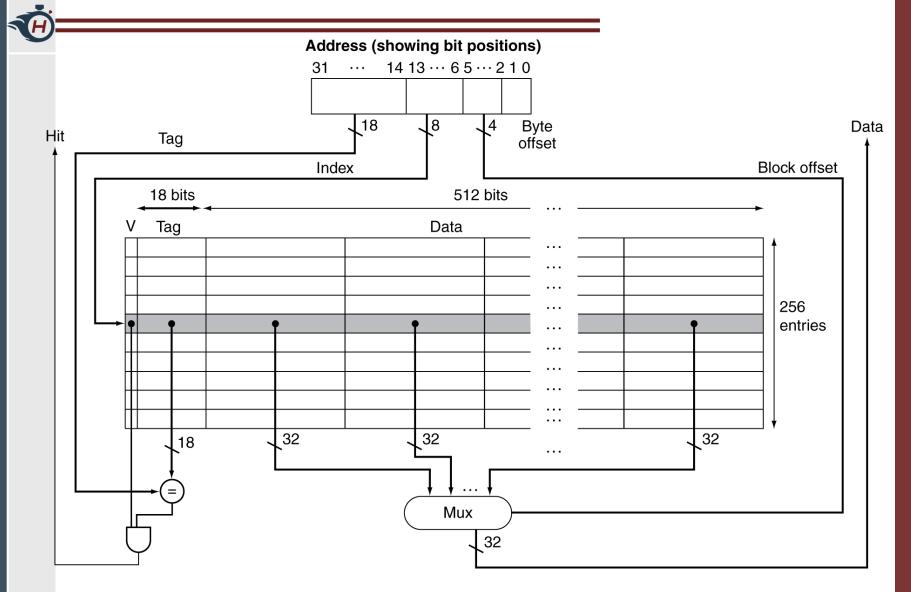
Write Allocation

- > What should happen on a write miss?
- > Alternatives for write-through
 - Allocate on miss: fetch the block
 - Write around: don't fetch the block
 - Since programs often write a whole block before reading it (e.g., initialization)
- > For write-back
 - Usually fetch the block

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



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 = $1 + 4 \times 15 + 4 \times 1 = 65$ bus cycles
 - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle

Measuring Cache Performance

> Components of CPU time

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

Memory stall cycles

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

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

Cache Performance Example

> Given

- I-cache miss rate = 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 × 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

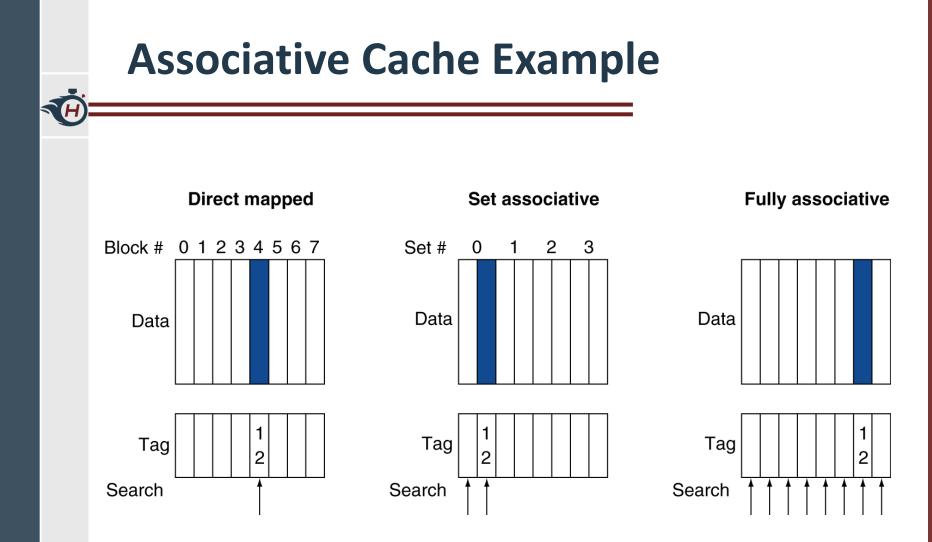
Performance Summary

- > 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

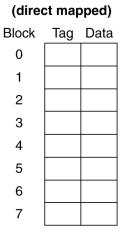
- Allow a given block to go in any cache entry
- Requires all entries to be searched at once
- Comparator per entry (expensive)
- > n-way set associative
 - Each set contains *n* entries
 - Block number determines which set
 - > (Block number) modulo (#Sets in cache)
 - Search all entries in a given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



Spectrum of Associativity

> For a cache with 8 entries

One-way set associative



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														

Associativity Example

> Compare 4-block caches

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

> Direct mapped

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

Associativity Example

> 2-way set associative

Block	Cache	Hit/miss	(Cache conter	nt after access			
address	index		Se	et O	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]				

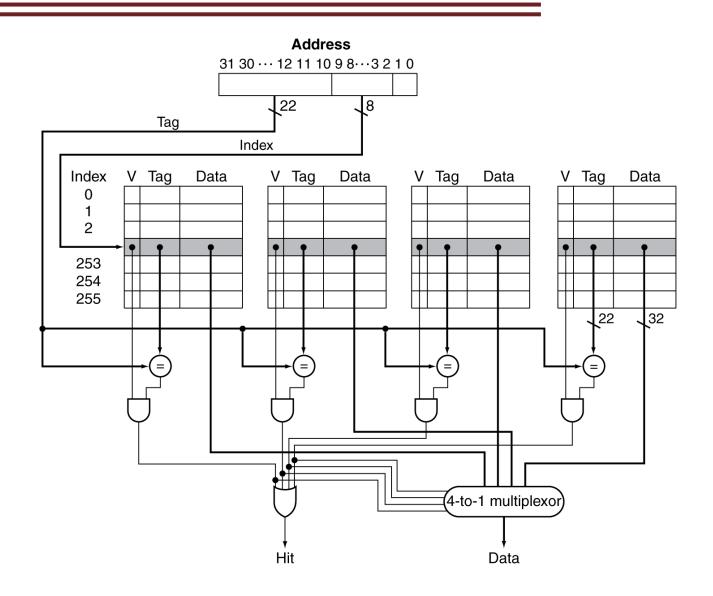
Fully associative

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

How Much Associativity

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

Set Associative Cache Organization



Replacement Policy

- > Direct mapped: no choice
- > Set associative
 - Prefer non-valid entry, if there is one
 - Otherwise, choose among entries in the set
- > Least-recently used (LRU)
 - Choose the one unused for the longest time
 - Simple for 2-way, manageable for 4-way, too hard beyond that
- > Random
 - Gives approximately the same performance as LRU for high associativity

Multilevel Caches

- > Primary cache attached to CPU
 - Small, but fast
- > Level-2 cache services misses from primary cache
 - Larger, slower, but still faster than main memory
- > Main memory services L-2 cache misses
- > Some high-end systems include L-3 cache

Multilevel Cache Example

> Given

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

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 = 500 cycles
- > CPI = 1 + 0.02 × 20 + 0.005 × 500 = 3.9
- > Performance ratio = 9/3.9 = 2.3

Multilevel Cache Considerations

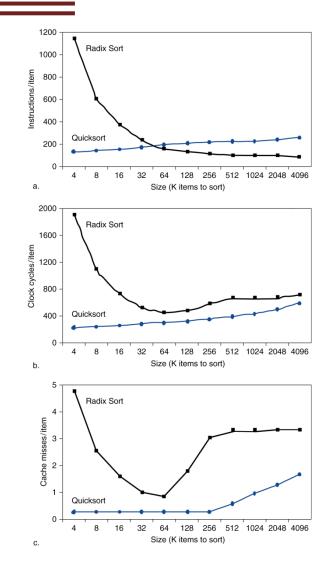
- > Primary cache
 - Focus on minimal hit time
- > L-2 cache
 - Focus on low miss rate to avoid main memory access
 - Hit time has less overall impact
- > Results
 - L-1 cache usually smaller than a single cache
 - L-1 block size smaller than L-2 block size

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 analyse
 - Use system simulation

Interactions with Software

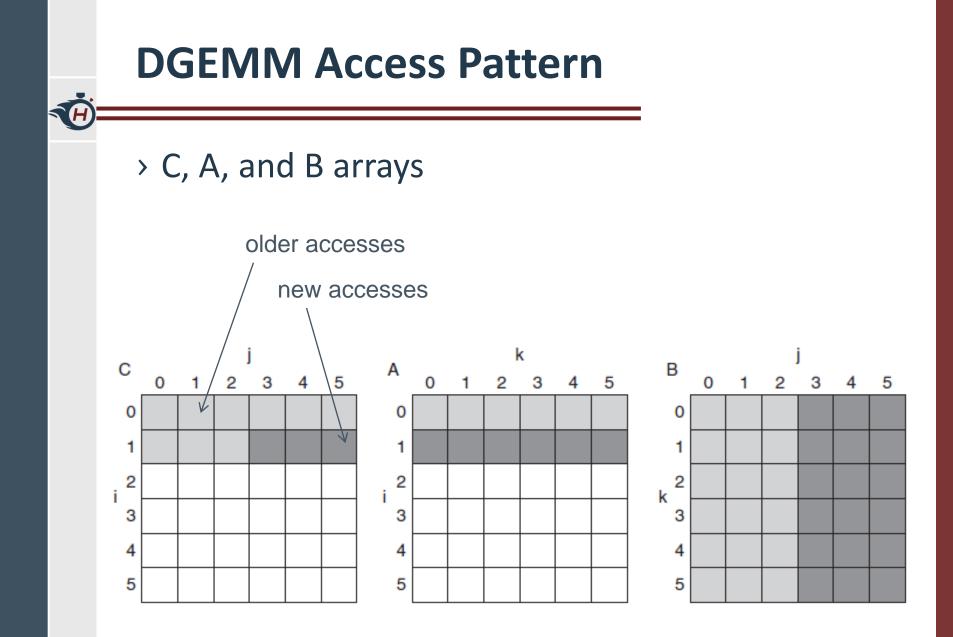
- Misses depend on memory access patterns
 - Algorithm behavior
 - Compiler optimization for memory access



Software Optimization via Blocking

- > Goal: maximize accesses to data before it is replaced
- > Consider inner loops of DGEMM:

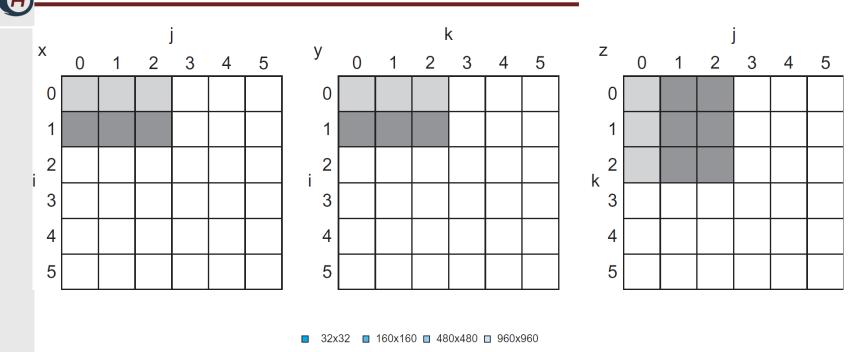
```
for (int j = 0; j < n; ++j)
{
    double cij = C[i+j*n];
    for( int k = 0; k < n; k++ )
        cij += A[i+k*n] * B[k+j*n];
    C[i+j*n] = cij;
}</pre>
```

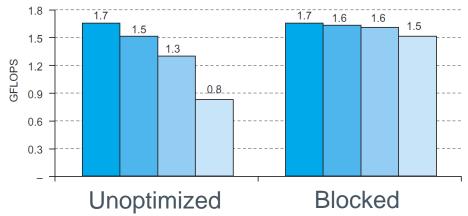


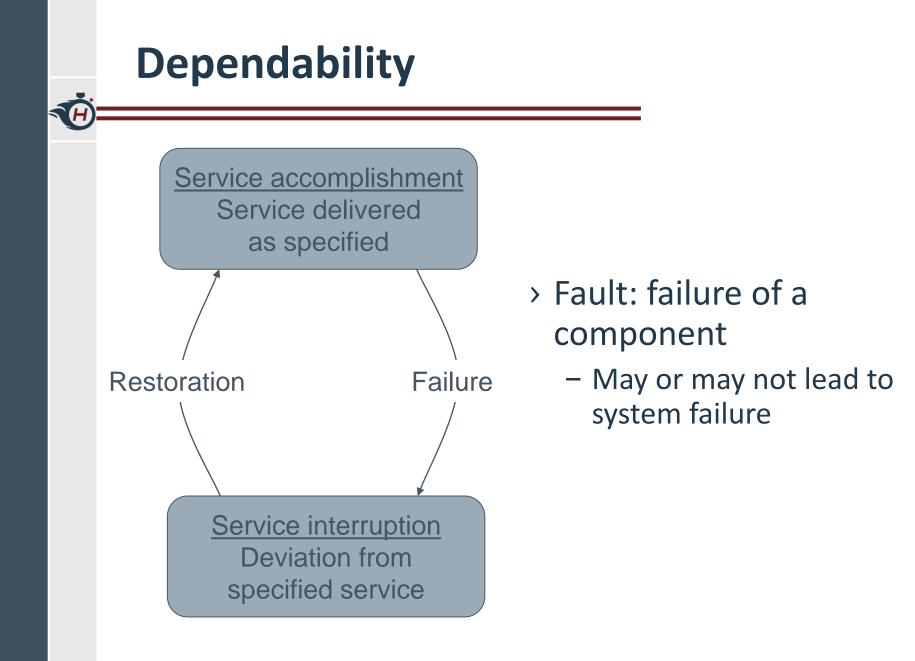
Cache Blocked DGEMM

```
1 #define BLOCKSIZE 32
2 void do block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
  for (int i = si; i < si+BLOCKSIZE; ++i)</pre>
5
   for (int j = sj; j < sj+BLOCKSIZE; ++j)
6
7
    {
8
     double cij = C[i+j*n];/* cij = C[i][j] */
9
    for ( int k = sk; k < sk+BLOCKSIZE; k++ )
    cij += A[i+k*n] * B[k+j*n]; /* cij+=A[i][k]*B[k][j] */
10
11 C[i+j*n] = cij;/* C[i][j] = cij */
12 }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16 for (int sj = 0; sj < n; sj += BLOCKSIZE )
17
    for ( int si = 0; si < n; si += BLOCKSIZE )</pre>
    for ( int sk = 0; sk < n; sk += BLOCKSIZE )</pre>
18
19
     do block(n, si, sj, sk, A, B, C);
20 }
```

Blocked DGEMM Access Pattern







Dependability Measures

- > Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- > Mean time between failures
 - 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

Encoding SEC

- > 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
	p1	Х		Х		Х		Х		Х		Х	
Parity bit	p2		Х	Х			Х	Х			Х	Х	
coverate	p4				Х	Х	Х	Х					Х
	p8								Х	Х	Х	Х	Х

Example

> Given a data 10011010_{bin}

- > Leave positions for parity bits: __1_001_1010
- > Add parity bits:
 - <u>0</u>_1_001_1010
 0<u>1</u>1_001_1010
 0111<u>1001_1010
 011110010101010
 </u>

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
- > E.g., flipping bit 10 from previous example: 011100101<u>1</u>10 Gives

$$0 1 1 1 0 0 1 0 1 1 1 0 \rightarrow p_1 = 0$$

$$0 1 1 1 0 0 1 1 1 1 1 0 \rightarrow p_2 = 1$$

$$0 1 1 1 0 0 1 1 1 1 1 0 \rightarrow p_3 = 0$$

$$0 1 1 1 0 0 1 0 1 1 1 0 \rightarrow p_4 = 1$$

$$1010_{bin} = 10$$

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

- 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 comptuers
- > Examples
 - IBM VM/370 (1970s technology!)
 - VMWare
 - Microsoft Virtual PC

Virtual Machine Monitor - Hypervisor

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

Example: Timer Virtualization

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

Instruction Set Support

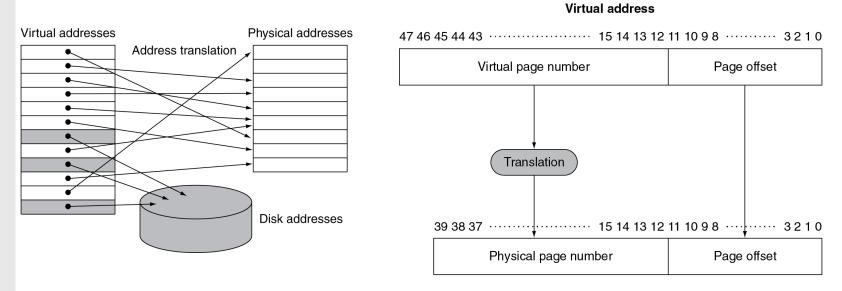
- > User and System modes
- Privileged instructions only available in system mode
 - Trap to system if executed in user mode
- All physical resources only accessible using privileged instructions
 - Including page tables, interrupt controls, I/O registers
- > Renaissance of virtualization support
 -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

Virtual Memory

- > Use main memory as a "cache" for secondary (disk) storage
 - Gives the illusion of a virtually infinite memory
 - 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
 - Virtual Memory "block" is called a page
 - Virtual Memory translation "miss" is called a page fault

Address Translation

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



Physical address

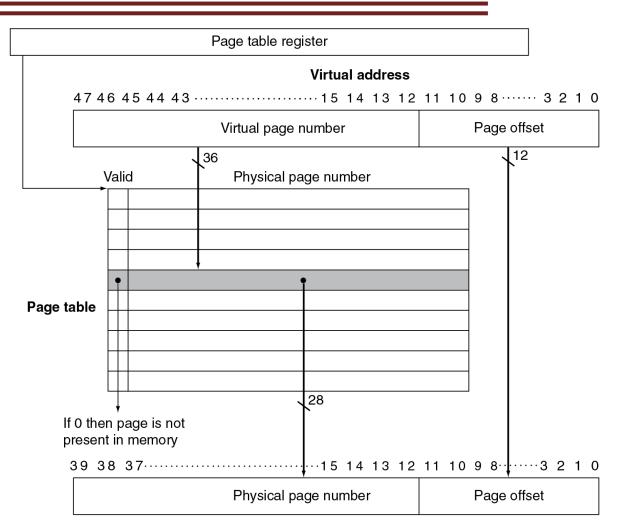
Page Fault Penalty

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

Page Tables

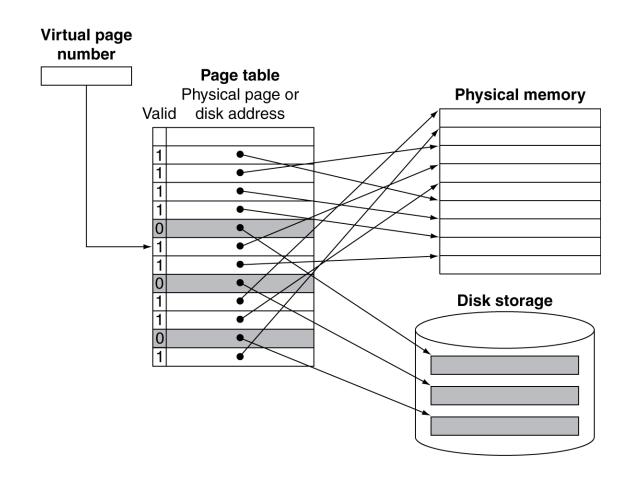
- > Stores placement information
 - Array of page table entries, indexed by virtual page number
 - Page table register in CPU points to page table in physical memory
- > If page is present in memory
 - PTE stores the physical page number
 - Plus other status bits (referenced, dirty, ...)
- > If page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

Translation Using a Page Table



Physical address

Mapping Pages to Storage



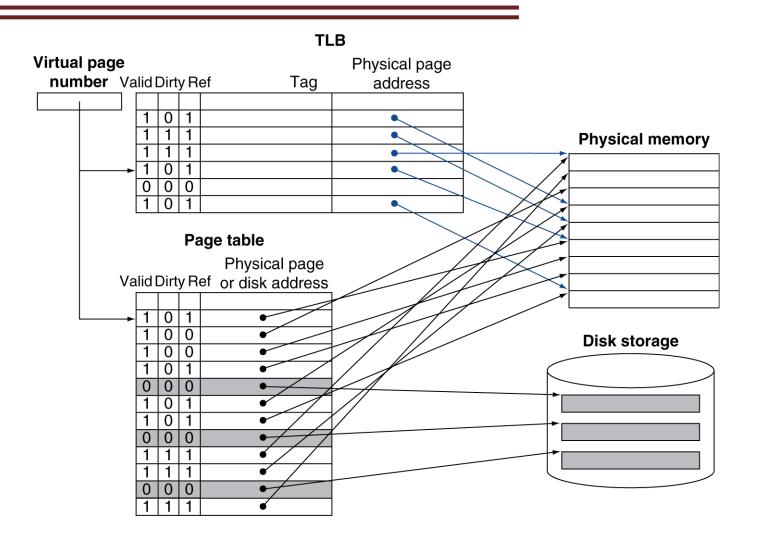
Replacement and Writes

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

Fast Translation Using a TLB

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

Fast Translation Using a TLB



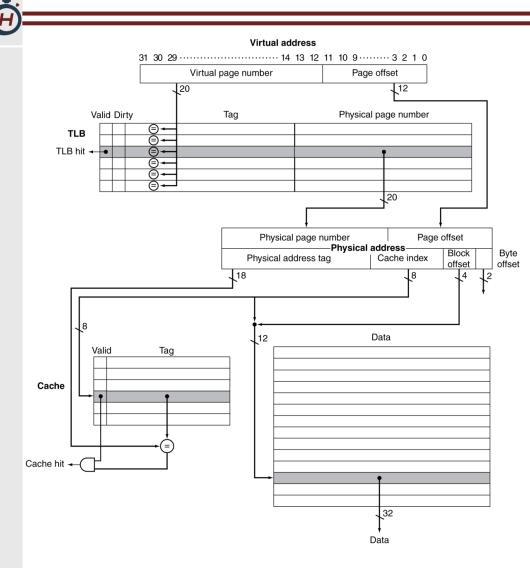
TLB Misses

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

Page Fault Handler

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

TLB and Cache Interaction



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

Memory Protection

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

The Memory Hierarchy

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

Block Placement

> Determined by associativity

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

Finding a Block

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

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

Replacement

> Choice of entry to replace on a miss

- Least recently used (LRU)
 - > Complex and costly hardware for high associativity
- Random
 - > Close to LRU, easier to implement
- > Virtual memory
 - LRU approximation with hardware support

Write Policy

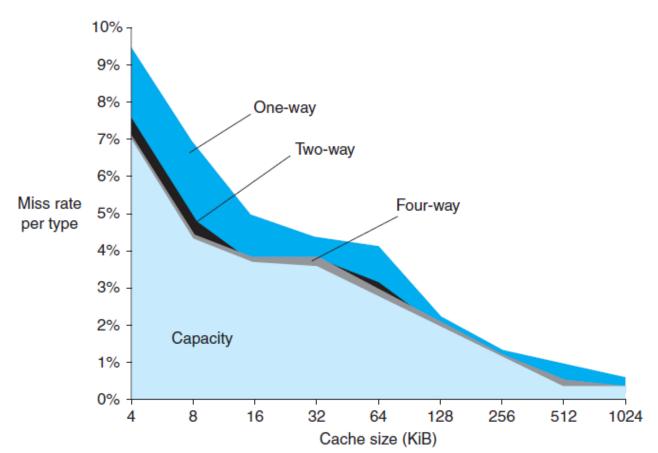
- > Write-through
 - Update both upper and lower levels
 - Simplifies replacement, but may require write buffer
- > Write-back
 - Update upper level only
 - Update lower level when block is replaced
 - Need to keep more state
- > Virtual memory
 - Only write-back is feasible, given disk write latency

Sources of Misses

> Compulsory misses (aka cold start misses)

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

Miss rate for three C's



Compulsory miss 0.006%

SPEC CPU2000 benchmark

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.

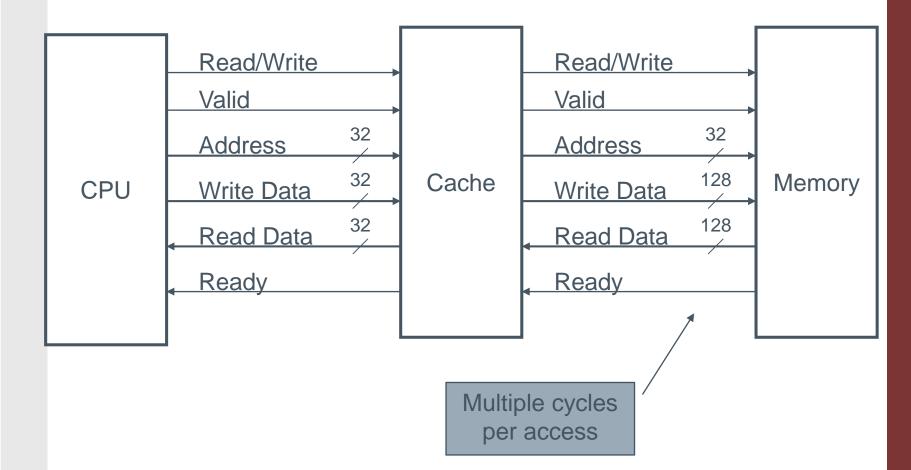
Cache Control

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



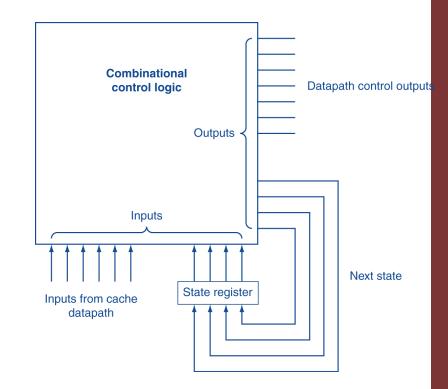
Interface Signals

H

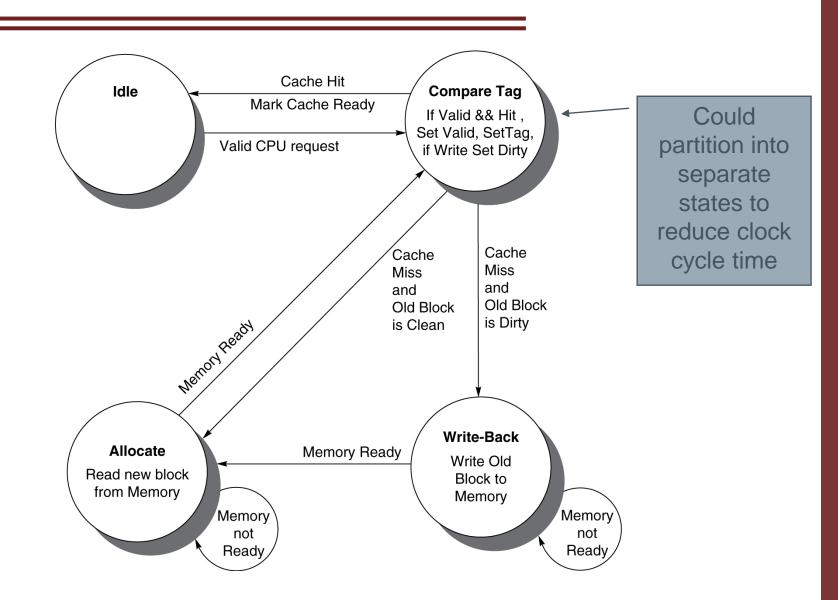


Finite State Machines

- > 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



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)
 ⇒ read returns written value
 - − P_1 writes X; P_2 reads X (sufficiently later) ⇒ read returns written value
 - > c.f. CPU B reading X after step 3 in example
 - P₁ writes X, P₂ 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 On-Chip Caches

H

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

2-Level TLB Organization

H

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	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	Both L1 TLBs are four-way set associative, LRU replacement
	TLB misses handled in hardware	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

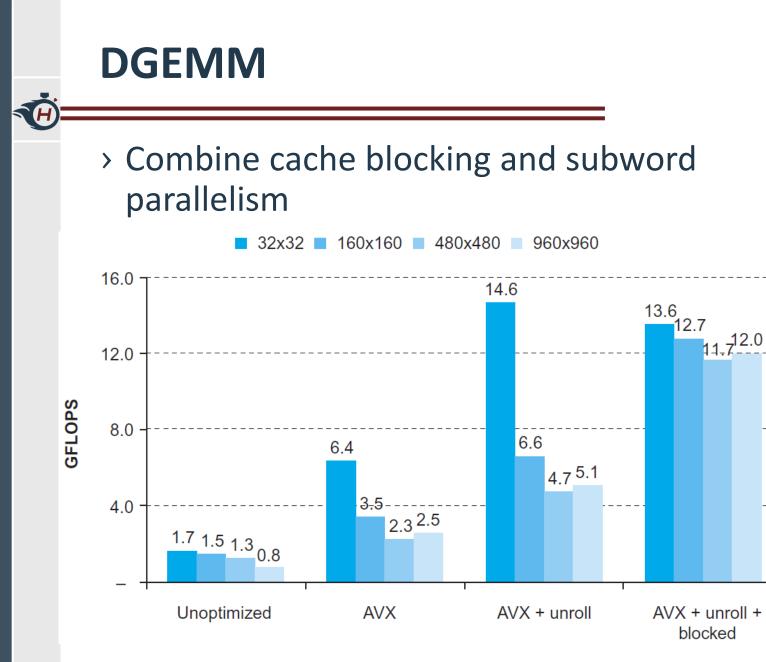
Supporting Multiple Issue

- Both have multi-banked caches that allow multiple accesses per cycle assuming no bank conflicts
- > Other optimizations
 - Return requested word first
 - Non-blocking cache
 - > Hit under miss
 - > Miss under miss
 - Data prefetching

RISC-V System Instructions

H

Туре	Mnemonic	Name
Mem ordering	fence.i	Instruction fence
	fence	Fence
	sfence.vm	Address translation fence
CSR access	csrrwi	CSR read/write immediate
	csrrsi	CSR read/set immediate
	csrrci	CSR read/clear immediate
	CSTTW	CSR read/write
	CSTTS	CSR read/set
	csrrc	CSR read/clear
System	ecall	Environment call
	ebreak	Environment breakpoint
	sret	Supervisor exception return
	wfi	Wait for interrupt



Pitfalls

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

Pitfalls

Ö

- > 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-oforder processors
 - Ignores effect of non-blocked accesses
 - Instead, evaluate performance by simulation

Pitfalls

- **Ö**
- > 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 igodot
 - Caching gives this illusion 🙂
- > Principle of locality
 - Programs use a small part of their memory space frequently
- > Memory hierarchy
 - L1 cache \leftrightarrow L2 cache $\leftrightarrow ... \leftrightarrow$ DRAM memory \leftrightarrow disk
- Memory system design is critical for multiprocessors