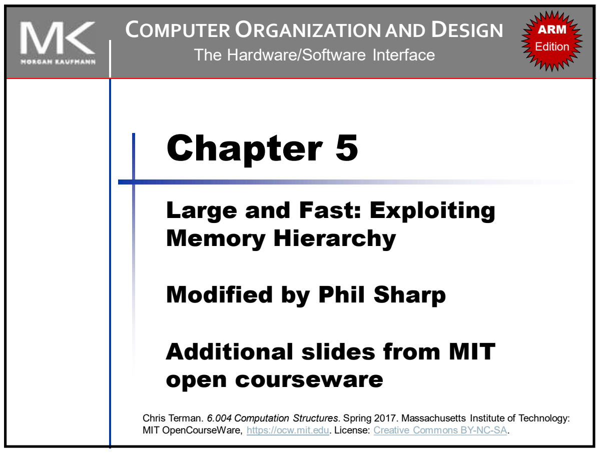


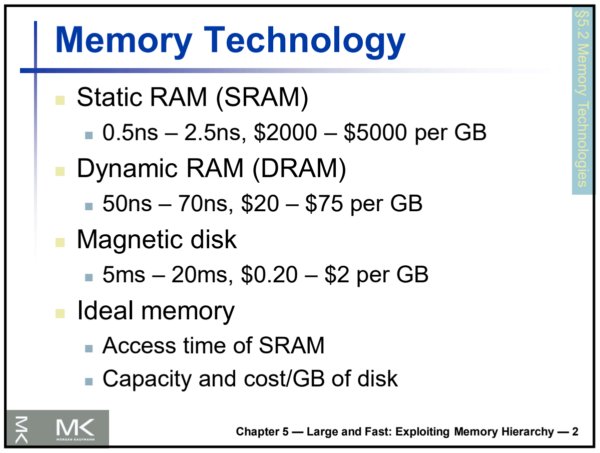
COMPUTER ORGANIZATION AND DESIGN
The Hardware/Software Interface
ARM Edition

# Chapter 5

**Large and Fast: Exploiting Memory Hierarchy**

**Modified by Phil Sharp**

**Additional slides from MIT open courseware**

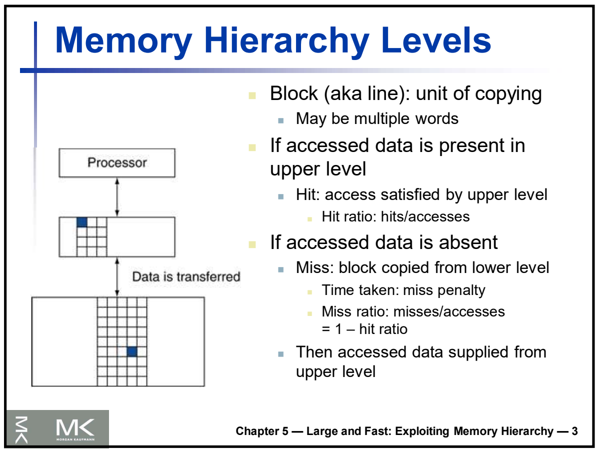


## Memory Technology

§5.2 Memory Technologies

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

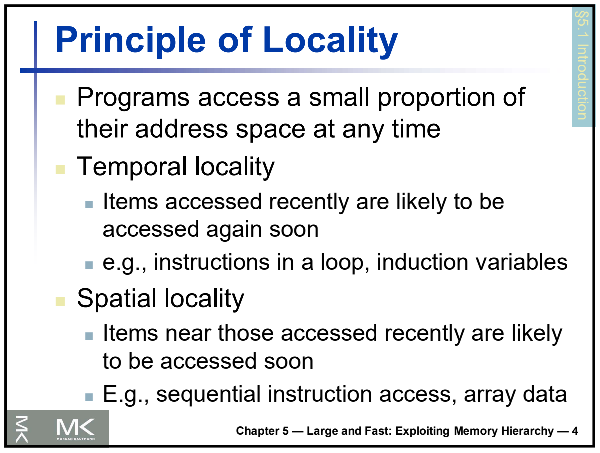
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 2


Processor
Data is transferred

## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 3

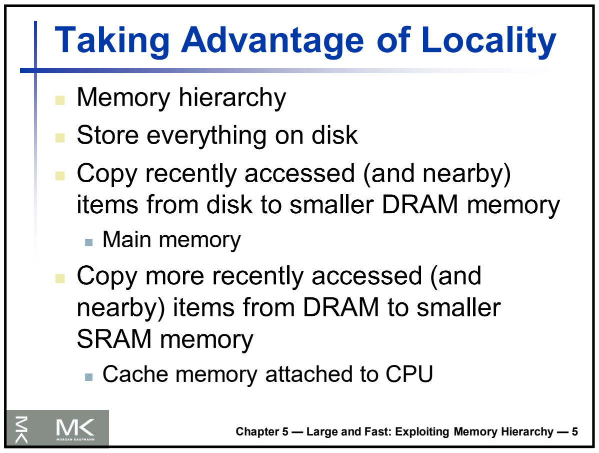


## Principle of Locality

§5.1 Introduction

- 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 4

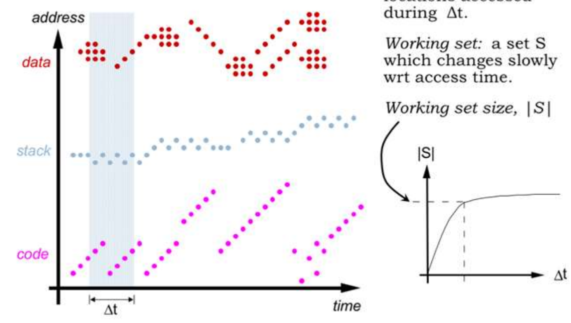


## Taking Advantage of Locality

- 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 5

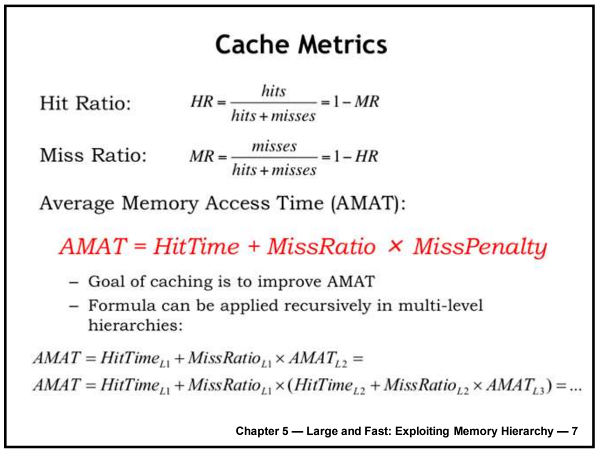
## Memory Reference Patterns



S is the set of locations accessed during Δt.

*Working set:* a set S which changes slowly wrt access time.

*Working set size,* |S|



## Cache Metrics

Hit Ratio: $HR = \dfrac{hits}{hits + misses} = 1 - MR$

Miss Ratio: $MR = \dfrac{misses}{hits + misses} = 1 - HR$

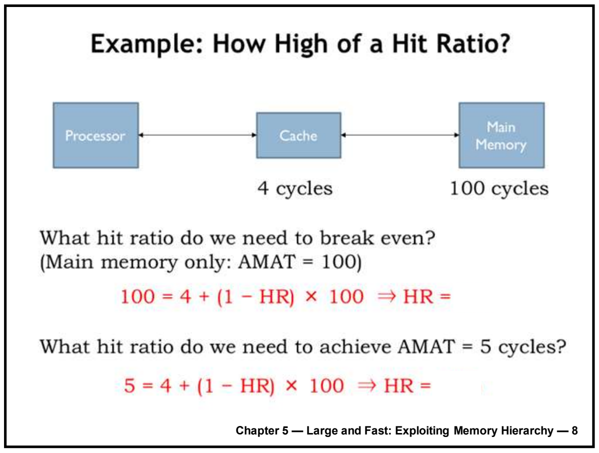
Average Memory Access Time (AMAT):

$$AMAT = HitTime + MissRatio \times MissPenalty$$

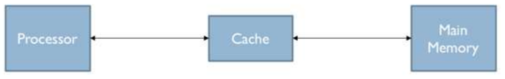
- Goal of caching is to improve AMAT
- Formula can be applied recursively in multi-level hierarchies:

$$AMAT = HitTime_{L1} + MissRatio_{L1} \times AMAT_{L2} =$$
$$AMAT = HitTime_{L1} + MissRatio_{L1} \times (HitTime_{L2} + MissRatio_{L2} \times AMAT_{L3}) = \ldots$$

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 7



## Example: How High of a Hit Ratio?



4 cycles 100 cycles

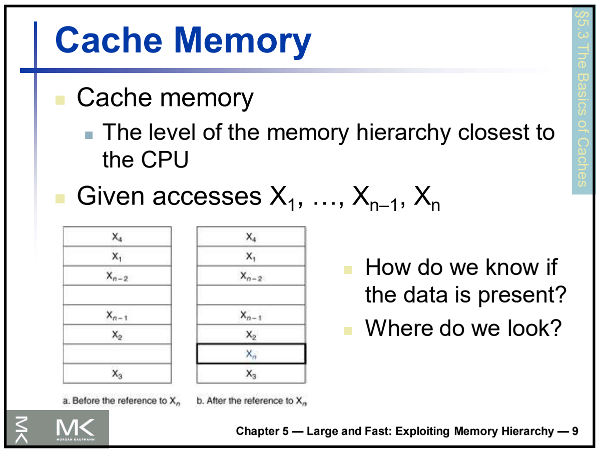
What hit ratio do we need to break even?
(Main memory only: AMAT = 100)

$100 = 4 + (1 - HR) \times 100 \Rightarrow HR =$

What hit ratio do we need to achieve AMAT = 5 cycles?

$5 = 4 + (1 - HR) \times 100 \Rightarrow HR =$

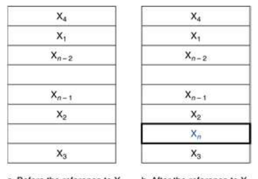
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 8



## Cache Memory

§5.3 The Basics of Caches

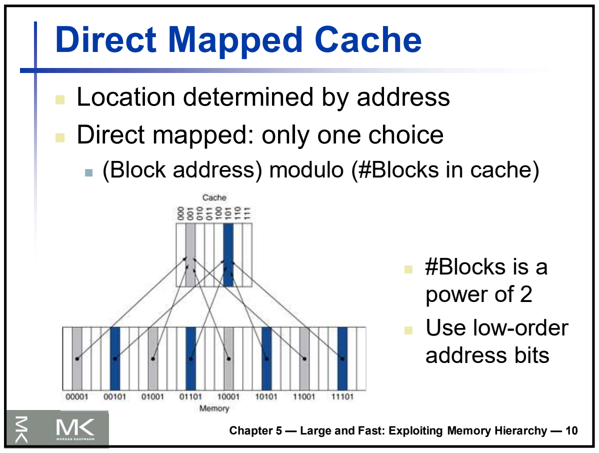
- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses $X_1, \ldots, X_{n-1}, X_n$



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

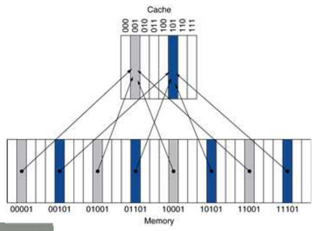
- How do we know if the data is present?
- Where do we look?

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 9



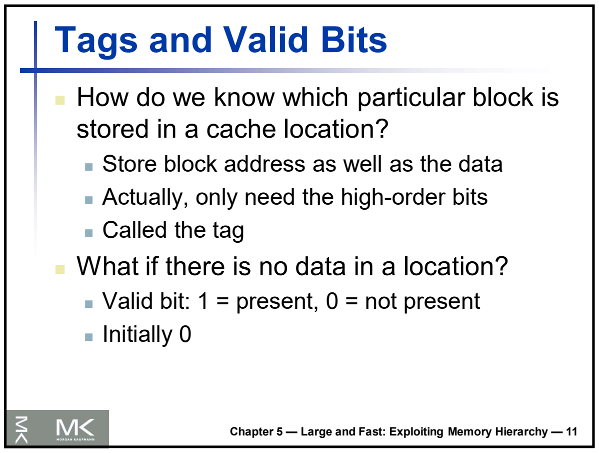
## Direct Mapped Cache

- 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

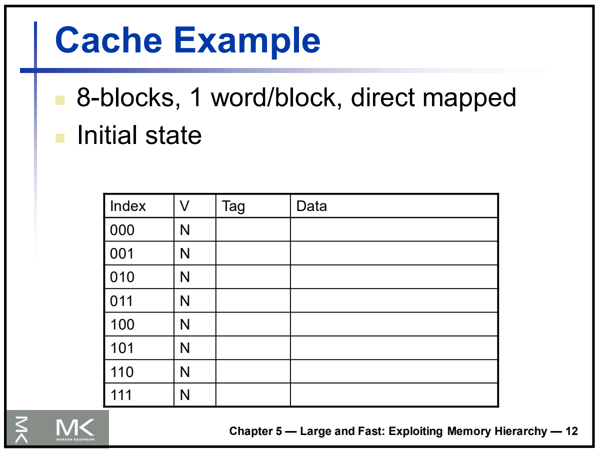
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 10



## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 11

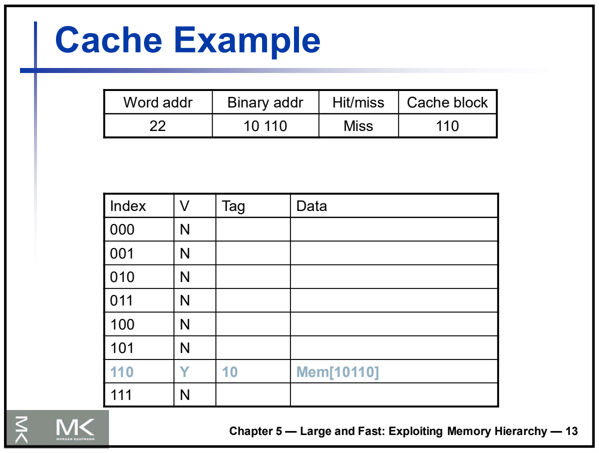


## Cache Example

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

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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 12

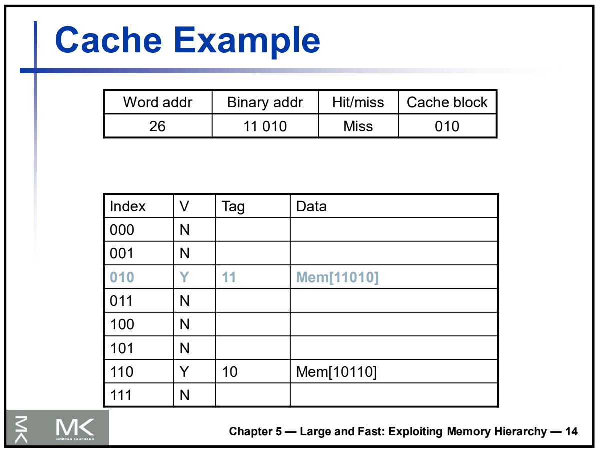


## Cache Example

| 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 | | |

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 13

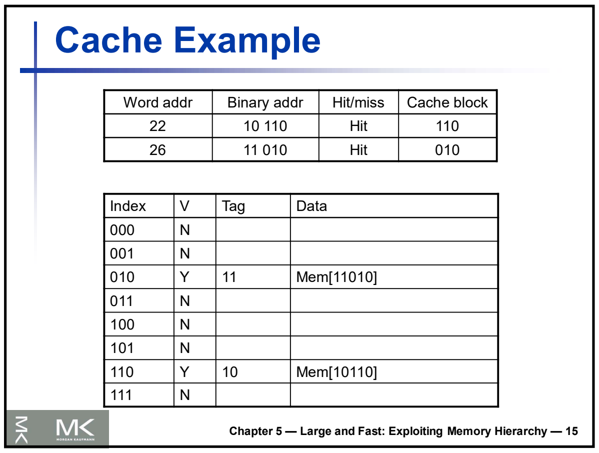


## Cache Example

| 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 | | |

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 14

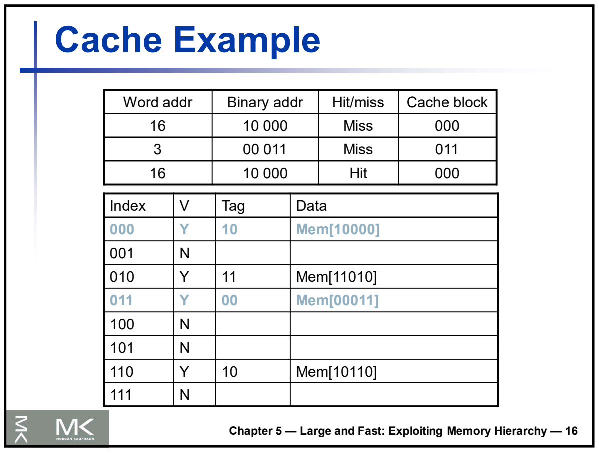


## Cache Example

| 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 | | |

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 15

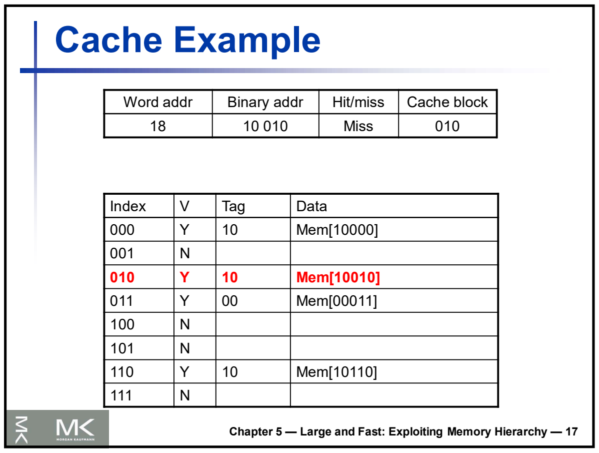


## Cache Example

| 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 | | |

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 16

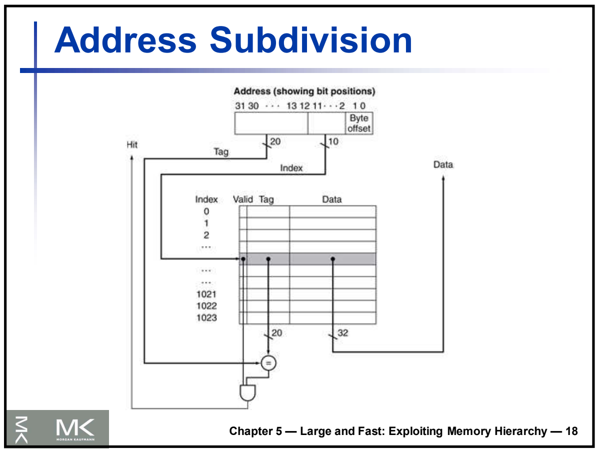


## Cache Example

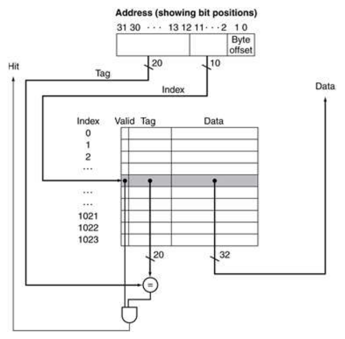
| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 18 | 10 010 | Miss | 010 |

| 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 | | |

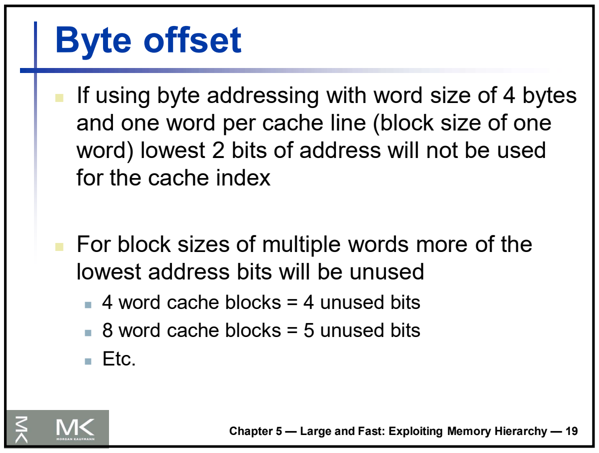
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 17



## Address Subdivision



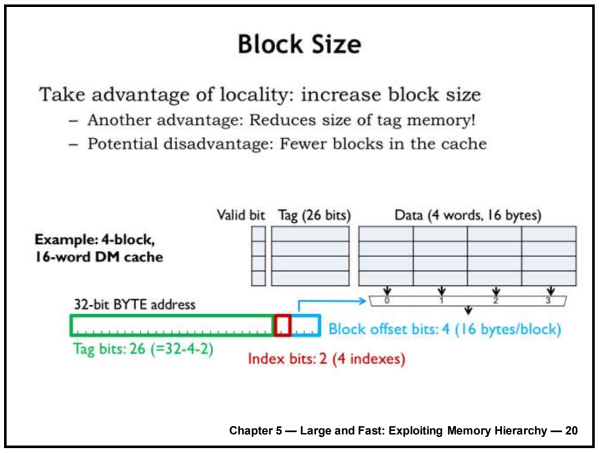
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 18



## Byte offset

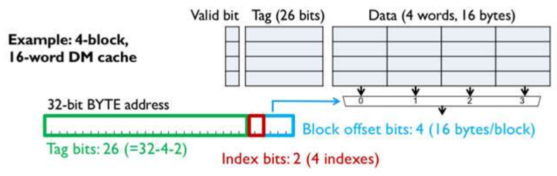
- If using byte addressing with word size of 4 bytes and one word per cache line (block size of one word) lowest 2 bits of address will not be used for the cache index
- For block sizes of multiple words more of the lowest address bits will be unused
  - 4 word cache blocks = 4 unused bits
  - 8 word cache blocks = 5 unused bits
  - Etc.

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 19

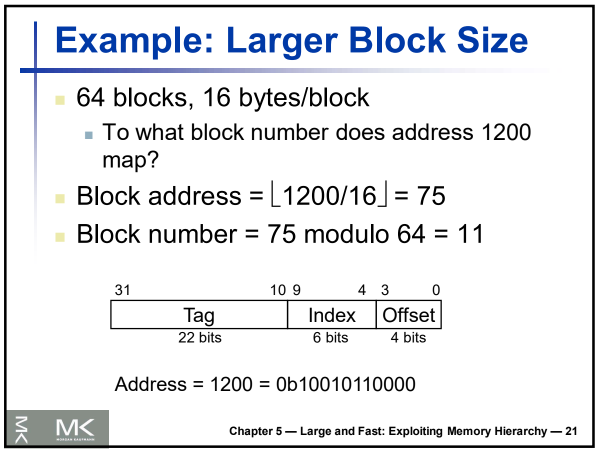


## Block Size

Take advantage of locality: increase block size
- Another advantage: Reduces size of tag memory!
- Potential disadvantage: Fewer blocks in the cache

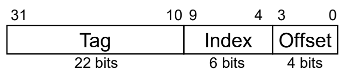

Valid bit
Tag (26 bits)
Data (4 words, 16 bytes)
Example: 4-block, 16-word DM cache
0
1
2
3
32-bit BYTE address
Block offset bits: 4 (16 bytes/block)
Tag bits: 26 (=32-4-2)
Index bits: 2 (4 indexes)

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 20



## 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



Address = 1200 = 0b10010110000

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 21

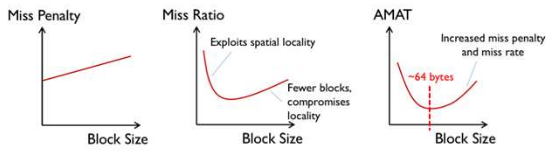
## Explanation

- Address = 1200 = 0b10010110000
- Byte addressing so each block covers 16 addresses
  - 16 addresses represented by 4 bits
- Remove lower 4 bits from 0b10010110000
  - 0b1001011 = 75 = block address
- Next 6 bytes are the index
  - 0b001011 = 11 = cache block number
- Remaining bits are the tag
  - 0b1 = tag

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 22

## Block Size Tradeoffs

- Larger block sizes…
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the block into the cache
  - Can increase the average hit time and miss rate
- Average Access Time (AMAT) = HitTime + MissPenalty*MR



Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 23

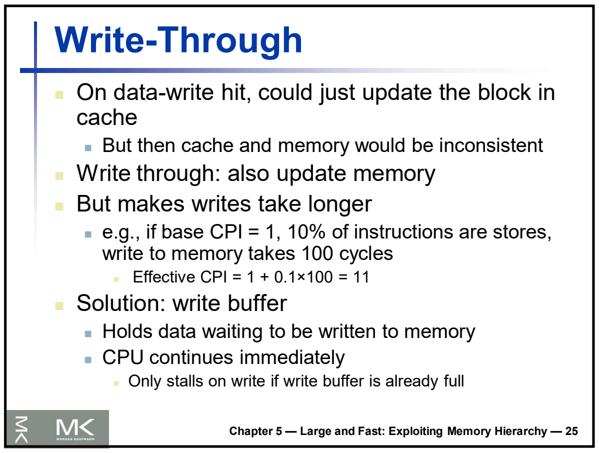
## Direct-Mapped Cache Problem: Conflict Misses

| | Word Address | Cache Line index | Hit/ Miss |
|---|---|---|---|
| Loop A: Pgm at 1024, data at 37: | 1024 | 0 | HIT |
| | 37 | 37 | HIT |
| | 1025 | 1 | HIT |
| | 38 | 38 | HIT |
| | 1026 | 2 | HIT |
| | 39 | 39 | HIT |
| | 1024 | 0 | HIT |
| | 37 | 37 | HIT |
| | … | | |
| Loop B: Pgm at 1024, data at 2048: | 1024 | 0 | MISS |
| | 2048 | 0 | MISS |
| | 1025 | 1 | MISS |
| | 2049 | 1 | MISS |
| | 1026 | 2 | MISS |
| | 2050 | 2 | MISS |
| | 1024 | 0 | MISS |
| | 2048 | 0 | MISS |
| | … | | |

Assume:
1024-line DM cache
Block size = 1 word
Consider looping code, in steady state
Assume WORD, not BYTE, addressing

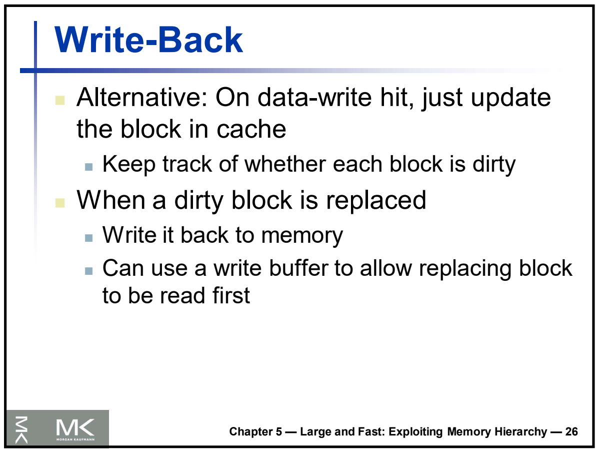
Inflexible mapping (each address can only be in one cache location) → Conflict misses!

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 24



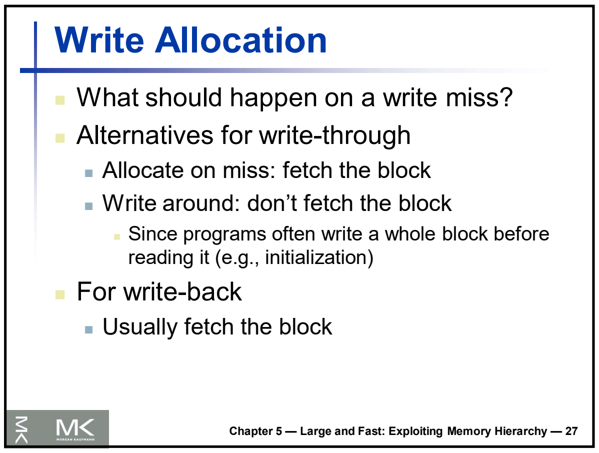
## Write-Through

- 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



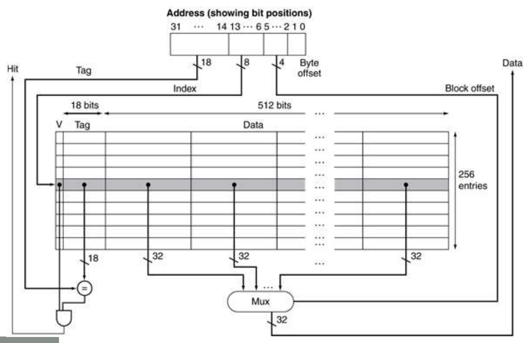
## Write Allocation

- 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

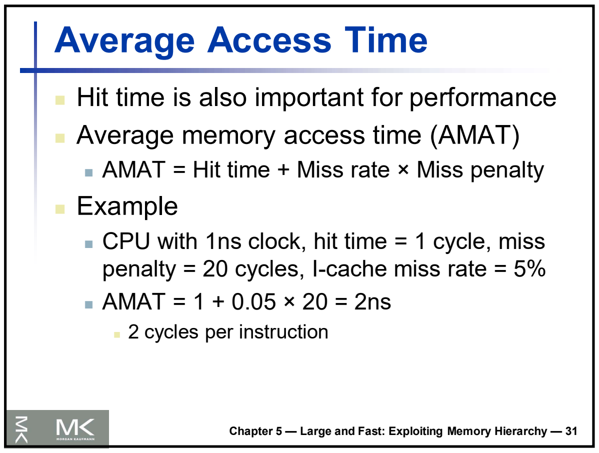


## Measuring Cache Performance

§5.4 Measuring and Improving Cache Performance

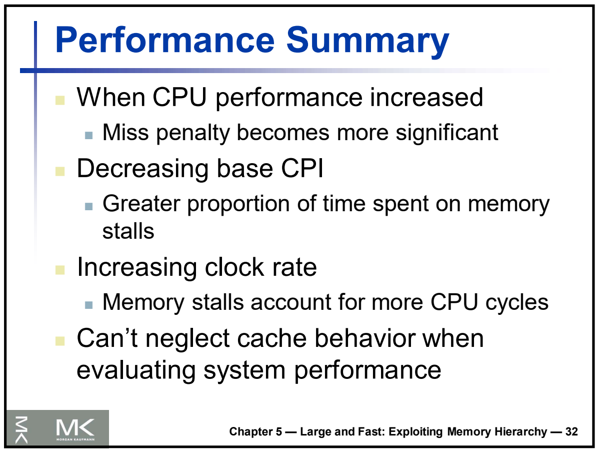
- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

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



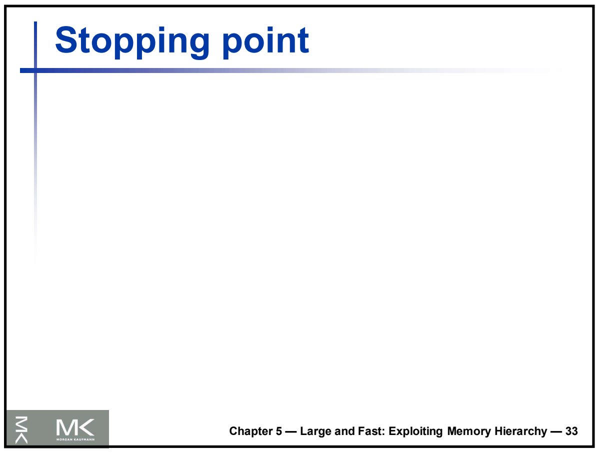
## Average Access Time

- 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

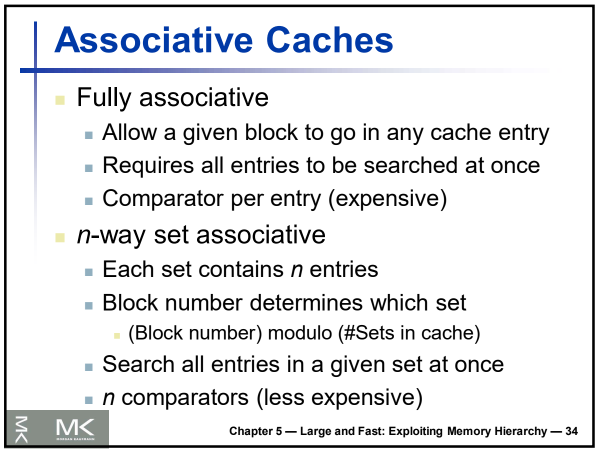


## Performance Summary

- 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



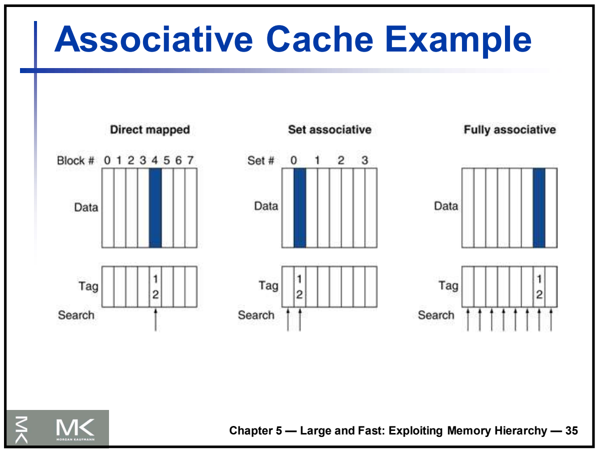
## Stopping point



## Associative Caches

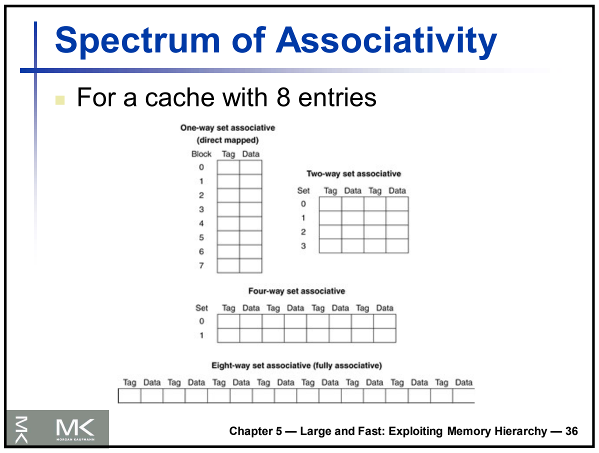
- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  -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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 34



## Associative Cache Example

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 35

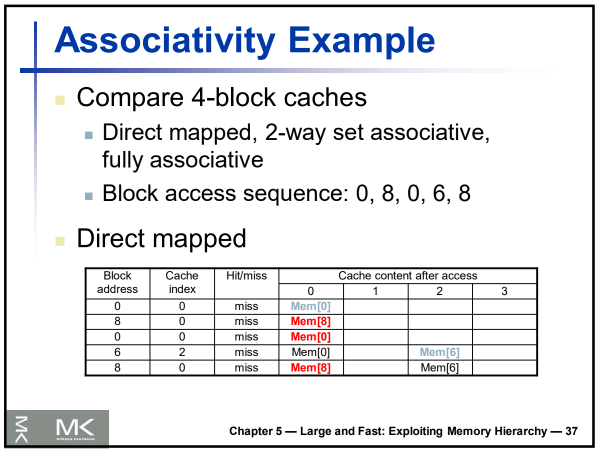


## Spectrum of Associativity

- 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 36

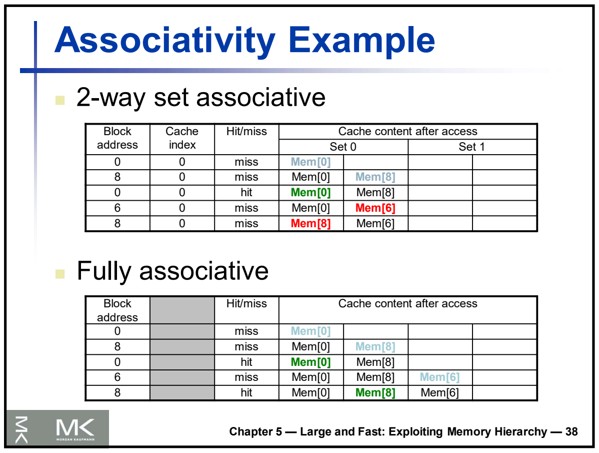


## Associativity Example

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

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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 37



## Associativity Example

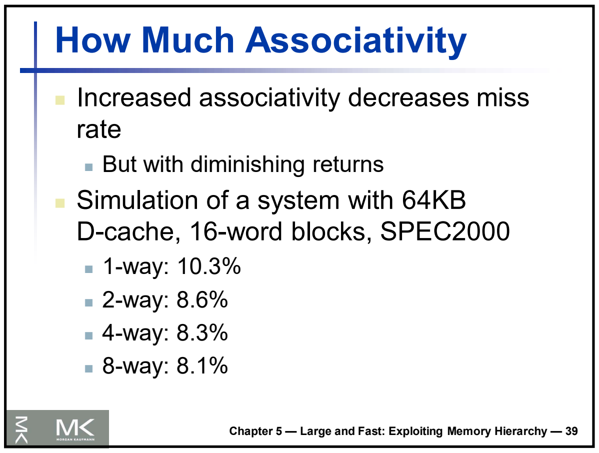
- 2-way set associative

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

- Fully associative

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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 38

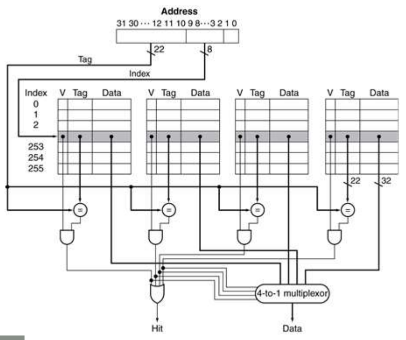


## How Much Associativity

- 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%

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 39

## Set Associative Cache Organization



Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 40

## Replacement Policy

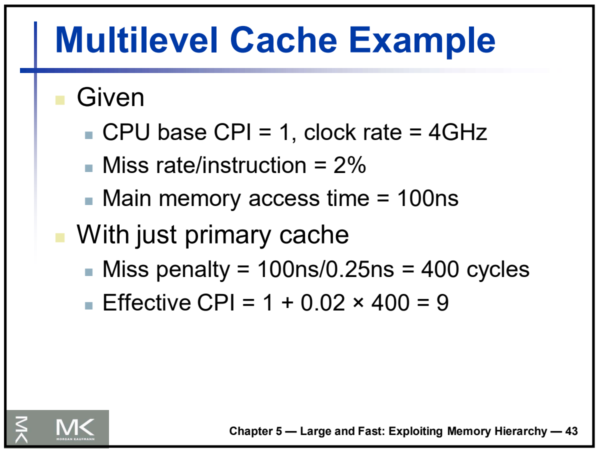
- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 41

## Multilevel Caches

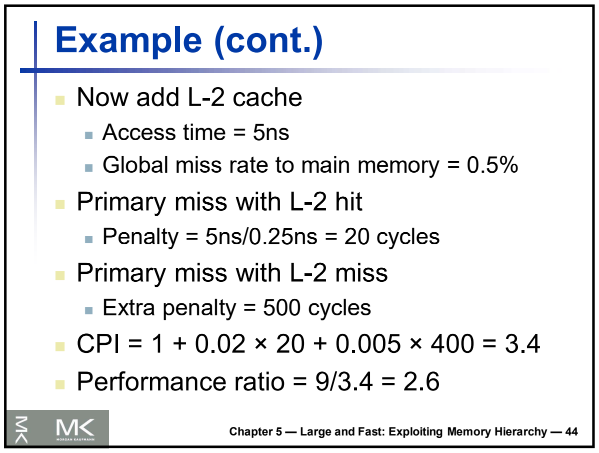
- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 42



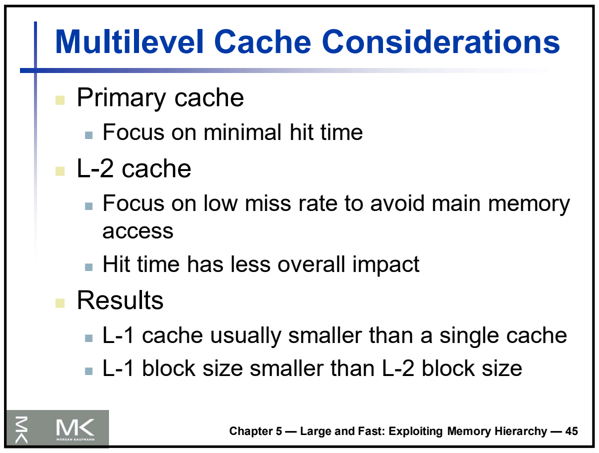
## Multilevel Cache Example

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



## Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6



## Multilevel Cache Considerations

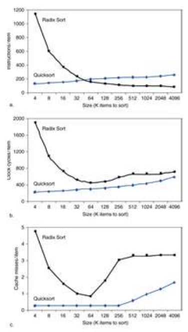
- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

## 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



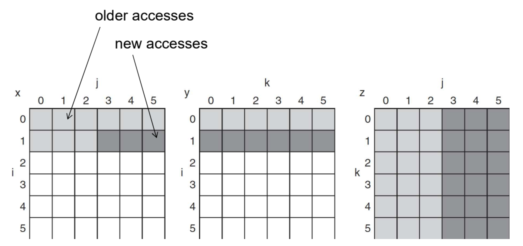
## Software Optimization via Blocking

- 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;
}
```

## DGEMM Access Pattern

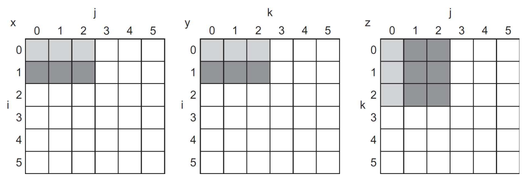
- C, A, and B arrays

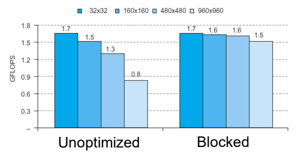


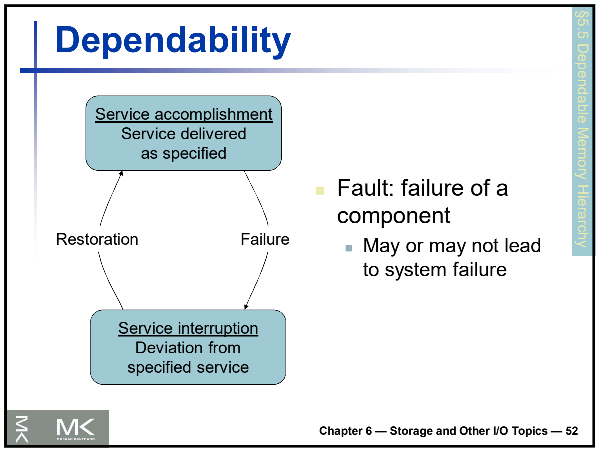
## Cache Blocked DGEMM

```
#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

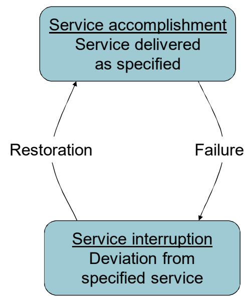






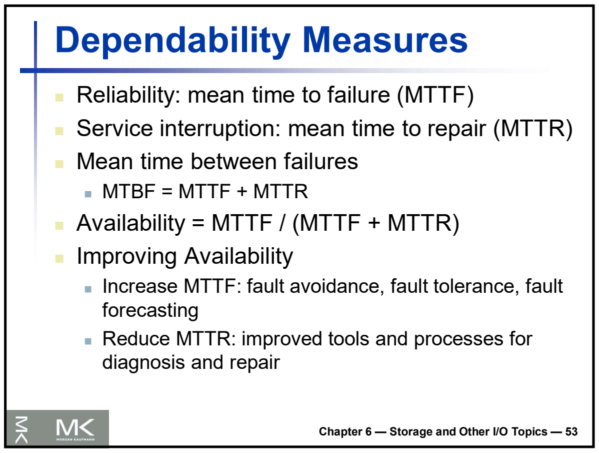
## Dependability

§5.5 Dependable Memory Hierarchy


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

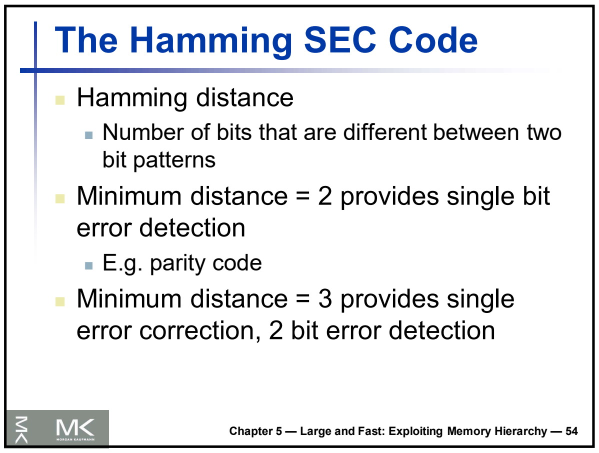
Chapter 6 — Storage and Other I/O Topics — 52



## 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

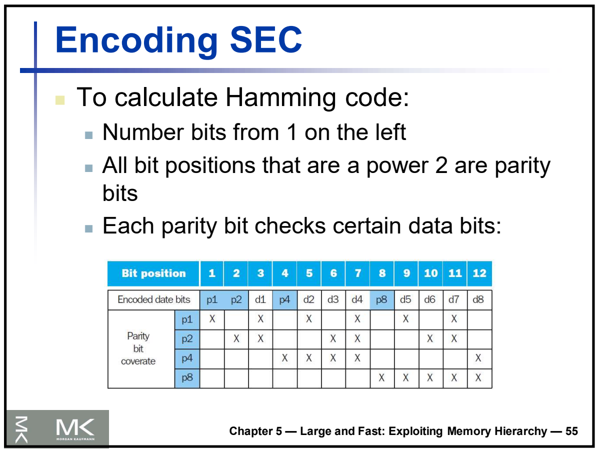
Chapter 6 — Storage and Other I/O Topics — 53



## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 54

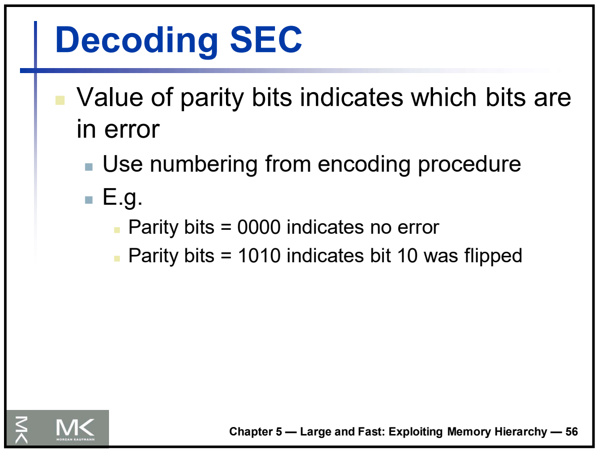


## Encoding SEC

- 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 |
| 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 |

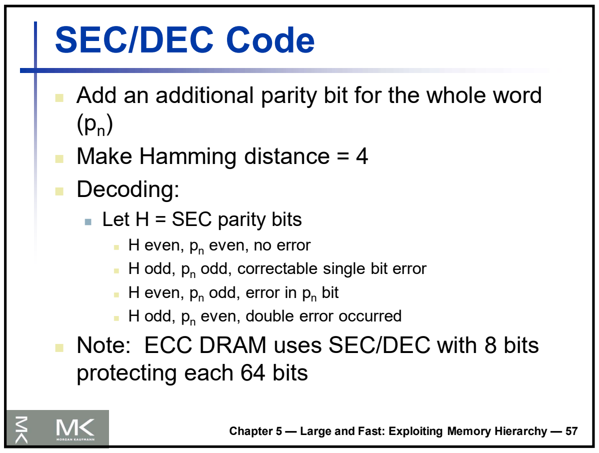
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 55



## 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

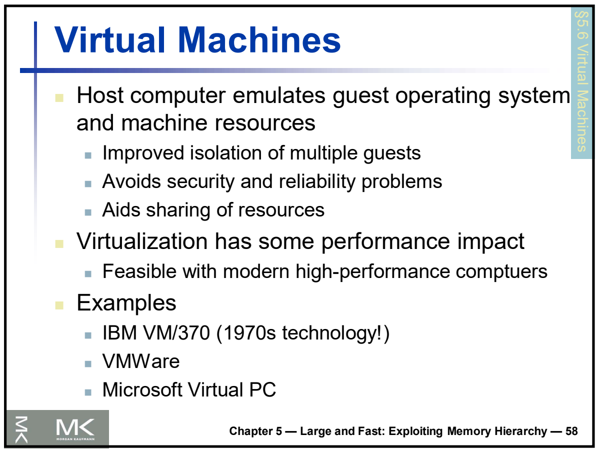
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 56



## SEC/DEC 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/DEC with 8 bits protecting each 64 bits

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 57

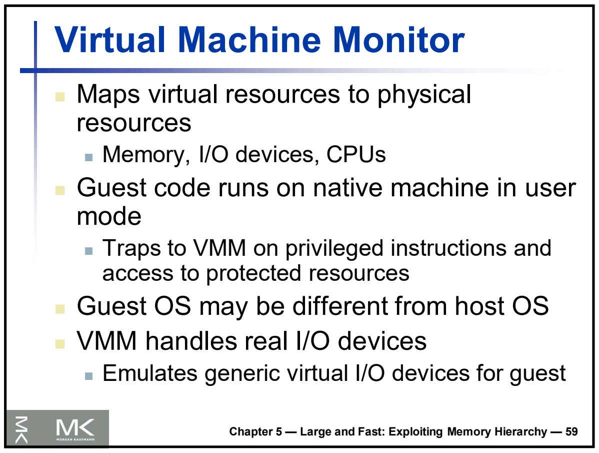


## Virtual Machines

§5.6 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

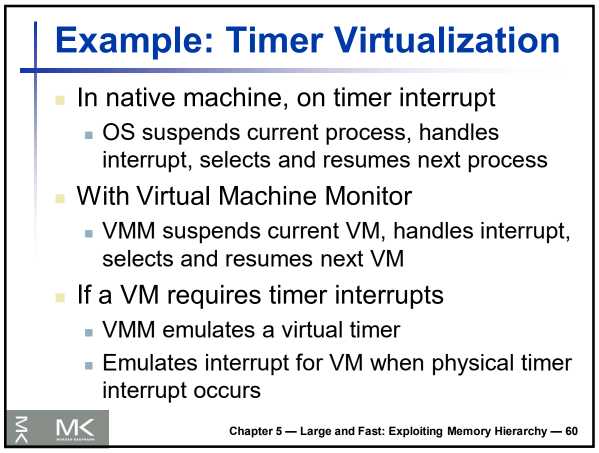
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 58



## Virtual Machine Monitor

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

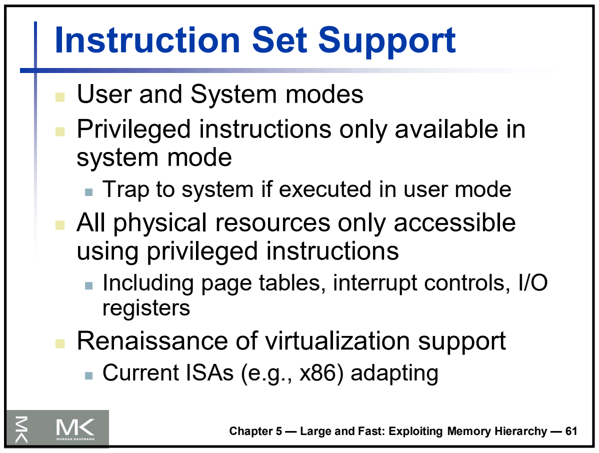
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 59



## 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

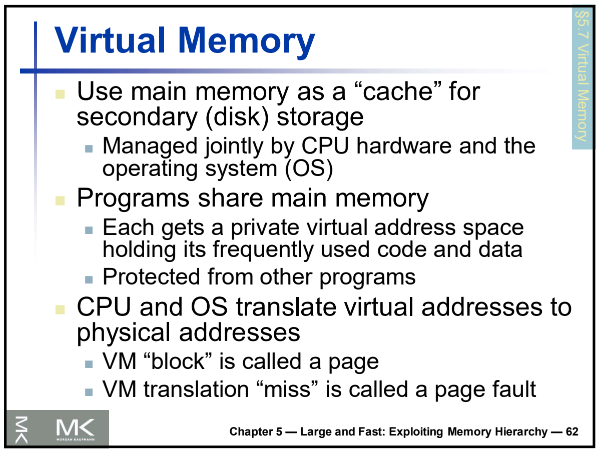
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 60



## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 61

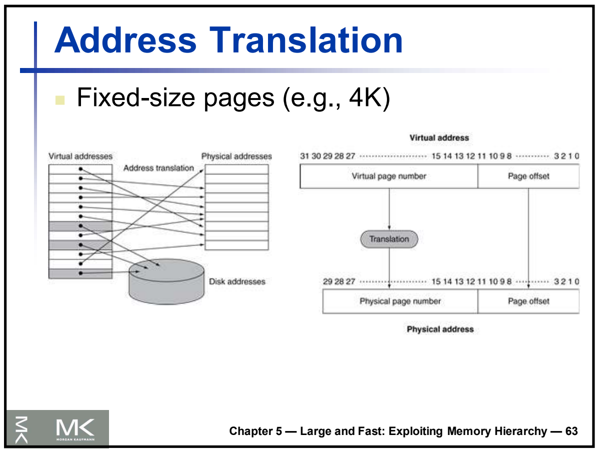


## Virtual Memory

§5.7 Virtual Memory

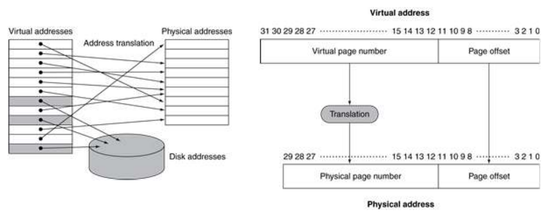
- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 62



## Address Translation

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



Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 63

## 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

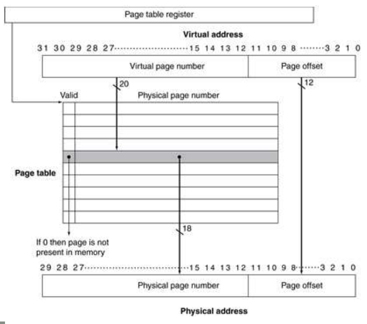
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 64

## Page Tables

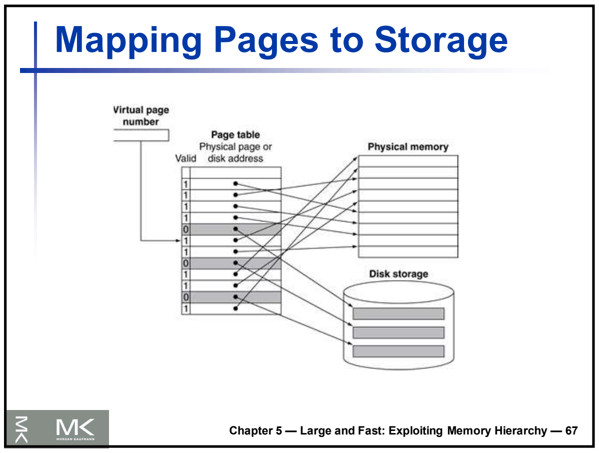
- Stores placement information
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- If page is not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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 65

## Translation Using a Page Table

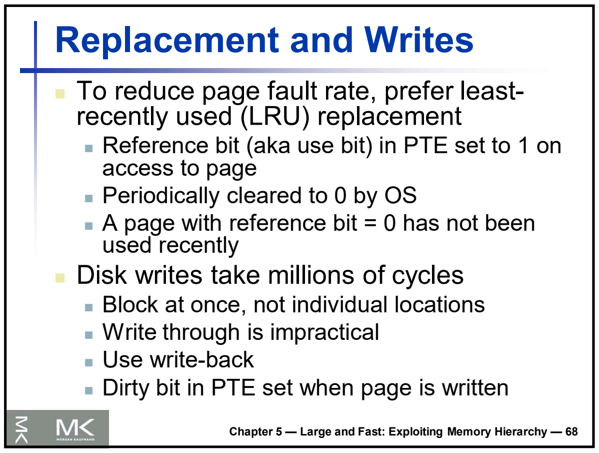


Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 66



## Mapping Pages to Storage

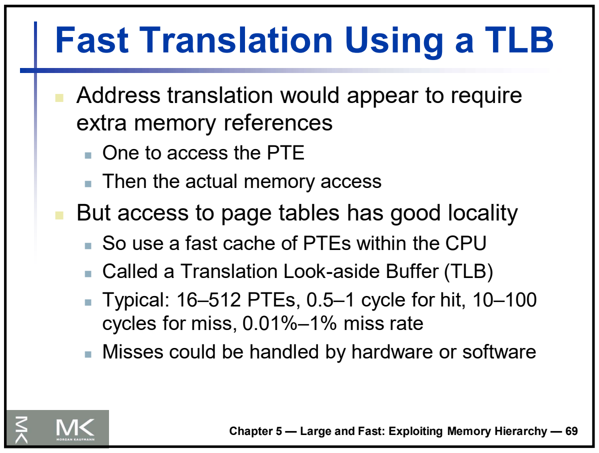
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 67



## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 68

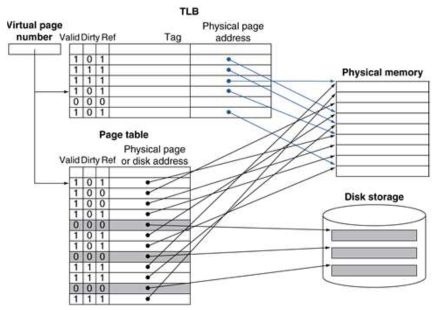


## Fast Translation Using a TLB

- 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 69

## Fast Translation Using a TLB



Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 70

## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 71

## TLB Miss Handler

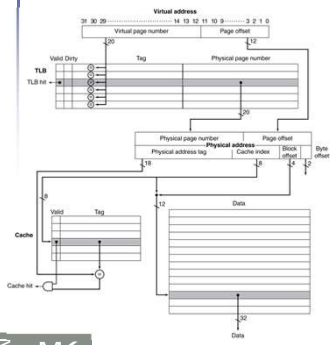
- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not preset
- Must recognize TLB miss before destination register overwritten
  - Raise exception
- Handler copies PTE from memory to TLB
  - Then restarts instruction
  - If page not present, page fault will occur

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 72

## Page Fault Handler

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

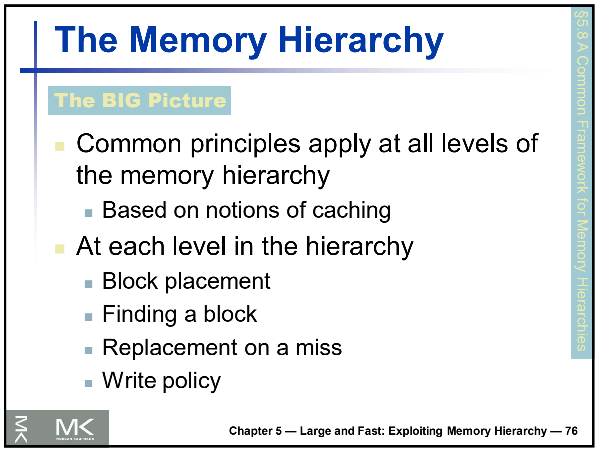
## TLB and Cache Interaction



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

## Memory Protection

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



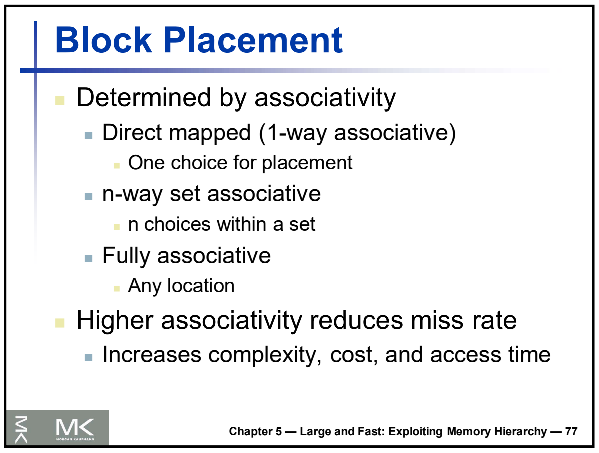
## The Memory Hierarchy

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

§5.8 A Common Framework for Memory Hierarchies

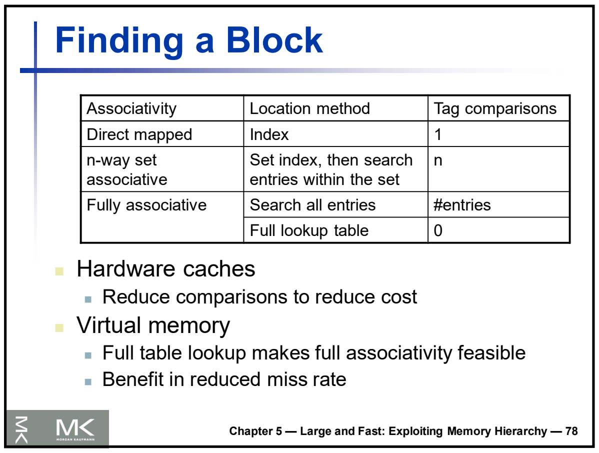
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 76



## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 77

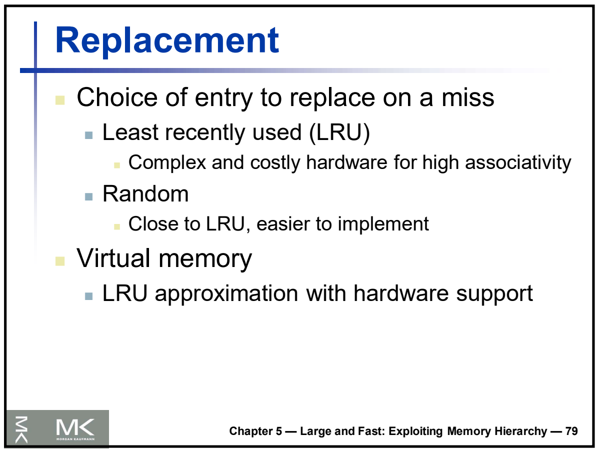


## Finding a Block

| 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

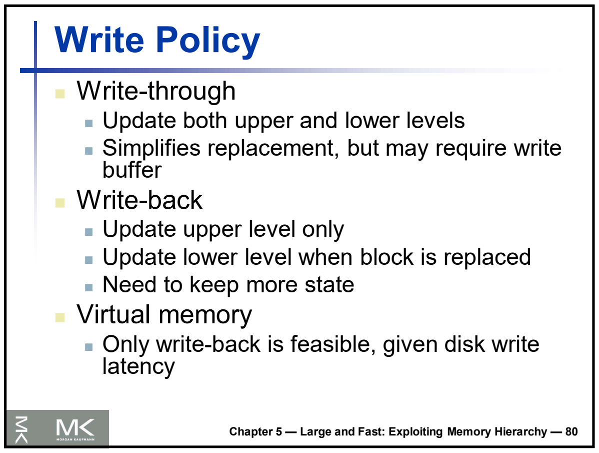
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 78



## 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

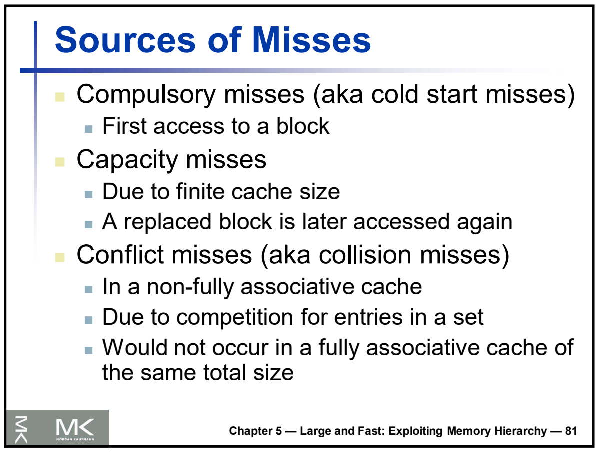
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 79



## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 80



## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 81

## Cache Design Trade-offs

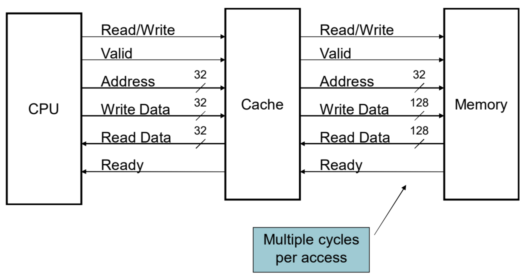
| Design change | Effect on miss rate | Negative performance effect |
|---|---|---|
| Increase cache size | Decrease capacity misses | May increase access time |
| Increase associativity | Decrease conflict misses | May increase access time |
| Increase block size | Decrease compulsory misses | Increases 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 byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

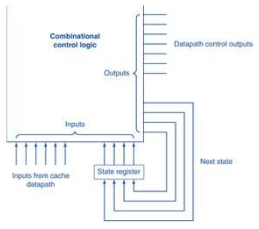
| 31 … 10 | 9 … 4 | 3 … 0 |
|---|---|---|
| Tag | Index | Offset |
| 18 bits | 10 bits | 4 bits |

## Interface Signals

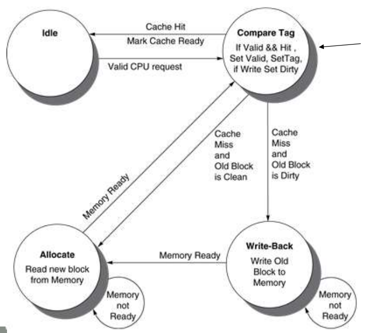


## Finite State Machines

- 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


Idle
Cache Hit
Mark Cache Ready
Valid CPU request
Compare Tag
If Valid && Hit , Set Valid, SetTag, if Write Set Dirty
Cache Miss and Old Block is Clean
Cache Miss and Old Block is Dirty
Memory Ready
Allocate
Read new block from Memory
Memory not Ready
Memory Ready
Write-Back
Write Old Block to Memory
Memory not Ready

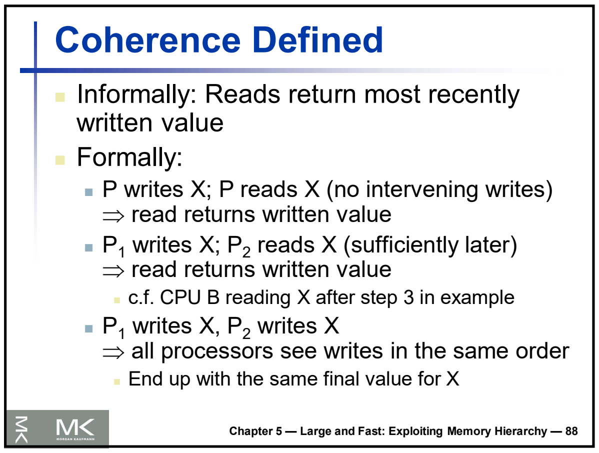
Could partition into separate states to reduce clock cycle time

## Cache Coherence Problem

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

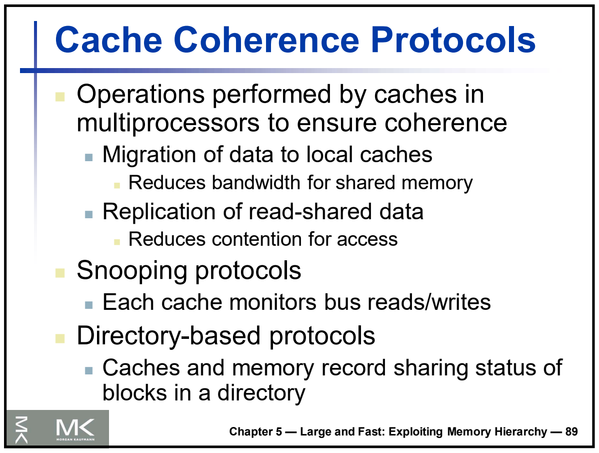
| Time step | Event | CPU A's cache | CPU B's cache | Memory |
|---|---|---|---|---|
| 0 | | | | 0 |
| 1 | CPU A reads X | 0 | | 0 |
| 2 | CPU B reads X | 0 | 0 | 0 |
| 3 | CPU A writes 1 to X | 1 | 0 | 1 |

§5.10 Parallelism and Memory Hierarchies: Cache Coherence



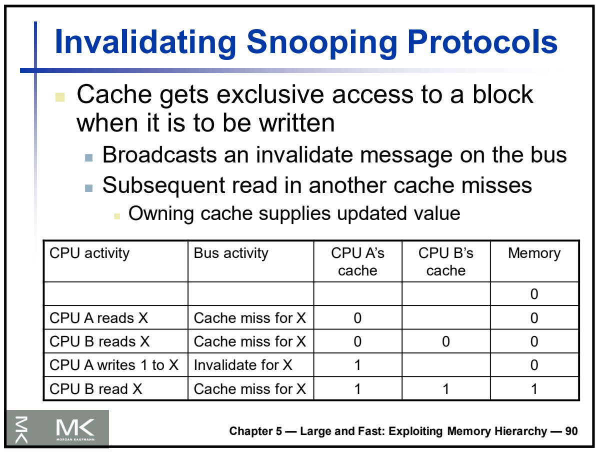
## Coherence Defined

- 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

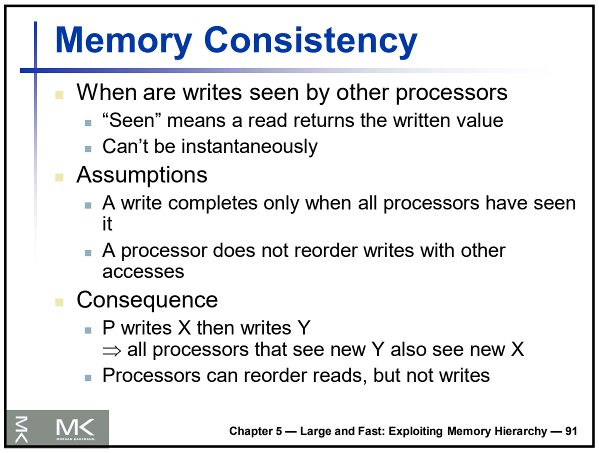
- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based 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

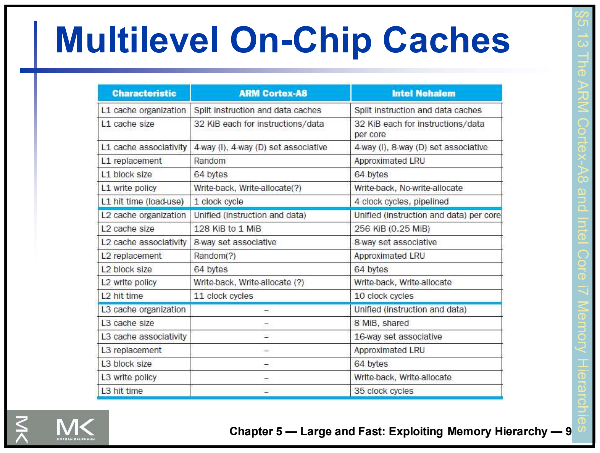
| CPU activity | Bus activity | CPU A's cache | CPU B's cache | Memory |
|---|---|---|---|---|
| | | | | 0 |
| CPU A reads X | Cache miss for X | 0 | | 0 |
| CPU B reads X | Cache miss for X | 0 | 0 | 0 |
| CPU A writes 1 to X | Invalidate for X | 1 | | 0 |
| CPU B read X | Cache miss for X | 1 | 1 |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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 91

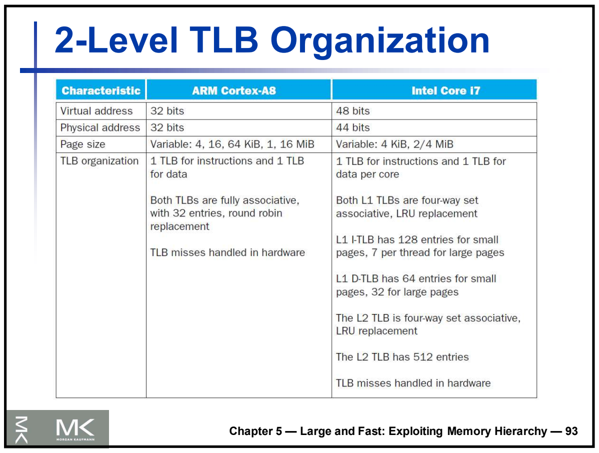


## Multilevel On-Chip Caches

§5.13 The ARM Cortex-A8 and Intel Core i7 Memory Hierarchies

| Characteristic | ARM Cortex-A8 | Intel Nehalem |
|---|---|---|
| L1 cache organization | Split instruction and data caches | Split instruction and data caches |
| L1 cache size | 32 KiB each for instructions/data | 32 KiB each for instructions/data per core |
| L1 cache associativity | 4-way (I), 4-way (D) set associative | 4-way (I), 8-way (D) set associative |
| L1 replacement | Random | Approximated LRU |
| L1 block size | 64 bytes | 64 bytes |
| L1 write policy | Write-back, Write-allocate(?) | Write-back, No-write-allocate |
| L1 hit time (load-use) | 1 clock cycle | 4 clock cycles, pipelined |
| L2 cache organization | Unified (instruction and data) | Unified (instruction and data) per core |
| L2 cache size | 128 KiB to 1 MiB | 256 KiB (0.25 MiB) |
| L2 cache associativity | 8-way set associative | 8-way set associative |
| L2 replacement | Random(?) | Approximated LRU |
| L2 block size | 64 bytes | 64 bytes |
| L2 write policy | Write-back, Write-allocate (?) | Write-back, Write-allocate |
| L2 hit time | 11 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 |

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 92



## 2-Level TLB Organization

| Characteristic | ARM Cortex-A8 | Intel Core i7 |
|---|---|---|
| Virtual address | 32 bits | 48 bits |
| Physical address | 32 bits | 44 bits |
| Page size | Variable: 4, 16, 64 KiB, 1, 16 MiB | Variable: 4 KiB, 2/4 MiB |
| TLB organization | 1 TLB for instructions and 1 TLB for data<br><br>Both TLBs are fully associative, with 32 entries, round robin replacement<br><br>TLB misses handled in hardware | 1 TLB for instructions and 1 TLB for data per core<br><br>Both L1 TLBs are four-way set associative, LRU replacement<br><br>L1 I-TLB has 128 entries for small pages, 7 per thread for large pages<br><br>L1 D-TLB has 64 entries for small pages, 32 for large pages<br><br>The L2 TLB is four-way set associative, LRU replacement<br><br>The L2 TLB has 512 entries<br><br>TLB misses handled in hardware |

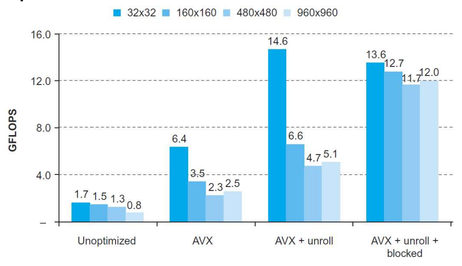
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 93

## Supporting Multiple Issue

- Both have multi-banked caches that allow multiple accesses per cycle assuming no bank conflicts
- Core i7 cache optimizations
  - Return requested word first
  - Non-blocking cache
    - Hit under miss
    - Miss under miss
  - Data prefetching

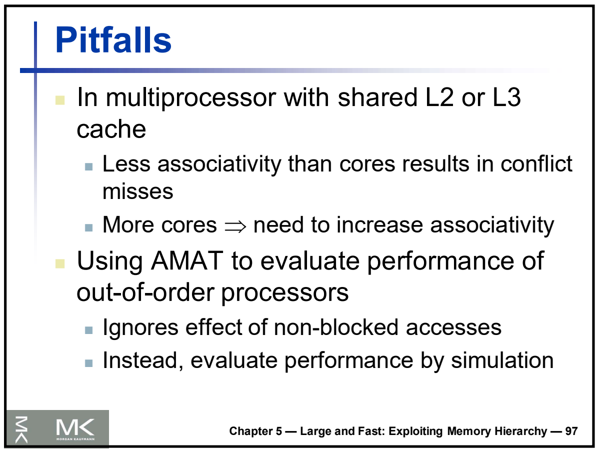
## DGEMM

- Combine cache blocking and subword parallelism



## Pitfalls

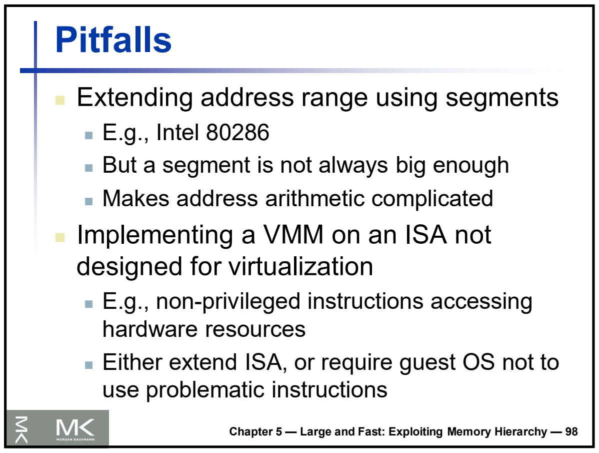
- Byte vs. word addressing
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4
- Ignoring memory system effects when writing or generating code
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality



## Pitfalls

- 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 performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation

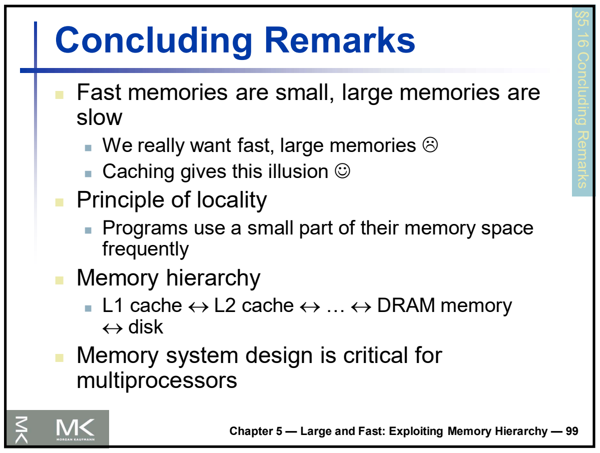
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 97



## 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 98



## Concluding Remarks

§5.16 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

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 99