The Memory Hierarchy & Cache

- **Review of Memory Hierarchy & Cache Basics (from 550):**
  - **Motivation for The Memory Hierarchy:**
    - CPU/Memory Performance Gap
    - The Principle Of Locality → Cache $$$$$
  - **Cache Basics:**
    - Block placement strategy & Cache Organization:
    - Block replacement policy
    - Unified vs. Separate Level 1 Cache
  - **CPU Performance Evaluation with Cache:**
    - Average Memory Access Time (AMAT)/Memory Stall cycles
    - Memory Access Tree

Review From 550

Cache exploits memory access locality to:

- Lower AMAT by hiding long main memory access latency. Thus cache is considered a memory latency-hiding technique.
- Lower demands on main memory bandwidth.

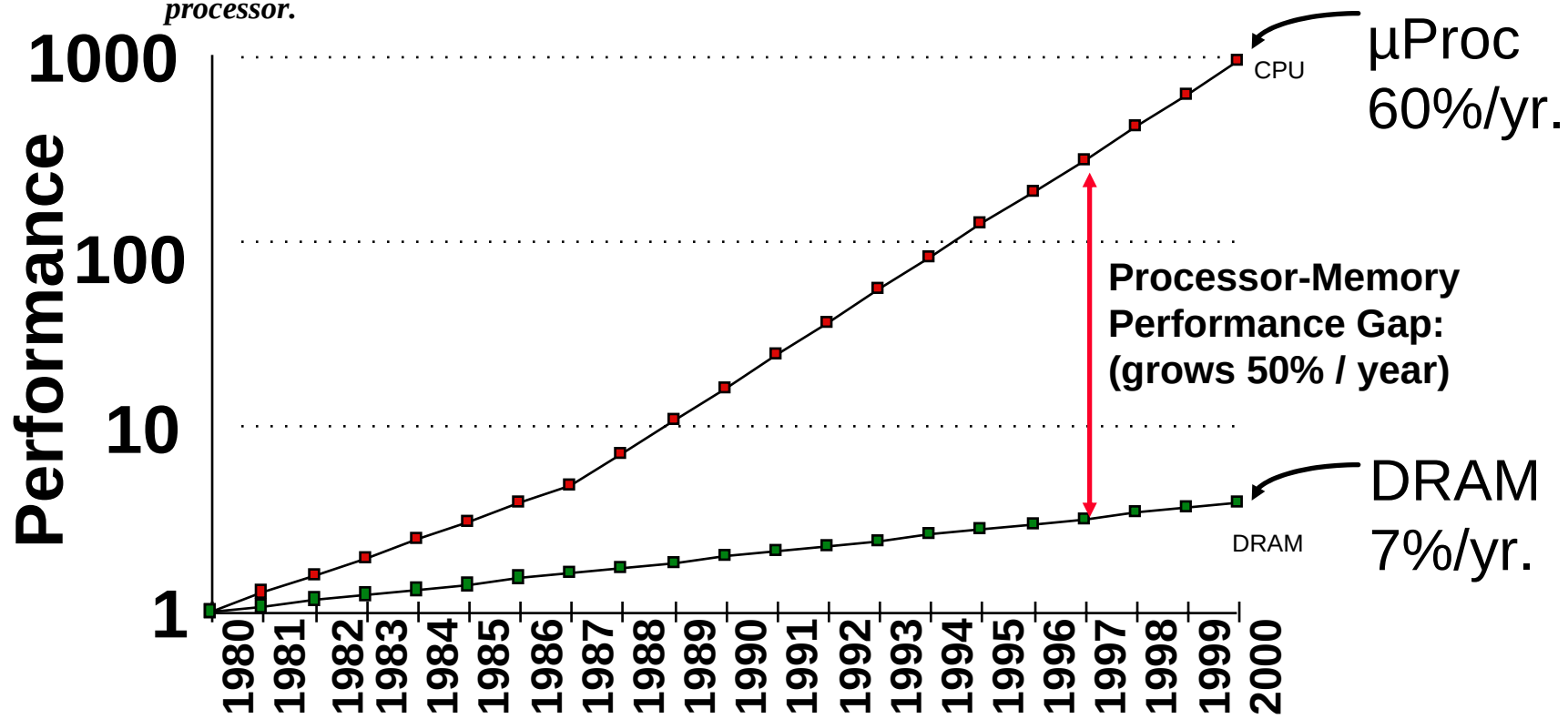
- **Classification of Steady-State Cache Misses:** *The Three C's of cache Misses*
- **Cache Write Policies/Performance Evaluation:**
  - Write Though
  - Write Back
- **Cache Write Miss Policies:** *Cache block allocation policy on a write miss.*
- **Multi-Level Caches:**
  - Miss Rates For Multi-Level Caches
  - 2-Level Cache Performance
  - Write Policy For 2-Level Cache
  - 3-Level Cache Performance

4th Edition: Chapter 5.1, Appendix C.1-C.3 (3rd Edition Chapter 5.1-5.4)

Memory Hierarchy: Motivation
Processor-Memory (DRAM) Performance Gap

i.e. Gap between memory access time (latency) and CPU cycle time

Memory Access Latency: ***The time between a memory access request is issued by the processor and the time the requested information (instructions or data) is available to the processor.***



Ideal Memory Access Time (latency) = 1 CPU Cycle
Real Memory Access Time (latency) >> 1 CPU cycle

(Review from 550)

Processor-DRAM Performance Gap Impact

- To illustrate the performance impact, assume a single-issue pipelined CPU with CPI = 1 using non-ideal memory.
- Ignoring other factors, the minimum cost of a full memory access in terms of number of wasted CPU cycles:

e.g cycles added to CPI

Year	CPU speed MHZ	CPU cycle ns	Memory Access ns	Minimum CPU memory stall cycles or instructions wasted
1986:	8	125	190	190/125 - 1 = 0.5
1989:	33	30	165	165/30 -1 = 4.5
1992:	60	16.6	120	120/16.6 -1 = 6.2
1996:	200	5	110	110/5 -1 = 21
1998:	300	3.33	100	100/3.33 -1 = 29
2000:	1000	1	90	90/1 - 1 = 89
2002:	2000	.5	80	80/.5 - 1 = 159
2004:	3000	.333	60	60.333 - 1 = 179

Or more

Ideal Memory Access Time (latency) = 1 CPU Cycle
Real Memory Access Time (latency) >> 1 CPU cycle

(Review from 550)

Addressing The CPU/Memory Performance Gap:

Memory Access Latency Reduction & Hiding Techniques

Memory Latency Reduction Techniques:

Reduce it!

- **Faster Dynamic RAM (DRAM) Cells:** Depends on VLSI processing technology.
- **Wider Memory Bus Width:** Fewer memory bus accesses needed (e.g 128 vs. 64 bits)
- **Multiple Memory Banks:**
  - At DRAM chip level (SDR, DDR SDRAM), module or channel levels.
- **Integration of Memory Controller with Processor:** e.g AMD's processor architecture + Intel's i7
- **New Emerging Faster RAM Technologies:** e.g. Magnetoresistive Random Access Memory (MRAM)

Memory Latency Hiding Techniques:

Hide it!

- **Memory Hierarchy**: One or more levels of smaller and faster memory (SRAM-based **cache**) on- or off-chip that exploit **program access locality** to hide long main memory latency.
- **Pre-Fetching:** Request instructions and/or data from memory before actually needed to hide long memory access latency.

Get it from main memory into cache before you need it !

What about dynamic scheduling?

Memory Hierarchy: Motivation

- The gap between CPU performance and main memory has been widening with higher performance CPUs creating performance bottlenecks for memory access instructions.

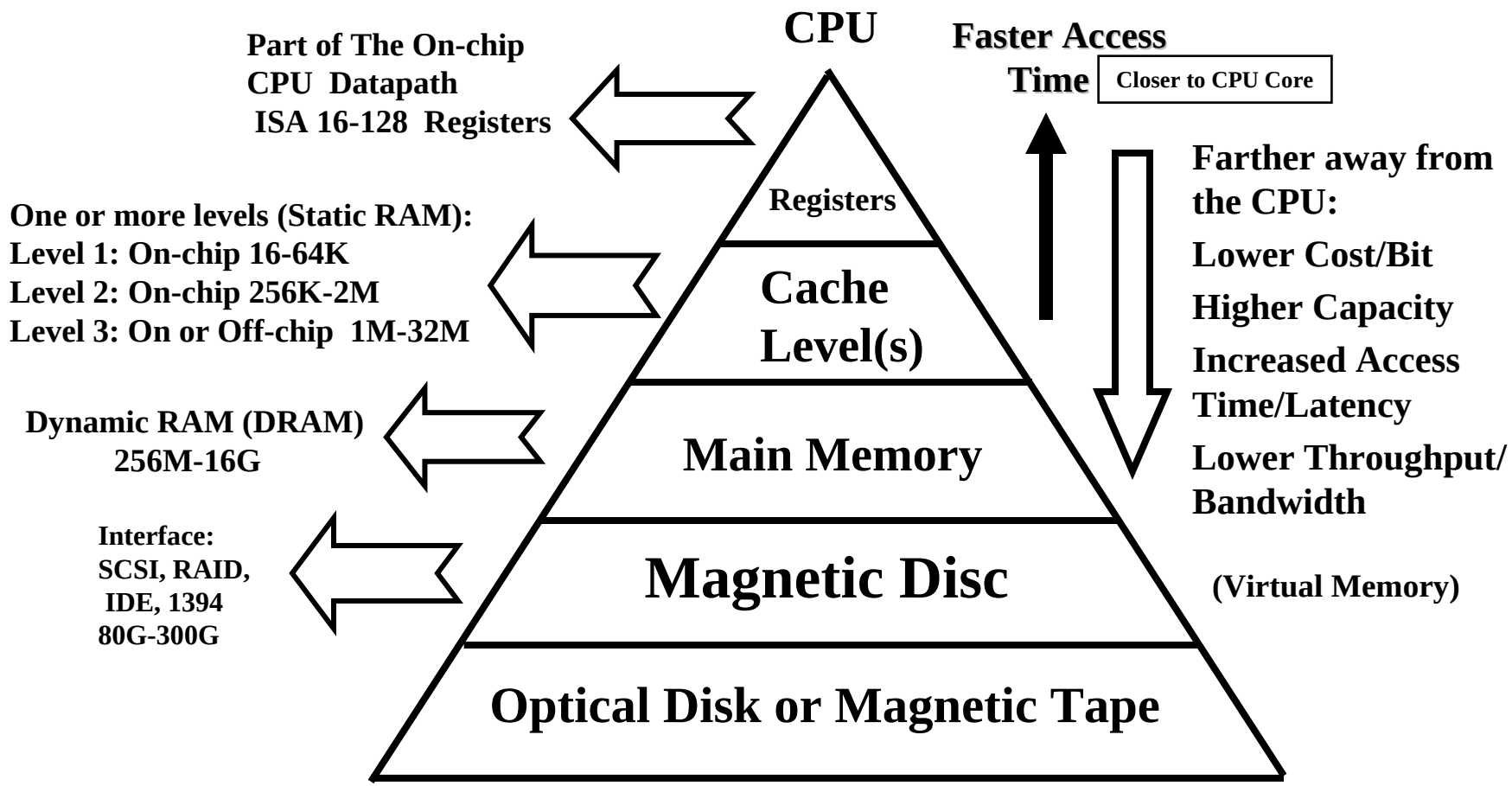
For Ideal Memory: Memory Access Time or latency = 1 CPU cycle

- To hide long memory access latency, the memory hierarchy is organized into several levels of memory with the smaller, faster SRAM-based memory levels closer to the CPU: registers, then primary Cache Level (L_1), then additional secondary cache levels (L_2, L_3…), then DRAM-based main memory, then mass storage (virtual memory).
- Each level of the hierarchy is usually a subset of the level below: data found in a level is also found in the level below (farther from CPU) but at lower speed (longer access time).
- Each level maps addresses from a larger physical memory to a smaller level of physical memory closer to the CPU.
- This concept is greatly aided by the principal of locality both temporal and spatial which indicates that programs tend to reuse data and instructions that they have used recently or those stored in their vicinity leading to working set of a program.

(Review from 550)

Both Editions: Chapter 5.1

Levels of The Memory Hierarchy



(Review from 550)

Memory Hierarchy: Motivation

The Principle Of Locality

- **Programs usually access a relatively small portion of their address space (instructions/data) at any instant of time (program working set).**

Thus: Memory Access Locality → Program Working Set

Often used data + instructions

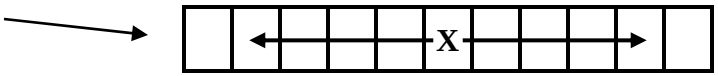
- **Two Types of access locality:**

1 – ***Temporal Locality:*** **If an item (instruction or data) is referenced, it will tend to be referenced again soon.**

  - **e.g. instructions in the body of inner loops**

2 – ***Spatial locality:*** **If an item is referenced, items whose addresses are close will tend to be referenced soon.**

  - **e.g. sequential instruction execution, sequential access to elements of array**



- **The presence of locality in program behavior (memory access patterns), makes it possible to satisfy a large percentage of program memory access needs (both instructions and data) using faster memory levels (cache) with much less capacity than program address space.**

Cache utilizes faster memory (SRAM)

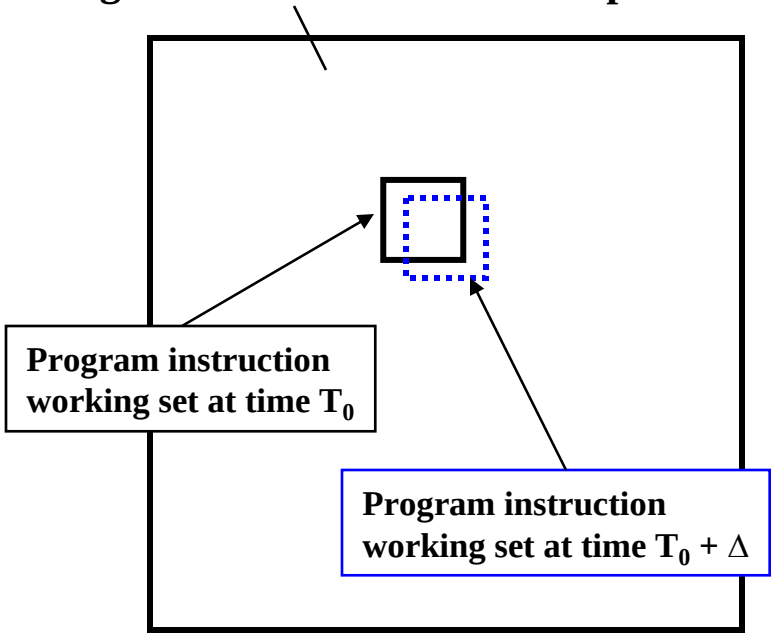
(Review from 550)

Access Locality & Program Working Set

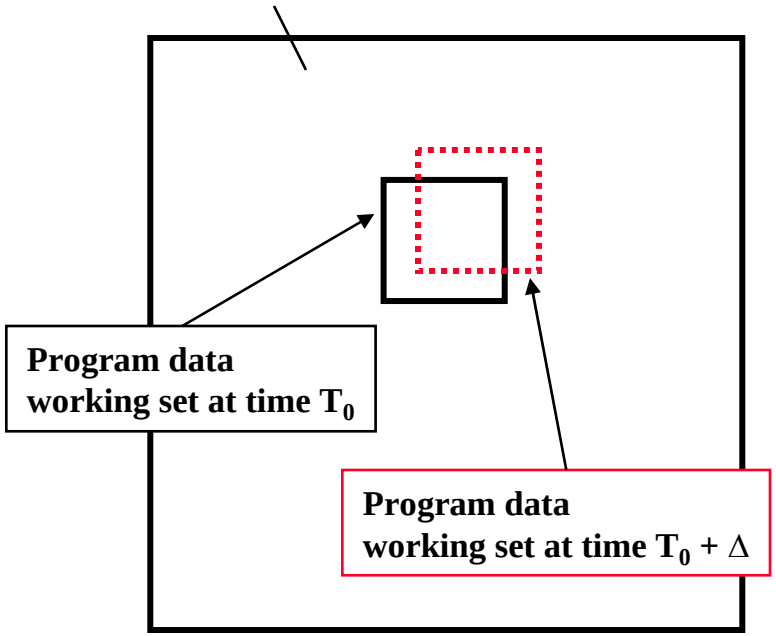
- **Programs usually access a relatively small portion of their address space (instructions/data) at any instant of time (program working set).**
- **The presence of locality in program behavior and memory access patterns, makes it possible to satisfy a large percentage of program memory access needs using faster memory levels with *much less capacity* than program address space.**
  (i.e Cache)

Using Static RAM (SRAM)

Program Instruction Address Space



Program Data Address Space



Locality in program memory access → Program Working Set

Memory Hierarchy Operation

- If an instruction or operand is required by the CPU, the levels of the memory hierarchy are searched for the item starting with the level closest to the CPU (Level 1 cache): L_1 Cache

Cache Hit

  - If the item is found, it's delivered to the CPU resulting in a cache hit without searching lower levels. Hit rate for level one cache = H_1

Cache Miss

  - If the item is missing from an upper level, resulting in a cache miss, the level just below is searched. Miss rate for level one cache = 1 – Hit rate = 1 - H_1
  - For systems with several levels of cache, the search continues with cache level 2, 3 etc.
  - If all levels of cache report a miss then main memory is accessed for the item.
    - CPU ↔ cache ↔ memory: Managed by hardware.
  - If the item is not found in main memory resulting in a page fault, then disk (virtual memory), is accessed for the item.
    - Memory ↔ disk: Managed by the operating system with hardware support

(Review from 550)

Memory Hierarchy: Terminology

- **A Block:** The smallest unit of information transferred between two levels.
- **Hit:** Item is found in some block in the upper level (example: Block X)
  - **Hit Rate:** The fraction of memory access found in the upper level.
  - **Hit Time:** Time to access the upper level which consists of (S)RAM access time + Time to determine hit/miss
- **Miss:** Item needs to be retrieved from a block in the lower level (Block Y)
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the missed block to the processor
- **Hit Time << Miss Penalty** M

e. g. H1

Hit rate for level one cache = H_1

Ideally = 1 Cycle

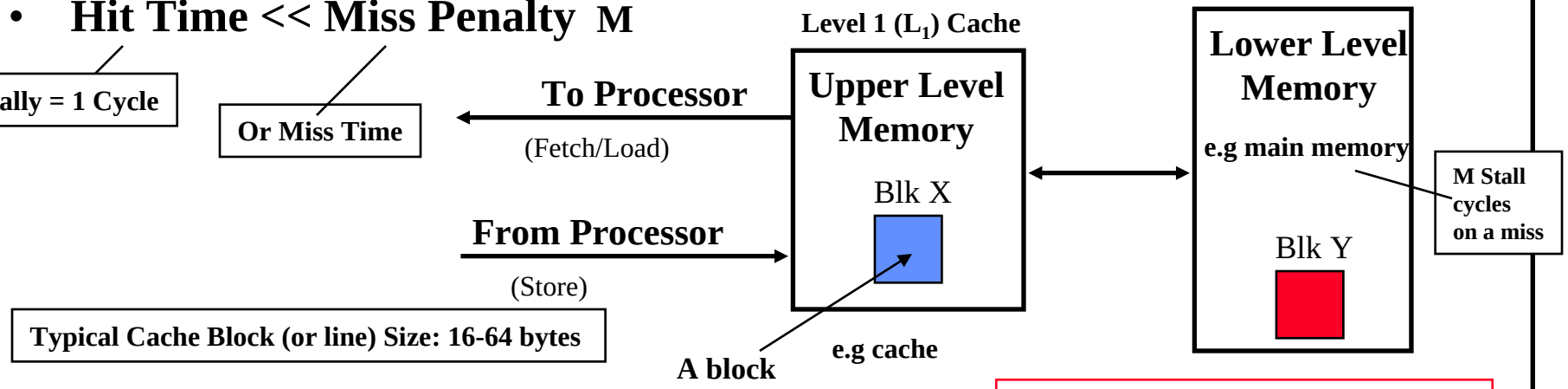
e. g. 1- H1

Miss rate for level one cache = 1 – Hit rate = 1 - H_1

M

Ideally = 1 Cycle

Or Miss Time



Typical Cache Block (or line) Size: 16-64 bytes

Hit if block is found in cache

(Review from 550)

Basic Cache Concepts

- **Cache is the first level of the memory hierarchy once the address leaves the CPU and is searched first for the requested data.**
- **If the data requested by the CPU is present in the cache, it is retrieved from cache and the data access is a cache hit otherwise a cache miss and data must be read from main memory.**
- **On a cache miss a block of data must be brought in from main memory to cache to possibly *replace* an existing cache block.**
- **The allowed block addresses where blocks can be mapped (placed) into cache from main memory is determined by *cache placement strategy*.**
- **Locating a block of data in cache is handled by cache block identification mechanism: Tag matching.**
- **On a cache miss choosing the cache block being removed (replaced) is handled by the *block replacement strategy* in place.**
- **When a write to cache is requested, a number of main memory update strategies exist as part of *the cache write policy*.**

(Review from 550)

Basic Cache Design & Operation Issues

- **Q1: Where can a block be placed cache?** **Block placement**
  (Block placement strategy & Cache organization)
  - **Fully Associative, Set Associative, Direct Mapped.**
    Very complex *Most common* *Simple but suffers from conflict misses*
- **Q2: How is a block found if it is in cache?** **Locating a block**
  (Block identification) **Cache Hit/Miss?**
  - **Tag/Block.** **Tag Matching**
- **Q3: Which block should be replaced on a miss?**
  (Block replacement) **Block replacement**
  - **Random, LRU, FIFO.**
- **Q4: What happens on a write?**
  (Cache write policy)
  - **Write through, write back.**

Not covered in 550 will be covered here

+ Cache block write allocation policy

4th Edition: Appendix C.1 (3rd Edition Chapter 5.2)

(Review from 550)

Cache Organization & Placement Strategies

Placement strategies or mapping of a main memory data block onto cache block frames divide cache designs into three organizations:

1 **Direct mapped cache:** **A block can be placed in only one location (cache block frame), given by the mapping function:**

Least complex to implement
suffers from conflict misses

Mapping Function

index = (Block address) MOD (Number of blocks in cache)

2 **Fully associative cache:** **A block can be placed anywhere in cache. (no mapping function).**

Most complex cache organization to implement

3 **Set associative cache:** **A block can be placed in a restricted set of places, or cache block frames. A set is a group of block frames in the cache. A block is first mapped onto the set and then it can be placed anywhere within the set. The set in this case is chosen by:**

Mapping Function

index = (Block address) MOD (Number of sets in cache)

If there are *n* blocks in a set the cache placement is called *n*-way set-associative.

Most common cache organization

(Review from 550)

Cache Organization:

Direct Mapped Cache

A block can be placed in one location only, given by:

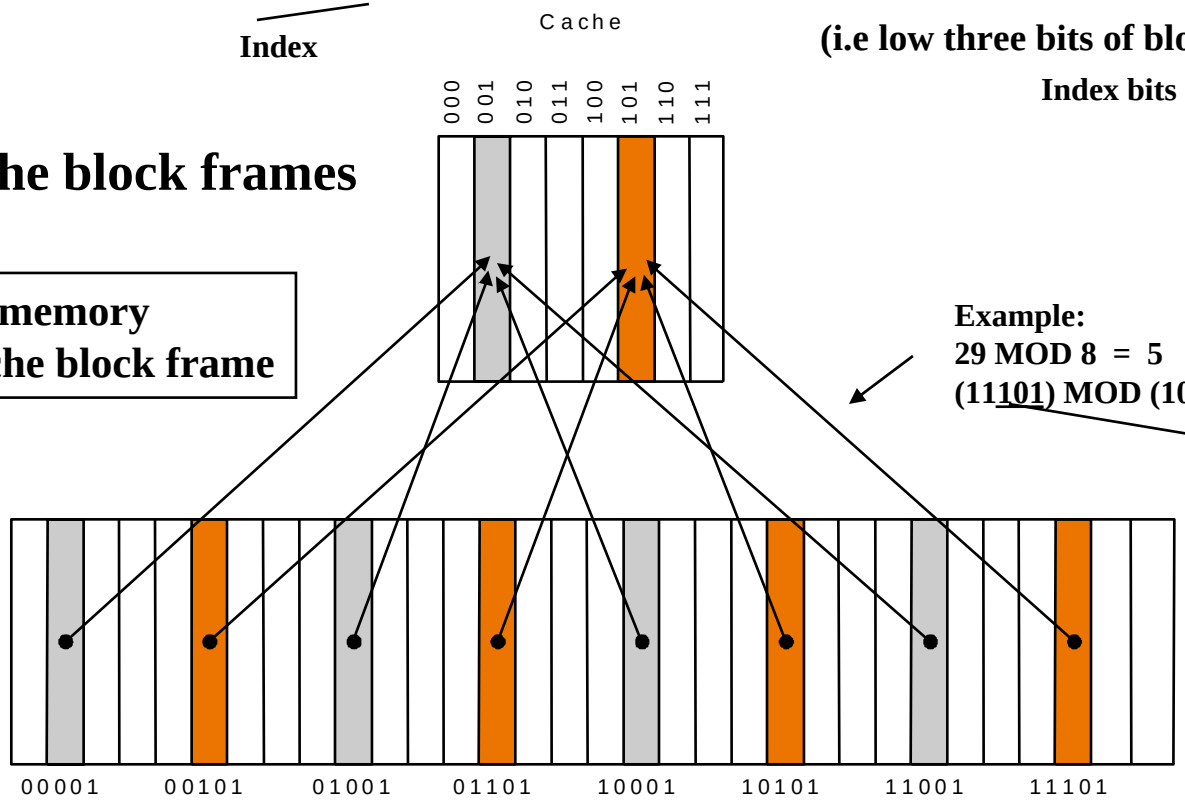
(Block address) MOD (Number of blocks in cache)

In this case, mapping function: (Block address) MOD (8) **= index**

Index

(i.e low three bits of block address)

Index bits



8 cache block frames

Here four blocks in memory map to the same cache block frame

32 memory blocks cacheable

Example:
29 MOD 8 = 5
(11101) MOD (1000) = 101

index

Index size = Log_2 8 = 3 bits

<u>Limitation of Direct Mapped Cache:</u> Conflicts between memory blocks that map to the same cache block frame may result in conflict cache misses

(Review from 550)

4KB Direct Mapped Cache Example

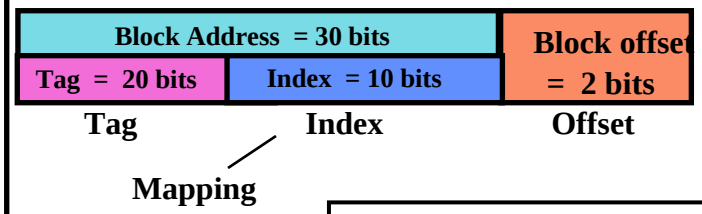
1K = 2^{10} = 1024 Blocks
Each block = one word
(4 bytes)

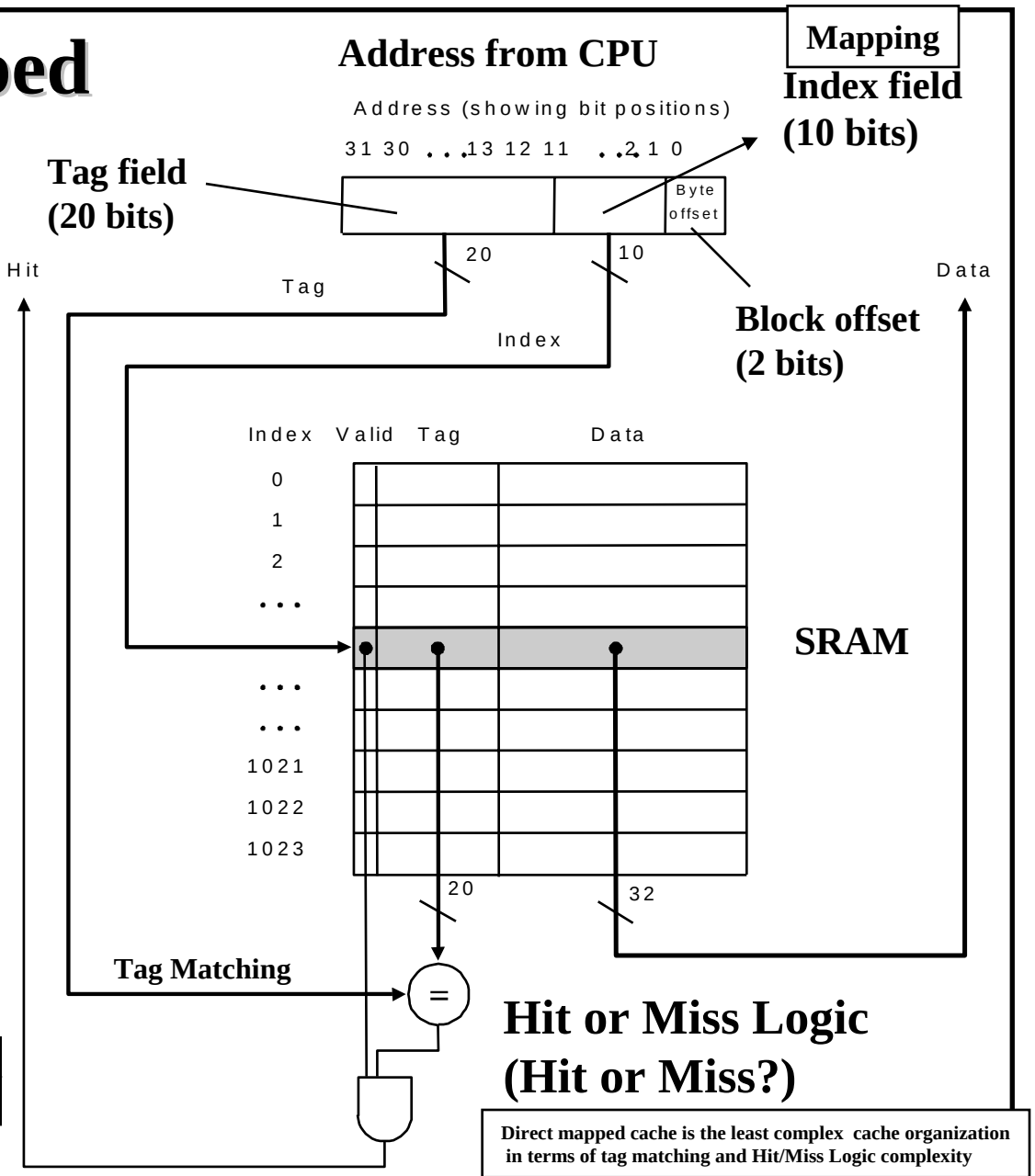
Can cache up to
2^{32} bytes = 4 GB
of memory

Mapping function:

Cache Block frame number =
(Block address) MOD (1024)

i.e . Index field or 10 low bits of block address



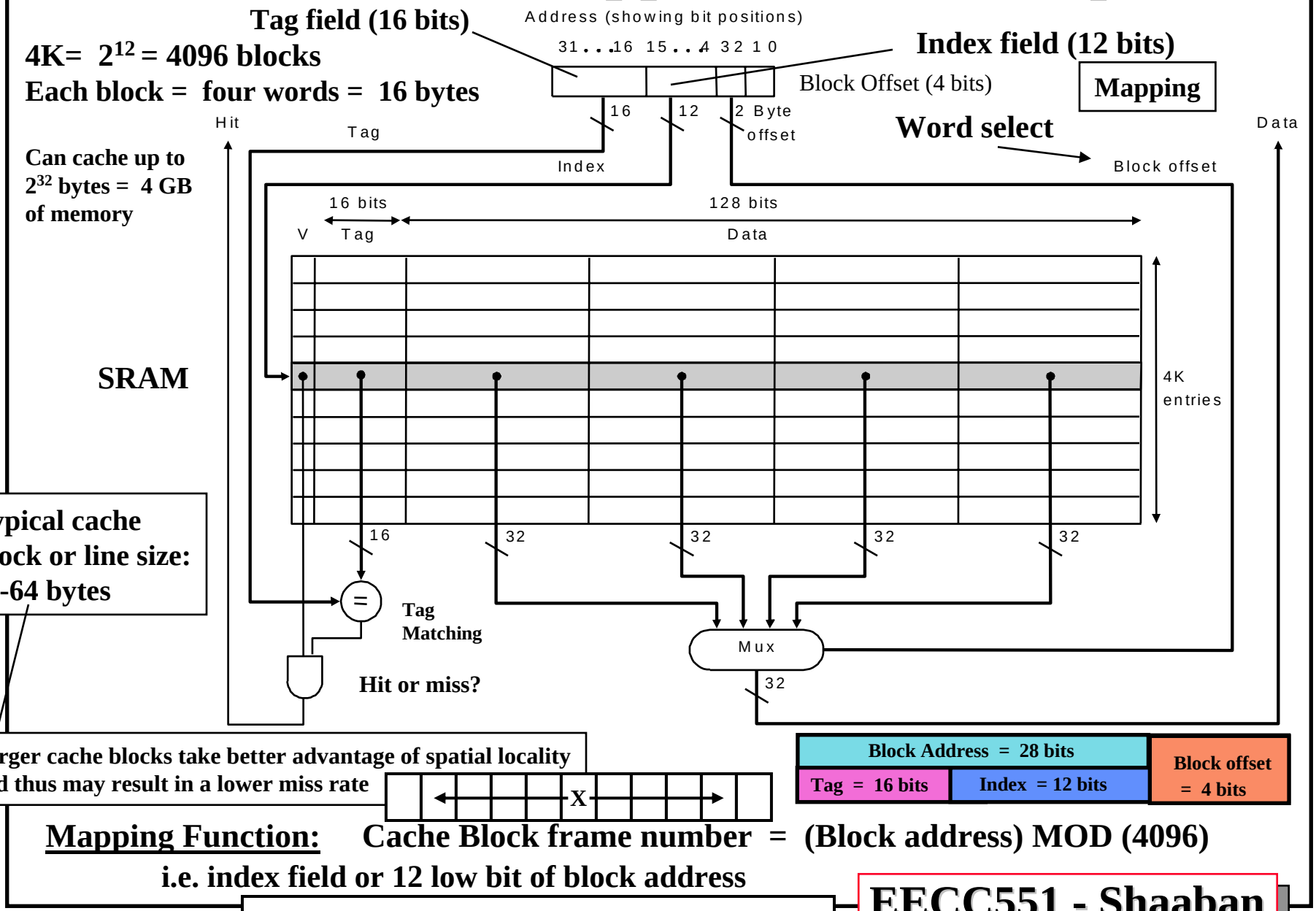


Direct mapped cache is the least complex cache organization in terms of tag matching and Hit/Miss Logic complexity

Hit Access Time = SRAM Delay + Hit/Miss Logic Delay

(Review from 550)

64KB Direct Mapped Cache Example



Mapping Function: Cache Block frame number = (Block address) MOD (4096)

i.e. index field or 12 low bit of block address

Hit Access Time = SRAM Delay + Hit/Miss Logic Delay

(Review from 550)

Cache Organization: Set Associative Cache

V	Tag	Data

Cache Block Frame

Why set associative?

Set associative cache reduces cache misses by <u>reducing conflicts</u> between blocks that would have been mapped to the same cache block frame in the case of direct mapped cache

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

1-way set associative: (direct mapped)
1 block frame per set

Here 8 sets

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

2-way set associative:
2 blocks frames per set

Here 4 sets

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

4-way set associative:
4 blocks frames per set

Here 2 sets

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

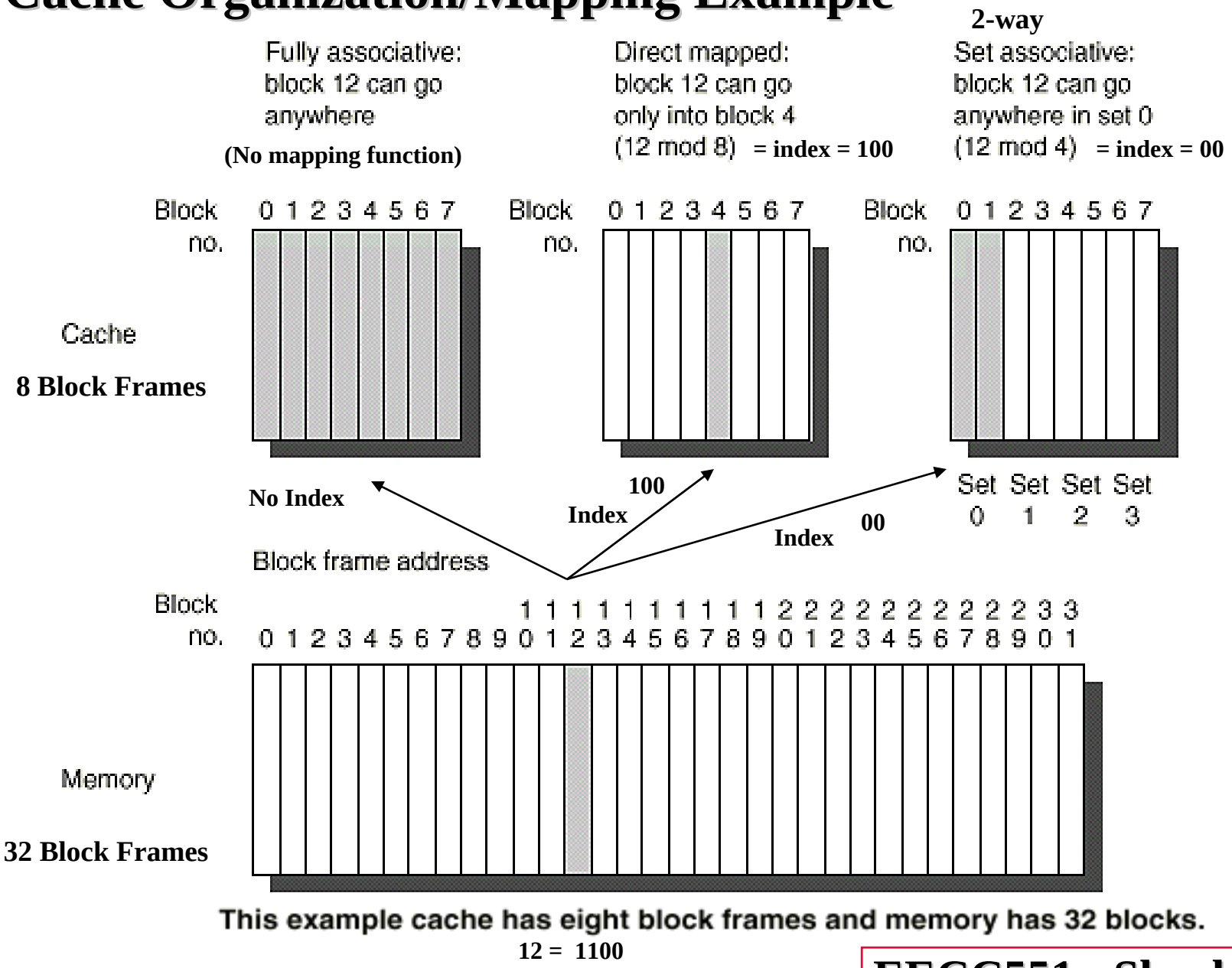
8-way set associative:
8 blocks frames per set
In this case it becomes fully associative since total number of block frames = 8

One set (no mapping)

A cache with a total of 8 cache block frames shown above

(Review from 550)

Cache Organization/Mapping Example

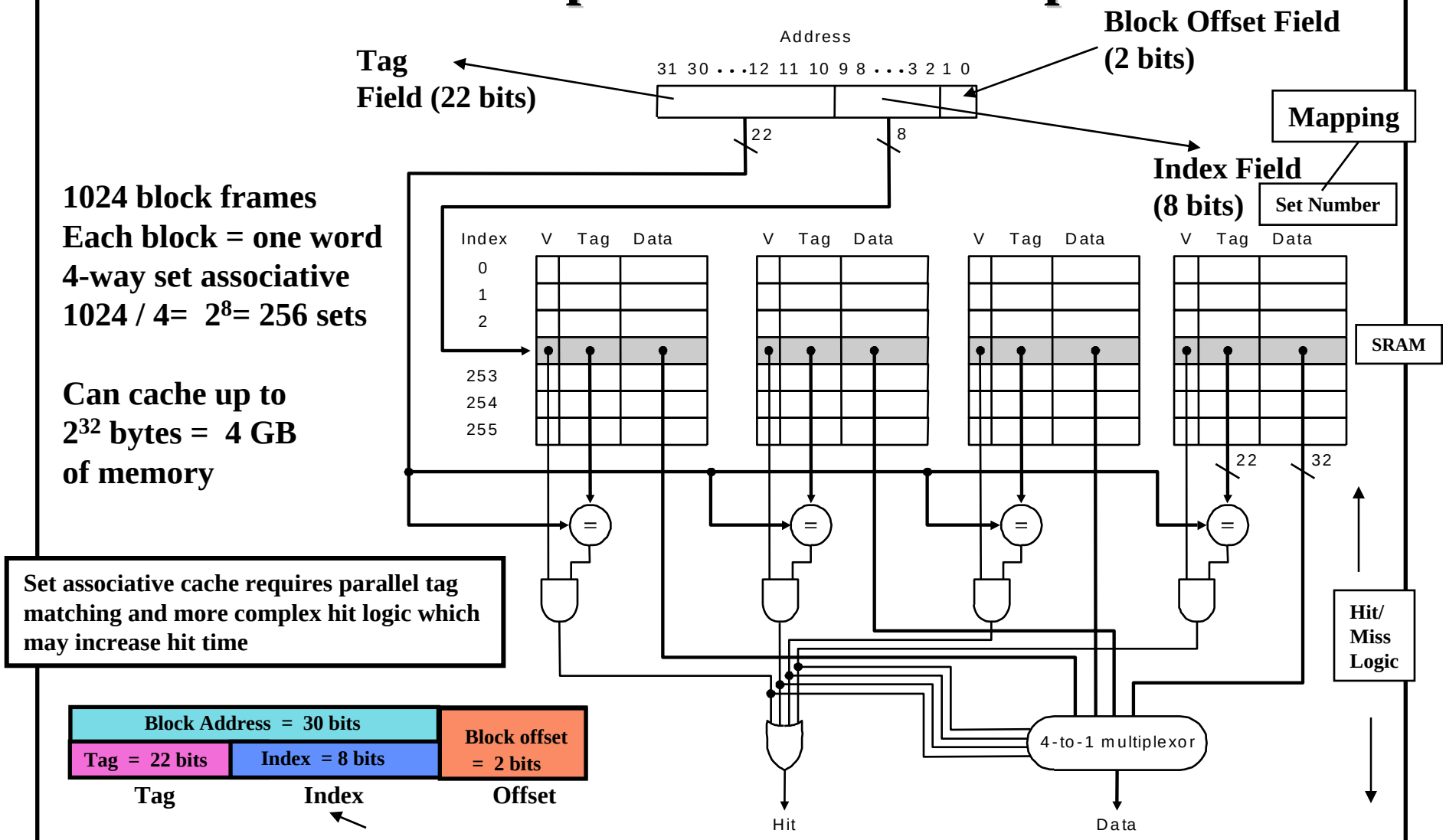


This example cache has eight block frames and memory has 32 blocks.

12 = 1100

(Review from 550)

4K Four-Way Set Associative Cache: MIPS Implementation Example



1024 block frames
Each block = one word
4-way set associative
1024 / 4= 2^8= 256 sets

Can cache up to
2^{32} bytes = 4 GB
of memory

Set associative cache requires parallel tag matching and more complex hit logic which may increase hit time

Mapping Function: Cache Set Number = index= (Block address) MOD (256)

Hit Access Time = SRAM Delay + Hit/Miss Logic Delay

(Review from 550)

Locating A Data Block in Cache

- Each block frame in cache has an address tag.
- The tags of every cache block that might contain the required data are checked in parallel.
- A valid bit is added to the tag to indicate whether this entry contains a valid address.
- The address from the CPU to cache is divided into:
  - A block address, further divided into:
    - An index field to choose a block frame/set in cache.
      (no index field when fully associative).
    - A tag field to search and match addresses in the selected set.
  - A block offset to select the data from the block.

Address Field Sizes/Mapping

Physical Memory Address Generated by CPU

(size determined by amount of physical main memory cacheable)

Mapping

Block offset size = $\log_2$(block size)

Index size = $\log_2$(Total number of blocks/associativity)

Tag size = address size - index size - offset size

Number of Sets in cache

Mapping function:

Cache set or block frame number = Index =

= (Block Address) MOD (Number of Sets)

No index/mapping function for fully associative cache

(Review from 550)

Cache Replacement Policy

Which block to replace on a cache miss?

- When a cache miss occurs the cache controller may have to select a block of cache data to be removed from a cache block frame and replaced with the requested data, such a block is selected by one of three methods:

(No cache replacement policy in direct mapped cache)

No choice on which block to replace

1 – **Random:**

- Any block is randomly selected for replacement providing uniform allocation.
- Simple to build in hardware. Most widely used cache replacement strategy.

2 – **Least-recently used (LRU):**

- Accesses to blocks are recorded and and the block replaced is the one that was not used for the longest period of time.
- Full LRU is *expensive* to implement, as the number of blocks to be tracked increases, and is usually approximated by block usage bits that are cleared at regular time intervals.

3 – **First In, First Out (FIFO:**

- Because LRU can be complicated to implement, this approximates LRU by determining the oldest block rather than LRU

Miss Rates for Caches with Different Size, Associativity & Replacement Algorithm

Sample Data

Associativity:	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Program steady state cache miss rates are given
Initially cache is empty and miss rates ~ 100%

FIFO replacement miss rates (not shown here) is better than random but worse than LRU

For SPEC92

(Review from 550)

Unified vs. Separate Level 1 Cache

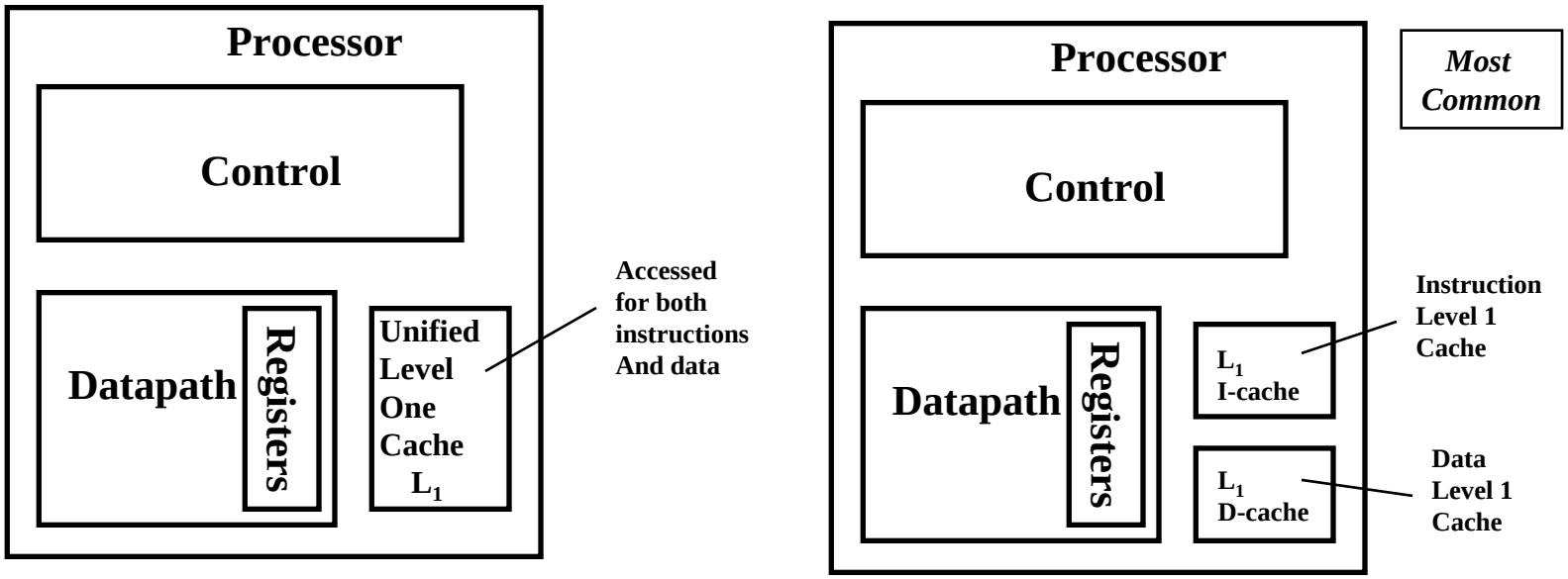
- **Unified Level 1 Cache (Princeton Memory Architecture).** AKA Shared Cache

  A single level 1 (L_1) cache is used for both instructions and data.

Or Split

- **Separate instruction/data Level 1 caches (Harvard Memory Architecture):**

  The level 1 (L_1) cache is split into two caches, one for instructions (instruction cache, L_1 I-cache) and the other for data (data cache, L_1 D-cache).



AKA shared

Unified Level 1 Cache (Princeton Memory Architecture)

Separate (Split) Level 1 Caches (Harvard Memory Architecture)

Why?

Split Level 1 Cache is more preferred in pipelined CPUs to avoid instruction fetch/Data access structural hazards

(Review from 550)

Memory Hierarchy Performance:

Average Memory Access Time (AMAT), Memory Stall cycles

- **The Average Memory Access Time (AMAT):** The number of cycles required to complete an average memory access request by the CPU.
- **Average memory stall cycles per memory access:** The number of stall cycles added to CPU execution cycles for one memory access.
- **Memory stall cycles per average memory access = (AMAT -1)**
- For ideal memory: AMAT = 1 cycle, this results in zero memory stall cycles.
- Memory stall cycles per average instruction =

Number of memory accesses per instruction

Instruction Fetch

x Memory stall cycles per average memory access

= (1 + fraction of loads/stores) x (AMAT -1)

Base CPI = $CPI_{execution}$ = CPI with ideal memory

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

(Review from 550)

cycles = CPU cycles

Cache Performance: Single Level L1 Princeton (Unified) Memory Architecture

(Ignoring Write Policy)

(Review from 550)

CPUtime = Instruction count x CPI x Clock cycle time

$CPI_{execution}$ = CPI with ideal memory

$$CPI = CPI_{execution} + \text{Mem Stall cycles per instruction}$$

Mem Stall cycles per instruction =

Memory accesses per instruction x Memory stall cycles per access

Assuming no stall cycles on a cache hit (cache access time = 1 cycle, stall = 0)

Cache Hit Rate = H1 Miss Rate = 1- H1

i.e No hit penalty

Memory stall cycles per memory access = Miss rate x Miss penalty = (1- H1) x M

AMAT = 1 + Miss rate x Miss penalty = 1 + (1- H1) x M

Memory accesses per instruction = (1 + fraction of loads/stores)

Miss Penalty = M = the number of stall cycles resulting from missing in cache

= Main memory access time - 1

Thus for a unified L1 cache with no stalls on a cache hit:

$$CPI = CPI_{execution} + (1 + \text{fraction of loads/stores}) \times (1 - H1) \times M$$

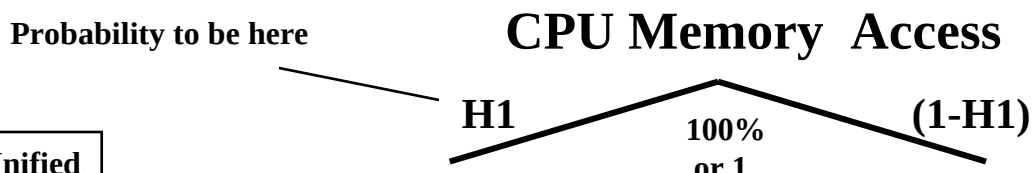
$$AMAT = 1 + (1 - H1) \times M$$

$CPI = CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access

$= CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

Memory Access Tree: For Unified Level 1 Cache

(Ignoring Write Policy)



Unified L_1

L1 Hit:
% = Hit Rate = H1
Hit Access Time = 1
Stall cycles per access = 0
Stall= H1 x 0 = 0
(No Stall)

L1 Miss:
% = (1- Hit rate) = (1-H1)
Access time = M + 1
Stall cycles per access = M
Stall = M x (1-H1)

Assuming:
Ideal access on a hit

Hit Rate — Hit Time — Miss Rate — Miss Time

$$AMAT = H1 \times 1 + (1 - H1) \times (M + 1) = 1 + M \times (1 - H1)$$

Stall Cycles Per Access = AMAT - 1 = M x (1 -H1)

$$CPI = CPI_{execution} + (1 + \text{fraction of loads/stores}) \times M \times (1 - H1)$$

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H1 = Level 1 Hit Rate 1- H1 = Level 1 Miss Rate

AMAT = 1 + Stalls per average memory access

(Review from 550)

Cache Performance Example

- Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.
- $CPI_{execution}$ = 1.1 (i.e base CPI with ideal memory)
- Instruction mix: 50% arith/logic, 30% load/store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of M= 50 cycles.

CPI = $CPI_{execution}$ + mem stalls per instruction

Mem Stalls per instruction = Mem accesses per instruction x Miss rate x Miss penalty

(Miss rate: (1- H1); Miss penalty: M)

Mem accesses per instruction = 1 + .3 = 1.3

(1: Instruction fetch; .3: Load/store)

Mem Stalls per memory access = (1- H1) x M = .015 x 50 = .75 cycles

AMAT = 1 +.75 = 1.75 cycles

Mem Stalls per instruction = 1.3 x .015 x 50 = 0.975

CPI = 1.1 + .975 = 2.075

The ideal memory CPU with no misses is 2.075/1.1 = 1.88 times faster

M = Miss Penalty = stall cycles per access resulting from missing in cache

Cache Performance Example

- **Suppose for the previous example we double the clock rate to 400 MHz, how much faster is this machine, assuming similar miss rate, instruction mix?**
- **Since memory speed is not changed, the miss penalty takes more CPU cycles:**

Miss penalty = M = 50 x 2 = 100 cycles.

CPI = 1.1 + 1.3 x .015 x 100 = 1.1 + 1.95 = 3.05

Speedup = $(CPI_{old} \times C_{old})/ (CPI_{new} \times C_{new})$

= 2.075 x 2 / 3.05 = 1.36

The new machine is only 1.36 times faster rather than 2 times faster due to the increased effect of cache misses.

→ ***CPUs with higher clock rate, have more cycles per cache miss and more memory impact on CPI.***

(Ignoring Write Policy)

Usually:
Data Miss Rate >> Instruction Miss Rate

Data Level 1 Cache → L_1 D-cache

L_1 I-cache ← Instruction Level 1 Cache

Miss rate = 1 – data H1

Miss rate = 1 – instruction H1

Cache Performance:
Single Level L1 Harvard (Split) Memory Architecture

For a CPU with separate or split level one (L1) caches for instructions and data (Harvard memory architecture) and no stalls for cache hits:

As assumption
In realty, even a hit may result in few stalls (2-3)

$$\text{CPUtime} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

$$\text{CPI} = \text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}$$

This is one method to find stalls per instruction
another method is shown in next slide →

$$\text{Mem Stall cycles per instruction} = \text{Instruction Fetch Miss rate} \times M + \text{Data Memory Accesses Per Instruction} \times \text{Data Miss Rate} \times M$$

1- Instruction H1

Fraction of Loads and Stores

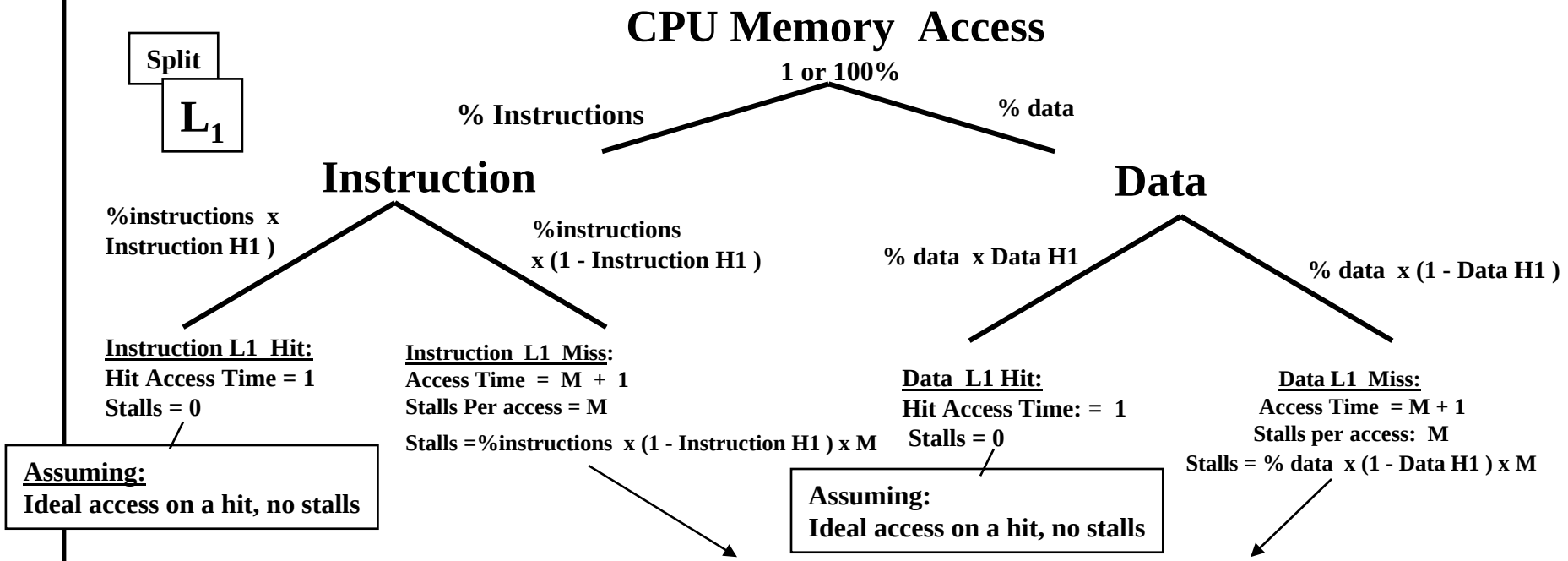
1- Data H1

M = Miss Penalty = stall cycles per access to main memory resulting from missing in cache

$\text{CPI}_{\text{execution}}$ = base CPI with ideal memory

Memory Access Tree For Separate Level 1 Caches

(Ignoring Write Policy)



Stall Cycles Per Access = % Instructions x (1 - Instruction H1) x M + % data x (1 - Data H1) x M

AMAT = 1 + Stall Cycles per access

Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access

CPI = $CPI_{execution}$ + Stall cycles per instruction
= $CPI_{execution}$ + (1 + fraction of loads/stores) x Stall Cycles per access

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
Data H1 = Level 1 Data Hit Rate 1- Data H1 = Level 1 Data Miss Rate
Instruction H1 = Level 1 Instruction Hit Rate 1- Instruction H1 = Level 1 Instruction Miss Rate
% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses

(Review from 550)

Split L1 Cache Performance Example

- **Suppose a CPU uses separate level one (L1) caches for instructions and data (Harvard memory architecture) with different miss rates for instruction and data access:**
  - **A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes.**
  - **$CPI_{execution}$ = 1.1** **(i.e base CPI with ideal memory)**
  - **Instruction mix: 50% arith/logic, 30% load/store, 20% control**
  - **Assume a cache miss rate of 0.5% for instruction fetch and a cache data miss rate of 6%.**
  - **A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes.**
- **Find the resulting stalls per access, AMAT and CPI using this cache?** **M** ***(Ignoring Write Policy)***

CPI = $CPI_{execution}$ + mem stalls per instruction

Memory Stall cycles per instruction = Instruction Fetch Miss rate x Miss Penalty + Data Memory Accesses Per Instruction x Data Miss Rate x Miss Penalty

Memory Stall cycles per instruction = 0.5/100 x 200 + 0.3 x 6/100 x 200 = 1 + 3.6 = 4.6 cycles
Stall cycles per average memory access = 4.6/1.3 = 3.54 cycles
AMAT = 1 + 3.54 = 4.54 cycles

CPI = $CPI_{execution}$ + mem stalls per instruction = 1.1 + 4.6 = 5.7 cycles

- **What is the miss rate of a single level unified cache that has the same performance?**

4.6 = 1.3 x Miss rate x 200 which gives a miss rate of 1.8 % for an equivalent unified cache

- **How much faster is the CPU with ideal memory?**

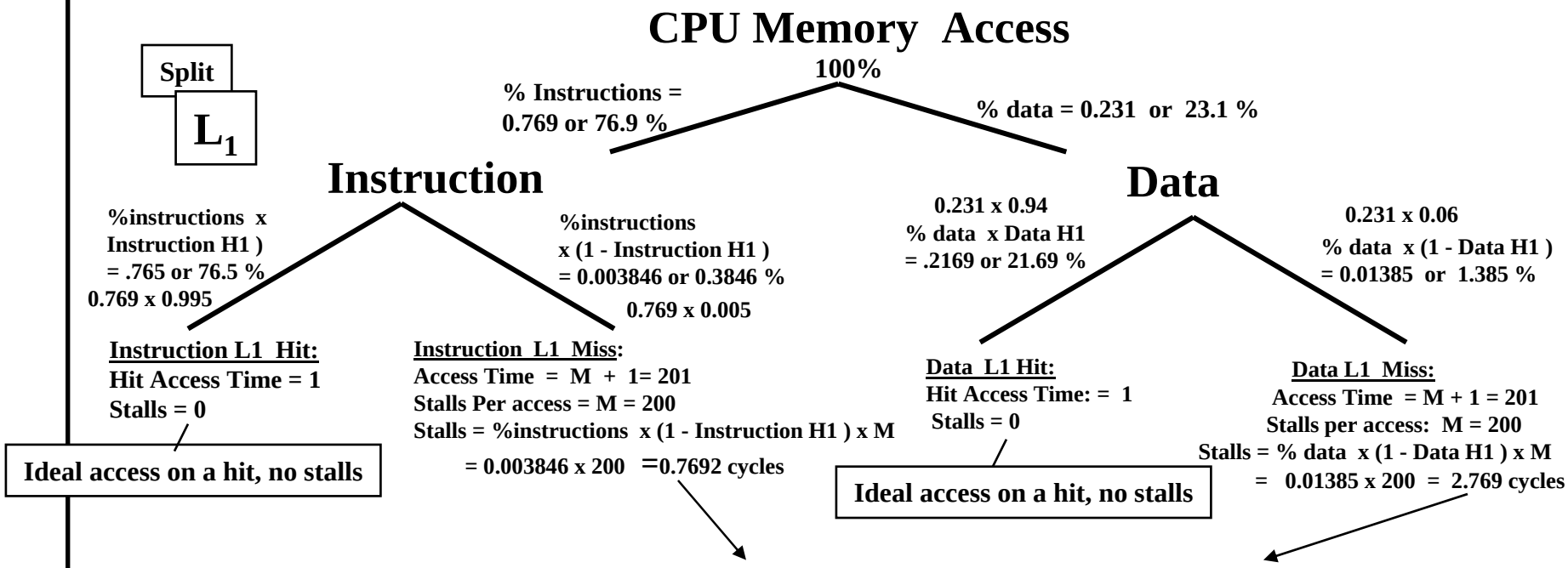
The CPU with ideal cache (no misses) is 5.7/1.1 = 5.18 times faster
With no cache at all the CPI would have been = 1.1 + 1.3 X 200 = 261.1 cycles !!

Memory Access Tree For Separate Level 1 Caches Example

(Ignoring Write Policy)

30% of all instructions executed are loads/stores, thus:
Fraction of instruction fetches out of all memory accesses = 1/ (1+0.3) = 1/1.3 = 0.769 or 76.9 %
Fraction of data accesses out of all memory accesses = 0.3/ (1+0.3) = 0.3/1.3 = 0.231 or 23.1 %

For Last Example



Stall Cycles Per Access = % Instructions x (1 - Instruction H1) x M + % data x (1 - Data H1) x M
= 0.7692 + 2.769 = 3.54 cycles

AMAT = 1 + Stall Cycles per access = 1 + 3.5 = 4.54 cycles

Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access = 1.3 x 3.54 = 4.6 cycles

CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 4.6 = 5.7

Given as 1.1

M = Miss Penalty = stall cycles per access resulting from missing in cache = 200 cycles
M + 1 = Miss Time = Main memory access time = 200+1 =201 cycles L1 access Time = 1 cycle
Data H1 = 0.94 or 94% 1- Data H1 = 0.06 or 6%
Instruction H1 = 0.995 or 99.5% 1- Instruction H1 = 0.005 or 0.5 %
% Instructions = Percentage or fraction of instruction fetches out of all memory accesses = 76.9 %
% Data = Percentage or fraction of data accesses out of all memory accesses = 23.1 %

(Review from 550)

Typical Cache Performance Data Using SPEC92

Usually Date Miss Rate >> Instruction Miss Rate

Size	Instruction cache	Data cache	Unified cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

Miss rates for instruction, data, and unified caches of different sizes.

Program steady state cache miss rates are given
Initially cache is empty and miss rates ~ 100%

Types of Cache Misses: *The Three C's*

(of Cache Misses)

1 **Compulsory:** On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.

- Initially upon program startup: Miss rate ~ 100% All compulsory misses

Can be reduced by increasing cache block size and pre-fetching

2 **Capacity:** Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity).

i.e. cache filled

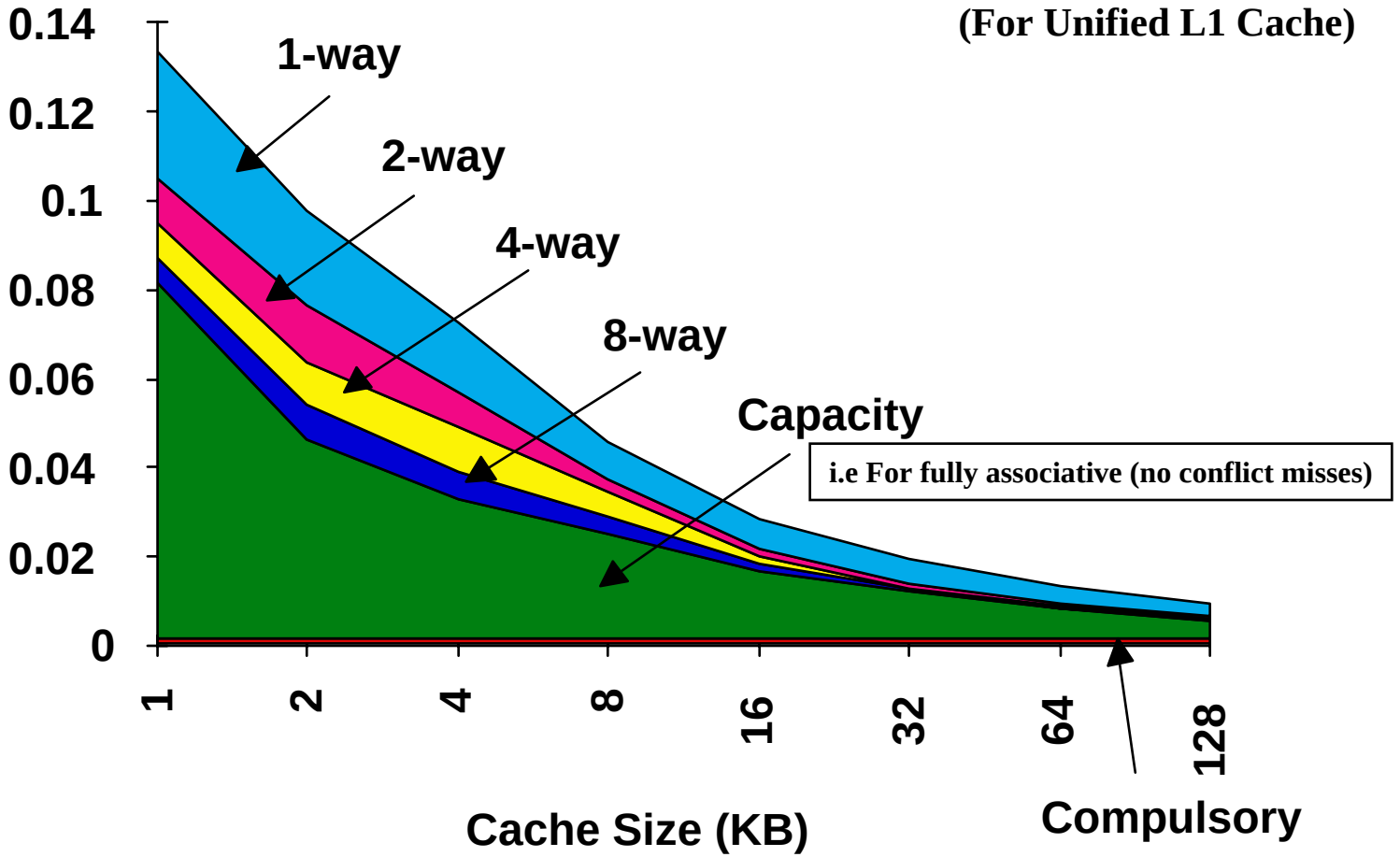
Can be reduced by increasing total cache size

3 **Conflict:** In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.

Can be reduced by increasing cache associativity

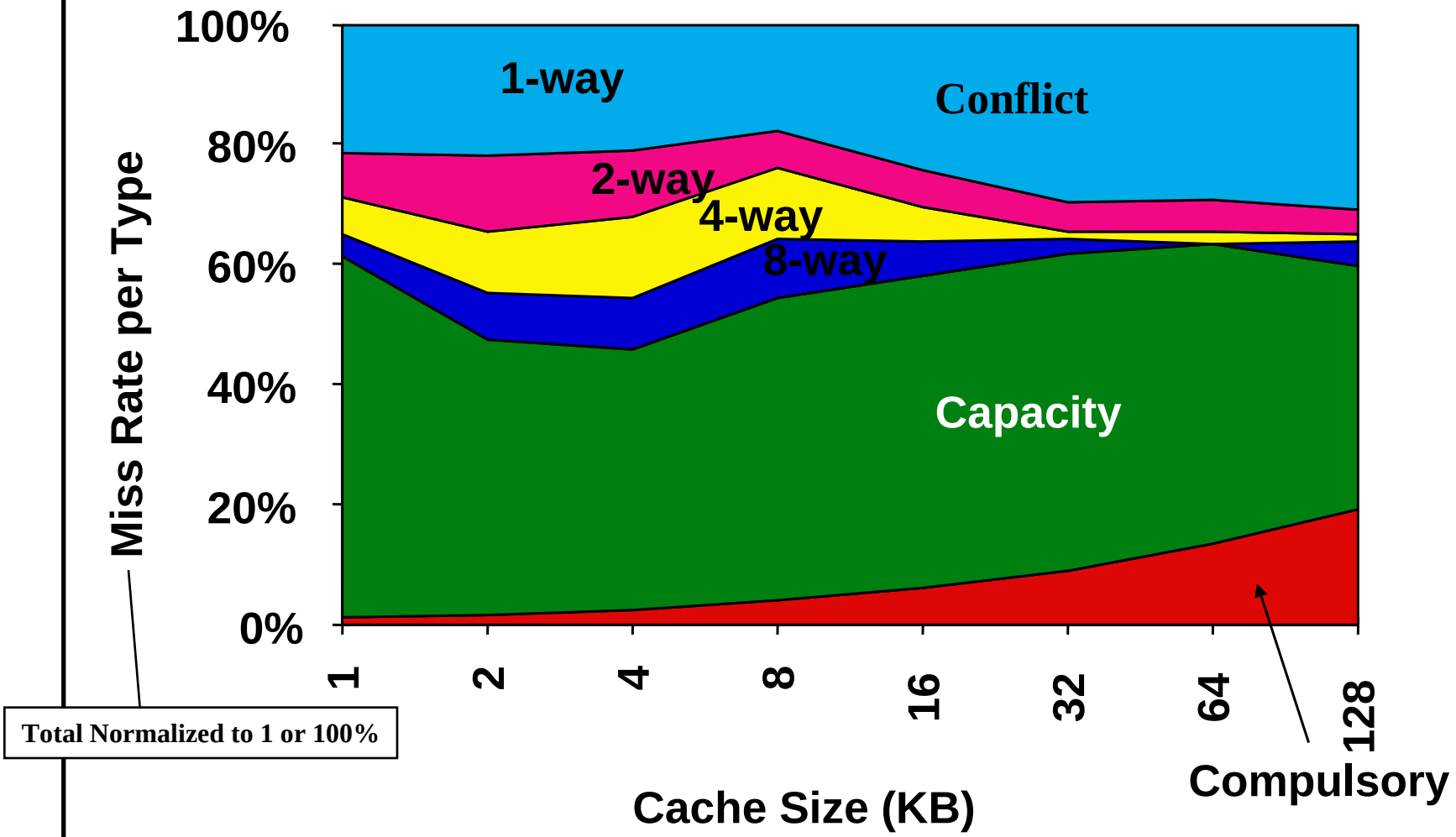
The 3 Cs of Cache:

Absolute Steady State Miss Rates (SPEC92)



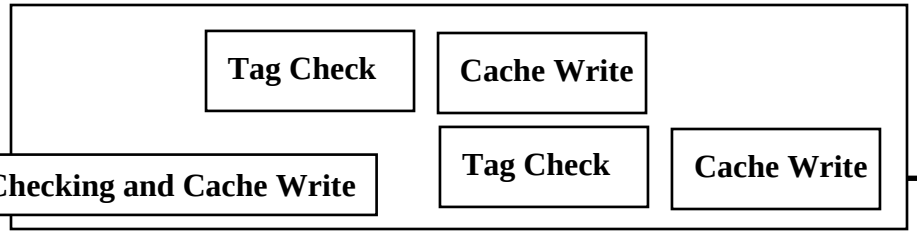
The 3 Cs of Cache:

Relative Steady State Miss Rates (SPEC92)



Cache Read/Write Operations

- **Statistical data suggest that reads (*including instruction fetches)* dominate processor cache accesses (writes account for ~ 25% of data cache traffic).**
  - i.e stores
- **In cache reads, a block is read at the same time while the tag is being compared with the block address. If the read is a hit the data is passed to the CPU, if a miss it ignores it.**
- **In cache writes, modifying the block cannot begin until the tag is checked to see if the address is a hit.**
  - i.e write hit (we have old block to modify in cache)
- **Thus for cache writes, tag checking cannot take place in parallel, and only the specific data (between 1 and 8 bytes) requested by the CPU can be modified.**
  - **Solution: Pipeline tag checking and cache write.**
- **Cache can be classified according to the write and memory update strategy in place as: write through, or write back cache.**



Pipelining of Tag Checking and Cache Write

Cache Write Strategies

1 **Write Though: Data is written to both the cache block and to a block of main memory.** (i.e written though to memory)

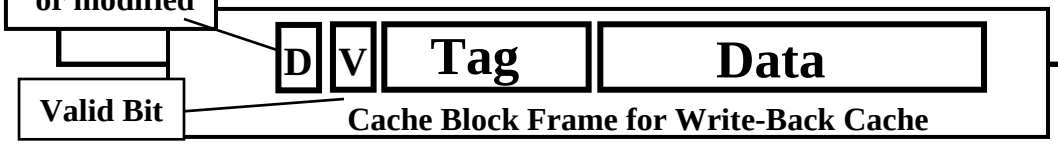
- The lower level always has the most updated data; an important feature for I/O and multiprocessing.
- Easier to implement than write back.
- A write buffer is often used to reduce CPU write stall while data is written to memory.

The updated cache block is marked as modified or dirty

2 **Write Back: Data is written or updated only to the cache block. The modified or dirty cache block is written to main memory when it's being replaced from cache.** back

- Writes occur at the speed of cache
- A status bit called a dirty or modified bit, is used to indicate whether the block was modified while in cache; if not the block is not written back to main memory when replaced. i.e discarded
- Advantage: Uses less memory bandwidth than write through.

D = Dirty Or Modified Status Bit
0 = clean
1 = dirty or modified

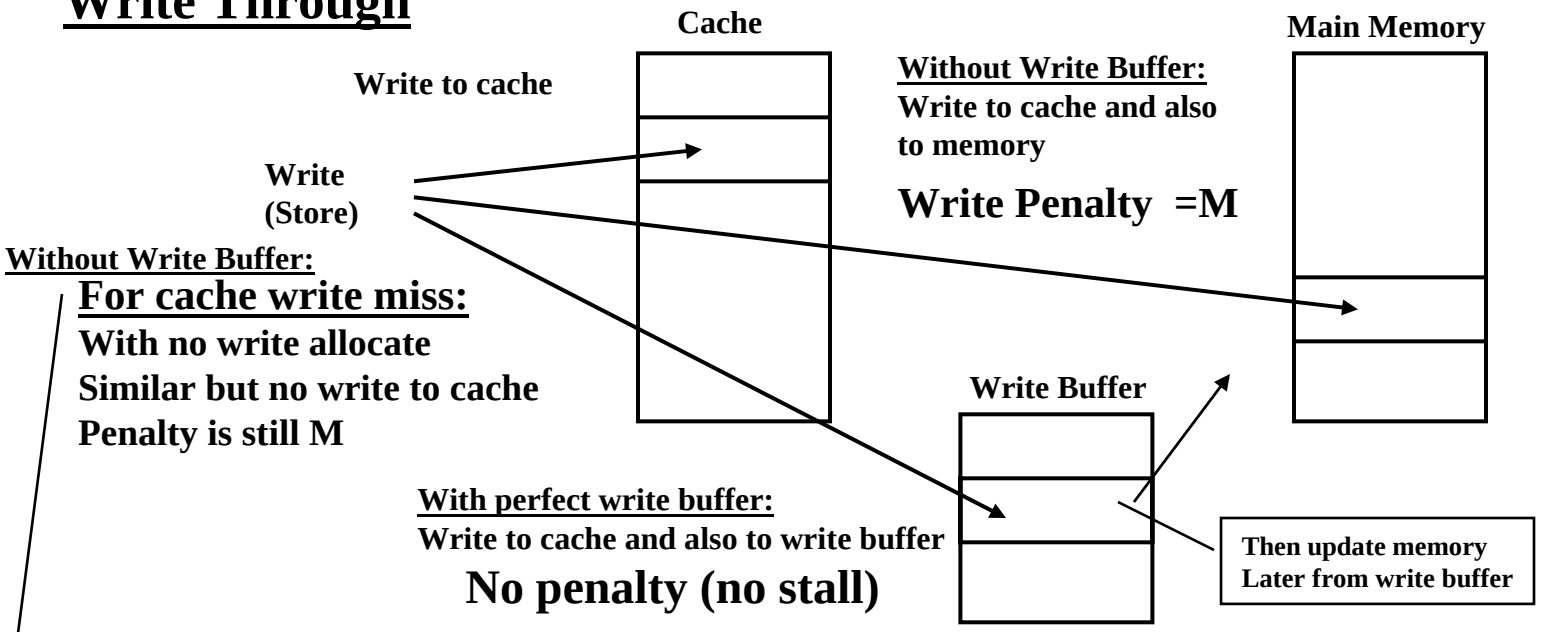


Cache Write Strategies:

Cache Write Hit = block to be modified is found in cache

Write Hit Operation (block to be written to is in cache)

Write Through



Without Write Buffer:

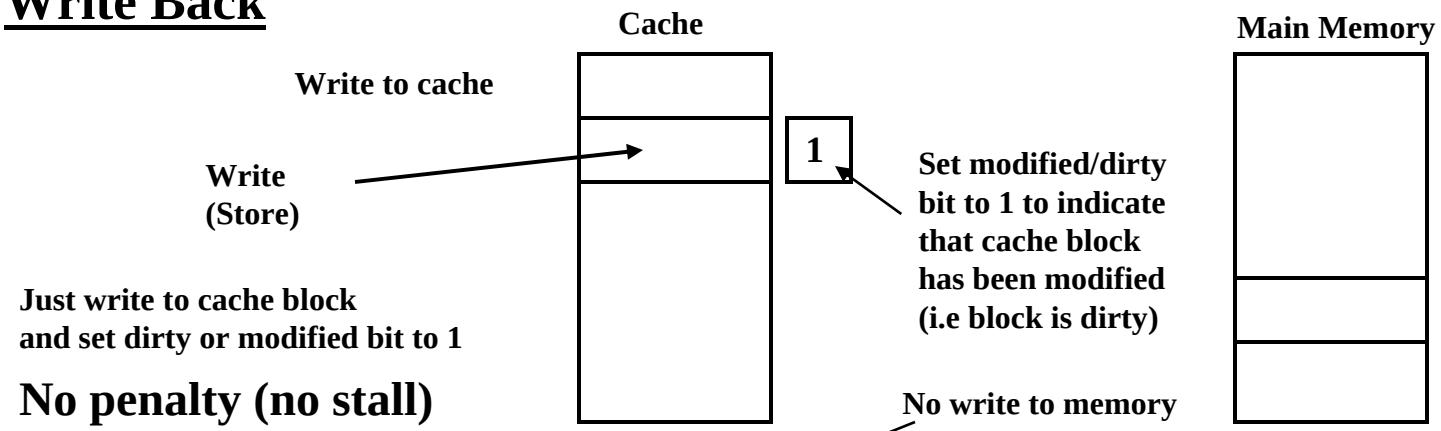
For cache write miss:

With no write allocate
Similar but no write to cache
Penalty is still M

With perfect write buffer:
Write to cache and also to write buffer

No penalty (no stall)

Write Back



Just write to cache block
and set dirty or modified bit to 1

No penalty (no stall)

Write back to memory when replaced in cache

Cache Write Miss = block to be modified is not in cache

Cache Write Miss Policy

- **Since data is usually not needed immediately on a write miss two options exist on a cache write miss:**

Write Allocate: *(Bring old block to cache then update it)*

The missed cache block is loaded into cache on a write miss followed by write hit actions.

i.e A cache block frame is allocated for the block to be modified (written-to)

No-Write Allocate:

i.e A cache block frame is not allocated for the block to be modified (written-to)

The block is modified in the lower level (lower cache level, or main memory) and not loaded (written or updated) into cache.

While any of the above two write miss policies can be used with either write back or write through:

- **Write back caches always use write allocate to capture subsequent writes to the block in cache.**
- **Write through caches usually use no-write allocate since subsequent writes still have to go to memory.**

Cache Write Miss = Block to be modified is not in cache
Allocate = Allocate or assign a cache block frame for written data

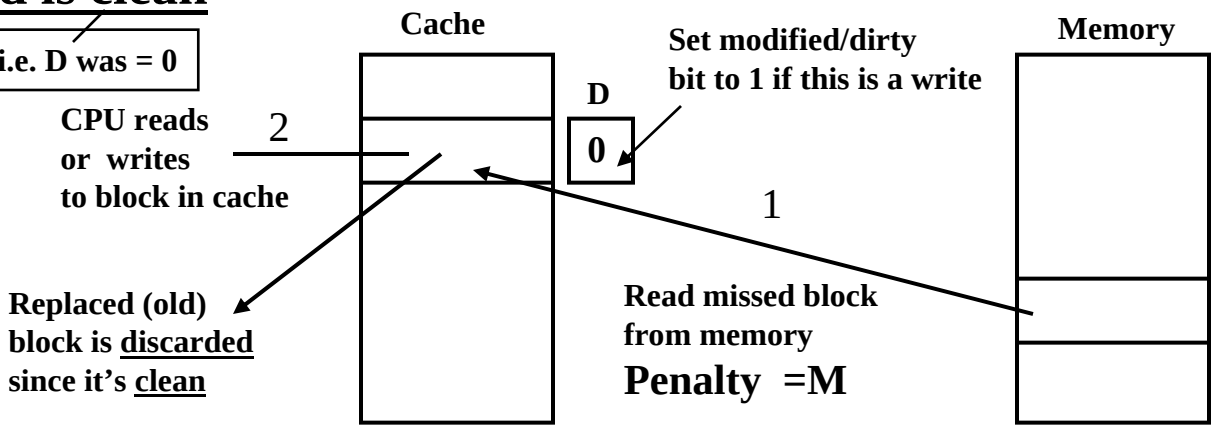
Write Back Cache With Write Allocate: Cache Miss Operation

(read or write miss)

Block to be replaced is clean

i.e. D was = 0

Miss Penalty = M



Block to be replaced is dirty (modified)

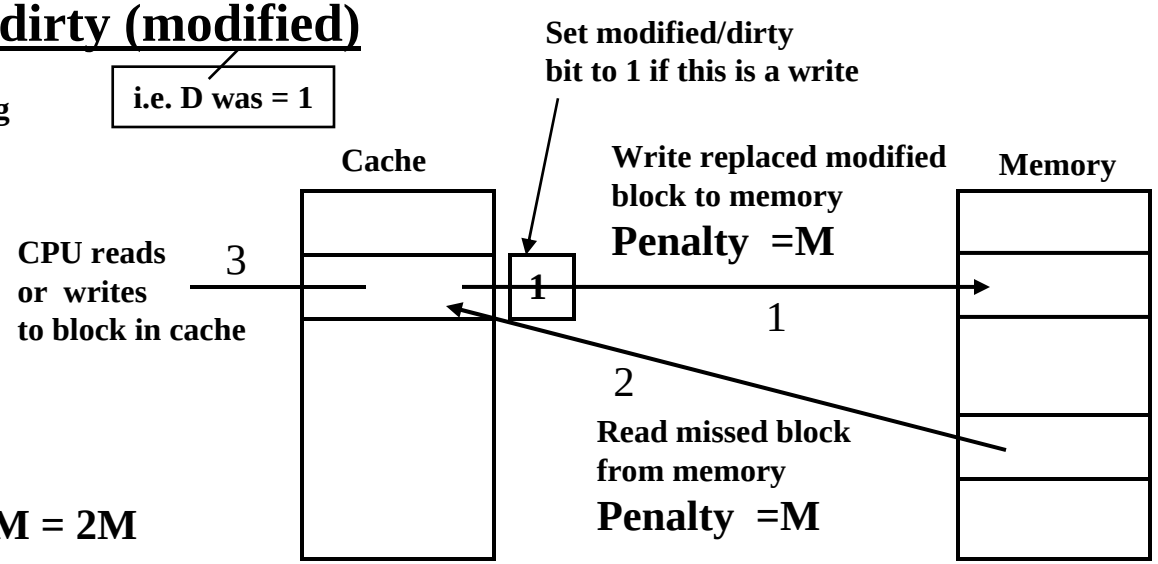
i.e. D was = 1

1 Write back modified block being replaced to memory
Penalty =M

2 Read missed block from memory
Penalty =M

Thus:

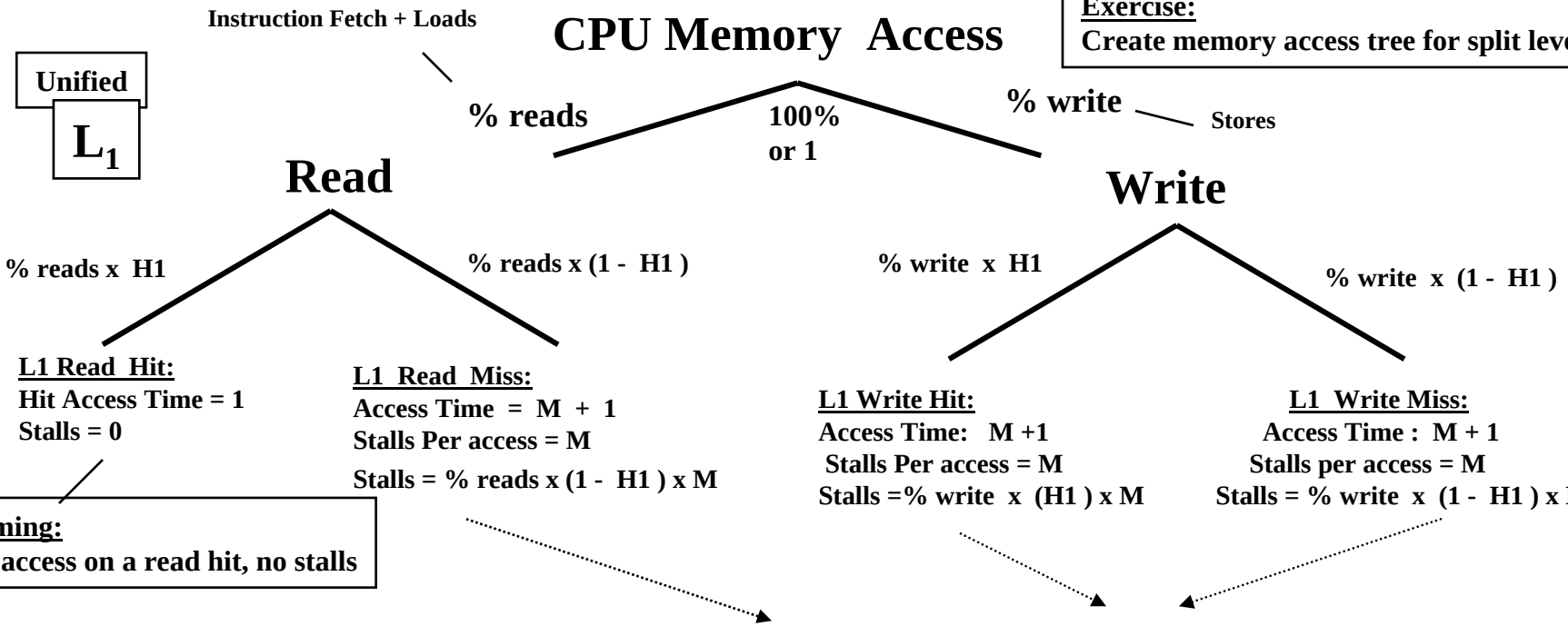
Total Miss Penalty = M + M = 2M



M = Miss Penalty = stall cycles per access resulting from missing in cache

Memory Access Tree, Unified L_1

Write Through, No Write Allocate, No Write Buffer



Exercise:
Create memory access tree for split level 1

Assuming:
Ideal access on a read hit, no stalls

Stall Cycles Per Memory Access = % reads x (1 - H1) x M + % write x M

AMAT = 1 + % reads x (1 - H1) x M + % write x M

CPI = $CPI_{execution}$ + (1 + fraction of loads/stores) x Stall Cycles per access

Stall Cycles per access = AMAT - 1

M = Miss Penalty
H1 = Level 1 Hit Rate
1- H1 = Level 1 Miss Rate

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H1 = Level 1 Hit Rate 1- H1 = Level 1 Miss Rate

Reducing Write Stalls For Write Though Cache Using Write Buffers

- **To reduce write stalls when write though is used, a write buffer is used to eliminate or reduce write stalls:**
  - **<u>Perfect write buffer:</u> All writes are handled by write buffer, no stalling for writes**
  - **In this case (for unified L1 cache):**

    Stall Cycles Per Memory Access = % reads x (1 - H1) x M

    (i.e No stalls at all for writes)
  - **<u>Realistic Write buffer:</u> A percentage of write stalls are not eliminated when the write buffer is full.**
  - **In this case (for unified L1 cache):**

Stall Cycles/Memory Access = (% reads x (1 - H1) + % write stalls not eliminated) x M

Write Through Cache Performance Example

- A CPU with $CPI_{execution}$ = 1.1 Mem accesses per instruction = 1.3
- Uses a unified L1 Write Through, No Write Allocate, with:
  - 1 – No write buffer.
  - 2 – Perfect Write buffer
  - 3 – A realistic write buffer that eliminates 85% of write stalls
- Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles. = M

CPI = $CPI_{execution}$ + mem stalls per instruction

% reads = 1.15/1.3 = 88.5% % writes = .15/1.3 = 11.5%

1 With No Write Buffer : Stall on all writes

Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5% + 11.5%) = 8.33 cycles

CPI = 1.1 + 8.33 = 9.43

2 With Perfect Write Buffer (all write stalls eliminated):

Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5%) = 0.86 cycles

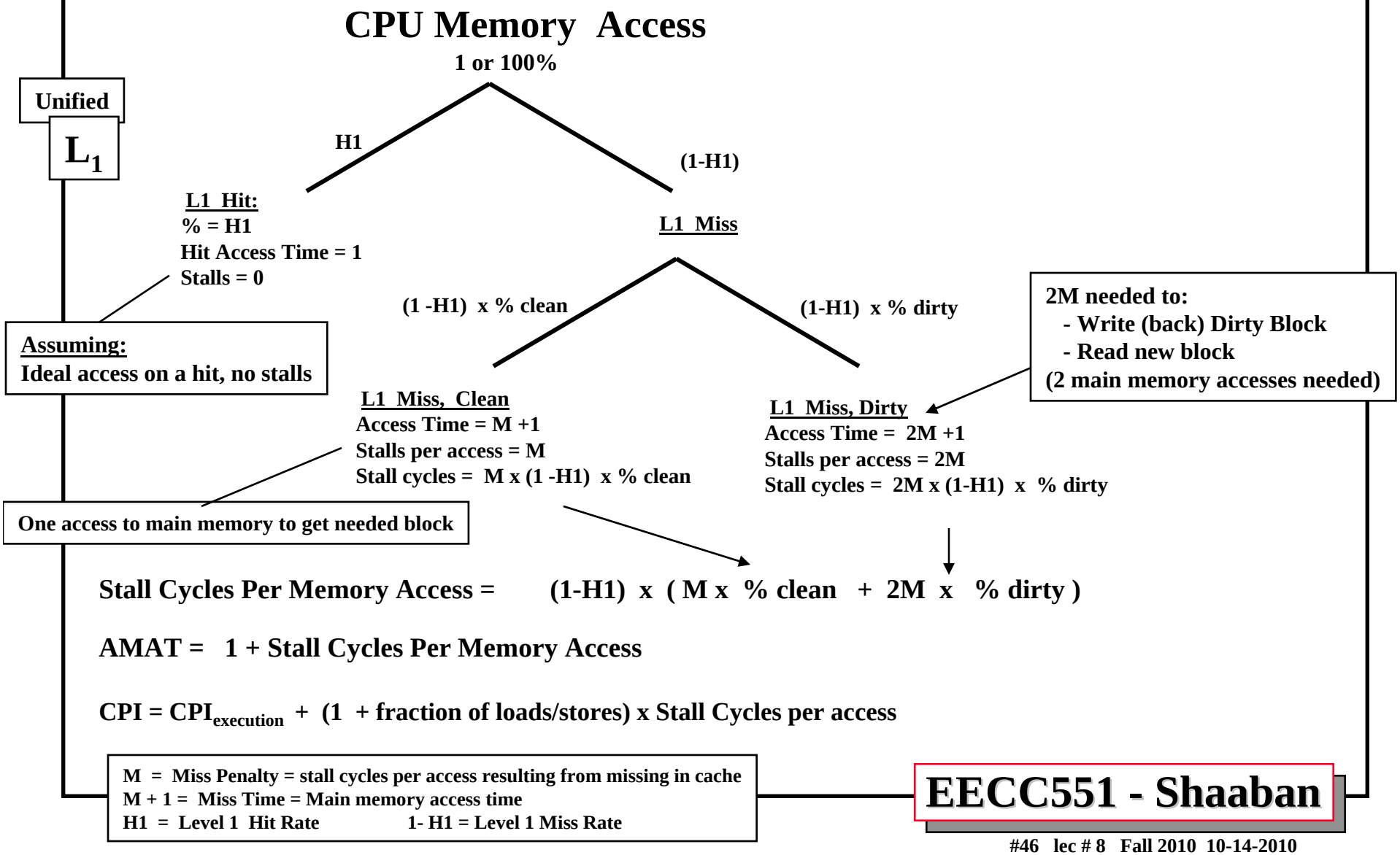
CPI = 1.1 + 0.86 = 1.96

3 With Realistic Write Buffer (eliminates 85% of write stalls)

Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5% + 15% x 11.5%) = 1.98 cycles

CPI = 1.1 + 1.98 = 3.08

Memory Access Tree Unified L_1 Write Back, With Write Allocate



Stall Cycles Per Memory Access = (1-H1) x (M x % clean + 2M x % dirty)

AMAT = 1 + Stall Cycles Per Memory Access

CPI = $CPI_{execution}$ + (1 + fraction of loads/stores) x Stall Cycles per access

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H1 = Level 1 Hit Rate 1- H1 = Level 1 Miss Rate

Write Back Cache Performance Example

- A CPU with $CPI_{execution}$ = 1.1 uses a unified L1 with with write back, with write allocate, and the probability a cache block is dirty = 10%
- Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

$$CPI = CPI_{execution}^{(1 - H1)} + \text{mem stalls per instruction}^{= M}$$

Mem Stalls per instruction =

Mem accesses per instruction x Stalls per access

Mem accesses per instruction = 1 + 0.3 = 1.3

Stalls per access = (1-H1) x (M x % clean + 2M x % dirty)

Stalls per access = 1.5% x (50 x 90% + 100 x 10%) = 0.825 cycles

AMAT = 1 + stalls per access = 1 + 0.825 = 1.825 cycles

Mem Stalls per instruction = 1.3 x 0.825 = 1.07 cycles

CPI = 1.1 + 1.07 = 2.17

The ideal CPU with no misses is 2.17/1.1 = 1.97 times faster

For Last Example

Memory Access Tree For Unified L_1
Write Back, With Write Allocate Example

CPU Memory Access

1 or 100%

H1 = 0.985 or 98.5%

(1-H1) = 0.015 or 1.5%

Unified L_1

L1 Hit:
% = H1 = 0.985 or 98.5%
Hit Access Time = 1
Stalls = 0

L1 Miss

(1 -H1) x % clean
= .015 x 0.9
= 0.0135 or 1.35%

(1-H1) x % dirty
= 0.015 x 0.1
= 0.0015 or 0.15%

Given Parameters:
H1 = 98.5% T1 = 0 cycles
M = 50 cycles
Stalls on a hit in L1
L1 Misses: 10% dirty 90% clean
$CPI_{execution}$ = 1.1
Memory accesses per instruction = 1.3

Assuming:
Ideal access on a hit in L_1

L1 Miss, Clean
Access Time = M +1 = 51
Stalls per access = M = 50
Stall cycles = M x (1 -H1) x % clean
= 50 x 0.0135 = 0.675 cycles

2M needed to Write Dirty Block and Read new block

L1 Miss, Dirty
Access Time = 2M +1= 101
Stalls per access = 2M = 100
Stall cycles = 2M x (1-H1) x % dirty
= 100 x 0.0015 = 0.15 cycles

Stall Cycles Per Memory Access = (M x (1-H1) x % clean + 2M x (1-H1) x % dirty)
= 0.675 + 0.15 = 0.825 cycles

AMAT = 1 + Stall Cycles Per Memory Access =1 + 0.825 = 1.825 cycles

Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access = 1.3 x 0.825 = 1.07 cycles

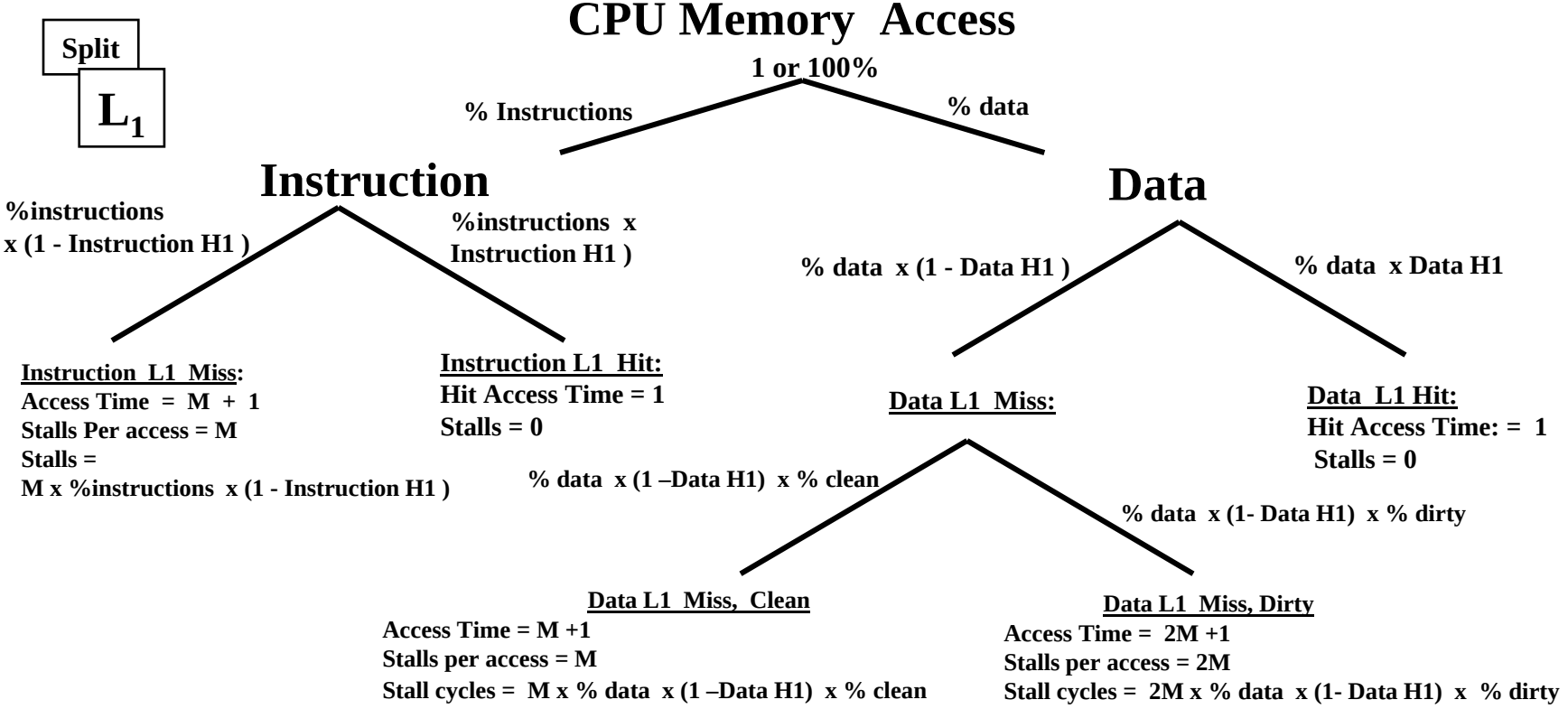
$CPI = CPI_{execution}$ + Stall cycles per instruction = 1.1 + 1.07 = 2.17

Given as 1.1

M = Miss Penalty = 50 cycles
M + 1 = Miss Time = 50 + 1 = 51 cycles L1 access Time = 1 cycle
H1 = 0.985 or 98.5% 1- H1 = 0.015 or 1.5%

Memory Access Tree Structure
For Separate Level 1 Caches, Write Back, With Write Allocate

(AKA Split)



Exercise: Find expression for: Stall cycles per average memory access, AMAT

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
Data H1 = Level 1 Data Hit Rate 1- Data H1 = Level 1 Data Miss Rate
Instruction H1 = Level 1 Instruction Hit Rate 1- Instruction H1 = Level 1 Instruction Miss Rate
% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses
% Clean = Percentage or fraction of data L1 misses that are clean
% Dirty = Percentage or fraction of data L1 misses that are dirty = 1 - % Clean

Assuming:
Ideal access on a hit in L_1

Improving Cache Performance: Multi-Level Cache

2 Levels of Cache: L_1, L_2

Basic Design Rule for L_1 Cache:
K.I.S.S
(e.g low degree of associatively and capacity to keep it fast)

3-4 cycles access time
2-8 way set associative

CPU

L_1 Cache

Assuming
Ideal access on a hit in L_1

Hit Rate= H_1
Hit Access Time = 1 cycle (No Stall)
Stalls for hit access = T_1 = 0

L_2 has slower access time than L_1, but has more capacity and higher associativity

10-15 cycles access time
8-16 way set associative

L_2 Cache

Local Hit Rate= H_2
Stalls per hit access= T_2
Hit Access Time = T_2 + 1 cycles

Main Memory

Slower (longer access time) than L_2

L_1 = Level 1 Cache
L_2 = Level 2 Cache

Memory access penalty, M
(stalls per main memory access)
Access Time = M +1

Goal of multi-level Caches:

Reduce the effective miss penalty incurred by level 1 cache misses by using additional levels of cache that capture some of these misses.

Thus hiding more main memory latency and reducing AMAT further

4th Edition: Appendix C.3
(3rd Edition Chapter 5.4)

Miss Rates For Multi-Level Caches

i.e that reach this level

- **Local Miss Rate:** This rate is the number of misses in a cache level divided by the number of memory accesses to this level (i.e those memory accesses that reach this level).

  Local Hit Rate = 1 - Local Miss Rate

- **Global Miss Rate:** The number of misses in a cache level divided by the total number of memory accesses generated by the CPU.
- Since level 1 receives all CPU memory accesses, for level 1:

  Local Miss Rate = Global Miss Rate = 1 - H1
- For level 2 since it only receives those accesses missed in 1:

  Local Miss Rate = Miss rate$_{L2}$ = 1- H2

  Global Miss Rate = Miss rate$_{L1}$ x Local Miss rate$_{L2}$

  = (1- H1) x (1 - H2)

For Level 3, global miss rate?

2-Level Cache (Both Unified) Performance (Ignoring Write Policy)

CPUtime = IC x (CPI$_{execution}$ + Mem Stall cycles per instruction) x C

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

- **For a system with 2 levels of unified cache, assuming no penalty when found in L_1 cache: ($T_1 = 0$)**

AMAT = 1 + Stall Cycles per access

Average stall cycles per memory access =

[miss rate L_1] x [Hit rate L_2 x Hit time L_2
+ Miss rate L_2 x Memory access penalty] =

(1-H1) x H2 x T2 + (1-H1)(1-H2) x M

L1 Miss, L2 Hit

Here we assume T1 =0 (no stall on L1 hit)

H1 = L1 Hit Rate
T1 = stall cycles per L1 access hit
H2 = Local L2 Hit Rate
T2 =stall cycles per L2 access hit

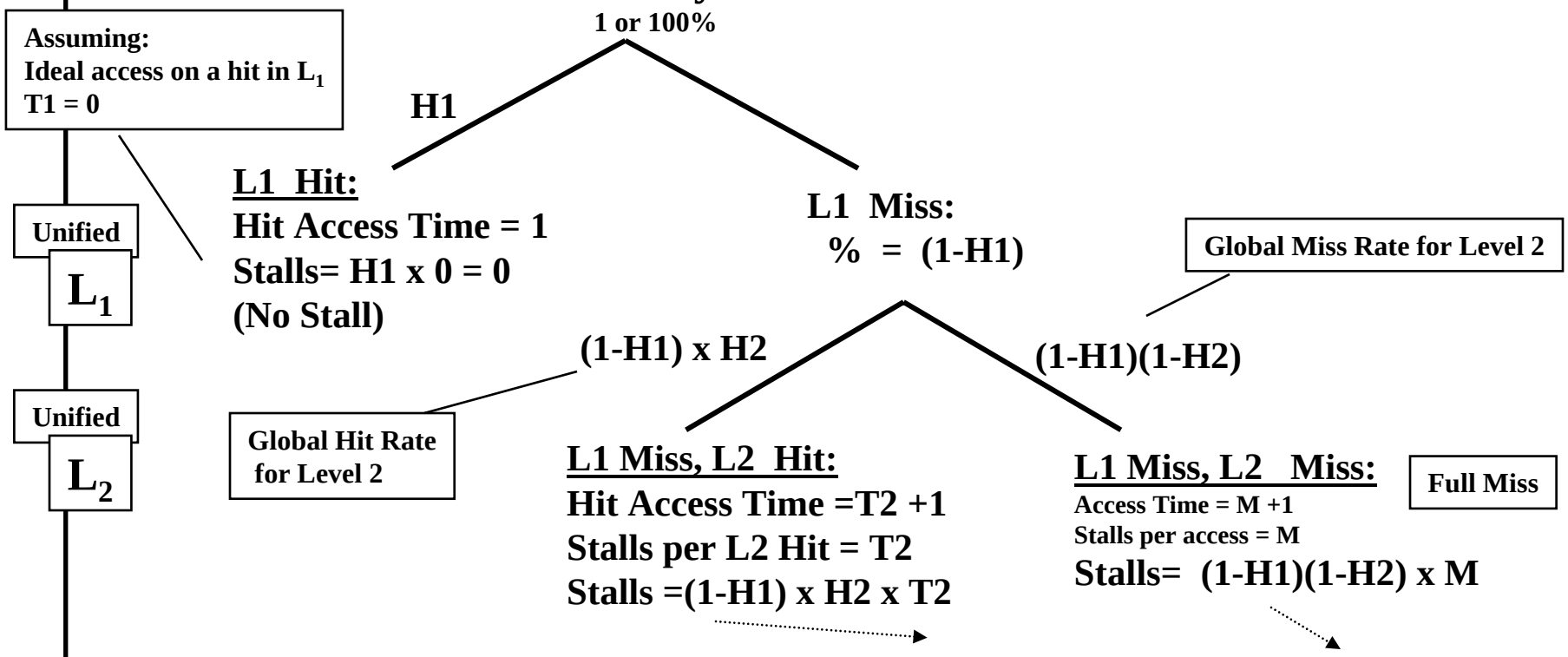
L1 Miss, L2 Miss: Must Access Main Memory

Full Miss

CPI = CPI$_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access
= CPI$_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

2-Level Cache (Both Unified) Performance Memory Access Tree (Ignoring Write Policy)

CPU Stall Cycles Per Memory Access



Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

AMAT = 1 + (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access
= $CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

Unified Two-Level Cache Example

(Ignoring Write Policy)

- CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- With two levels of cache (both unified)
- L_1 hit access time = 1 cycle (no stall on a hit, T1= 0), a miss rate of 5%
- L_2 hit access time = 3 cycles (T2= 2 stall cycles per hit) with local miss rate 40%,
- Memory access penalty, M = 100 cycles (stalls per access). **Find CPI ...**

i.e 1-H2

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

With No Cache, CPI = 1.1 + 1.3 x 100 = 131.1

With single L_1, CPI = 1.1 + 1.3 x .05 x 100 = 7.6

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

= 0.05 x .6 x 2 + 0.05 x 0.4 x 100

= 0.06 + 2 = 2.06 cycles

AMAT = 2.06 + 1 = 3.06 cycles

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

= 2.06 x 1.3 = 2.678 cycles

CPI = 1.1 + 2.678 = 3.778

Speedup = 7.6/3.778 = 2

Compared to CPU with L1 only

$CPI = CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access

$= CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

Memory Access Tree For 2-Level Cache (Both Unified) Example

For Last Example

CPU Stall Cycles Per Memory Access

(Ignoring Write Policy)

CPU Memory Access

1 or 100%

H1 = 0.95 or 95%

Unified L_1

L1 Hit:
Hit Access Time = 1
Stalls per L1 Hit = T1 = 0
Stalls= H1 x 0 = 0
(No Stall)

Ideal access on a hit in L_1
T1 = 0

L1 Miss:
(1-H1)= 0.05 or 5%

(1-H1) x H2
= 0.05 x 0.6
= 0.03 or 3%

(1-H1)(1-H2)
= 0.05 x 0.4
= 0.02 or 2%

Global Miss Rate for L_2

Unified L_2

Global Hit Rate for L_2

L1 Miss, L2 Hit:
Hit Access Time =T2 +1 = 3 cycles
Stalls per L2 Hit = T2 = 2 cycles
Stalls =(1-H1) x H2 x T2
= 0.03 x 2 = 0.06 cycles

L1 Miss, L2 Miss:
Access Time = M +1 = 100 + 1 =101 cycles
Stalls per access = M = 100 cycles
Stalls= (1-H1)(1-H2) x M
= 0.02 x 100 = 2 cycles

Full Miss

Given Parameters:
H1 = 95% T1 = 0 cycles
H2 = 60% T2 = 2 cycles
M = 100 cycles
Stalls on a hit
$CPI_{execution}$ = 1.1
Memory accesses per instruction = 1.3

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M
= 0.06 + 2 = 2.06 cycles

AMAT = 1 + Stall cycles per memory access = 1 + 2.06 = 3.06 cycles

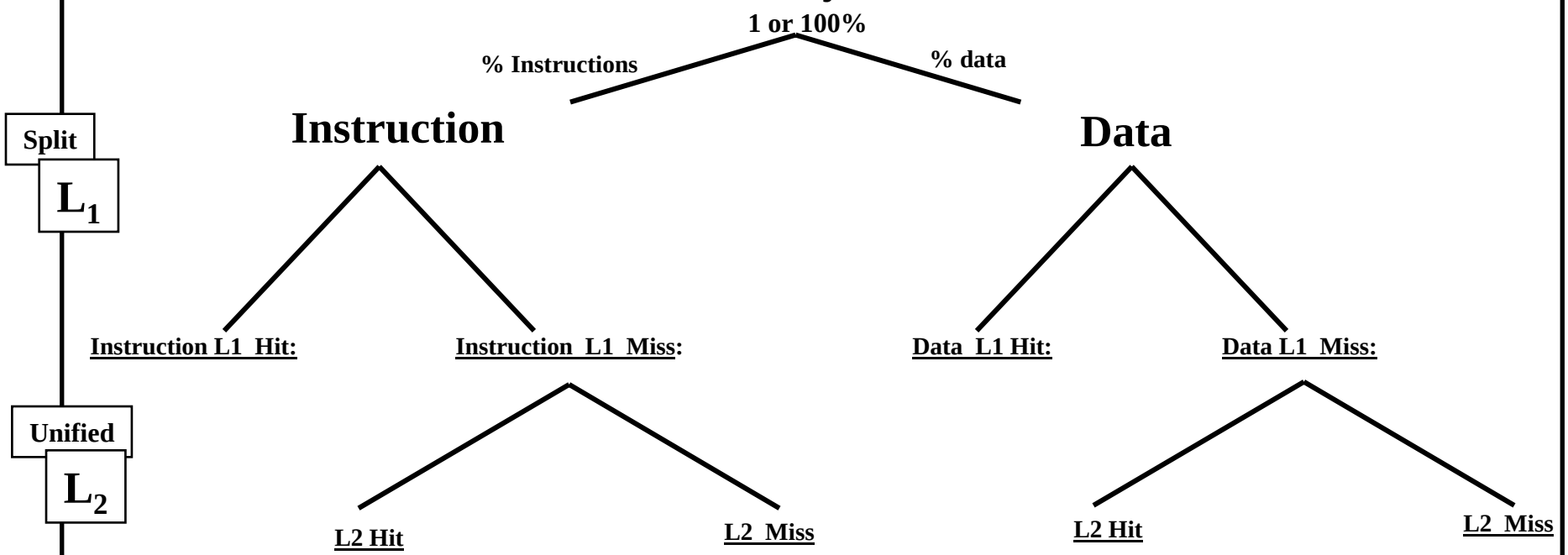
Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access
= 1.3 x 2.06 = 2.678 cycles

CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 2.678 = 3.778

CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access
= $CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

Memory Access Tree Structure For 2-Level Cache (Separate Level 1 Caches, Unified Level 2)

(Ignoring Write Policy)



Exercise: In terms of the parameters below, complete the memory access tree and find the expression for stall cycles per memory access

% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses

For L1: T1 = Stalls per hit access to level 1
Data H1 = Level 1 Data Hit Rate 1- Data H1 = Level 1 Data Miss Rate
Instruction H1 = Level 1 Instruction Hit Rate 1- Instruction H1 = Level 1 Instruction Miss Rate
For L2: T2 = Stalls per access to level 2
H2 = Level 2 local hit Rate 1-H2 = Level 2 local miss rate

M = Miss Penalty = stall cycles per access resulting from missing in cache level 2
M + 1 = Miss Time = Main memory access time

Common Write Policy For 2-Level Cache

L_1

- **Write Policy For Level 1 Cache:**
  - Usually Write through to Level 2. (not write through to main memory just to L2)
  - Write allocate is used to reduce level 1 read misses.
  - Use write buffer to reduce write stalls to level 2.

L_2

- **Write Policy For Level 2 Cache:**
  - Usually write back with write allocate is used.
    - To minimize memory bandwidth usage.

- The above 2-level cache write policy results in inclusive L2 cache since the content of L1 is also in L2
  - Common in the majority of all CPUs with 2-levels of cache
  - As opposed to exclusive L1, L2 (e.g AMD Athlon XP, A64)

i.e what is in L1 is not duplicated in L2

As if we have a single level of cache with one portion (L1) is faster than remainder (L2)

2-Level (Both Unified) Memory Access Tree

L1: Write Through to L2, Write Allocate, With Perfect Write Buffer
L2: Write Back with Write Allocate

Assuming:
Ideal access on a hit in L_1
T1 = 0

CPU Memory Access

1 or 100%

(H1) **(1-H1)**

Unified L_1

L1 Hit:
Hit Access Time = 1
Stalls Per access = 0

L1 Miss:

(1-H1) x H2

Global Miss Rate for L_2

Unified L_2

L1 Miss, L2 Hit:
Hit Access Time =T2 +1
Stalls per L2 Hit = T2
Stalls = (1-H1) x H2 x T2

(1-H1) x (1-H2)

L1 Miss, L2 Miss

(1 -H1) x (1-H2) x % clean **(1-H1) x (1-H2) x % dirty**

L1 Miss, L2 Miss, Clean
Access Time = M +1
Stalls per access = M
Stall cycles =
M x (1 -H1) x (1-H2) x % clean

L1 Miss, L2 Miss, Dirty
Access Time = 2M +1
Stalls per access = 2M
Stall cycles = 2M x (1-H1) x (1-H2) x % dirty

Stall cycles per memory access = (1-H1) x H2 x T2 + M x (1 -H1) x (1-H2) x % clean + 2M x (1-H1) x (1-H2) x % dirty

= (1-H1) x H2 x T2 + (1 -H1) x (1-H2) x (% clean x M + % dirty x 2M)

AMAT = 1 + Stall Cycles Per Memory Access
CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x Stall Cycles per access

Two-Level (Both Unified) Cache Example With Write Policy

- **CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz**
- **1.3 memory accesses per instruction. Two levels of cache (both unified)**
- **For L_1 :** **i.e. ideal access time = 1 cycle**
  - **Cache operates at 500 MHz (no stall on L1 Hit, T1 =0) with a miss rate of 1-H1 = 5%**
  - **Write though to L_2 with perfect write buffer with write allocate**
- **For L_2:**
  - **Hit access time = 3 cycles (T2= 2 stall cycles per hit) local miss rate 1- H2 = 40%**
  - **Write back to main memory with write allocate**
  - **Probability a cache block is dirty = 10%**
- **Memory access penalty, M = 100 cycles.**
- **Create memory access tree and find, stalls per memory access, AMAT, CPI.**
- Stall cycles per memory access = (1-H1) x H2 x T2 +
  (1 -H1) x (1-H2) x (% clean x M + % dirty x 2M)

  = .05 x .6 x 2 + .05 x .4 x (.9 x 100 + .1 x200)

  = .06 + 0.02 x 110 = .06 + 2.2 = 2.26
- AMAT = 2.26 + 1 = 3.26 cycles

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

= 2.26 x 1.3 = 2.938 cycles

CPI = 1.1 + 2.938 = 4.038 = 4

CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT –1)

Memory Access Tree For Two-Level (Both Unified) Cache Example With Write Policy

For Last Example

L1: Write Through to L2, Write Allocate, With Perfect Write Buffer
L2: Write Back with Write Allocate

CPU Memory Access

Given Parameters:

H1 = 95% T1 = 0 cycles
H2 = 60% T2 = 2 cycles (Stalls on a hit)
M = 100 cycles

L2 Misses: 10% dirty 90% clean
$CPI_{execution}$ = 1.1
Memory accesses per instruction = 1.3

Unified L_1

1 or 100%

(H1) = 0.95 or 95%

(1-H1) = 0.05 or 5%

L1 Hit:
Hit Access Time = 1
Stalls Per access = 0

L1 Miss:

(1-H1) x H2 = 0.05 x 0.6 = 0.03 or 3%

Unified L_2

L1 Miss, L2 Hit:
Hit Access Time = T2 +1 = 3 cycles
Stalls per L2 Hit = T2 = 2 cycles
Stalls = (1-H1) x H2 x T2
= 0.03 x 2 = 0.06 cycles

(1-H1) x (1-H2) = 0.05 x 0.4 = 0.02 or 2%

L1 Miss, L2 Miss

(1 -H1) x (1-H2) x % clean = 0.02 x 0.9 = 0.018 or 1.8%

(1-H1) x (1-H2) x % dirty = 0.02 x 0.1 = 0.002 or 0.2 %

L1 Miss, L2 Miss, Clean
Access Time = M +1 = 101 cycles
Stalls per access = M
Stall cycles = M x (1 -H1) x (1-H2) x % clean
= 100 x 0.018 = 1.8 cycles

L1 Miss, L2 Miss, Dirty
Access Time = 2M +1 = 200 + 1 = 201 cycles
Stalls per access = 2M = 200 cycles
Stall cycles = 2M x (1-H1) x (1-H2) x % dirty
= 200 x 0.002 = 0.4 cycles

Stall cycles per memory access = (1-H1) x H2 x T2 + M x (1 -H1) x (1-H2) x % clean + 2M x (1-H1) x (1-H2) x % dirty
= 0.06 + 1. 8 + 0.4 = 2.26 cycles

AMAT = 1 + Stall cycles per memory access = 1 + 2.26 = 3.26 cycles

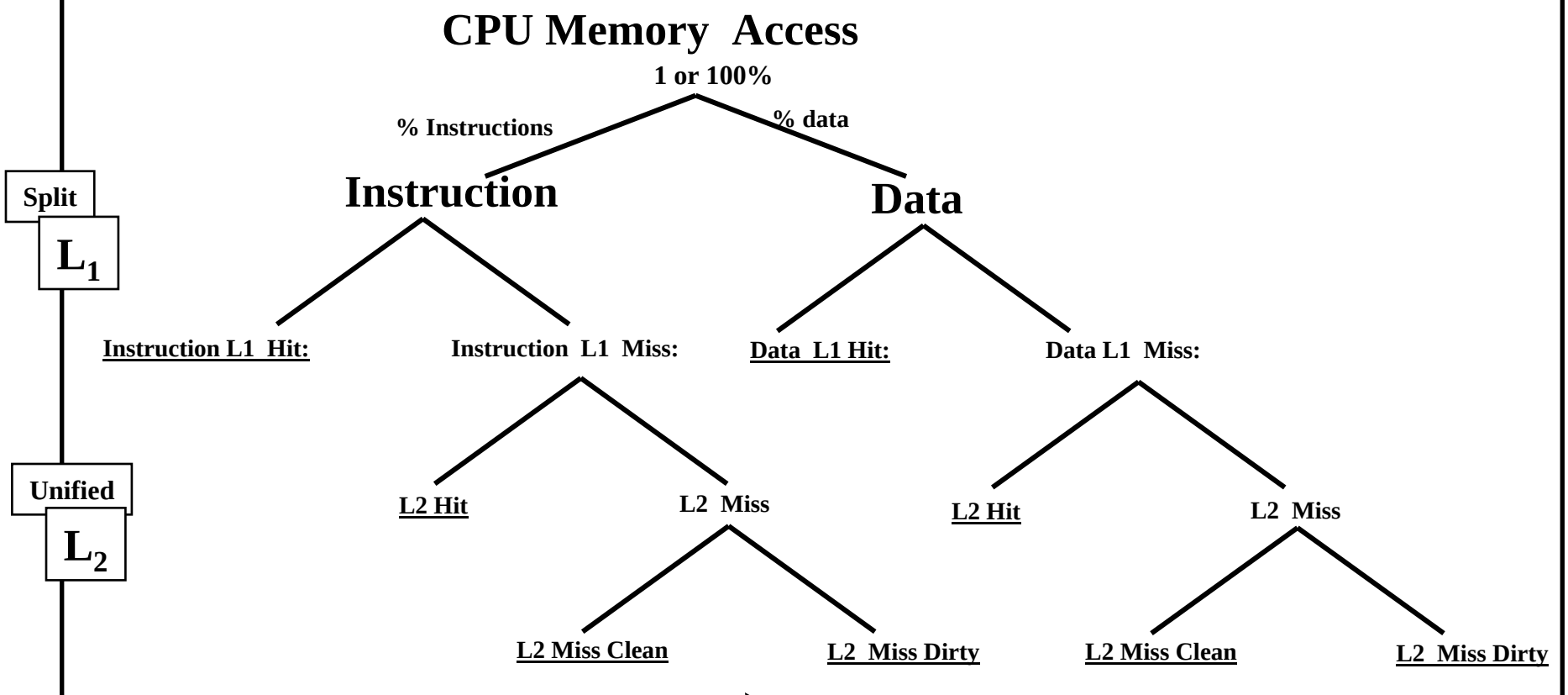
Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access
= 1.3 x 2.26 = 2.938 cycles

CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 2.938 = 4.038

AMAT = 1 + Stall Cycles Per Memory Access
CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x Stall Cycles per access

Memory Access Tree Structure For 2-Level Cache(Separate Level 1 Caches, Unified Level 2)

L1: Write Through to L2, Write Allocate, With Perfect Write Buffer L2: Write Back with Write Allocate



Exercise: In terms of the parameters below, complete the memory access tree and find the expression for stall cycles per memory access

% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses

For L1: T1 = Stalls per hit access to level 1
Data H1 = Level 1 Data Hit Rate 1- Data H1 = Level 1 Data Miss Rate
Instruction H1 = Level 1 Instruction Hit Rate 1- Instruction H1 = Level 1 Instruction Miss Rate
For L2: T2 = Stalls per access to level 2
H2 = Level 2 local hit Rate 1-H2 = Level 2 local miss rate
% Clean = Percentage or fraction of data L2 misses that are clean
% Dirty = Percentage or fraction of L2 misses that are dirty = 1 - % Clean

M = Miss Penalty = stall cycles per access resulting from missing in cache level 2
M + 1 = Miss Time = Main memory access time

Multi-Level Cache:

3 Levels of Cache

Basic Design Rule for L_1 Cache:
K.I.S.S
(e.g low degree of associatively and capacity to keep it fast)

3-4 cycles access time
2-8 way set associative

CPU

Assuming
Ideal access on a hit in L_1

Hit Rate= H_1,
Hit Access Time = 1 cycle (No Stall)
Stalls for hit access = T_1 = 0

Slower than L_1
But has more capacity and higher associativity

10-15 cycles access time
8-16 way set associative

L1 Cache

Local Hit Rate= H_2
Stalls per hit access= T_2
Hit Access Time = T_2 + 1 cycles

L2 Cache

Slower the L_2
But has more capacity and higher associativity

30-60 cycles access time
16-64 way set associative

L3 Cache

Local Hit Rate= H_3
Stalls per hit access= T_3
Hit Access Time = T_3 + 1 cycles

Main Memory

Slower the L_3

L_1 = Level 1 Cache
L_2 = Level 2 Cache
L_3 = Level 3 Cache

Memory access penalty, M
(stalls per main memory access)
Access Time = M +1

$$CPI = CPI_{execution} + (1 + \text{fraction of loads and stores}) \times \text{stall cycles per access}$$
$$= CPI_{execution} + (1 + \text{fraction of loads and stores}) \times (AMAT - 1)$$

3-Level (All Unified) Cache Performance

(Ignoring Write Policy)

CPUtime = IC x (CPI$_{execution}$ + Mem Stall cycles per instruction) x C

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

- **For a system with 3 levels of cache, assuming no penalty when found in L_1 cache: ($T_1 = 0$)**

Stall cycles per memory access =

[miss rate L_1] x [Hit rate L_2 x Hit time L_2

+ Miss rate L_2 x (Hit rate L3 x Hit time L_3

+ Miss rate L_3 x Memory access penalty)] =

(1-H1) x H2 x T2

+ (1-H1) x (1-H2) x H3 x T3

+ (1-H1)(1-H2) (1-H3)x M

L1 Miss, L2 Miss: Must Access Main Memory

Full Miss

L1 Miss, L2 Hit

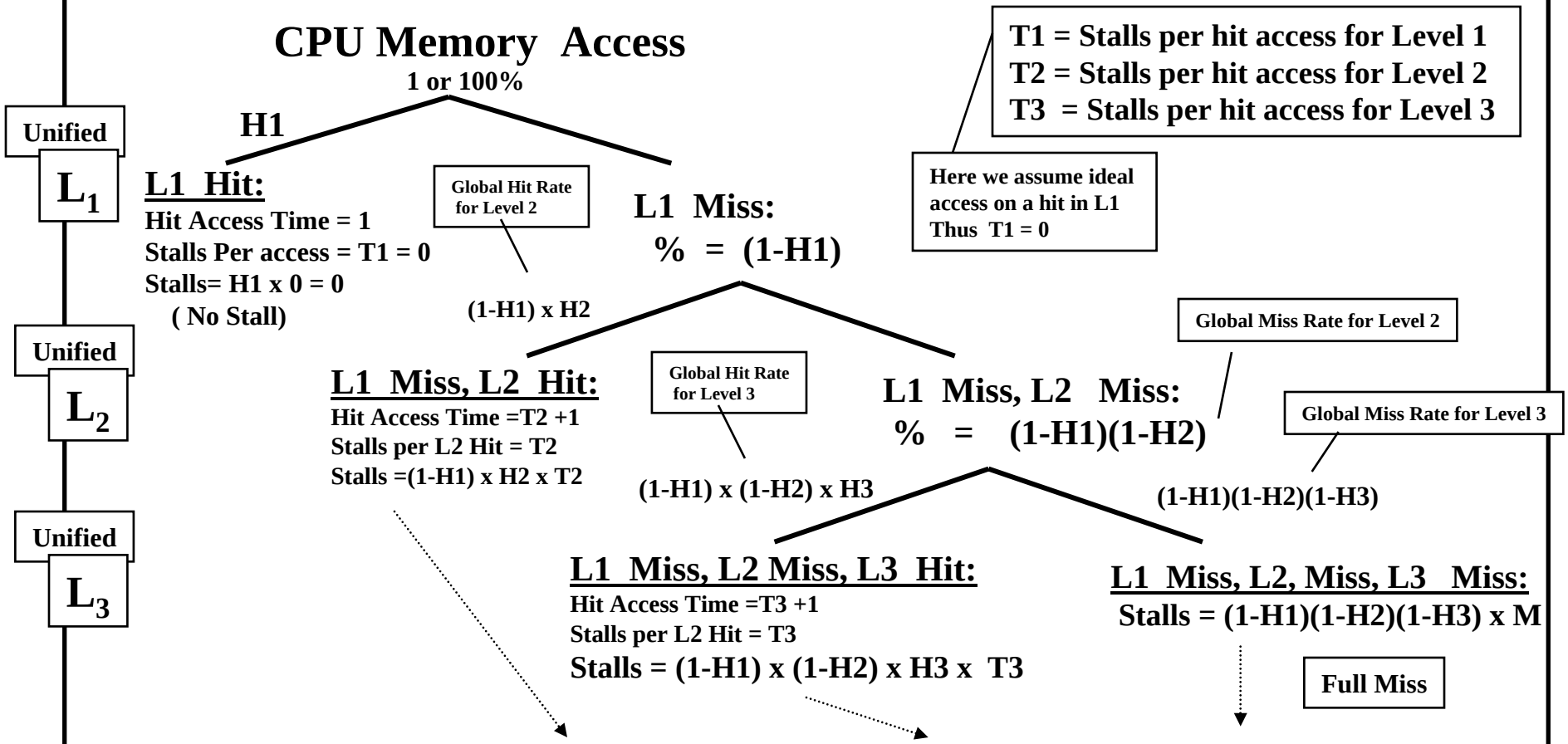
L1 Miss, L2 Miss, L3 Hit

CPI = CPI$_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access
= CPI$_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

3-Level (All Unified) Cache Performance

Memory Access Tree (Ignoring Write Policy)

CPU Stall Cycles Per Memory Access



Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2) (1-H3)x M

AMAT = 1 + Stall cycles per memory access

CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access
= $CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

Three-Level (All Unified) Cache Example

(Ignoring Write Policy)

- CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- L_1 cache operates at 500 MHz (no stalls on a hit in L1) with a miss rate of 5%
- L_2 hit access time = 3 cycles (T2= 2 stall cycles per hit), local miss rate 40%
- L_3 hit access time = 6 cycles (T3= 5 stall cycles per hit), local miss rate 50%,
- Memory access penalty, M= 100 cycles (stall cycles per access). Find CPI.

With No Cache, CPI = 1.1 + 1.3 x 100 = 131.1

With single L_1, CPI = 1.1 + 1.3 x .05 x 100 = 7.6

With L1, L2 CPI = 1.1 + 1.3 x (.05 x .6 x 2 + .05 x .4 x 100) = 3.778

$$CPI = CPI_{execution} + \text{Mem Stall cycles per instruction}$$

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2) (1-H3)x M

= .05 x .6 x 2 + .05 x .4 x .5 x 5 + .05 x .4 x .5 x 100

= .06 + .05 + 1 = 1.11

AMAT = 1.11 + 1 = 2.11 cycles (vs. AMAT = 3.06 with L1, L2, vs. 5 with L1 only)

$$CPI = 1.1 + 1.3 \times 1.11 = 2.54$$

Speedup compared to L1 only = 7.6/2.54 = 3

Speedup compared to L1, L2 = 3.778/2.54 = 1.49

All cache levels are unified, ignoring write policy

Memory Access Tree For 3-Level Cache (All Unified) Example

For Last Example

(Ignoring Write Policy)

CPU Memory Access

1 or 100%)

H1 = .95 or 95%

Unified L_1

L1 Hit:
Hit Access Time = 1
Stalls Per access = 0
Stalls= H1 x 0 = 0
(No Stall)

Global Hit Rate for Level 2

(1-H1) x H2
= 0.05 x .6
= 0.03 or 3%

L1 Miss:
(1-H1) = 0.05 or 5%

Given Parameters:
H1 = 95% T1 = 0 cycles
H2 = 60% T2 = 2 cycles
H3 = 50% T3 = 5 cycles
M = 100 cycles
Stalls on a hit
$CPI_{execution}$ = 1.1
Memory accesses per instruction = 1.3

Unified L_2

L1 Miss, L2 Hit:
Hit Access Time =T2 +1 = 3
Stalls per L2 Hit = T2 =2
Stalls =(1-H1) x H2 x T2
= .05 x .6 x 2 = .06

Global Hit Rate for Level 3

(1-H1) x (1-H2) x H3
= .05 x .4 x .5
= .01 or 1%

L1 Miss, L2 Miss:
(1-H1)(1-H2) = .05 x .4 = .02 or 2%

Global Miss Rate for Level 2

(1-H1)(1-H2)(1-H3)
= .05 x .4 x .5
= .01 or 1%

Global Miss Rate for Level 3

Unified L_3

L1 Miss, L2 Miss, L3 Hit:
Hit Access Time =T3 +1 = 6
Stalls per L2 Hit = T3 = 5
Stalls = (1-H1) x (1-H2) x H3 x T3
= .01 x 5 = .05 cycles

L1 Miss, L2, Miss, L3 Miss:
Miss Penalty = M = 100
Stalls = (1-H1)(1-H2)(1-H3) x M
= .01 x 100 = 1 cycle

Full Miss

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1) x (1-H2) x H3 x T3 + (1-H1)(1-H2) (1-H3)x M
= .06 + .05 + 1 = 1.11

AMAT = 1 + Stall cycles per memory access
= 1 + 1.11 = 2.11 cycles

Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access
= 1.3 x 1.11 = 1.443 cycles

CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 1.443 = 2.543

Exercise: Create the memory access tree for 3-level cache where level 1 is split and Levels 2, 3 are unified once ignoring write policy and once with write back for L3. Find the expression for memory stalls per access for either case.