**SFU**

# CMPT 450/750: Computer Architecture Fall 2022

## Memory Hierarchy

Alaa Alameldeen & Arrvindh Shriraman

# Revisiting Processor Performance

- **Program Execution Time =**

    **(CPU Clock Cycles + Memory Stall Cycles) x Clock Cycle Time**

- **For each instruction:**

    CPI = CPI(Perfect Cache) + Memory stall cycles per instruction

- **With no caches, all memory requests require accessing main memory**

    ➢ Very long latency

- **Caches filter out many memory accesses**

    ➢ Reduces execution time

    ➢ Reduces memory bandwidth & power

# Cache Performance

- **Memory stall cycles Per Instruction =**

  **Cache Misses per instruction x Miss Penalty**

- **Processor Performance:**

  CPI = CPI(Perfect Cache)

  + Misses per instruction x Miss Penalty

- **Average Memory Access Time =**

  Hit Time + Miss rate x Miss penalty

- **Cache hierarchies attempt to reduce average memory access time**

# Cache Performance Metrics

- **Hit rate: #hits / #accesses**

- **Miss rate: #misses / #accesses**

- **Misses per instruction (or 1000 instructions: MPKI)**
  - ➢Misses/Instruction = miss rate x memory accesses / Instruction count
    - = miss rate x memory accesses per instruction
  - ➢MPKI= 1000 x miss rate x memory accesses per instruction

- **Hit time: time from request issued to cache until data is returned to the processor**
  - ➢Depends on cache design parameters
  - ➢Bigger caches, larger associativity, or more ports increase hit time

- **Miss penalty: depends on memory hierarchy parameters**

- **We need a memory hierarchy to reduce the miss penalty**

# Cache Performance Example

**Program P running on a processor has an average IPC of 0.5. 40% of program P's instructions are loads and stores. P has an L1 miss rate of 10% and an average miss penalty of 30 cycles. How much faster will P run if all loads and stores are cache hits?**

- **CPI = 1/IPC = 2.0; Memory accesses per instruction = 40% = 0.4**

- **Misses per instruction = memory accesses per instruction x miss rate**

$$= 0.4 \quad x \quad 0.1 \quad = \quad 0.04$$

- **CPI = CPI(Perfect Cache) + misses per instruction x miss penalty**

  **2.0 = CPI(Perfect Cache) + 0.04 x 30**

- **CPI(Perfect Cache) = 2.0 – 0.04 x 30 = 0.8**

- **Speedup for perfect cache = CPI/CPI(Perfect Cache) = 2.0/0.8 = 2.5 x**
  - ➤ Perfect cache is 2.5 x faster (or 150% faster)

# Miss Rate OR Misses Per Instruction?

- **Miss rate used to compute average memory access time (AMAT)**
  Hit Time + Miss rate x Miss penalty

- **Misses Per Instruction (or MPKI) used to compute CPI & Execution Time**
  CPI = CPI(Perfect Cache) + Misses per instruction x Miss Penalty
  Execution Time = Inst/Program x CPI x Cycle Time

- **MPKI is more directly related to performance**

- **Is it possible to have worse performance with a better miss rate?**

# MPKI vs. Miss Rate Example

Programs P1 and P2 run on a processor with a 4GHz frequency, an L1 cache hit time of 1 ns and an L1 average miss penalty of 30 ns. P1 has a miss rate of 5% and an MPKI of 25. P2 has a miss rate of 10% and an MPKI of 10. Both programs have a CPI of 0.5 with a perfect L1 cache. Compare P1 and P2's AMAT and CPI.

Note: Cycle Time = 1/frequency = 0.25 ns

- AMAT(P1) = Hit Time + Miss Rate(P1) x Miss Penalty = 1 + 0.05 x 30 = 2.5 ns
- AMAT(P2) = Hit Time + Miss Rate(P2) x Miss Penalty = 1 + 0.1 x 30 = 4 ns
- CPI(P1) = CPI(Perfect Cache) + misses per instruction(P1) x miss penalty

    = 0.5 + (25/1000) x (30/0.25) =  3.5

- CPI(P2) = CPI(Perfect Cache) + misses per instruction(P2) x miss penalty

    = 0.5 + (10/1000) x (30/0.25) =  1.7

- P1 has lower average memory access time but worse performance. Why?

# Why Do Caches Work?

- **Spatial Locality**
  - If data at a certain address is accessed, it is likely that data located at nearby addresses will also be accessed in the (near) future
  - Implications:
    - ❑ Cache line (block) size tradeoff
    - ❑ Prefetching brings lines to the cache before they are demanded
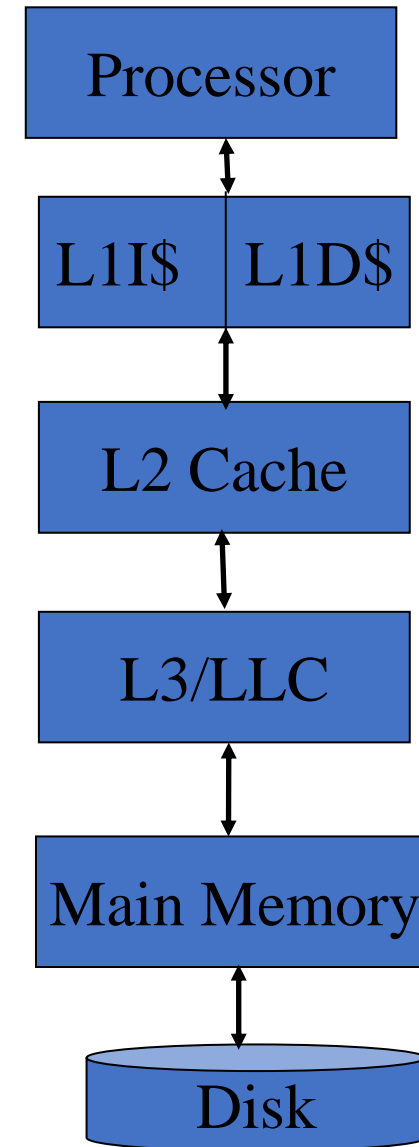
- **Temporal Locality**
  - If data at a certain address is accessed, it is likely the same data will be accessed in the (near) future
  - Implications:
    - ❑ Replacement policies try to predict which lines will be not be accessed (or will be accessed furthest) in the future
    - ❑ Insertion policies prioritize lines that will be accessed sooner
    - ❑ Dead block predictors predict which lines will be dead-on-arrival so they aren't allocated

- **We still need multiple cache levels in the memory hierarchy to bridge the gap between processor and memory speeds**

# Memory Hierarchy

- **First-level caches**
  - ➤ Usually Split I & D caches
  - ➤ Small and fast
- **Second-level caches**
  - ➤ Usually on-die
  - ➤ SRAM cells
- **Third-level… etc.?**
- **Main memory**
  - ➤ DRAM cells
  - ➤ focus on density
- **Solid-State Disk**
- **Hard Disk**
  - ➤ Usually magnetic device, non-volatile
  - ➤ Slow access time

Processor

L1I$ | L1D$

L2 Cache

L3/LLC

Main Memory

Disk

# Memory Hierarchy for a Mobile Device
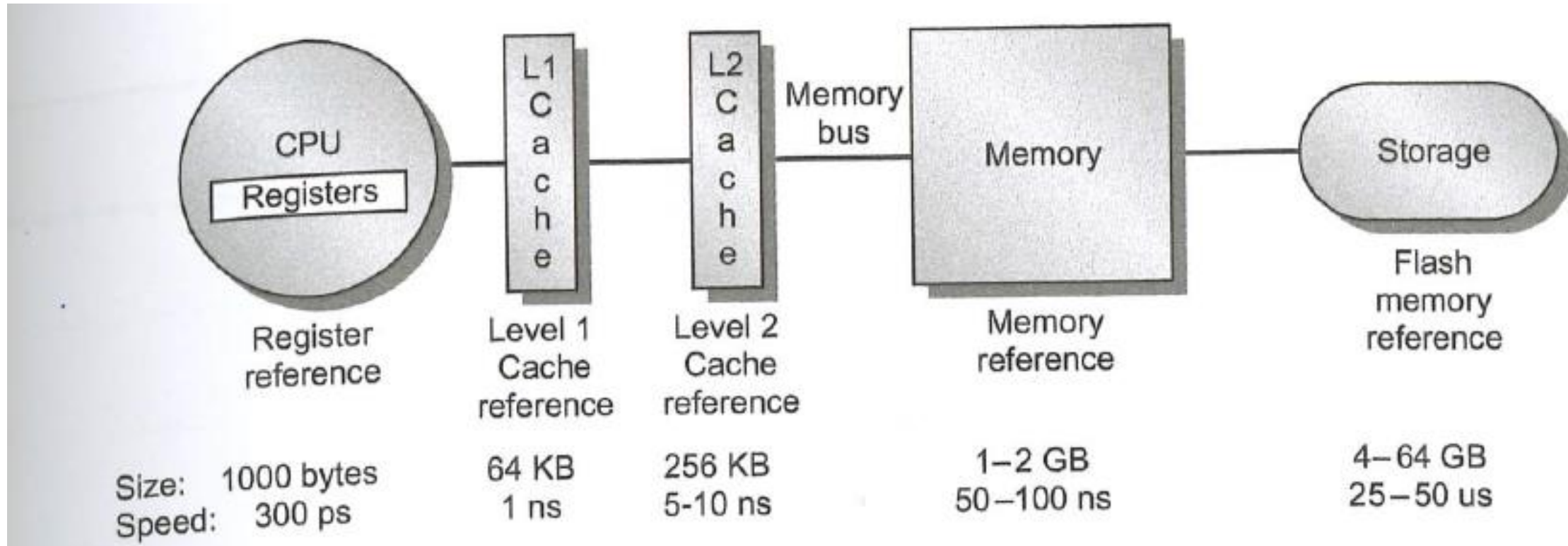
# Memory Hierarchy for a Desktop/Laptop

ARCH Figure 2.1(B)

| | | Register reference | Level 1 Cache reference | Level 2 Cache reference | Level 3 Cache reference | Memory reference | Flash memory reference |
|---|---|---|---|---|---|---|---|
| Laptop | Size: Speed: | 1000 bytes 300 ps | 64 KB 1 ns | 256 KB 3–10 ns | 4-8 MB 10–20 ns | 4–16 GB 50–100 ns | 256 GB-1 TB 50-100 uS |
| Desktop | Size: Speed: | 2000 bytes 300 ps | 64 KB 1 ns | 256 KB 3–10 ns | 8-32 MB 10–20 ns | 8–64 GB 50–100 ns | 256 GB-2 TB 50-100 uS |
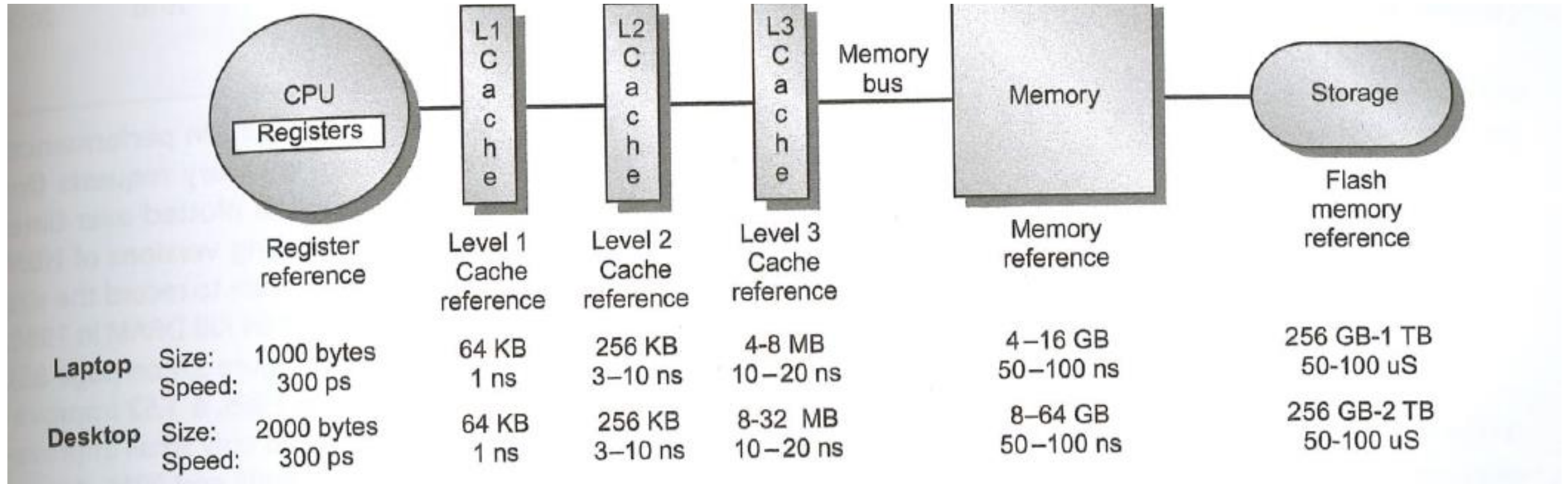
# Memory Hierarchy for a Server

| | Register reference | Level 1 Cache reference | Level 2 Cache reference | Level 3 Cache reference | Memory reference | Disk memory reference | Flash memory reference |
|---|---|---|---|---|---|---|---|
| Size: | 4000 bytes | 64 KB | 256 KB | 16-64 MB | 32–256 GB | 16–64 TB | 1-16 TB |
| Speed: | 200 ps | 1 ns | 3–10 ns | 10–20 ns | 50–100 ns | 5–10 ms | 100-200 us |
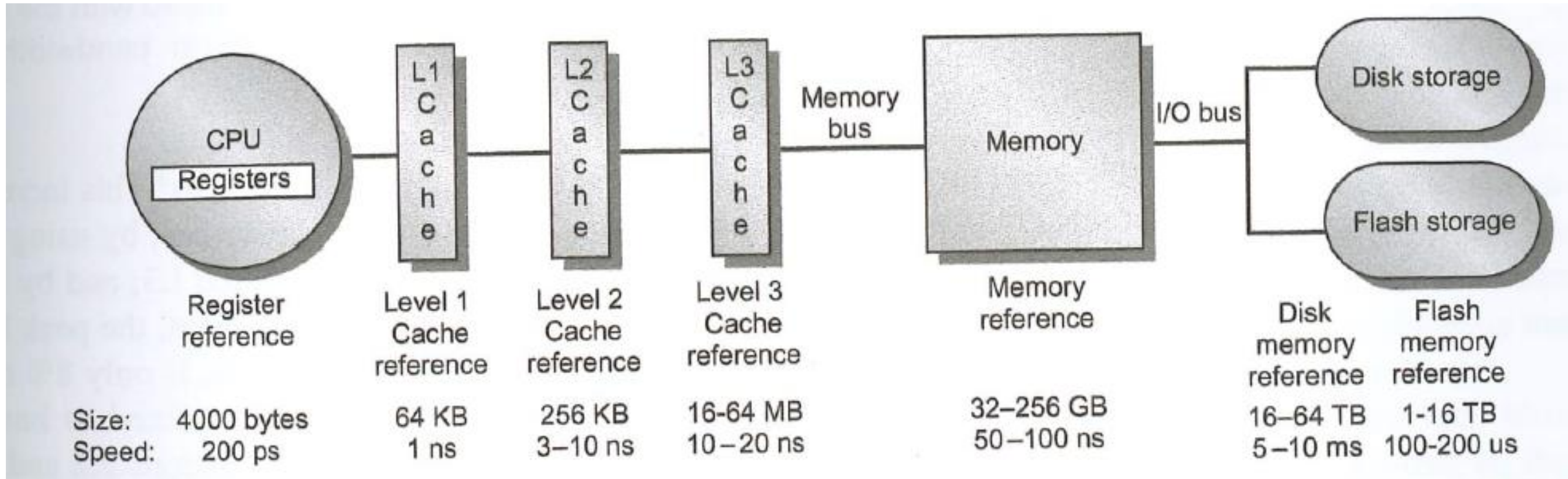
# Basic Cache Structure (Review)

- **Array of blocks (lines)**
  - ➢ Each block is usually 32-128 bytes

- **Finding a block in cache:**

| Data Address | Tag | Index | Offset |
| --- | --- | --- | --- |
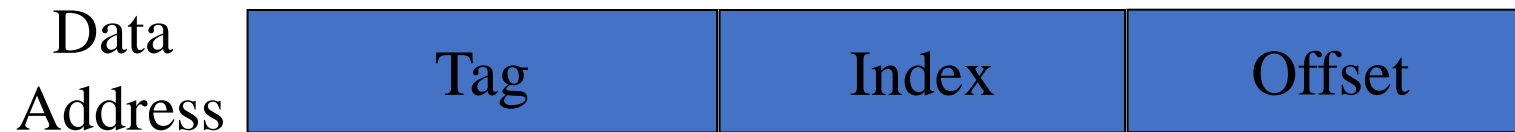
- **Offset: byte offset in block**

- **Index: Which set in the cache is the block located**

- **Tag: Needs to match address tag in cache**
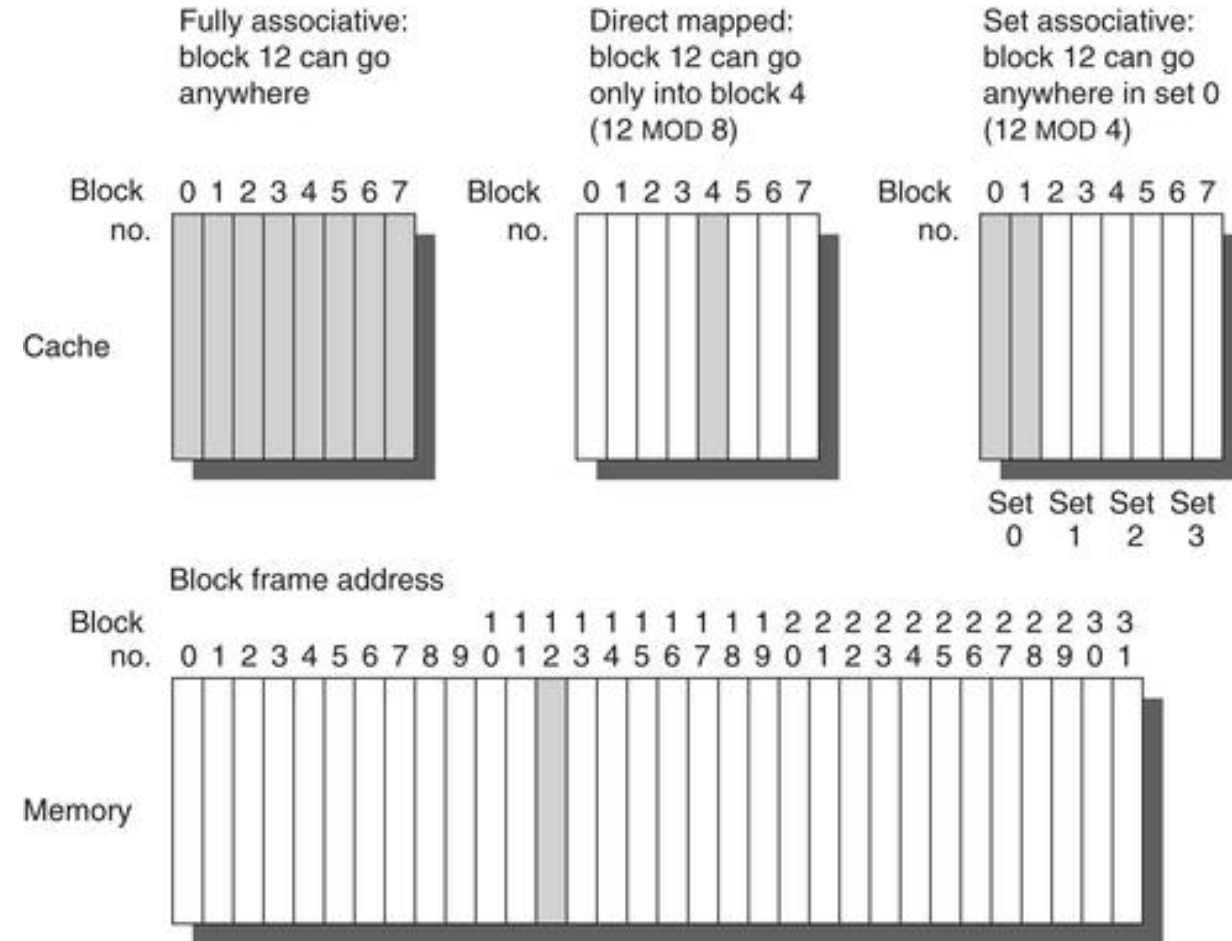
# Locating a Block in the Cache

- **Set associativity**
  - Set: Group of blocks corresponding to same index
  - Each block in the set is called a *Way*
  - 2-way set associative cache: each set contains two blocks
  - Direct-mapped cache: each set contains one block
  - Fully-associative cache: the whole cache is one set

- **Need to check all tags in a set to determine hit/miss status then select correct block**
  - Higher latency for set-associative caches



Fully associative: block 12 can go anywhere

Direct mapped: block 12 can go only into block 4 (12 MOD 8)

Set associative: block 12 can go anywhere in set 0 (12 MOD 4)

14

# Example: Cache Block Placement

- **Consider a 4-way, 32KB cache with 64-byte lines**

- **Where is 48-bit address 0x0000FFFFAB64?**

  ➢ Number of lines = cache size / line size = 32K / 64 = 512

  ➢ Each set contains 4 lines $\Rightarrow$ Number of sets = 512/4 = 128 sets

  ➢ Offset bits = $\log_2 (64)$ = 6: 0x24

  ➢ Index bits = $\log_2 (128)$ = 7: 0x2D

  ➢ Tag bits = 48-(6+7) = 35: 0x00007FFFD

# Cache Associativity Example

Program P runs on a processor with a 4 GHz frequency. The average memory access latency on a cache miss is 40 ns. Which one of these caches gets better performance for P?

1. 64KB direct-mapped cache with miss rate of 3%, hit latency = 3 cycles

2. 64KB 4-way set-associative cache with miss rate of 2%, hit latency = 4 cycles (due to extra latency of tag match/select)

- Cycle time = 1/frequency = 1/4,000,000,000 = 0.25 ns

- Hit Time = cycles/hit x cycle_time

- Average memory access time(1) = Hit Time(1) + Miss rate(1) x miss penalty

$$= 3 \times 0.25 + 0.03 \times 40 = 1.95 \text{ ns}$$

- Average memory access time(2) = Hit Time(2) + Miss rate(1) x miss penalty

$$= 4 \times 0.25 + 0.02 \times 40 = 1.8 \text{ ns}$$

- Cache 2 (4-way) is better even if hit time is higher.

- What if frequency is lower for set-associative cache?

# Types of Cache Misses

- **Compulsory (cold) misses**: **First access to a block. Compulsory misses occur even for infinite size cache**
  - ➢Could be reduced by prefetching blocks before they are demanded

- **Capacity misses**: **A cache cannot contain all blocks needed in a program. some blocks are discarded then later accessed. Capacity misses occur in a fully-associative cache.**
  - ➢Could be reduced with insertion/replacement policies and dead block prediction

- **Conflict misses**: **Blocks mapping to the same set may be discarded (in direct-mapped and set-associative caches).**
  - ➢Could be reduced by increasing associativity or better replacement/insertion policies

- **Coherence misses**: **Misses due to shared memory accesses**
  - ➢Discussed later this course

# Miss Distribution

**ARCH Figure B.8**

| Cache size (KB) | Degree associative | Total miss rate | Miss rate components (relative percent) (sum = 100% of total miss rate) | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Compulsory | | Capacity | | Conflict | |
| 4 | 1-way | 0.098 | 0.0001 | 0.1% | 0.070 | 72% | 0.027 | 28% |
| 4 | 2-way | 0.076 | 0.0001 | 0.1% | 0.070 | 93% | 0.005 | 7% |
| 4 | 4-way | 0.071 | 0.0001 | 0.1% | 0.070 | 99% | 0.001 | 1% |
| 4 | 8-way | 0.071 | 0.0001 | 0.1% | 0.070 | 100% | 0.000 | 0% |
| 8 | 1-way | 0.068 | 0.0001 | 0.1% | 0.044 | 65% | 0.024 | 35% |
| 8 | 2-way | 0.049 | 0.0001 | 0.1% | 0.044 | 90% | 0.005 | 10% |
| 8 | 4-way | 0.044 | 0.0001 | 0.1% | 0.044 | 99% | 0.000 | 1% |
| 8 | 8-way | 0.044 | 0.0001 | 0.1% | 0.044 | 100% | 0.000 | 0% |
| 16 | 1-way | 0.049 | 0.0001 | 0.1% | 0.040 | 82% | 0.009 | 17% |
| 16 | 2-way | 0.041 | 0.0001 | 0.2% | 0.040 | 98% | 0.001 | 2% |
| 16 | 4-way | 0.041 | 0.0001 | 0.2% | 0.040 | 99% | 0.000 | 0% |
| 16 | 8-way | 0.041 | 0.0001 | 0.2% | 0.040 | 100% | 0.000 | 0% |
| 32 | 1-way | 0.042 | 0.0001 | 0.2% | 0.037 | 89% | 0.005 | 11% |
| 32 | 2-way | 0.038 | 0.0001 | 0.2% | 0.037 | 99% | 0.000 | 0% |
| 32 | 4-way | 0.037 | 0.0001 | 0.2% | 0.037 | 100% | 0.000 | 0% |
| 32 | 8-way | 0.037 | 0.0001 | 0.2% | 0.037 | 100% | 0.000 | 0% |
| 64 | 1-way | 0.037 | 0.0001 | 0.2% | 0.028 | 77% | 0.008 | 23% |
| 64 | 2-way | 0.031 | 0.0001 | 0.2% | 0.028 | 91% | 0.003 | 9% |
| 64 | 4-way | 0.030 | 0.0001 | 0.2% | 0.028 | 95% | 0.001 | 4% |
| 64 | 8-way | 0.029 | 0.0001 | 0.2% | 0.028 | 97% | 0.001 | 2% |
| 128 | 1-way | 0.021 | 0.0001 | 0.3% | 0.019 | 91% | 0.002 | 8% |
| 128 | 2-way | 0.019 | 0.0001 | 0.3% | 0.019 | 100% | 0.000 | 0% |
| 128 | 4-way | 0.019 | 0.0001 | 0.3% | 0.019 | 100% | 0.000 | 0% |
| 128 | 8-way | 0.019 | 0.0001 | 0.3% | 0.019 | 100% | 0.000 | 0% |
| 256 | 1-way | 0.013 | 0.0001 | 0.5% | 0.012 | 94% | 0.001 | 6% |
| 256 | 2-way | 0.012 | 0.0001 | 0.5% | 0.012 | 99% | 0.000 | 0% |
| 256 | 4-way | 0.012 | 0.0001 | 0.5% | 0.012 | 99% | 0.000 | 0% |
| 256 | 8-way | 0.012 | 0.0001 | 0.5% | 0.012 | 99% | 0.000 | 0% |
| 512 | 1-way | 0.008 | 0.0001 | 0.8% | 0.005 | 66% | 0.003 | 33% |
| 512 | 2-way | 0.007 | 0.0001 | 0.9% | 0.005 | 71% | 0.002 | 28% |
| 512 | 4-way | 0.006 | 0.0001 | 1.1% | 0.005 | 91% | 0.000 | 8% |
| 512 | 8-way | 0.006 | 0.0001 | 1.1% | 0.005 | 95% | 0.000 | 4% |

# Common Cache Optimizations

- **Cache optimizations target reducing average memory access time**

**Average memory access time = Hit Time + Miss rate x Miss penalty**

| Technique | Hit time | Miss penalty | Miss rate | Hardware complexity | Comment |
|---|---|---|---|---|---|
| Larger block size | | – | + | 0 | Trivial; Pentium 4 L2 uses 128 bytes |
| Larger cache size | – | | + | 1 | Widely used, especially for L2 caches |
| Higher associativity | – | | + | 1 | Widely used |
| Multilevel caches | | + | | 2 | Costly hardware; harder if L1 block size ≠ L2 block size; widely used |
| Read priority over writes | | + | | 1 | Widely used |
| Avoiding address translation during cache indexing | + | | | 1 | Widely used |

**ARCH Figure B.18**

# Virtual vs. Physical Addressing

- **Using virtual addresses to access the L1 cache reduces latency**
  - ➤ Physical addresses need address translation

- **Issues with virtually-addressed caches**
  - ➤ Handing synonyms: multiple VAs mapping to same PA
  - ➤ Address translation needed on L1 misses
  - ➤ Reverse translation needed for coherence in a multiprocessor system
  - ➤ Need to invalidate whole cache on a context switch

- **Some L1 caches are "virtually-indexed, physically tagged" to parallelize cache access with address translation when indexing the cache. PA is still needed to match tags.**

# Multi-Level Cache Example

Program P with 30% loads/stores runs on a processor with a 4 GHz frequency. A main memory access needs 80 ns. Consider the following caches:

1. L1 data cache: 32KB 8-way cache with 4-cycle hit latency and a miss rate of 10%

2. L2 cache: 256KB 8-way cache with 10-cycle hit latency and a miss rate of 30%

3. L3 cache: 6MB 24-way cache with 35-cycle average hit latency and a miss rate of 50%

What is the average memory access time for a system with (1) L1 only; (2) L1 and L2; (3) L1,L2 and L3?

- Cycle time = 1/frequency = 1/4,000,000,000 = 0.25 ns; Hit Time = cycles/hit x cycle_time

- Average memory access time(L1) = Hit Time(L1) + Miss rate(L1) x miss penalty(Memory Access)

$$= 4 \times 0.25 + 0.1 \times 80 = 9 \text{ ns}$$

- Average memory access time(L1,L2) = Hit Time(L1) + Miss rate(L1) x miss penalty(L2 Access)

- L2 Access Latency = Hit Time (L2) + Miss rate (L2) x miss penalty(Memory Access)

$$= 10 \times 0.25 + 0.3 \times 80 = 26.5 \text{ ns}$$

- Average memory access time(L1,L2) = 4 x 0.25 + 0.1 x 26.5 = 3.65 ns

- Average memory access time(L1,L2,L3) = Hit Time(L1) + Miss rate(L1) x miss penalty(L2 Access)

   = Hit Time(L1) + Miss rate(L1) x (Hit Time (L2) + Miss rate(L2) x (Hit Time(L3) + Miss rate(L3) x Miss Penalty(Memory)))

   = 4 x 0.25 + 0.1 x (10 x 0.25 + 0.3 x (35 x 0.25 + 0.5 x 80)) = 2.71 ns

# Cache Management Policies

- **Cache replacement policy:**
  - ➢On a cache line fill, which victim line to replace?
  - ➢Only applicable to set-associative caches
    - ❑ Direct-mapped caches have only one line per set
  - ➢Examples: LRU, more advanced policies
  - ➢Discuss stack algorithms

- **Cache insertion policy:**
  - ➢When a cache line is filled, what would its priority be in the replacement stack?
  - ➢LRU: fill line is inserted in "Most Recently Used" position
  - ➢Other policies: LIP, BIP, DIP
  - ➢Dead block prediction helps determine lines that won't be reused (either bypassed or inserted in LRU position)

# Miss Penalty in Out-of-Order Processors

- **Recall:**

**Memory stall cycles/Instruction =  Cache Misses/instruction x miss penalty**

- **This assumes that the whole miss penalty is observed for all instructions**

- **In modern OoO processors, miss penalty for a single miss may be overlapped with other latencies**
  - ➢Overlapped with executions of other instructions in the instruction window
  - ➢Overlapped with other memory accesses if cache is non-blocking (called memory-level parallelism)

- **So we only need to include non-overlapped miss penalty:**

**Memory stall cycles/Instruction**

  **=  Cache Misses/instruction x (miss latency – overlapped miss latency)**

# Non-Blocking Cache Hierarchy

- **Superscalar processors can reduce average memory latency by overlapping multiple misses**

- **Cache hierarchies can simultaneously service multiple memory requests**
  - Do not block cache references that do not need the miss data (Called hit-under-miss optimization)
  - Service multiple miss requests to memory concurrently (Called hit-under-multiple-miss OR miss-under-miss optimization)
    - Only useful if memory can service multiple requests in parallel

- **These caches are called non-blocking (or lockup-free) caches**

- **Miss penalty with memory-level parallelism (MLP):**

**Memory stall cycles/Instruction**

    **= Cache Misses/instruction x (miss latency / average outstanding misses)**

# Memory-Level Parallelism Example

Program P has runs on a processor with a 4 GHz frequency. P has 10 billion instructions, and has a CPI of 0.5 with a perfect cache. The L1 cache is a non-blocking cache that can enable up to 16 outstanding misses at a time. The average memory access latency on a cache miss is 40 cycles. The L1 cache is a 32KB 8-way set-associative cache with a miss rate of 3%. Compare P's execution time when the average number of outstanding misses changes from 1 to 2.

- Cycle time = 1/frequency = 1/4,000,000,000 = 0.25 ns

- CPI = CPI(Perfect Cache) + miss rate x miss penalty

- For MLP = 1:

  CPI = 0.5 + 0.03 x 40 = 1.7

Execution Time = Instructions/Program x CPI x Cycle time = 10B x 1.7 x 0.25 ns = 4.25 seconds

- For MLP = 2:

  CPI = 0.5 + 0.03 x 40 / 2 = 1.1

Execution Time = Instructions/Program x CPI x Cycle time = 10B x 1.1 x 0.25 ns = 2.75 seconds (55% faster)

# Implementing Non-Blocking Caches

- Caches use Miss Status Holding (Handling) Registers to facilitate non-blocking memory level parallelism

- MSHRs are used to track address, data, and status for multiple outstanding cache misses

- Need to provide correct memory ordering, respond to CPU requests, and maintain cache coherence

- Design details (and names) vary widely between different processors but basic functions are similar

# Example MSHR Structure & Operation

- **Each MSHR contains the following information**
  - Data address of requested cache block
  - Block Valid Bit
  - PC of requesting instruction
  - For each word in cache line: Valid bit, destination (register where data will be stored), format bits (e.g., load width, int vs. fp, byte address bits, whether word is to be sign-extended)
  - Partial write codes: Indicates which bytes in a word has been written to the cache

- **On a cache miss, one MSHR is assigned**
  - Valid bit set
  - Data address saved
  - PC of requesting instruction saved
  - Appropriate word valid bits set and other cleared
  - Appropriate destination and format fields set for valid words
  - Partial write codes cleared
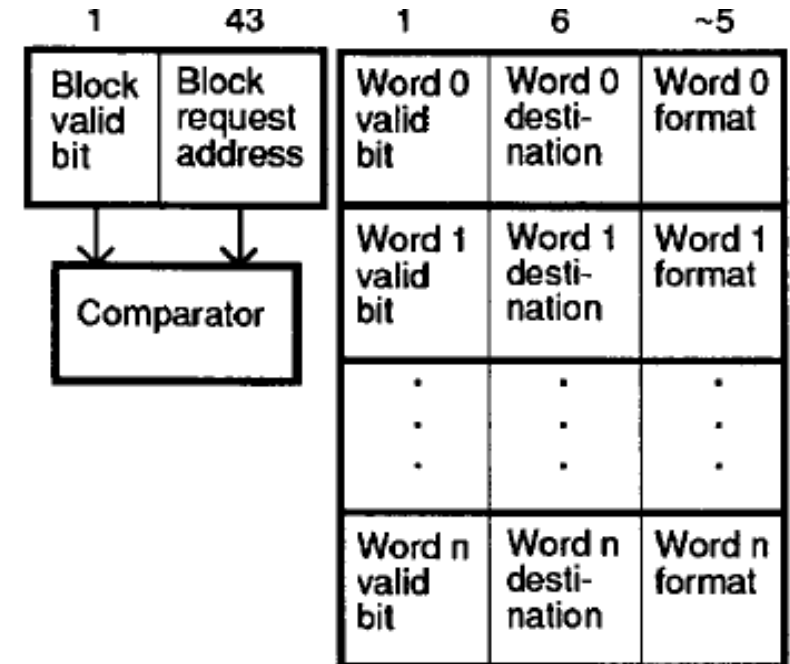


Farkas and Jouppi,
"Complexity/Performance Tradeoffs with Non-Blocking Loads", ISCA 1994

27

# Reducing Cache Misses

- **Cache misses are very costly**

- **Need multiple cache levels with**
  - ➢ High associativity or/and victim caches to reduce conflict misses
  - ➢ Effective replacement algorithms
  - ➢ Insertion policies
  - ➢ Data and instruction prefetching
  - ➢ Dead block prediction

- **Several mechanisms discussed next week**

# Reading Assignments

- ARCH Chapter 2.1, 2.2, 2.3, 2.4  (Read)

- ARCH Appendix B (Skim, Covered in 295/Quiz 1)