Removing The Ideal Memory Assumption:

The Memory Hierarchy & Cache

- **The impact of real memory on CPU Performance.**
- **Main memory basic properties:**
  - Memory Types: DRAM vs. SRAM
- **The Motivation for The Memory Hierarchy:**
  - CPU/Memory Performance Gap
  - The Principle Of Locality
- **Memory Hierarchy Structure & Operation**
- **Cache Concepts:**
  - Block placement strategy & Cache Organization:
    - **Fully Associative, Set Associative, Direct Mapped.**
  - Cache block identification: Tag Matching
  - Block replacement policy
  - Cache storage requirements
  - Unified vs. Separate Cache
- **CPU Performance Evaluation with Cache:**
  - Average Memory Access Time (AMAT)
  - Memory Stall cycles
  - Memory Access Tree

Removing The Ideal Memory Assumption

- **So far we have assumed that ideal memory is used for both instruction and data memory in all CPU designs considered:**
  - Single Cycle, Multi-cycle, and Pipelined CPUs.
- **Ideal memory is characterized by a short delay or memory access time (one cycle) comparable to other components in the datapath.**
  - i.e 2ns which is similar to ALU delays.
- **Real memory utilizing Dynamic Random Access Memory (DRAM) has a much higher access time than other datapath components (80ns or more).** Memory Access Time >> 1 CPU Cycle
- **Removing the ideal memory assumption in CPU designs leads to a large increase in clock cycle time and/or CPI greatly reducing CPU performance.**

Ideal Memory Access Time ≤ 1 CPU Cycle
Real Memory Access Time >> 1 CPU cycle

Removing The Ideal Memory Assumption

- **For example if we use real (non-ideal) memory with 80 ns access time (instead of 2ns) in our CPU designs then:**
- **Single Cycle CPU:**
  - Loads will require 80ns + 1ns + 2ns + 80ns + 1ns = 164ns = C
  - The CPU clock cycle time C increases from 8ns to 164ns (125MHz to 6 MHz)
  - CPU is 20.5 times slower
- **Multi Cycle CPU:**
  - To maintain a CPU cycle of 2ns (500MHz) instruction fetch and data memory now take 80/2 = 40 cycles each resulting in the following CPIs
    - **Arithmetic Instructions CPI = 40 + 3 = 43 cycles**
    - **Jump/Branch Instructions CPI = 40 + 2 = 42 cycles**
    - **Store Instructions CPI = 80 + 2 = 82 cycles**
    - **Load Instructions CPI = 80 + 3 = 83 cycles**
    - **Depending on instruction mix, CPU is 11-20 times slower++**
- **Pipelined CPU:**
  - To maintain a CPU cycle of 2ns, a pipeline with 83 stages is needed.
  - Data/Structural hazards over instruction/data memory access may lead to 40 or 80 stall cycles per instruction.
  - Depending on instruction mix CPI increases from 1 to 41-81 and the CPU is 41-81 times slower!

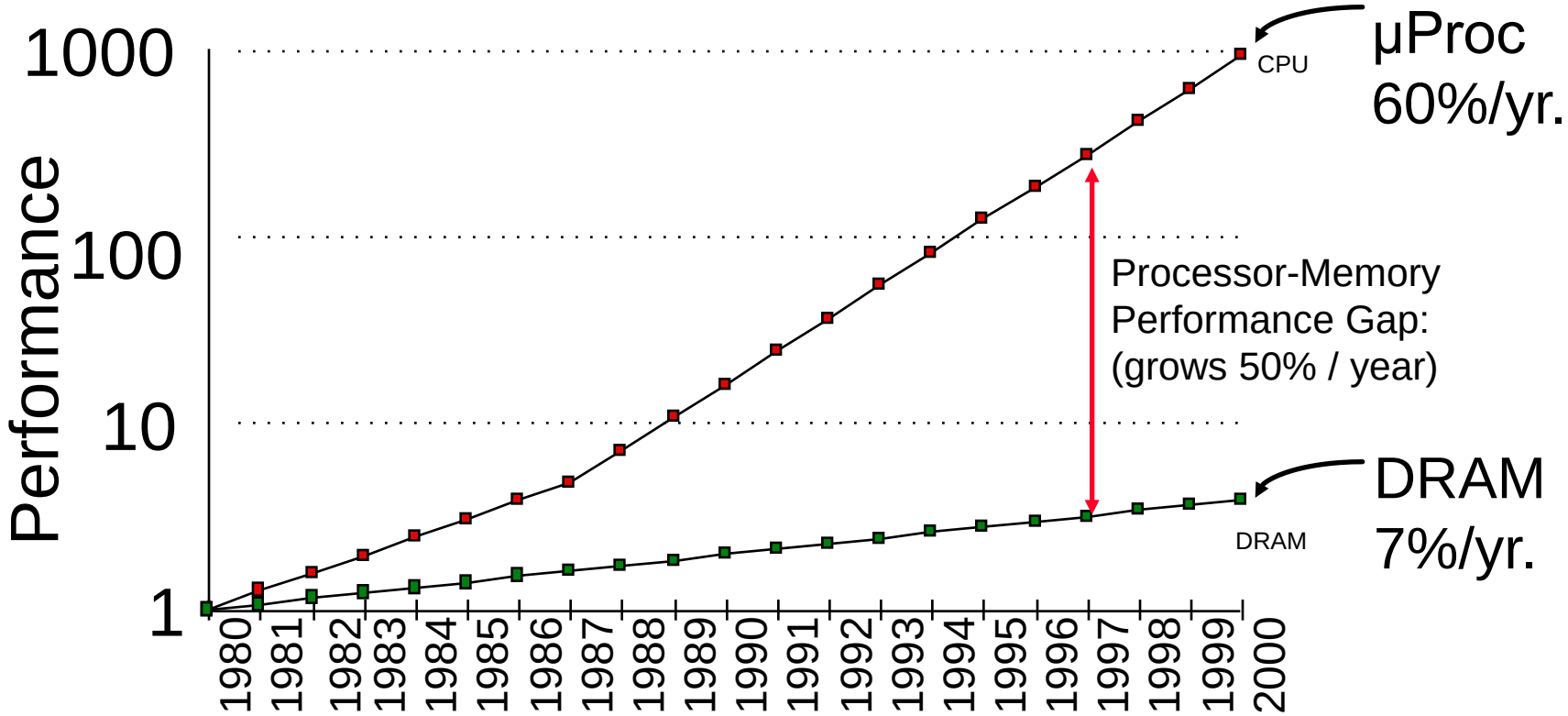
T = I x CPI x C

Ideal Memory Access Time ≤ 1 CPU Cycle
Real Memory Access Time >> 1 CPU cycle

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

Processor-DRAM Performance Gap: Impact of Real Memory on CPI

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

i.e wait cycles added to CPI

Ideal Memory Access Time ≤ 1 CPU Cycle
Real Memory Access Time >> 1 CPU cycle

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

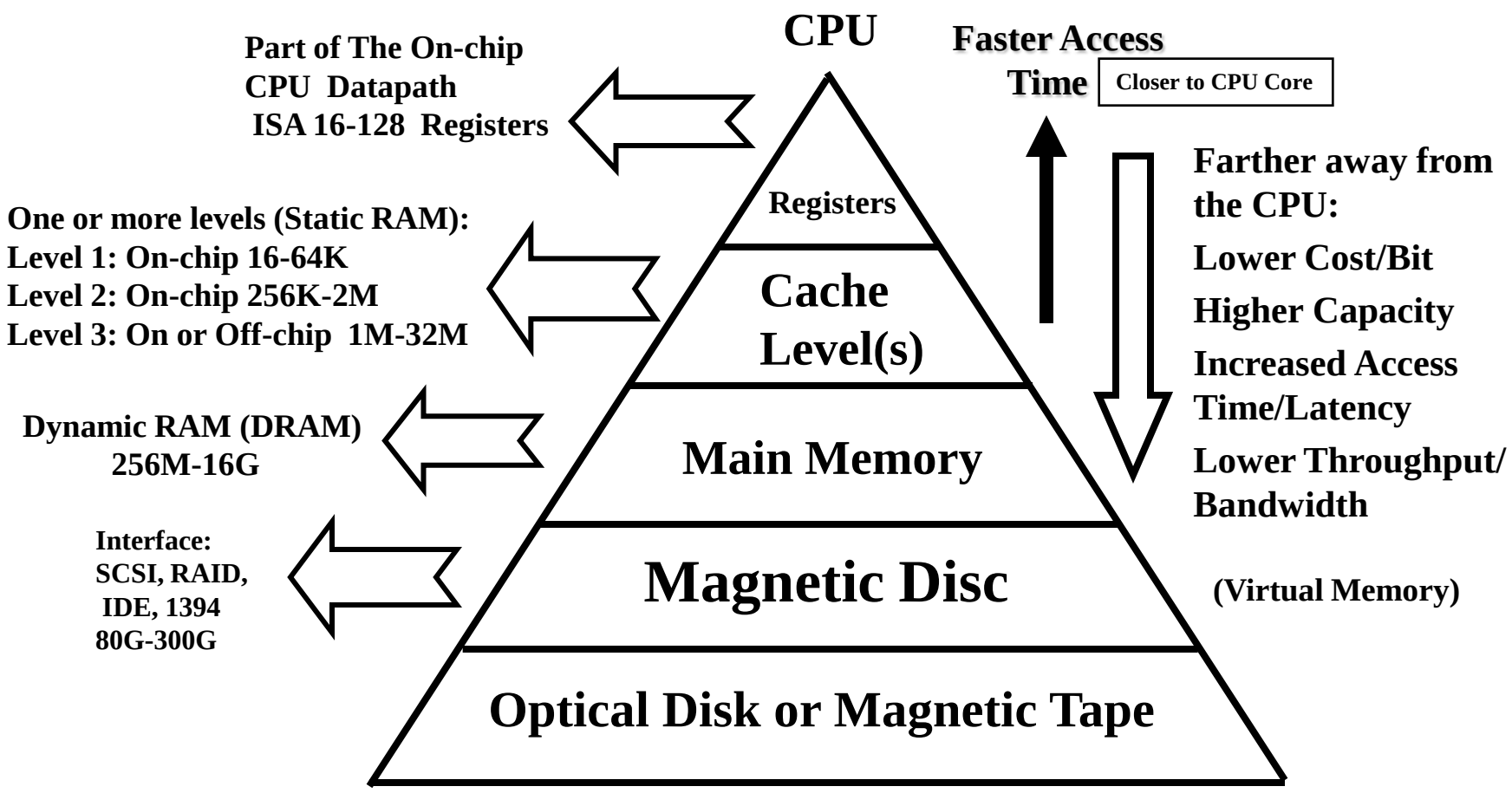
- **To hide long memory access latency, the memory hierarchy is organized into several levels of memory with the smaller, faster SRAM-based memory levels closer to the CPU: registers, then primary Cache Level (L_1), then additional secondary cache levels (L_2, L_3…), then DRAM-based main memory, then 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.**

Levels of The Memory Hierarchy



The Principle Of Locality

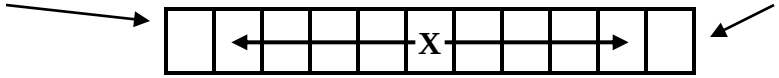
- **Programs usually access a relatively small portion of their address space (instructions/data) at any instant of time (program working set).**

  Thus: Memory Access Locality → Program Working Set

- **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.**

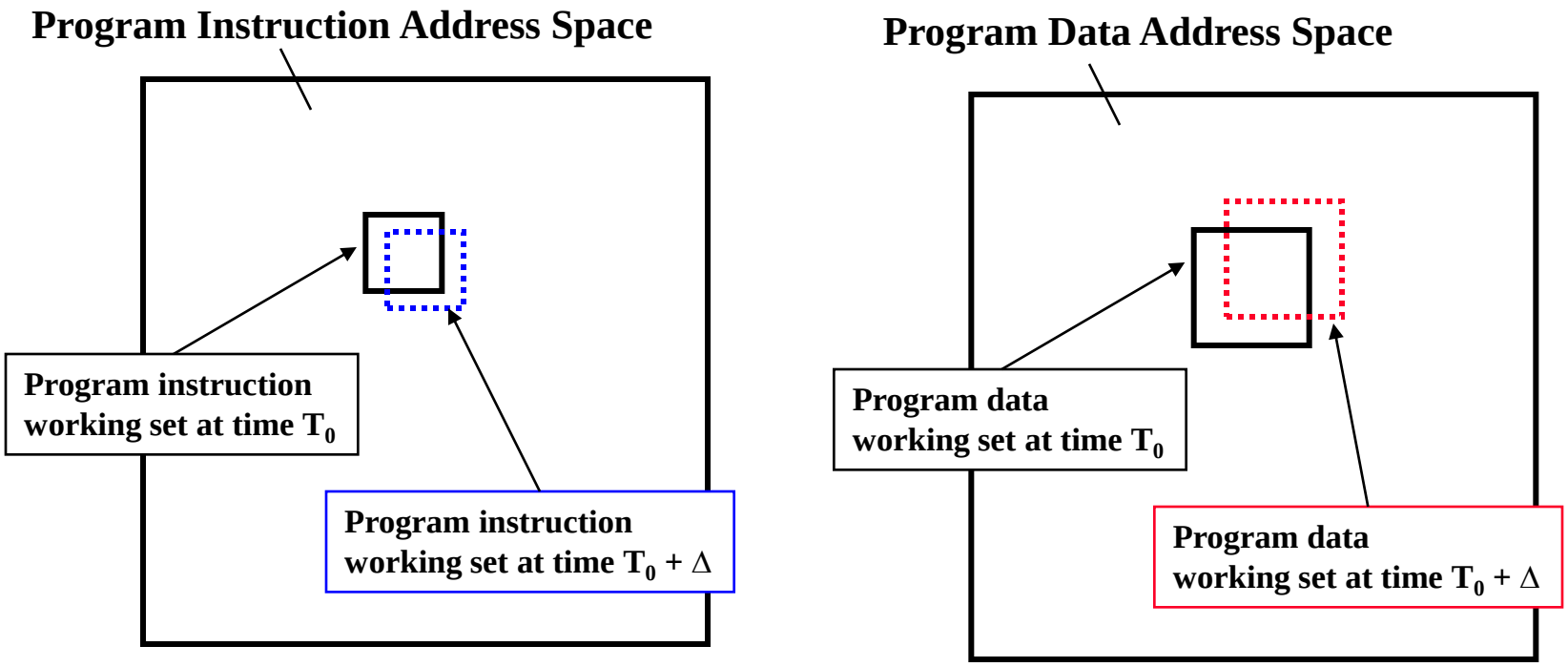
  Cache utilizes faster memory (SRAM)

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)



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
  - 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
  - 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

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

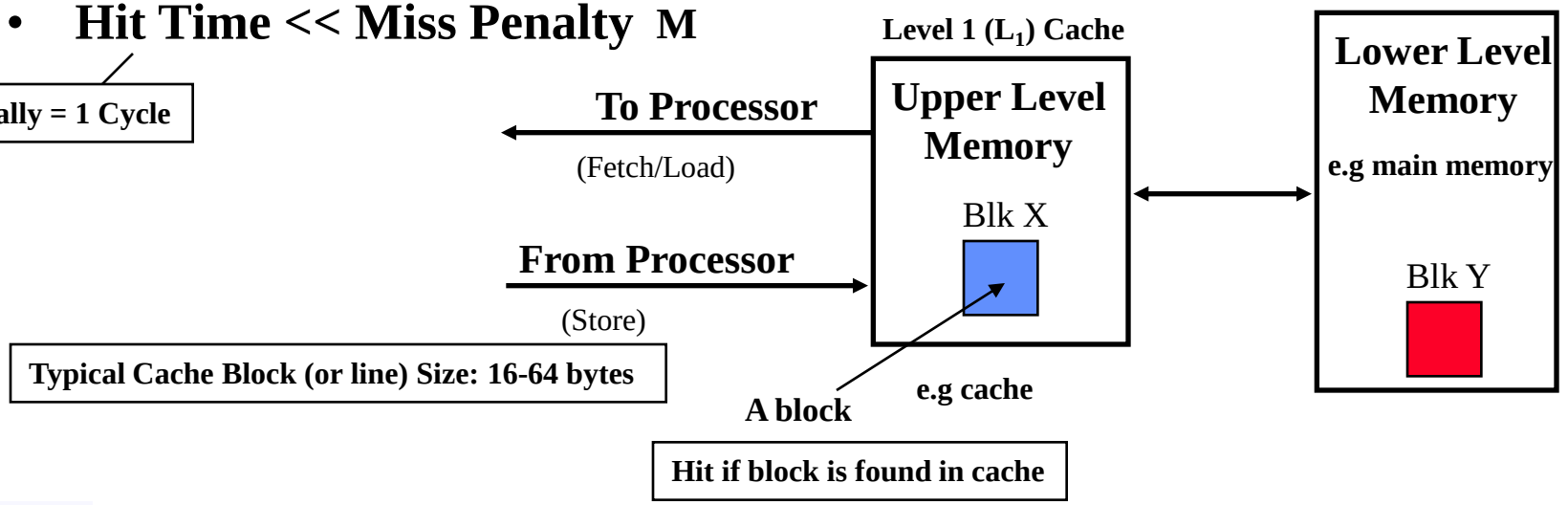
Ideally = 1 Cycle

e. g. 1- H1

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

M

Ideally = 1 Cycle



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

Hit if block is found in cache

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*.**

Basic Cache Design & Operation Issues

- **Q1: Where can a block be placed cache?**
  (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?**
  (Block identification)
  - **Tag/Block.**

Cache Hit/Miss?

- **Q3: Which block should be replaced on a miss?**
  (Block replacement)
  - **Random, LRU, FIFO.**

- **Q4: What happens on a write?**
  (Cache write policy)
  - **Write through, write back.**

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:**

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

Least complex to implement
suffers from conflict misses

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:**

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

Cache Organization:
Direct Mapped Cache

V	Tag	Data

Cache Block Frame

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

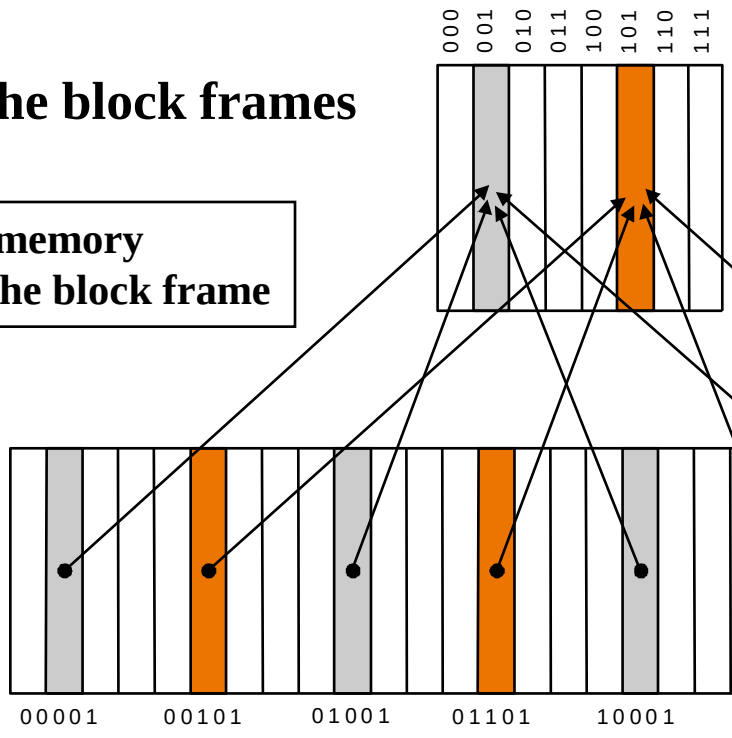
(Block address) MOD (Number of blocks in cache)

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

4KB Direct Mapped Cache Example

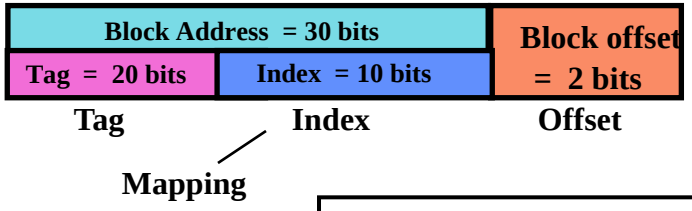
1K = 2^{10} = 1024 Blocks
Each block = one word
(4 bytes)

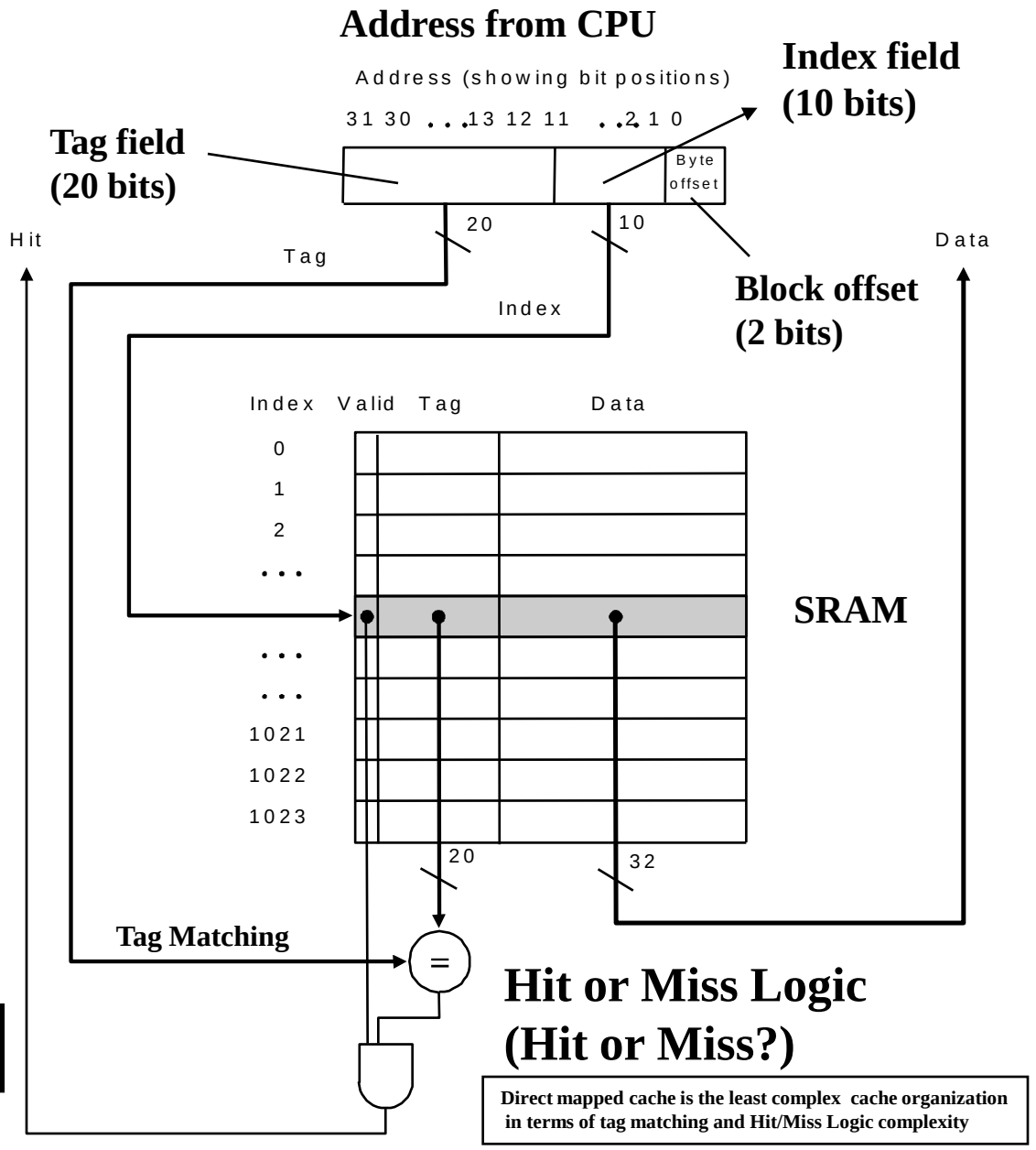
Can cache up to
2^{32} bytes = 4 GB
of memory

Mapping function:

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

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

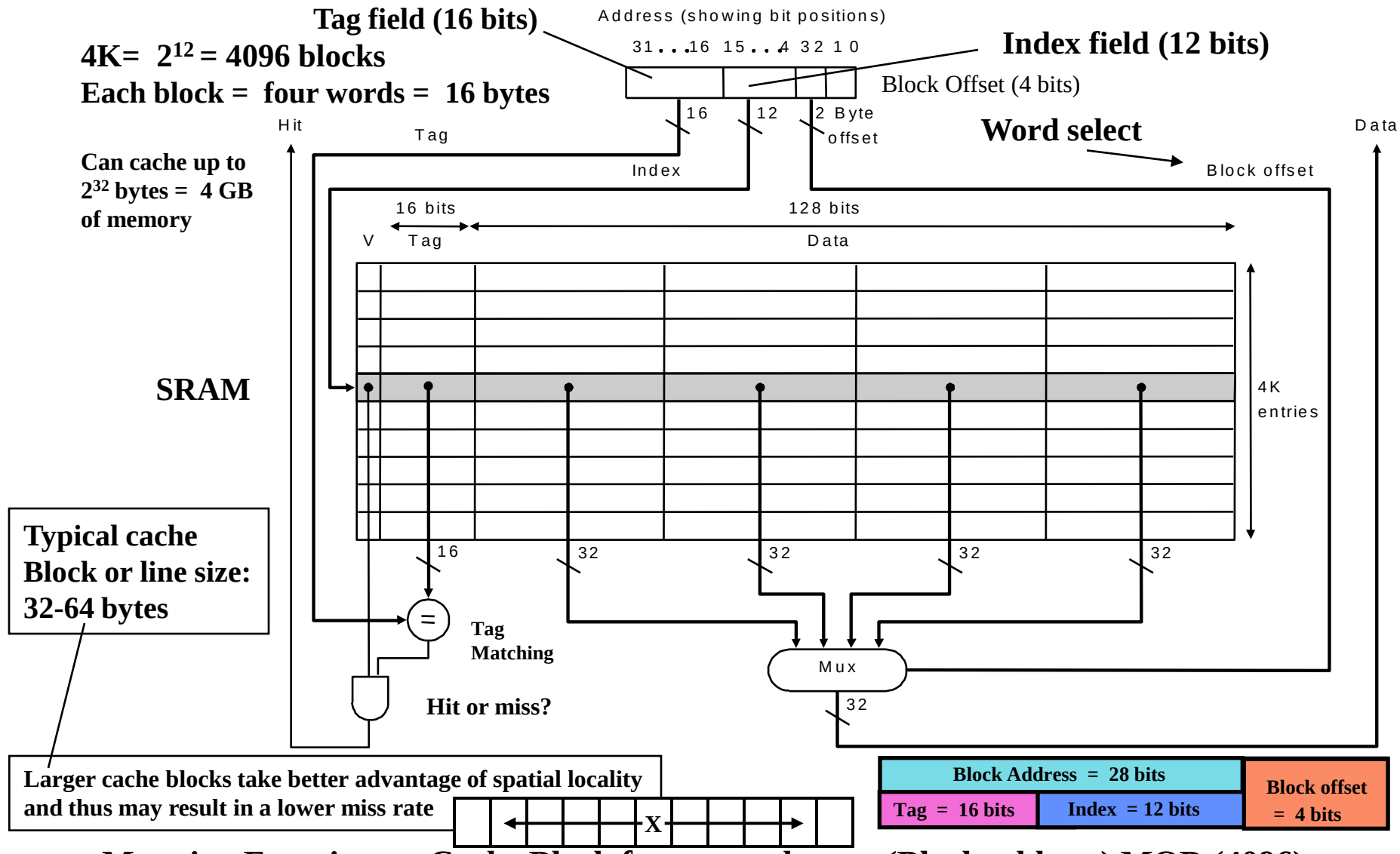




Direct mapped cache is the least complex cache organization in terms of tag matching and Hit/Miss Logic complexity

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

64KB Direct Mapped Cache Example

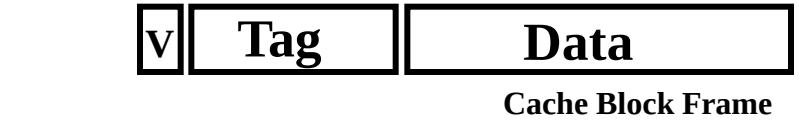


Mapping Function: Cache Block frame number = (Block address) MOD (4096)

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

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

Cache Organization: Set Associative Cache



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

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

2-way set associative:
2 blocks frames per set

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

4-way set associative:
4 blocks frames per set

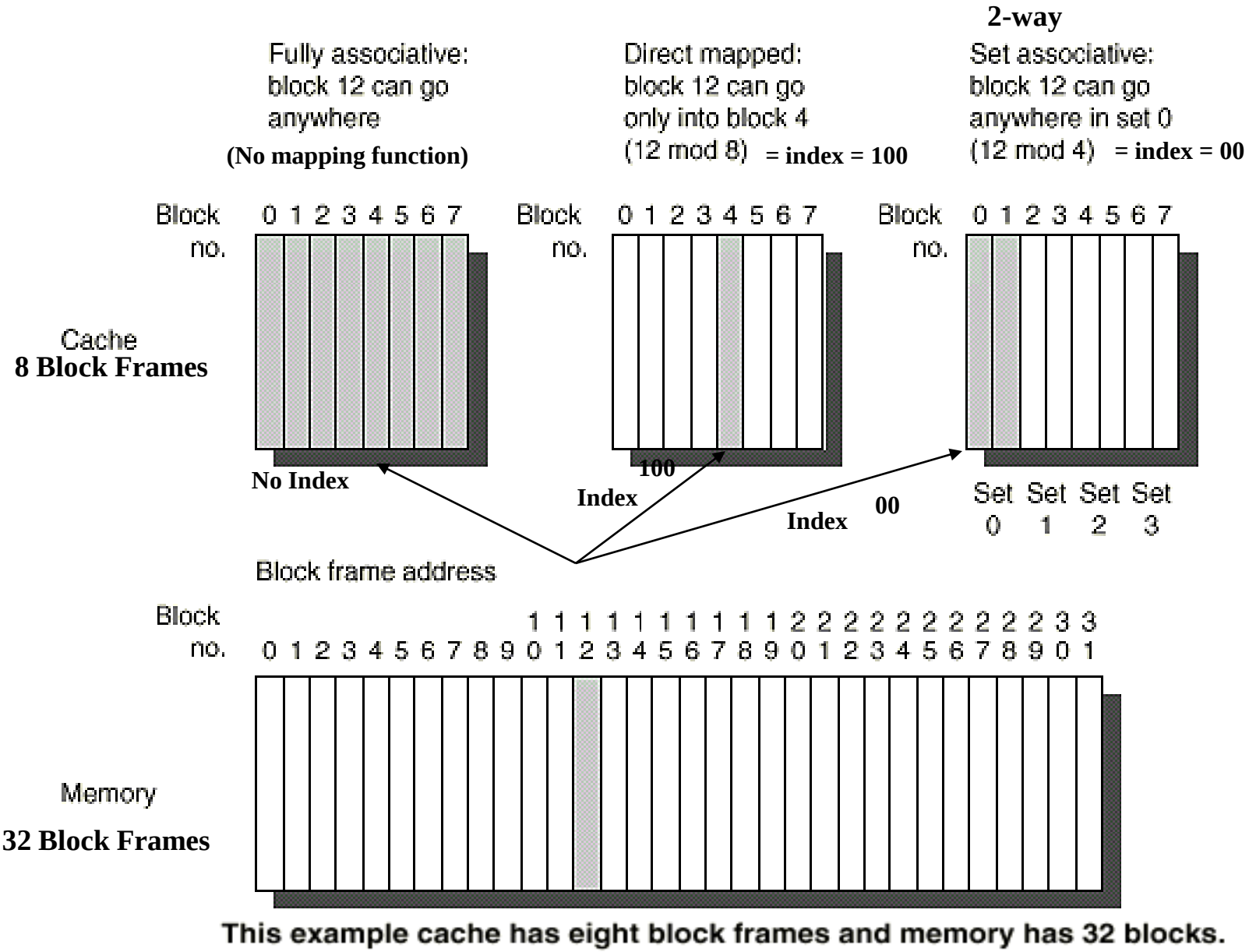
Eight-way set associative (fully associative)

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

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

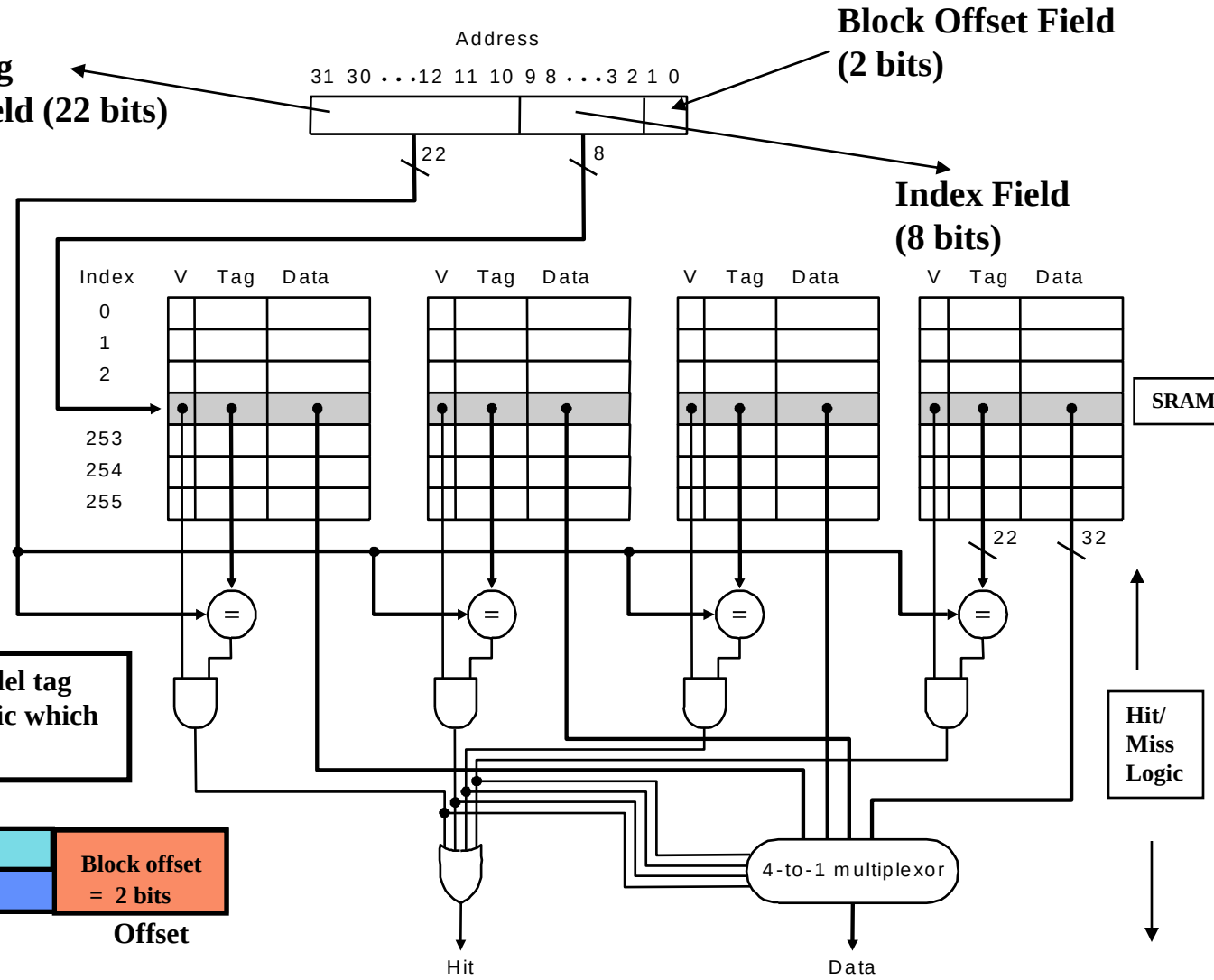
Cache Organization/Mapping Example



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

12 = 1100

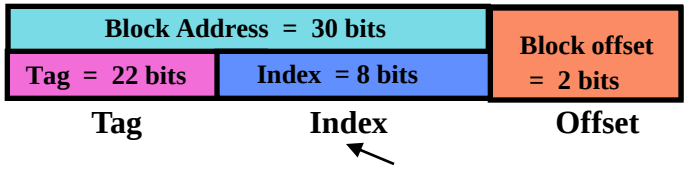
4K Four-Way Set Associative Cache: MIPS Implementation Example



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



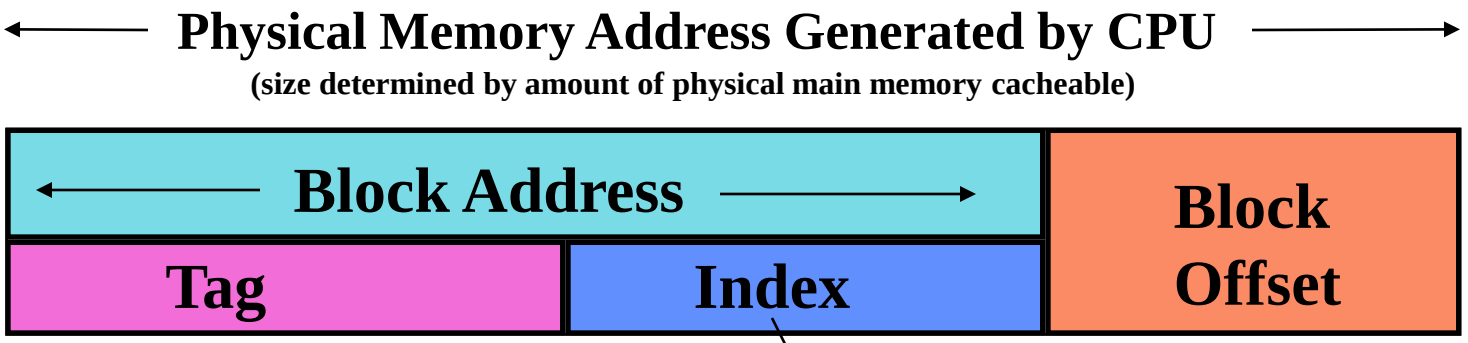
Mapping Function: Cache Set Number = index= (Block address) MOD (256)

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

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



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

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

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

Unified vs. Separate Level 1 Cache

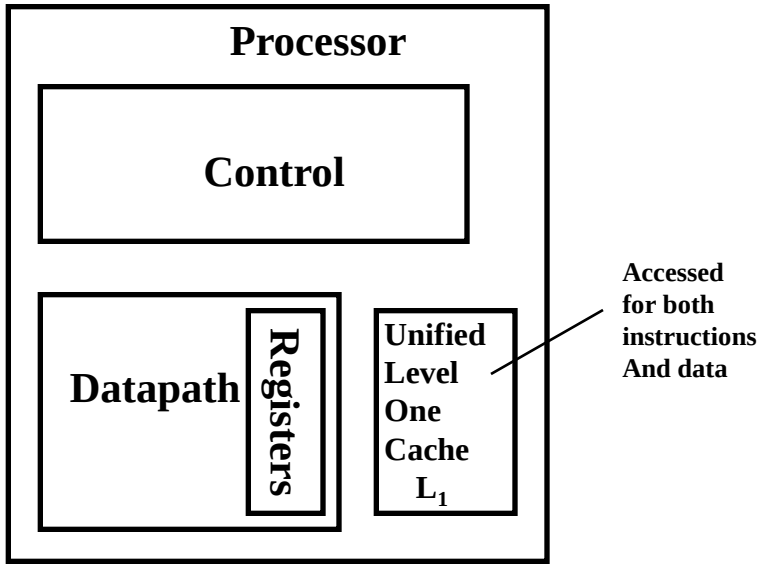
- **Unified Level 1 Cache (Princeton Memory Architecture).** AKA Shared Cache

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

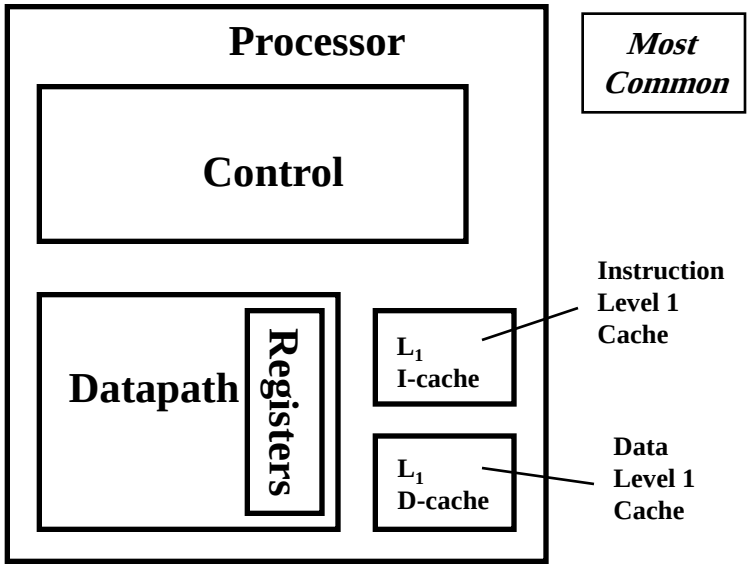
Or Split

- **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).



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

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.
- **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

cycles = CPU cycles

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

(Ignoring Write Policy)

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:

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

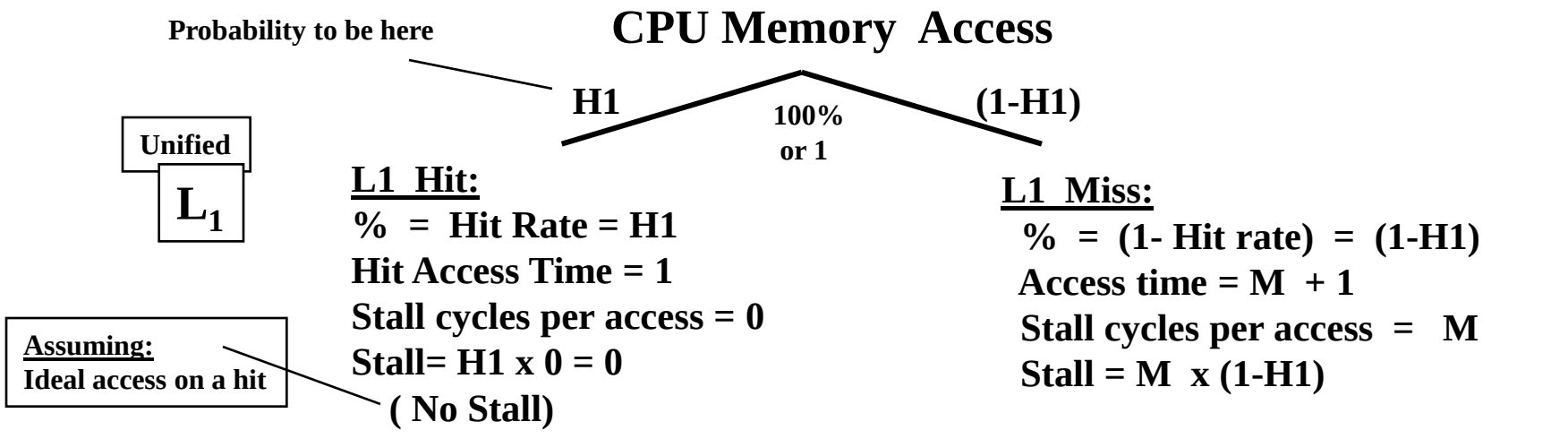
$$AMAT = 1 + (1 - H1) \times M$$

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

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

Memory Access Tree: For Unified Level 1 Cache

(Ignoring Write Policy)



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 + fraction of loads/stores) x M x (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 = $(CPI_{old} \times C_{old})/ (CPI_{new} \times C_{new})$

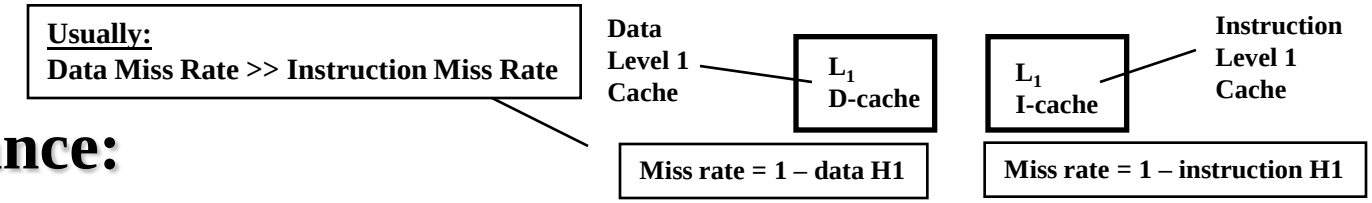
= 2.075 x 2 / 3.05 = 1.36

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.***

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:

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

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

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

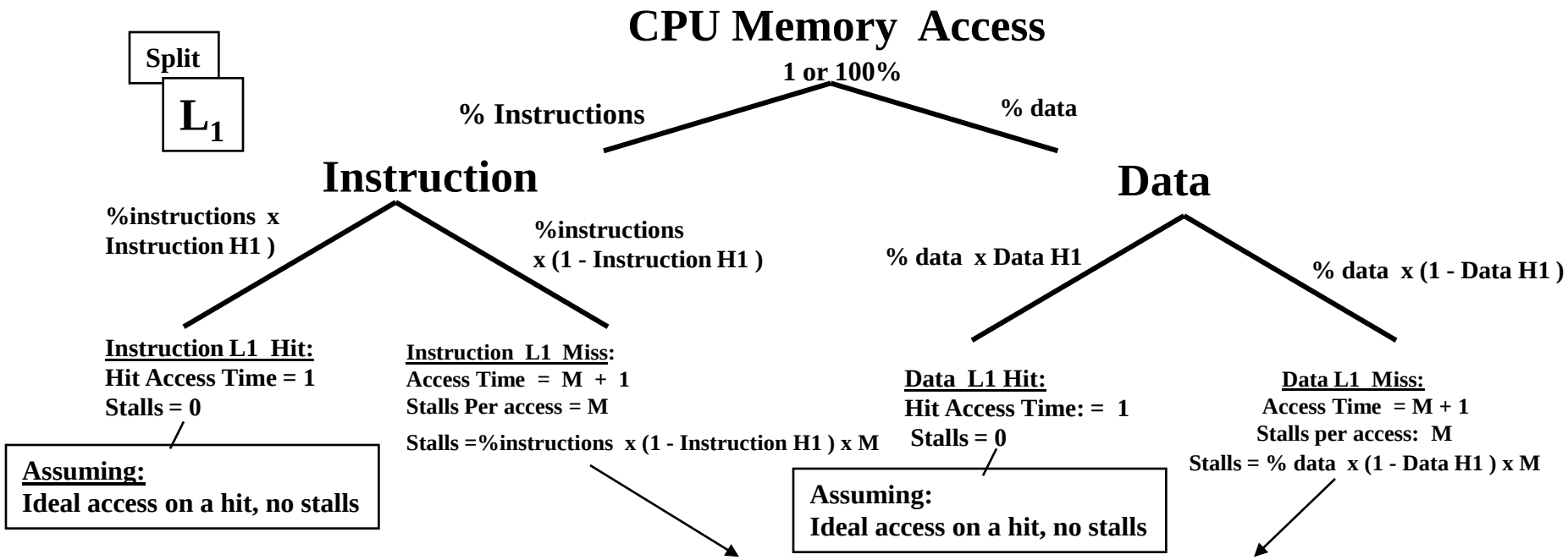
1- Data H1

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

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?**

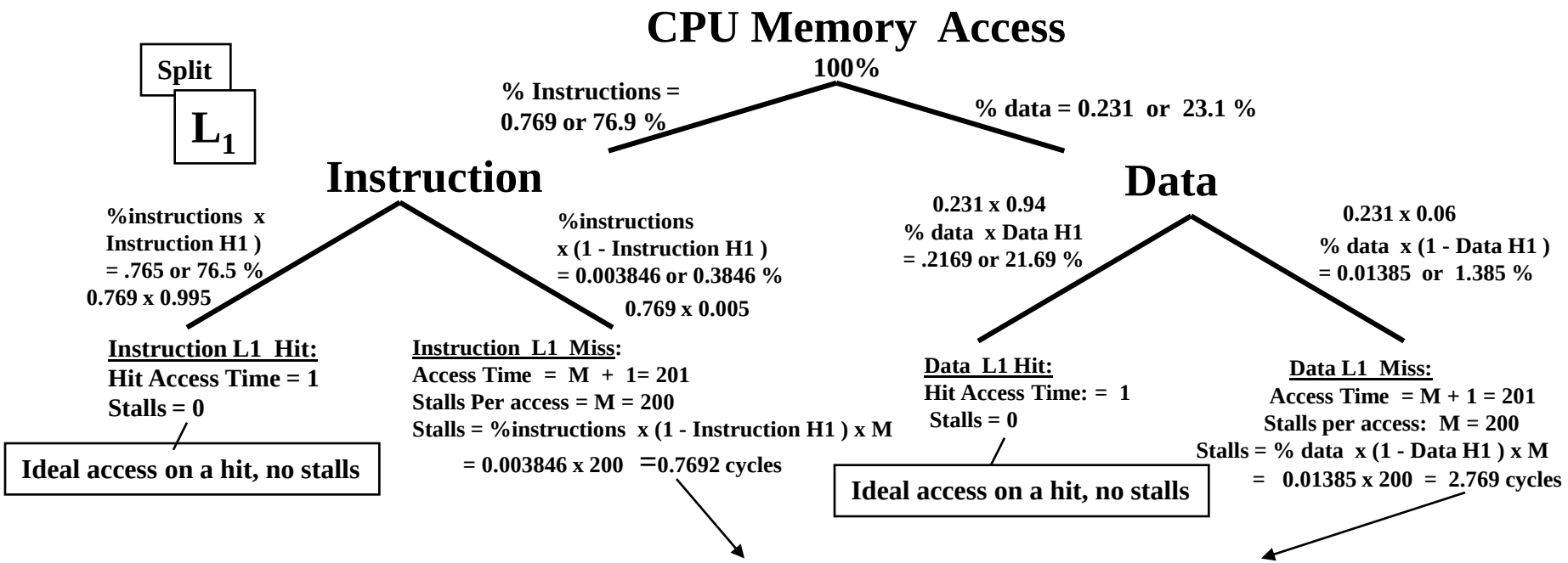
The CPU with ideal cache (no misses) is 5.7/1.1 = 5.18 times faster
With no cache at all the CPI would have been = 1.1 + 1.3 X 200 = 261.1 cycles !!

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 %

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%

Cache Read/Write Operations

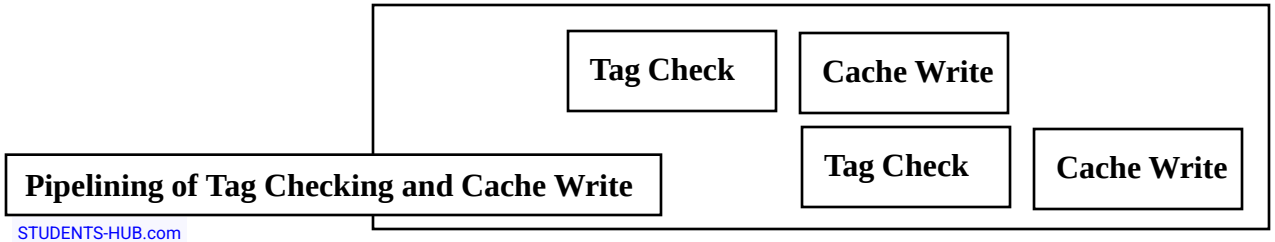
- Statistical data suggest that reads (*including instruction fetches)* dominate processor cache accesses (writes account for ~ 25% of data cache traffic).

  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.



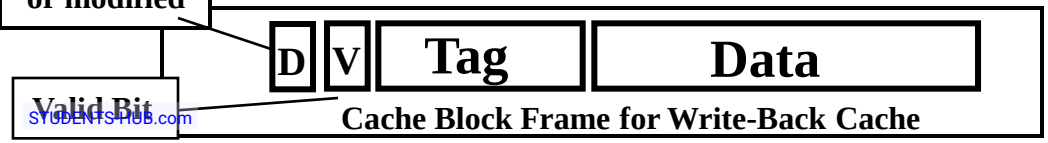
Cache Write Strategies

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

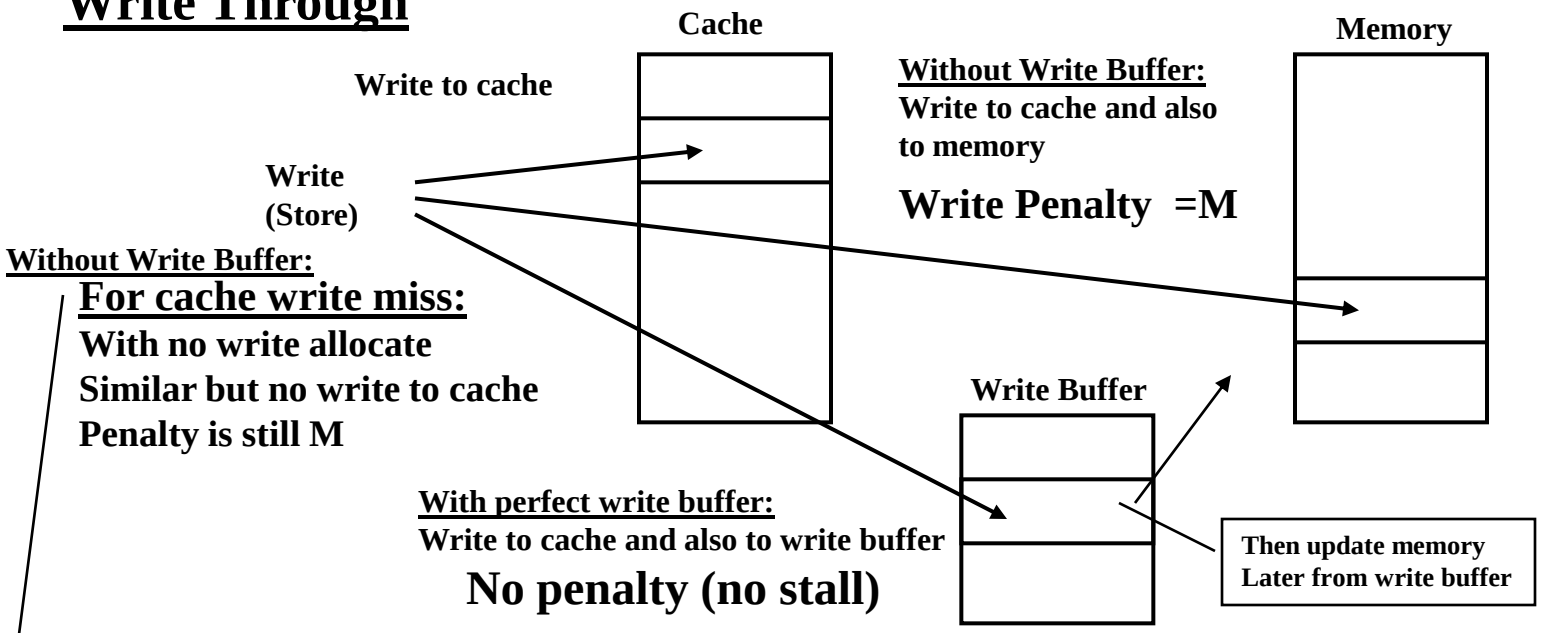


Cache Write Strategies:

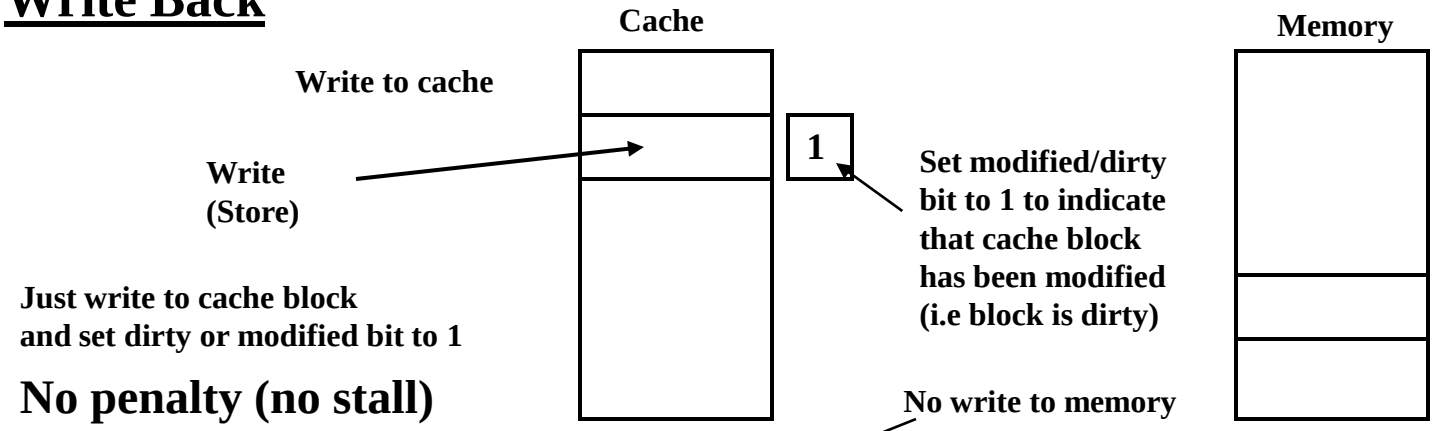
Cache Write Hit = block to be modified is found in cache

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

Write Through



Write Back



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

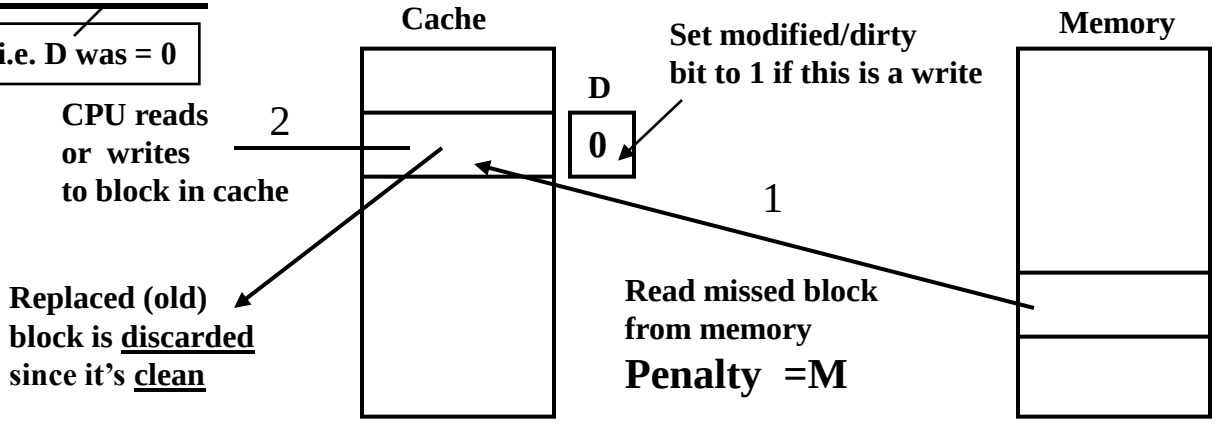
Write Back Cache With Write Allocate: Cache Miss Operation

(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)

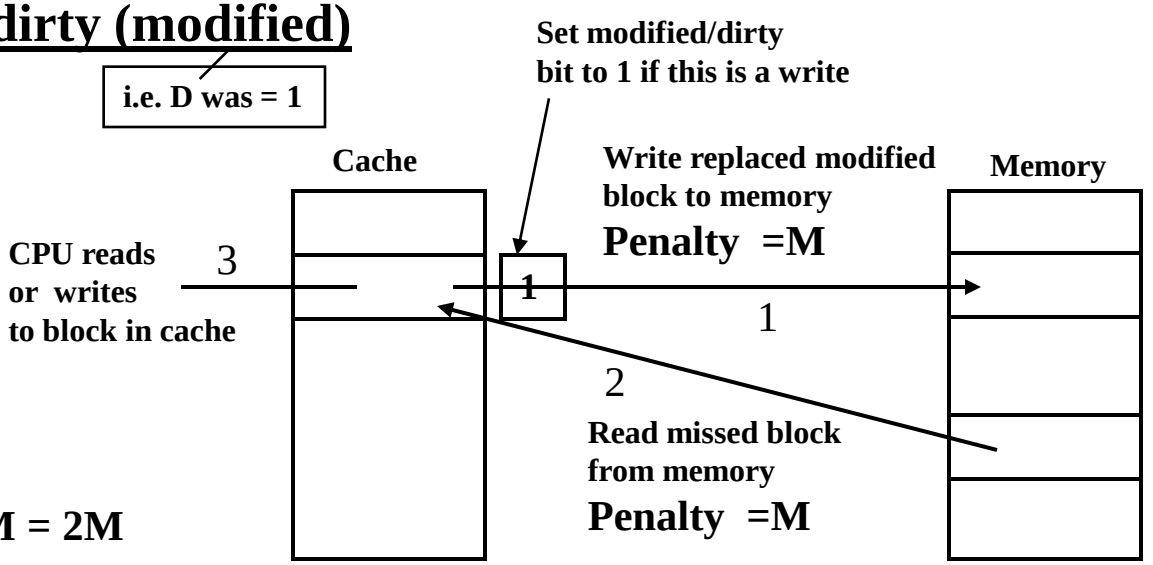
i.e. D was = 1

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

2 Read missed block from memory
Penalty =M

Thus:

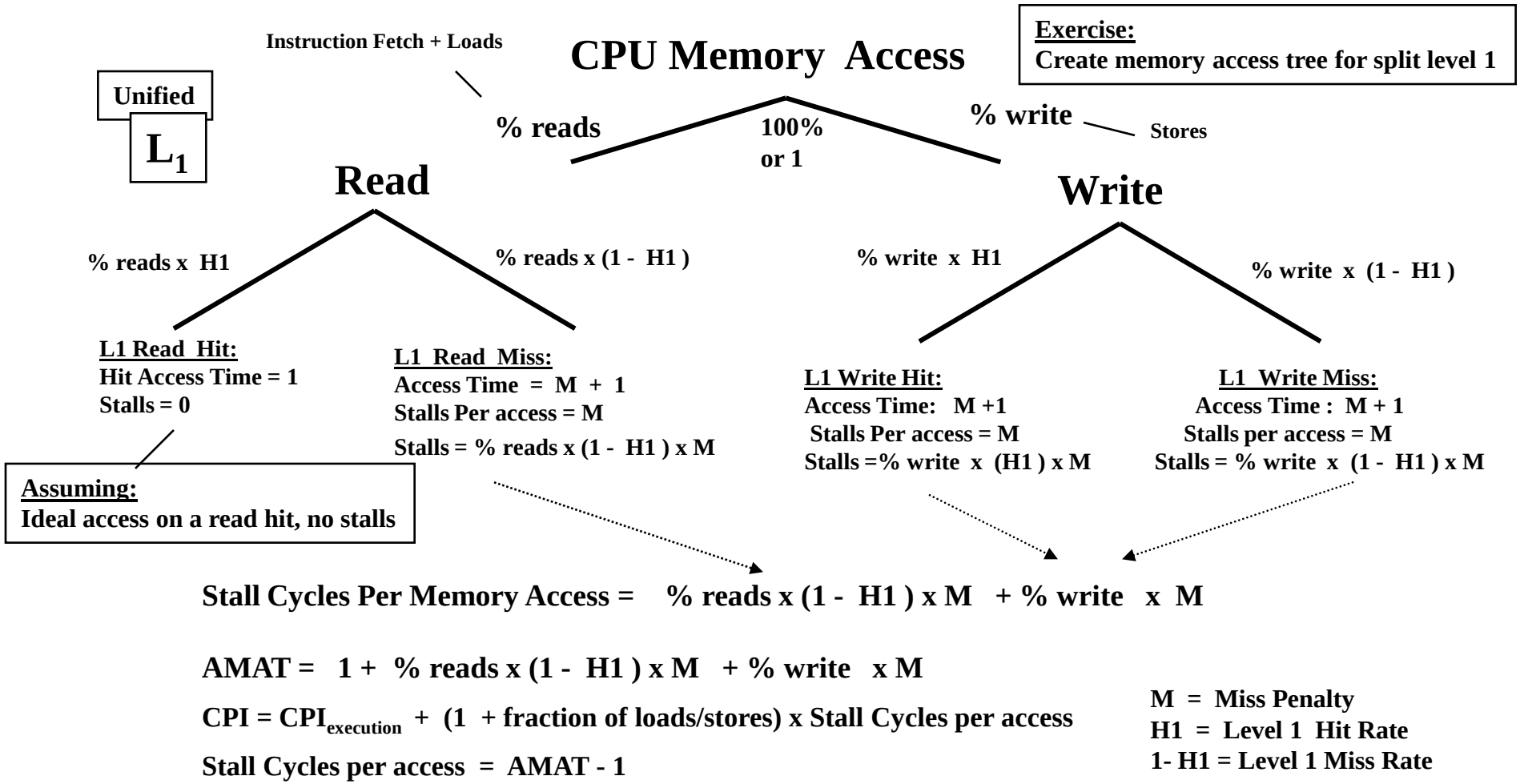
Total Miss Penalty = M + M = 2M



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



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

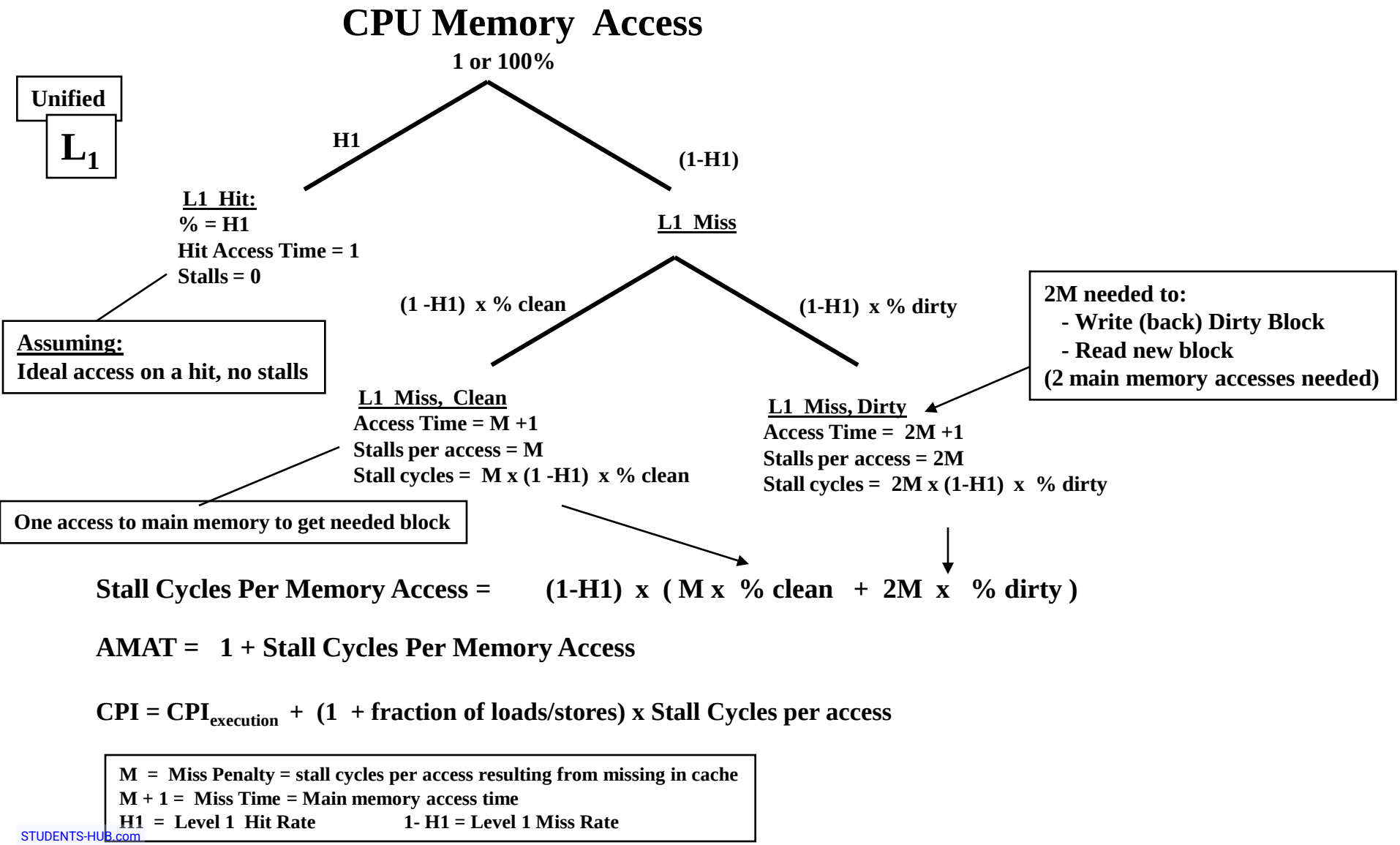
CPI = 1.1 + 0.86 = 1.96

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

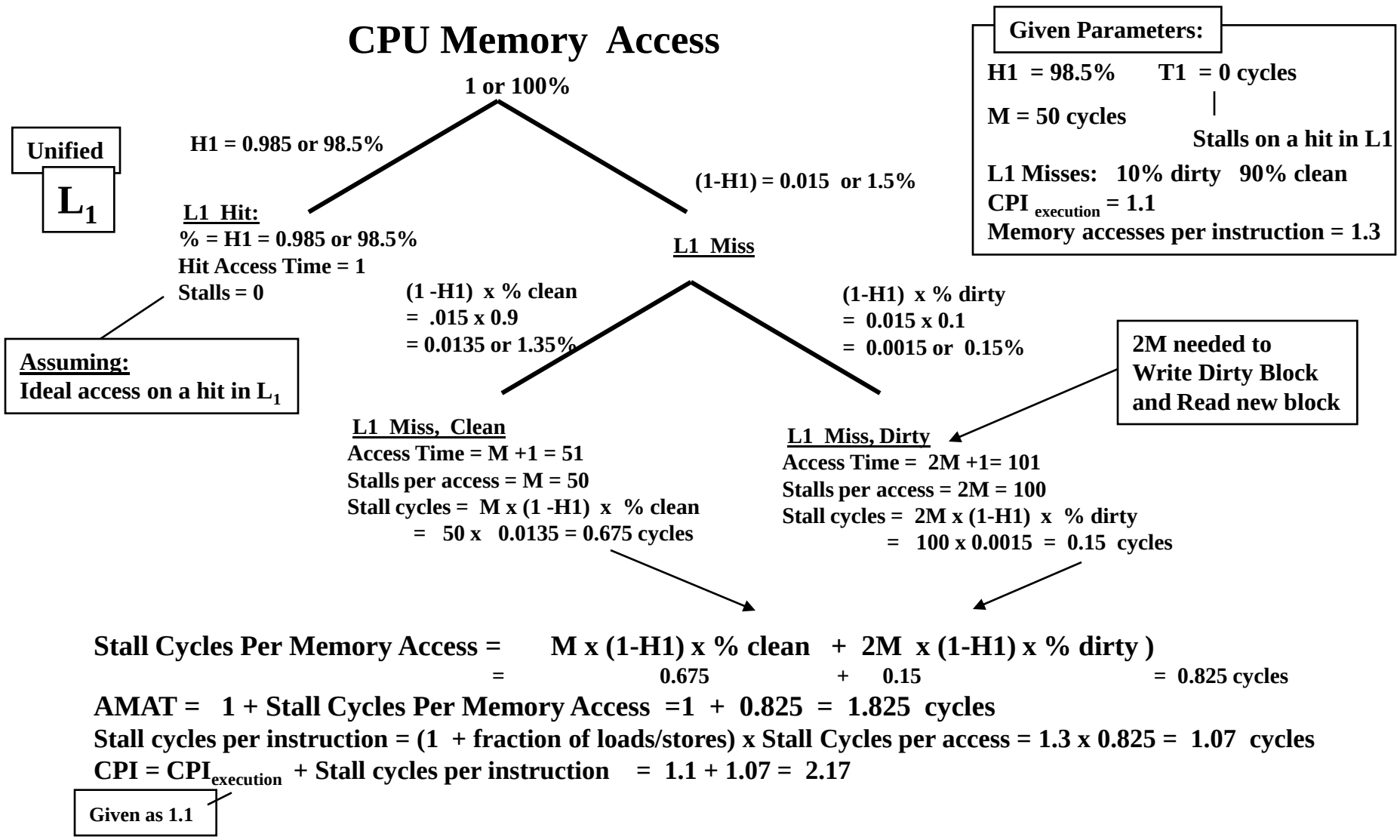
AMAT = 1 + stalls per access = 1 + 0.825 = 1.825 cycles

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

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



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

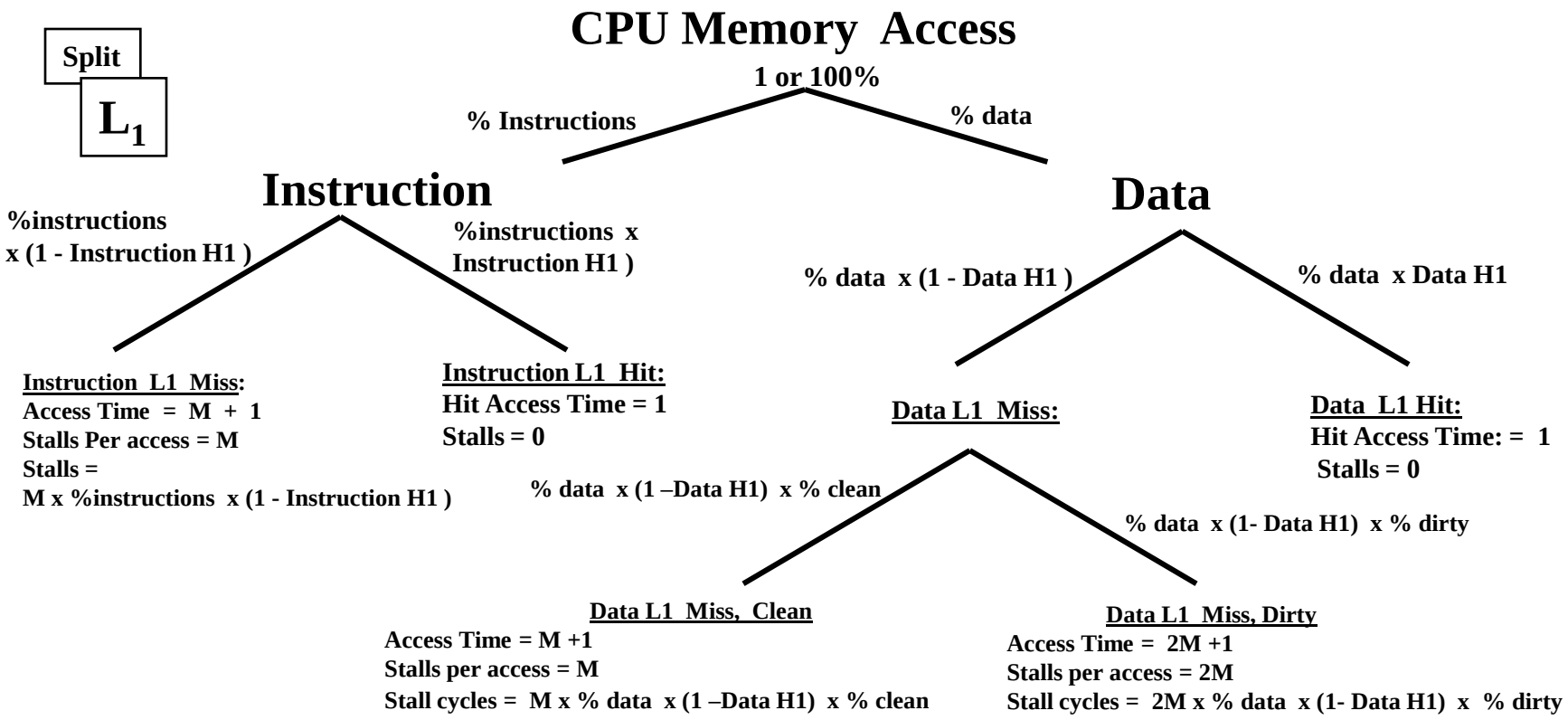
Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access = 1.3 x 0.825 = 1.07 cycles

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



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)

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 (5-8 cycles typical) But has more capacity and higher associativity

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

- **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)

i.e that reach this level

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

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

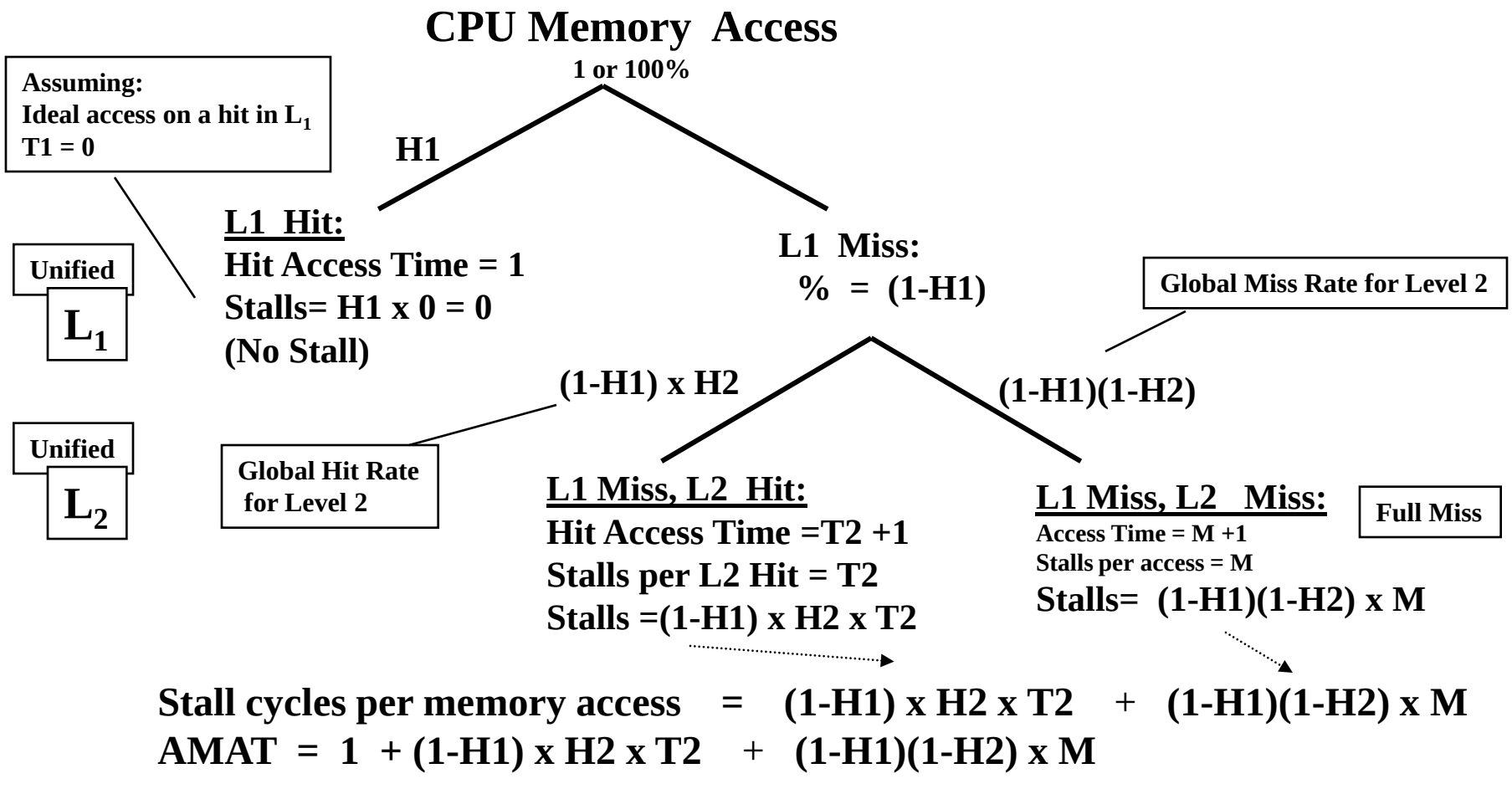
L1 Miss, L2 Miss: Must Access Main Memory

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

- 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 ...**

(Ignoring Write Policy)

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

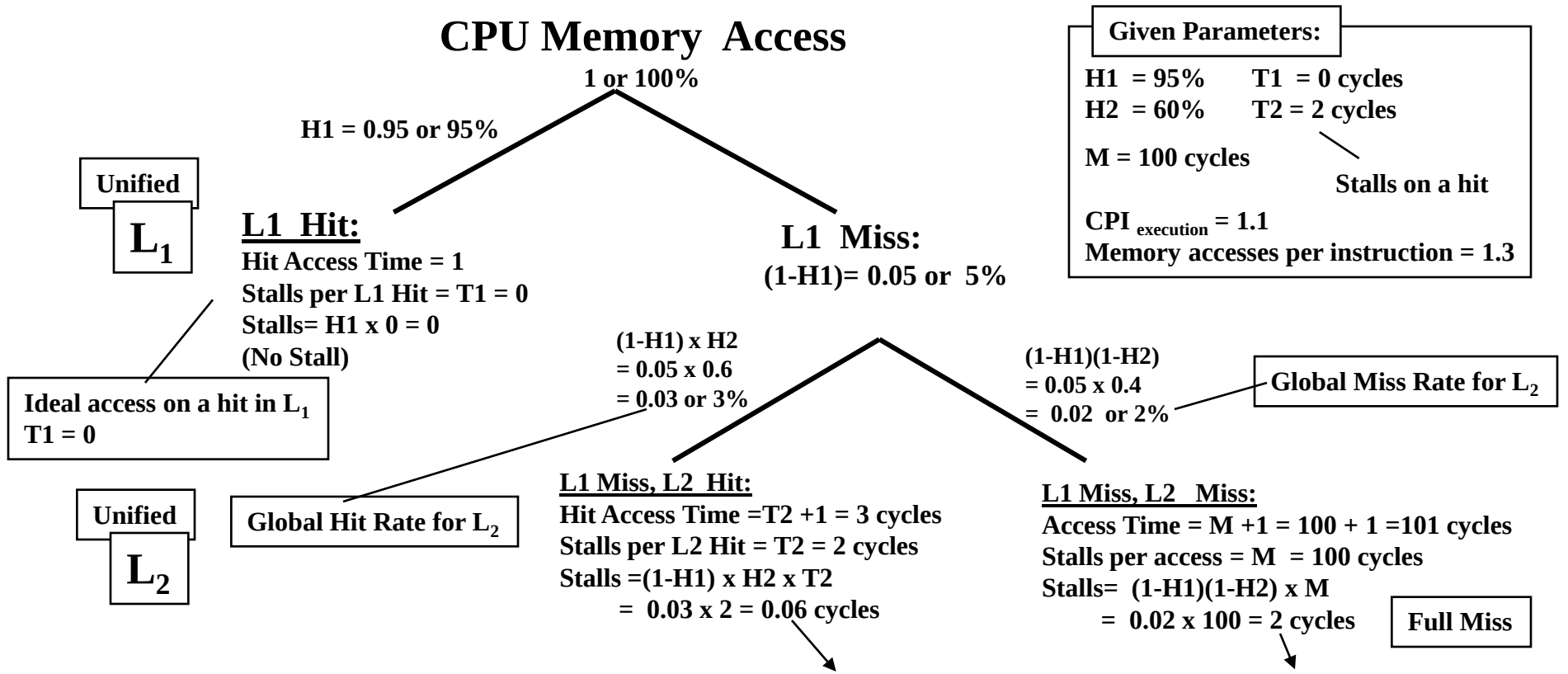
CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access

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

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

CPU Stall Cycles Per Memory Access

(Ignoring Write Policy)



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

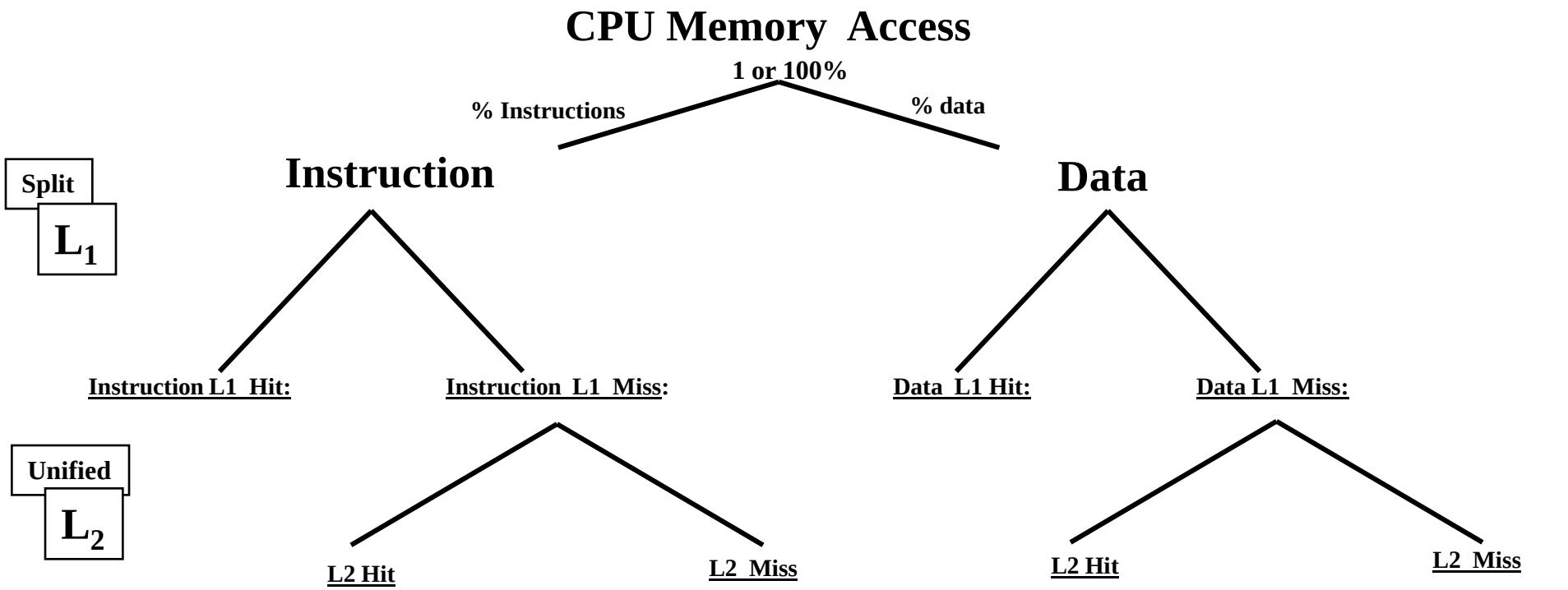
$CPI = CPI_{execution}$ + Stall cycles per instruction = 1.1 + 2.678 = 3.778

$CPI = CPI_{execution}$ + (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 = L2 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.

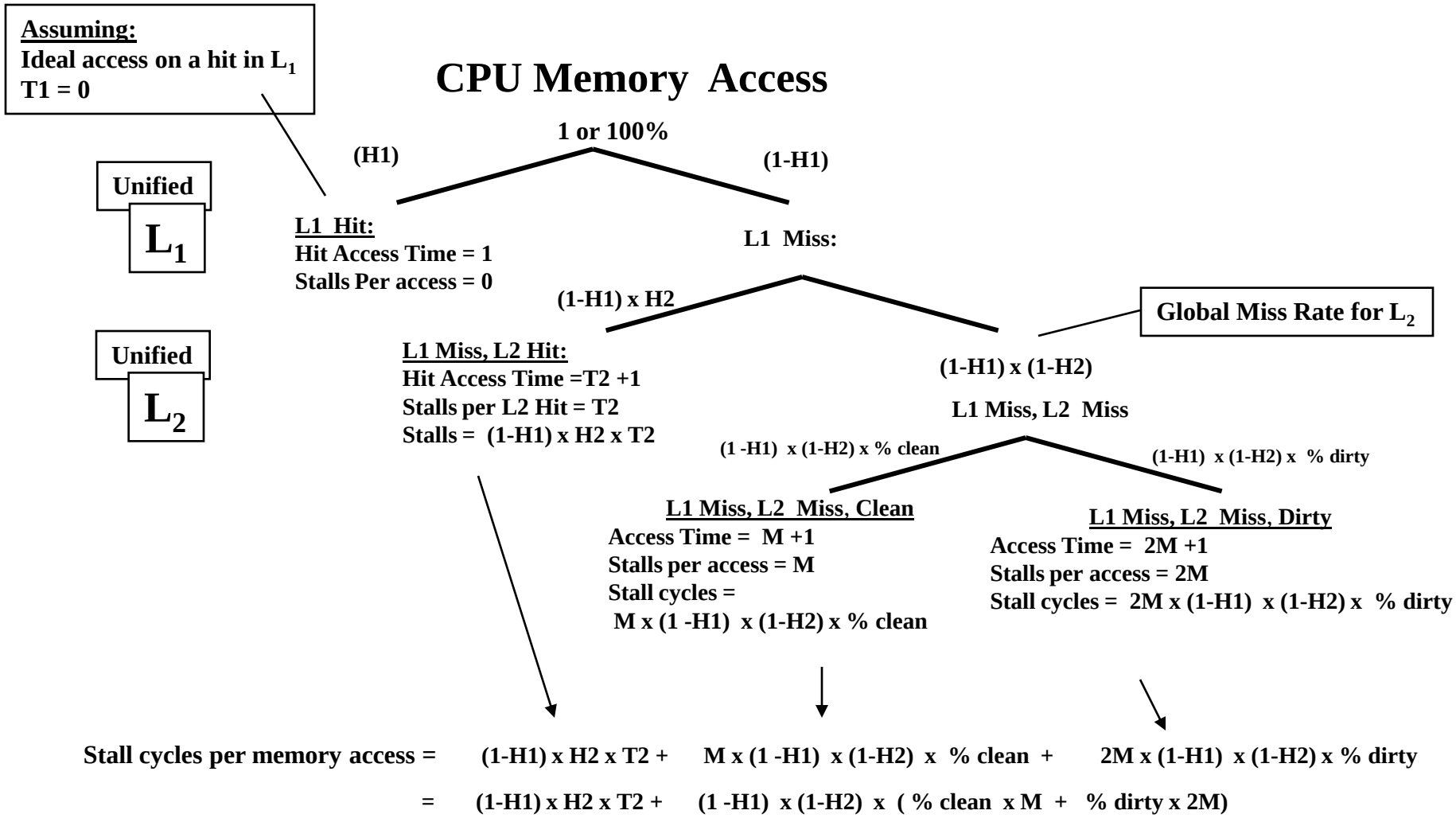
- The above 2-level cache write policy results in inclusive L2 cache since the content of L1 is also in L2
  - Common in the majority of all CPUs with 2-levels of cache
  - As opposed to exclusive L1, L2 (e.g AMD Athlon XP, A64)

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

i.e what is in L1 is not duplicated in 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



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

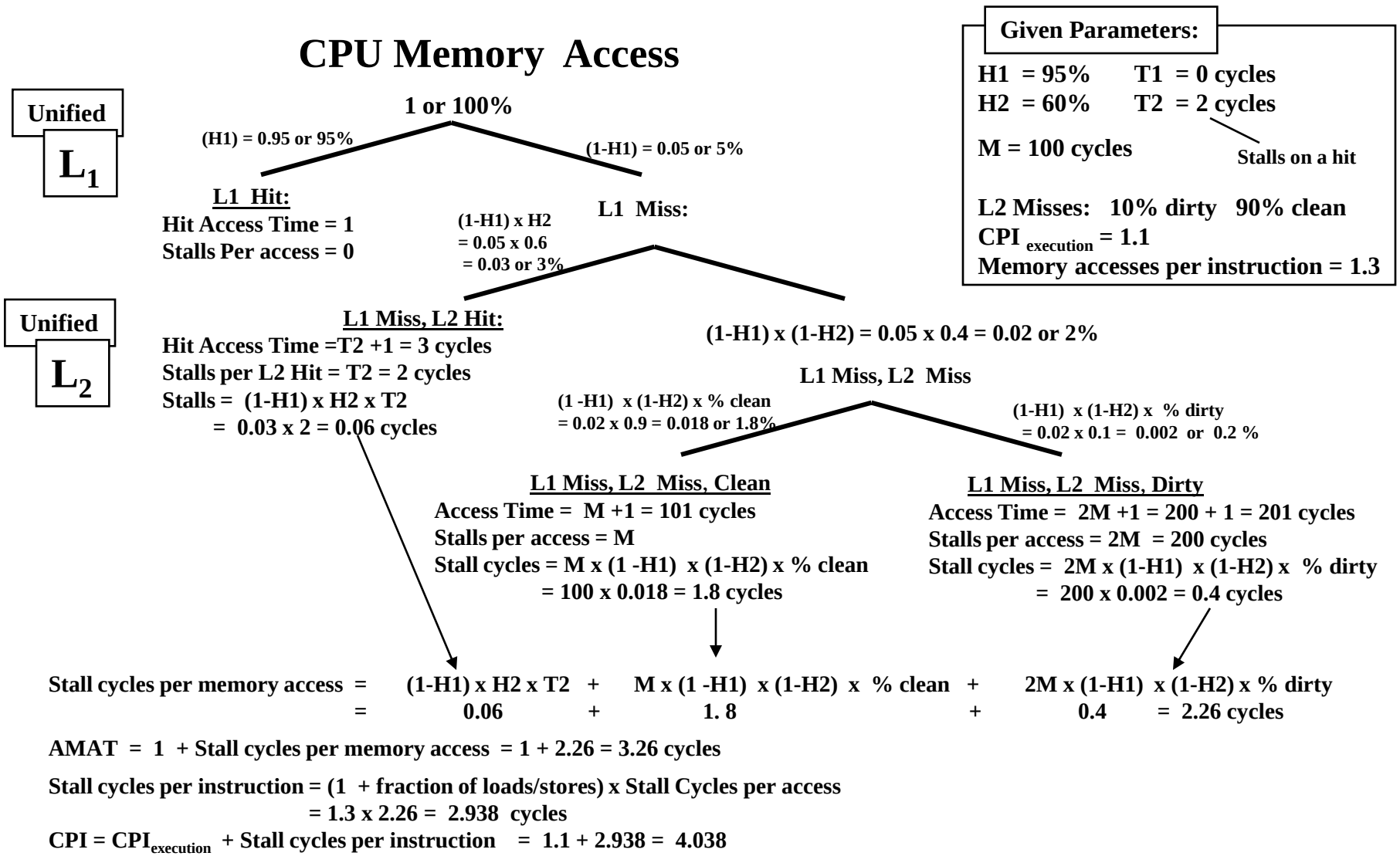
= 2.26 x 1.3 = 2.938 cycles

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

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



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

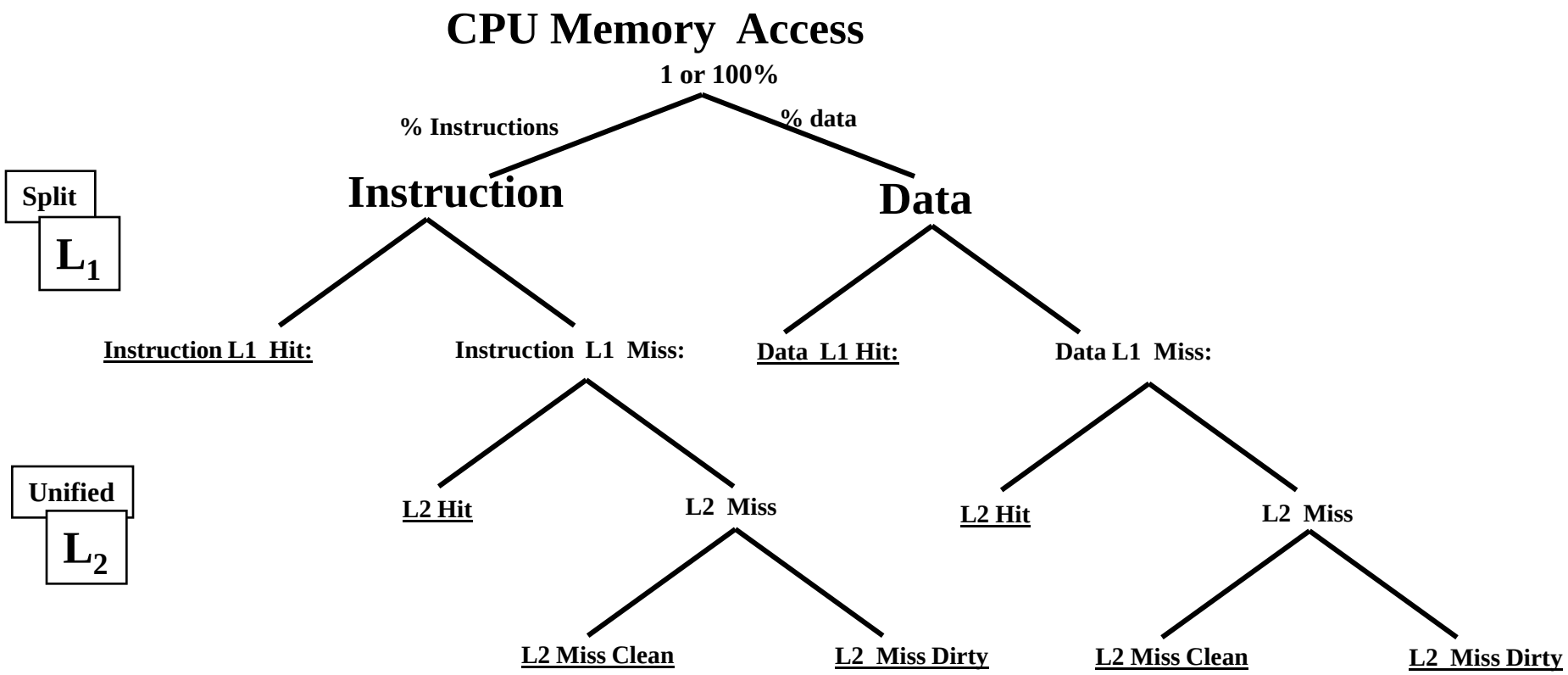
CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 2.938 = 4.038

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

Memory Hierarchy—Ways to Reduce Misses

Review Cache Performance Equations

- **CPUtime** = (CPU execution cycles + Mem stall cycles) * Cycle time
- **Mem stall cycles** = Mem accesses * Miss rate * Miss penalty
- **CPUtime** = IC * (CPI_{exe} + Mem accesses per instr * Miss rate * Miss penalty) * Cycle time
- **Misses per instr** = Mem accesses per instr * Miss rate
- **CPUtime** = IC * (CPI_{exe} + Misses per instr * Miss penalty) * Cycle time

Reducing Misses

- **Classifying Misses: 3 Cs**
  - *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*.
    (Misses in Infinite Cache)
  - *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.
    (Misses in Size)
  - *Conflict*—If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses* or *interference misses*.
    (Misses in N-way Associative)

Improving Cache Performance

1. Reduce the miss rate,

2. Reduce the miss penalty, or

3. Reduce the time to hit in the cache.

How to Reduce Miss Rate?

- **Increase Block Size**
- **Increase Associativity**
- **Use a Victim Cache**
- **Use a Pseudo Associative Cache**
- **Hardware Prefetching**
- **Compiler-Controlled Prefetching**
- **Compiler Optimizations**

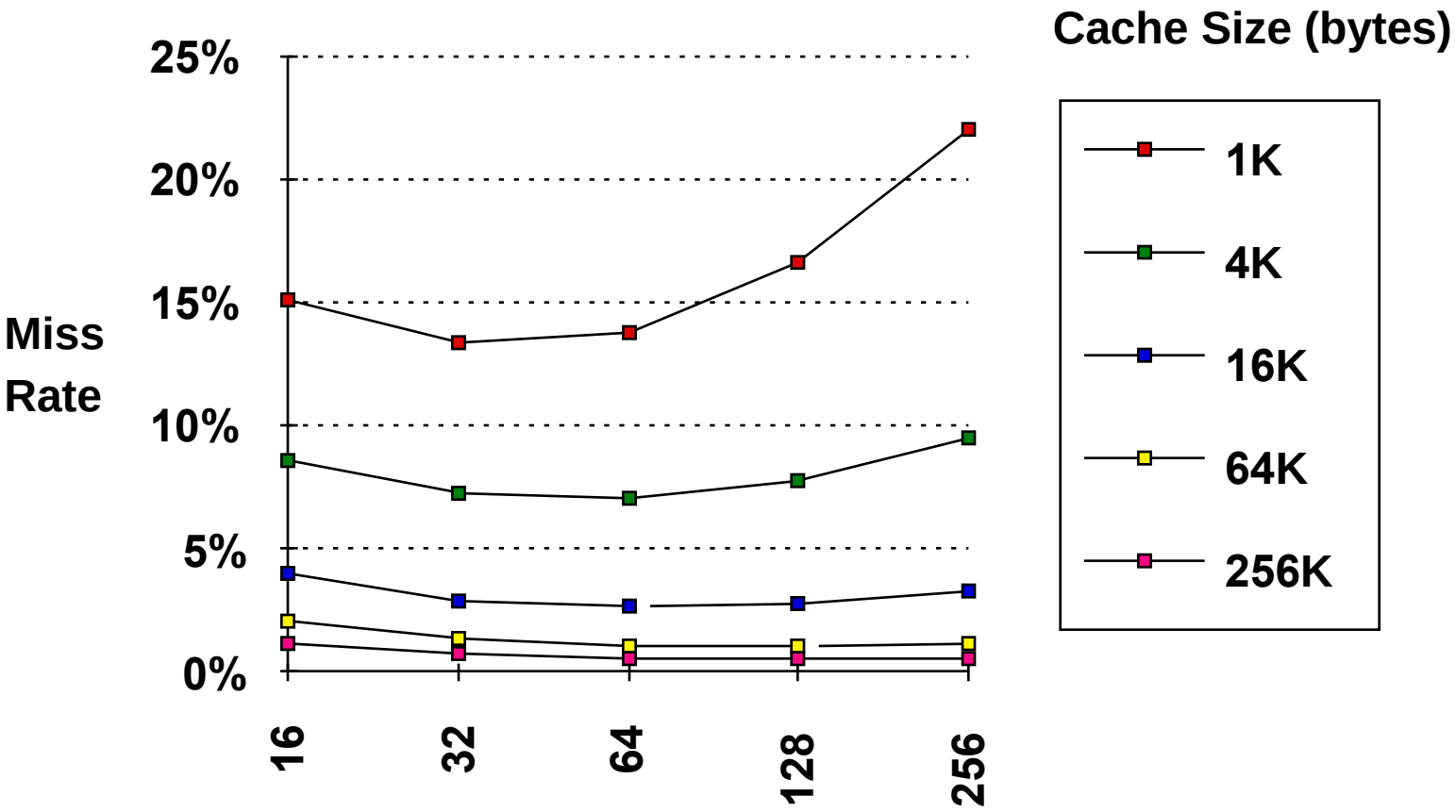
1. Increase cache Block Size

- **One way to reduce the miss rate is to increase the block size**
  - Reduce compulsory misses - why?
    - **Take advantage of spacial locality**
      - **Helps improve miss rate b/c of principle of locality:**
      - **Temporal locality says that if something is accessed once, it'll probably be accessed again soon**
      - **Spatial locality says that if something is accessed, something nearby it will probably be accessed**
        - Larger block sizes help with spatial locality

- **However, larger blocks have disadvantages**
  - May increase the miss penalty (need to get more data)
    - **Generally, larger blocks reduce # of total blocks in cache**
  - May increase hit time (need to read more data from cache and larger mux)
  - May increase conflict and capacity misses

- **Increasing the block size can help, but don't overdo it.**

1. Increase cache Block Size (cont.)



Example Increase cache Block Size (1/3)

- **Assume that to access lower-level of memory hierarchy you:**
  - Incur a 40 clock cycle overhead
  - Get 16 bytes of data every 2 clock cycles
- **I.e. get 16 bytes in 42 clock cycles, 32 in 44, etc…**
- **Using data below, which block size has minimum average memory access time?**

Cache sizes

Block Size	1K	4K	16K	64K	256K
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

Miss rates

Example Increase cache Block Size (3/3)

- Assume a cache hit otherwise takes 1 clock cycle –independent of block size
- So, for a 16-byte block in a 1-KB cache…

  Average memory access time =

  1 + (15.05% X 42) = 7.321 clock cycles
- And for a 256-byte block in a 256-KB cache…

  Average memory access time =

  1 + (0.49% X 72) = 1.353 clock cycles

Cache sizes

Block Size	Miss Penalty	1K	4K	16K	64K	256K
16	42	7.321	4.599	2.655	1.857	1.485
32	44	6.870	4.186	2.263	1.594	1.308
64	48	7.605	4.360	2.267	1.509	1.245
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

Red entries are lowest average time for a particular configuration

Note: All of these block sizes are common in processor's today

Note: Data for cache sizes in units of "clock cycles"

Summary Increase cache Block Size

- **We want to minimize cache miss rate & cache miss penalty at same time!**

- **Selection of block size depends on latency and bandwidth of lower-level memory:**
  - High latency, high bandwidth encourage large block size
    - **Cache gets many more bytes per miss for a small increase in miss penalty**
  - Low latency, low bandwidth encourage small block size
    - **Twice the miss penalty of a small block may be close to the penalty of a block twice the size**
    - **Larger # of small blocks may reduce conflict misses**

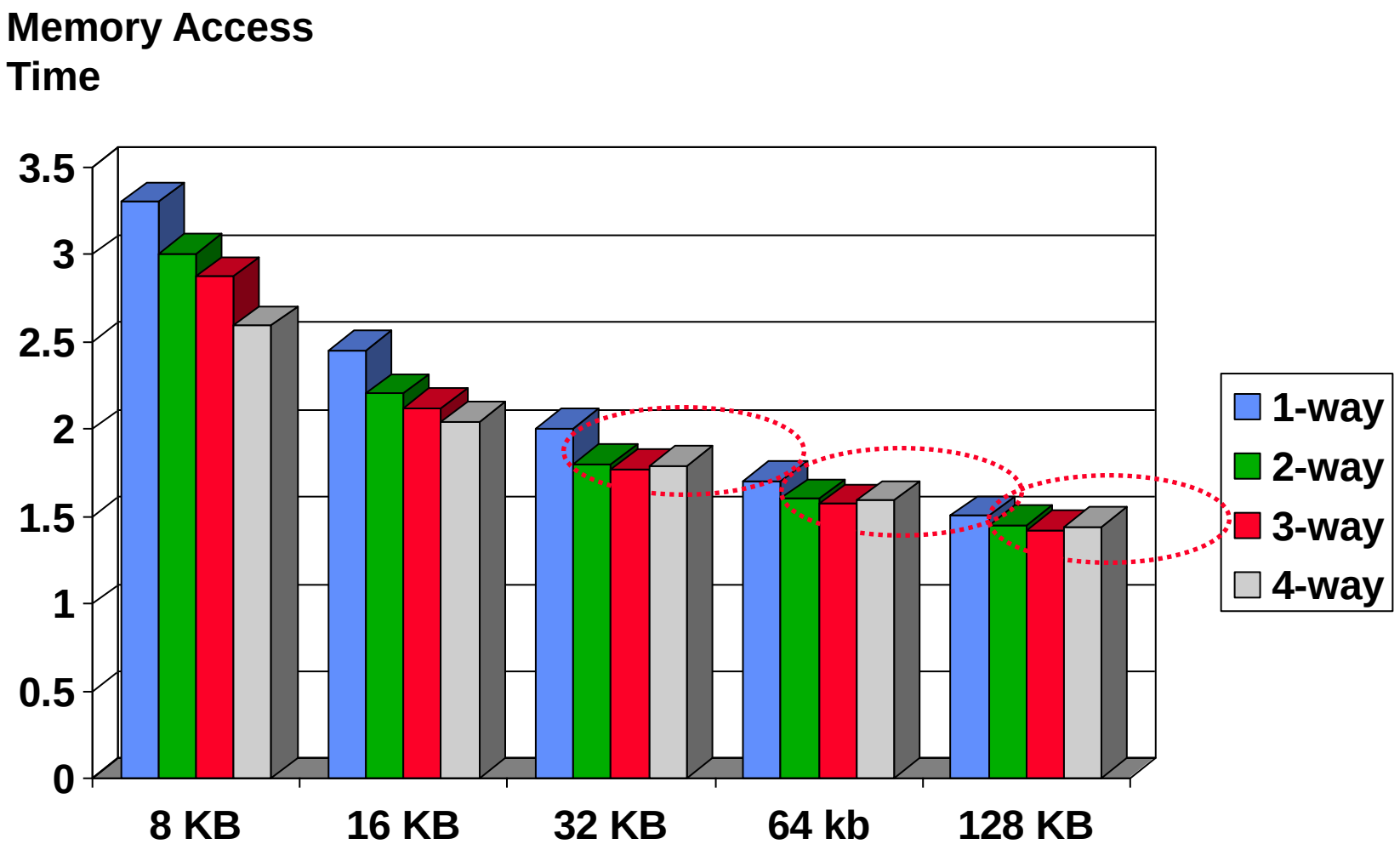
2. Reduce Misses via Higher Associativity

- **Increasing associativity helps reduce conflict misses**
- **2:1 Cache Rule:**
  - The miss rate of a direct mapped cache of size N is about equal to the miss rate of a 2-way set associative cache of size N/2
- **Disadvantages of higher associativity**
  - Need to do large number of comparisons
  - Need n-to-1 multiplexer for n-way set associative
  - Could increase hit time

Cache Size (KB)	Associativity 1-way	2-way	4-way	8-way
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	1.79
64	1.70	1.60	1.57	1.59
128	1.50	1.45	1.42	1.44

(Red means A.M.A.T. not improved by more associativity)

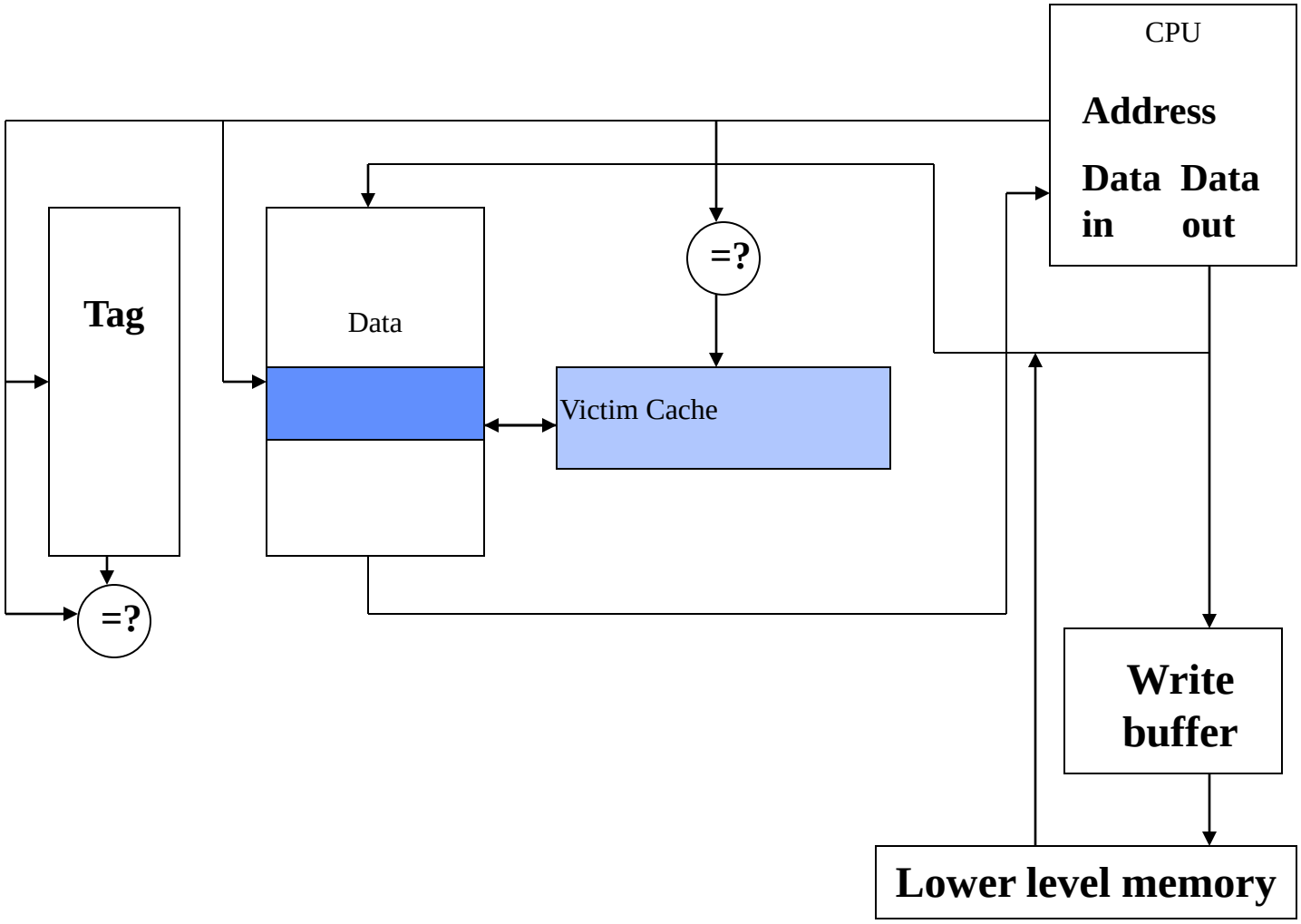
Avg. Memory Access vs Cache Size vs Associativity



3. Reducing Misses via Victim Cache

- **What is a "victim cache"?**
  - A victim cache temporarily stores blocks that have been discarded from the main cache (usually not that big)
- **How does it help us?**
  - If there's a cache miss, instead of immediately going down to the next level of memory hierarchy we check the victim cache first
  - If the entry is there, we swap the victim cache block with the actual cache block
- **Research shows:**
  - Victim caches with 1-5 entries help reduce conflict misses
  - For a 4KB direct mapped cache victim caches:
    - **Removed 20% - 95% of conflict misses!**

3. Reducing Misses via Victim Cache



4. Reducing Misses via Pseudo-Associativity

- **This techniques should help:**
  - The miss rate of set-associative caches
  - The hit speed of direct mapped caches
- **Also called: "column associated cache"**
- **Access proceeds normally as for a direct mapped cache**
  - But, on a miss, we look at another entry before going to a lower level of memory hierarchy
  - Usually done by:
    - **Inverting the most significant bit of index field to find the other block in the psuedo-set**
- **Pseudo-associative caches usually have 1 fast and 1 slow hit time (regular, psuedo hit respectively)**
  - In addition to the miss penalty that is…

5. HW Prefetching of Instruction & Data

- **E.g., Instruction Prefetching**
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in stream buffer
  - On miss check stream buffer
- **Works with data blocks too**
- **Prefetching relies on extra memory bandwidth that can be used without penalty**

- **What is the effective miss rate for the Alpha using instruction prefetching?**
- **How much larger of an instruction cache would we need if the Alpha to match the average access time if prefetching was removed?**
  - Assume:
    - **It takes 1 extra clock cycle if the instruction misses the cache but is found in the prefetch buffer**
    - **The prefetch hit rate is 25%**
    - **Miss rate for 8-KB instruction cache is 1.10%**
    - **Hit time is 2 clock cycles**
    - **Miss penalty is 50 clock cycles**

5. HW Prefetching of Instruction & Data

- **We need a revised memory access time formula:**
  - Say: Average memory access timeprefetch =
    - **Hit time + miss rate * prefetch hit rate * 1 + miss rate * (1 – prefetch hit rate) * miss penalty**
- **Plugging in numbers to the above, we get:**
  - 2 + (1.10% * 25% * 1) + (1.10% * (1 – 25%) * 50) = 2.415
- **To find the miss rate with equivalent performance, we start with the original formula and solve for miss rate:**
  - Average memory access timeno prefetching =
    - **Hit time + miss rate * miss penalty**
  - Results in: (2.415 – 2) / 50 = 0.83%
- **Calculation suggests effective miss rate of prefetching with 8KB cache is 0.83%**
- **Actual miss rates for 16KB = 0.64% and 8KB = 1.10%**

6. Reducing Misses by Compiler Optimizations

- **Instructions**
  - Reorder procedures in memory so as to reduce misses
  - Profiling to look at conflicts between groups of instructions
    - **McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache with 4 byte blocks**

- **Data**
  - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays (prevents the same indicies being used)
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

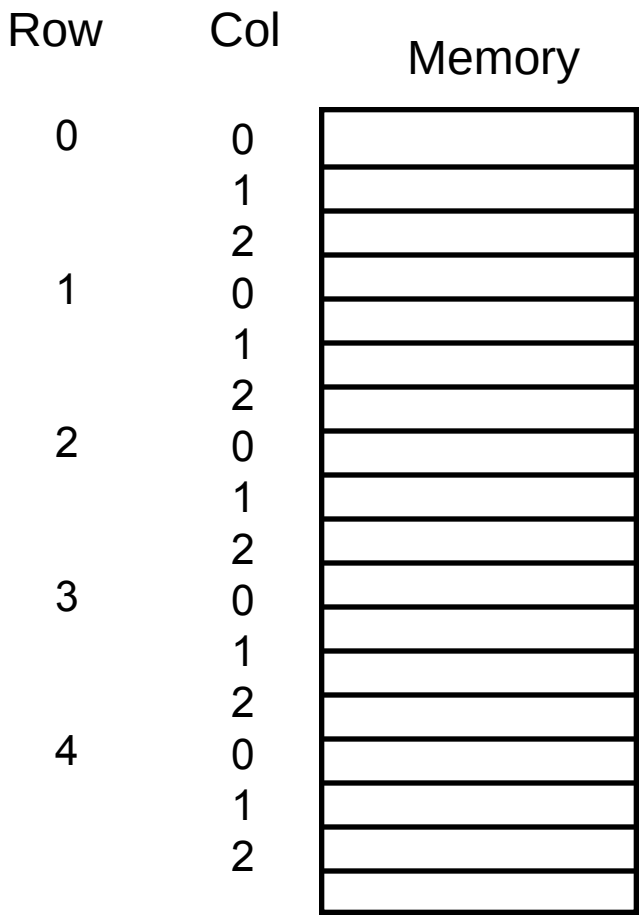
Merging arrays

- **This works by improving spatial locality**
- **For example, some programs may reference multiple arrays of the same size at the same time**
  - Could be bad:
    - **Accesses may interfere with one another in the cache**
- **A solution: Generate a single, compound array…**

```
/* Before:*/
int tag[SIZE]
int byte1[SIZE]
int byte2[SIZE]
int dirty[size]
```

```
/* After */
struct merge {
        int tag;
        int byte1;
        int byte2;
        int dirty;
}
struct merge cache_block_entry[SIZE]
```

Array Organizations

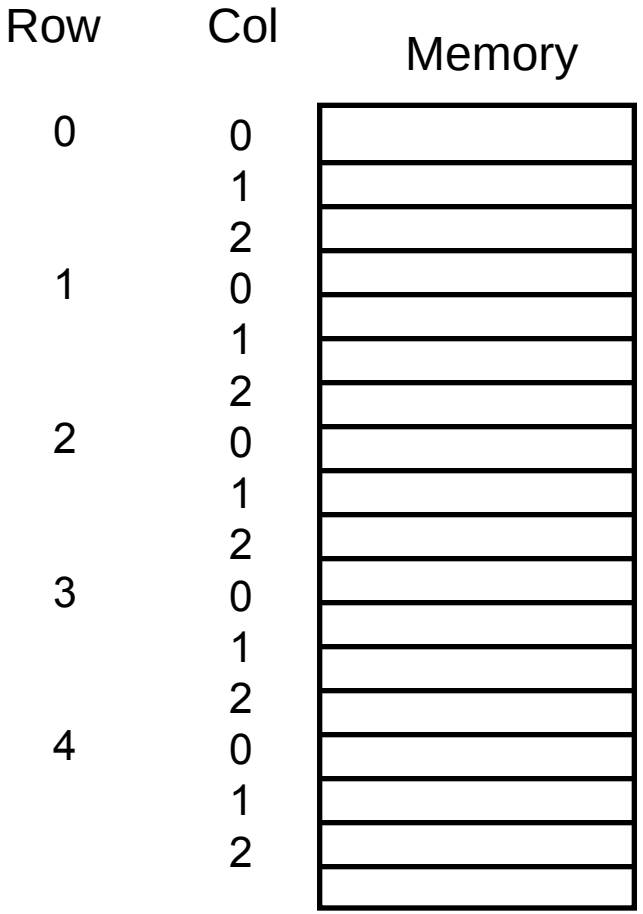


X[i][j]

Row Major Ordering

Loop Interchange Example

```
/* Before */
   for (j = 0; j < 100; j = j+1)
       for (i = 0; i < 5000; i = i+1)
             x[i][j] = 2 * x[i][j];
/* After */
   for (i = 0; i < 5000; i = i+1)
       for (j = 0; j < 100; j = j+1)
             x[i][j] = 2 * x[i][j];
```



- **Some programs have nested loops that access memory in non-sequential order**
  - Simply changing the order of the loops may make them access the data *in* sequential order…
  - Sequential accesses Instead of striding through memory every 100 words

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
       a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
       d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
   {   a[i][j] = 1/b[i][j] * c[i][j];
       d[i][j] = a[i][j] + c[i][j];
   }
```

2 misses per access to a & c vs. one miss per access

Blocking

- **This is probably the most "famous" of compiler optimizations to improve cache performance**
- **Tries to reduce misses by improving temporal locality**

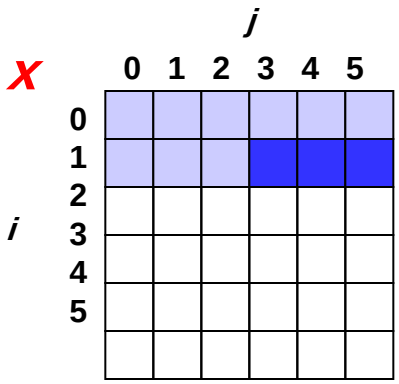
- **1st of all, we need to realize that arrays can be accessed/indexed "differently":**
  - Some arrays are accessed by rows, others by columns
  - Storing array data row-by-row is called *row major order*
  - Storing array data column-by-column is called *column major order*
- **In some code this won't help b/c array data is going to be accessed both by rows and by columns!**
  - Things like loop interchange don't help…
- **Blocking tries to create "submatricies" – or *blocks* to maximize accesses to data loaded in the cache before its replaced.**

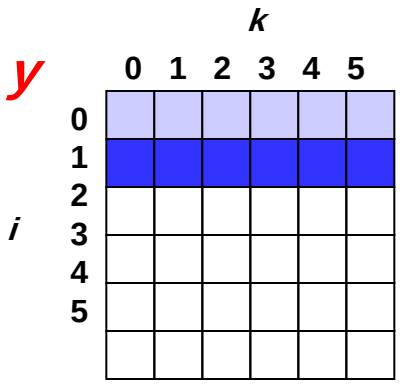
Example blocking (Before)

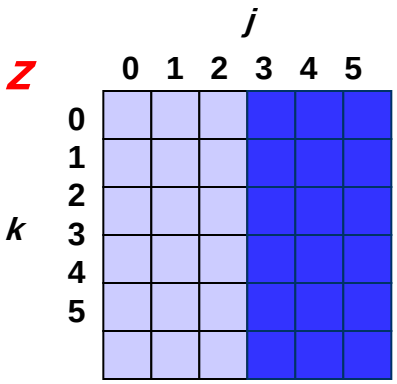
```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
   { r = 0;
      for (k = 0; k < N; k = k+1)
      {
               r = r + y[i][k]*z[k][j];
       };
       x[i][j] = r;
    };
```

2 inner loops read all N x N elements of *z*, access the same N elements in a row of *y* repeatedly, and write one row of N elements of *x*.

Pictorially what happens is:







White block = *not accessed*
Light block = *older access*
Dark block = *newer access*

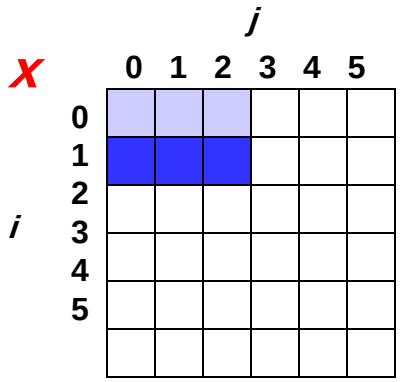
Blocking (some comments)

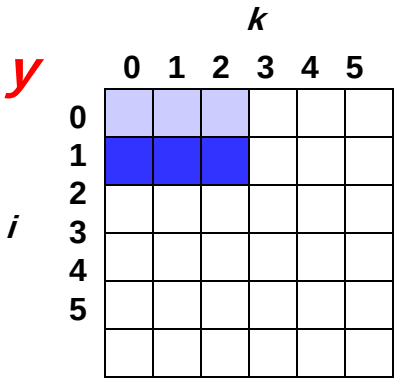
- **In the matrix multiply code, the # of capacity misses is going to depend upon:**
  - The factor N (i.e. the sizes of the matrices)
  - The size of the cache
- **Some possible outcomes:**
  - The cache can hold all N x N matrices (great!)
    - **Provided there are no conflict misses**
  - The cache can hold 1 N x N matrix and one row of size N
    - **Maybe ith row of y and matrix z may stay in the cache**
  - The cache can't hold even this much
    - **Misses will occur for both x and z**
- **In the worst case there will be $2N^3 + N^2$ memory reads – for N^3 memory operations!**

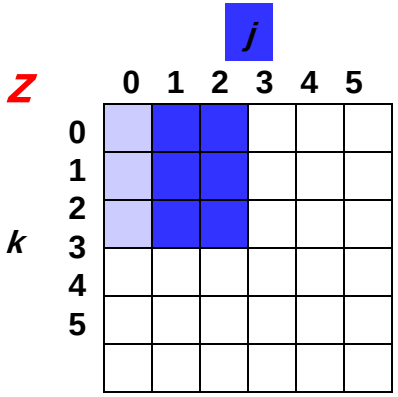
Example blocking (After)

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i = i+1)
      for (j = jj; j < min(jj+B-1,N); j = j+1)
      { r = 0;
        for (k = kk; k < min(kk+B-1,N); k = k+1)
        {
              r = r + y[i][k]*z[k][j];
         };
              x[i][j] = x[i][j] + r;
      };
```

Pictorially what happens is:







To ensure that the elements accessed will all fit/stay in the cache, the code is changed to operate on submatrices of size B x B.

The 2 inner loops compute in steps of size B instead of going from the beginning to the end of x and z.

B is called the *blocking factor*.

Smaller # of elements accessed but they're all in the cache!

Summary blocking

- **What might happen with regard to capacity misses?**
  - Total # of memory words accessed is $2N^3/B + N^2$
  - This is an improvement by a factor of B
- **Blocking thus exploits a combination of spatial and temporal locality**
  - y matrix benefits from spatial locality and z benefits from temporal locality
- **Usually, blocking aimed at reducing capacity misses:**
  - Assumes that conflict misses are not significant or…
  - …can be eliminated by more associative caches
- **Blocking reduces # of words active in a cache at 1 point in time – therefore small block size helps with conflicts…**

Improving Cache Performance

1. Reduce the miss rate,

2. Reduce the miss penalty

3. Reduce the time to hit in the cache.

1. Reducing Miss Penalty: Read Priority over Write on Miss

- **Reads are the common case – make them fast!**
- **Write buffers helped us with cache writes but…**
  - They complicate memory accesses b/c they might hold updated value of a location on a read miss

- **Example:**

SW 512(R0), R3 ; M[512] ← R3 (cache index 0)
LW R1, 1024(R0) ; R1 ← M[1024] (cache index 0)
LW R2, 512(R0) ; R2 ← M[512] (cache index 0)

•Assume direct mapped, write through cache
•(512, 1024 mapped to the same location)
•Assume a 4 word write buffer
•Will the value in R2 always be equal to the value in R3?

1. Reducing Miss Penalty: Read Priority over Write on Miss

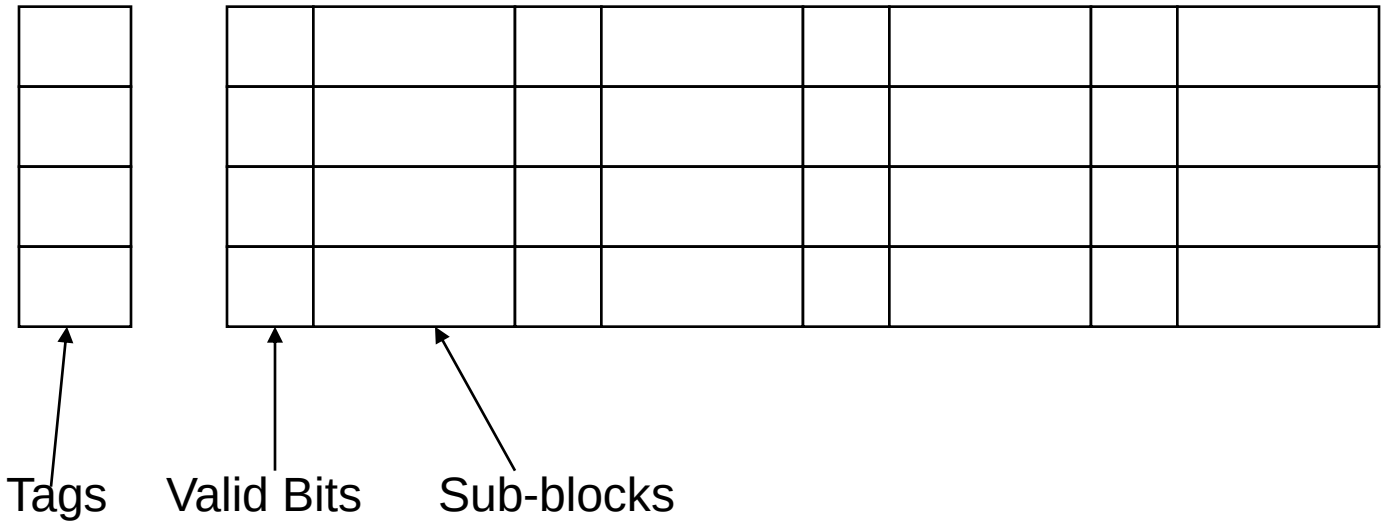
- **Example continue:**
  - This code generates a RAW hazard in memory
  - A cache access might work as follows:
    - **Data in R3 placed into the write buffer after the store**
    - **Next load uses same cache index – we get a miss**
      - (i.e. b/c the store data is there)
    - **Next load tries to put value in location 512 into R2**
    - **This also results in a cache miss**
      - (i.e. b/c 512 has been updated)
    - **If write buffer hasn't finished writing to location 512, reading location 512 will put the wrong, old value into the cache block and then into R2**
    - **R3 would not be equal to R2 – which is a bad thing!**

1. Reducing Miss Penalty: Read Priority over Write on Miss

- **First solution to this problem is to handle read misses only if the write buffer is empty**
  - (Causes quite a performance hit however!)
- **Alternative is to check contents of the write buffer on a read miss**
  - If there are no conflicts & memory system is available, let read miss continue
- **Can also reduce the cost of writes within a processor with a write-back cache…**
  - What if a read miss should replace a dirty memory block?
    - **Could: write to memory, read memory**
    - **Or: copy the dirty block to a buffer, read memory, then write memory – lets the CPU "not wait"**

2. Subblock Placement to Reduce Miss Penalty

- **Instead of replacing a whole complete block of a cache, we only replace one of its subblocks:**
  - Note: We'll have to make a hardware change to do this. What is it???
- **Subblocks should have a smaller miss penalty then full blocks**



3. Reduce Miss Penalty: Early Restart and Critical Word First

- **Don't wait for full block to be loaded before restarting CPU**
  - *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - *Critical Word First*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called *wrapped fetch* and *requested word first*
- **Generally useful only in large blocks,**
- **Spatial locality a problem; tend to want next sequential word, so not clear if benefit by early restart**

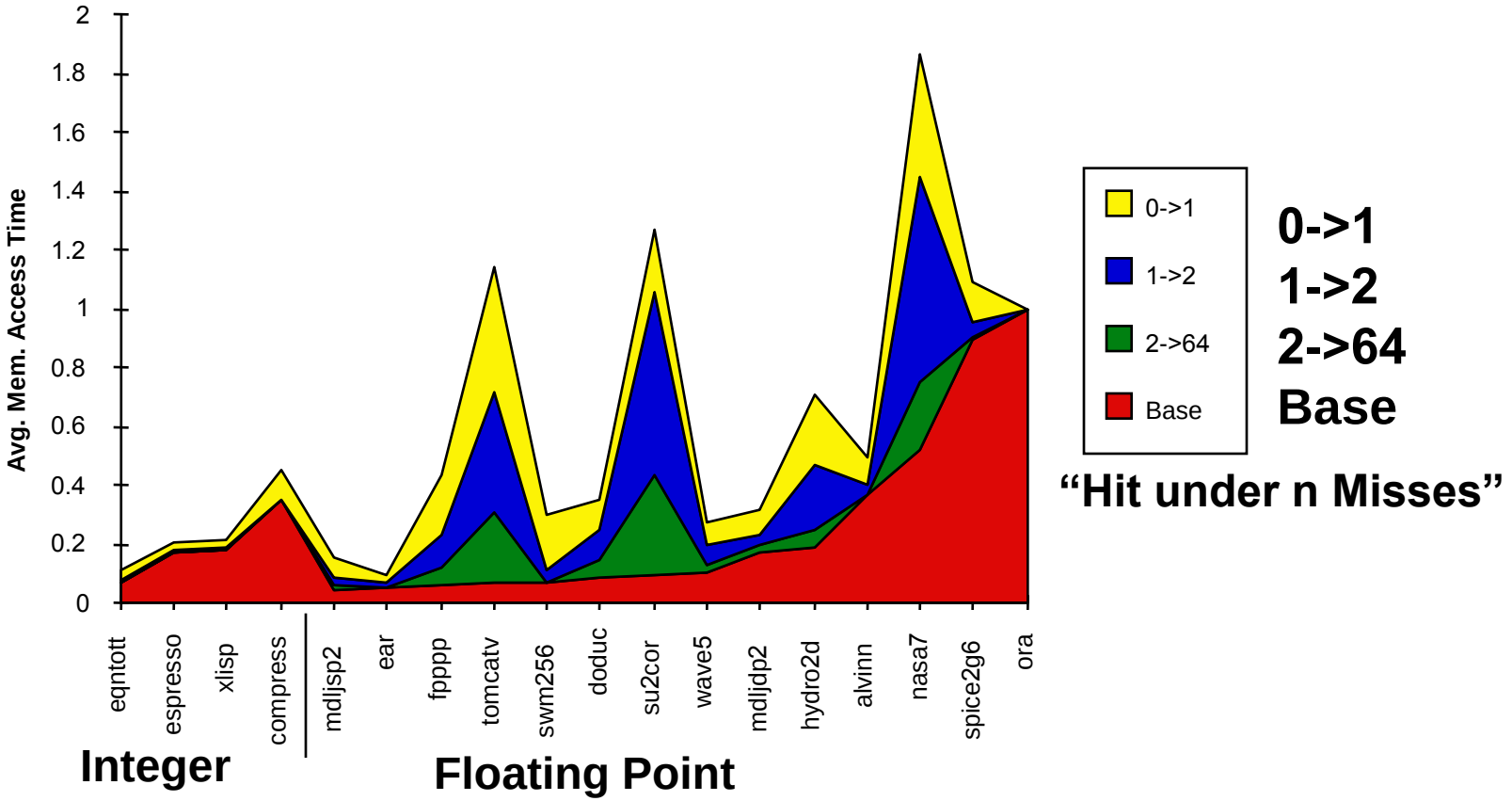
block

4. Reduce Miss Penalty: Non-blocking Caches to reduce stalls on misses

- ***Non-blocking cache*** **or** ***lockup-free cache*** **allow data cache to continue to supply cache hits during a miss**
  - requires out-of-order executuion CPU
- **"*hit under miss*" reduces the effective miss penalty by working during miss vs. ignoring CPU requests**
- **"*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses**
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - Requires muliple memory banks (otherwise cannot support)
  - Penium Pro allows 4 outstanding memory misses

Value of Hit Under Miss for SPEC

- **FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26**
- **Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19**
- **8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss**



5th Miss Penalty

- **L2 Equations**

  AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Miss Penalty$_{L1}$

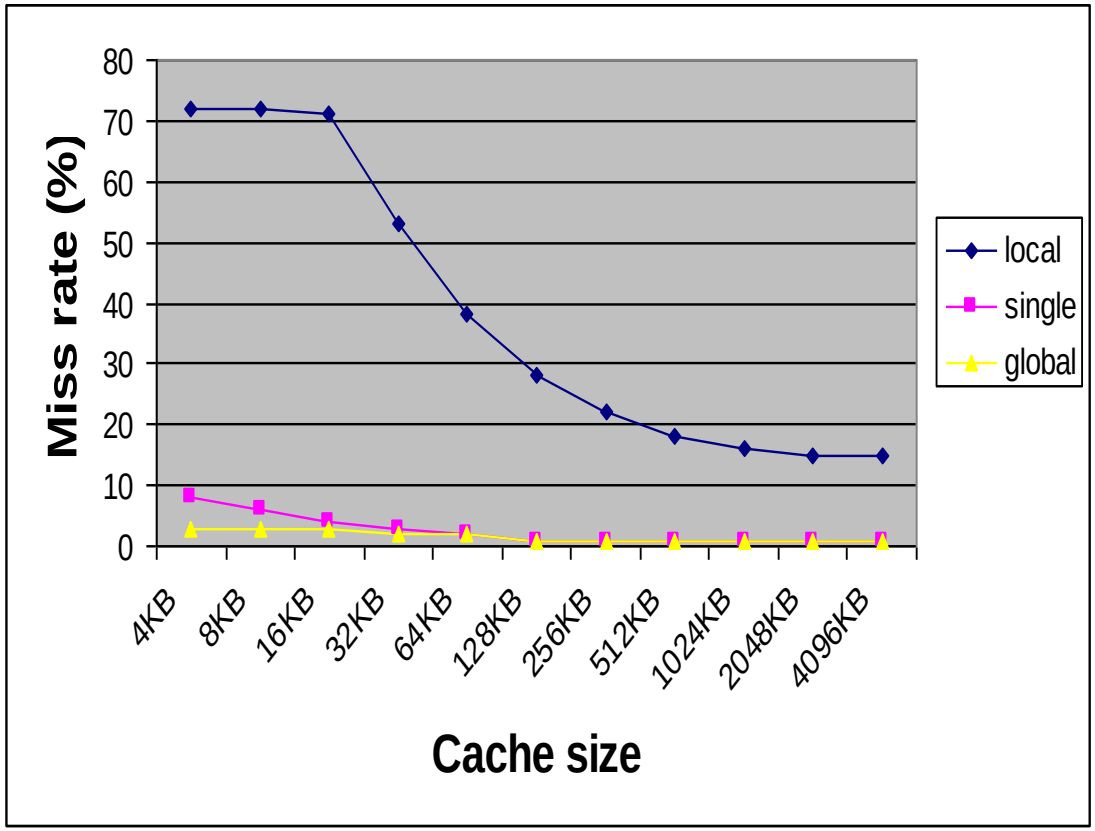
  Miss Penalty$_{L1}$ = Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$

  AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x (Hit Time$_{L2}$ + Miss Rate$_{L2}$ + Miss Penalty$_{L2}$)

- **Definitions:**
  - *Local miss rate*— misses in this cache divided by the total number of memory accesses *to this cache* (Miss rate$_{L2}$)
  - *Global miss rate*—misses in this cache divided by the total number of memory accesses *generated by the CPU* (Miss Rate$_{L1}$ x Miss Rate$_{L2}$)
  - Global Miss Rate is what matters

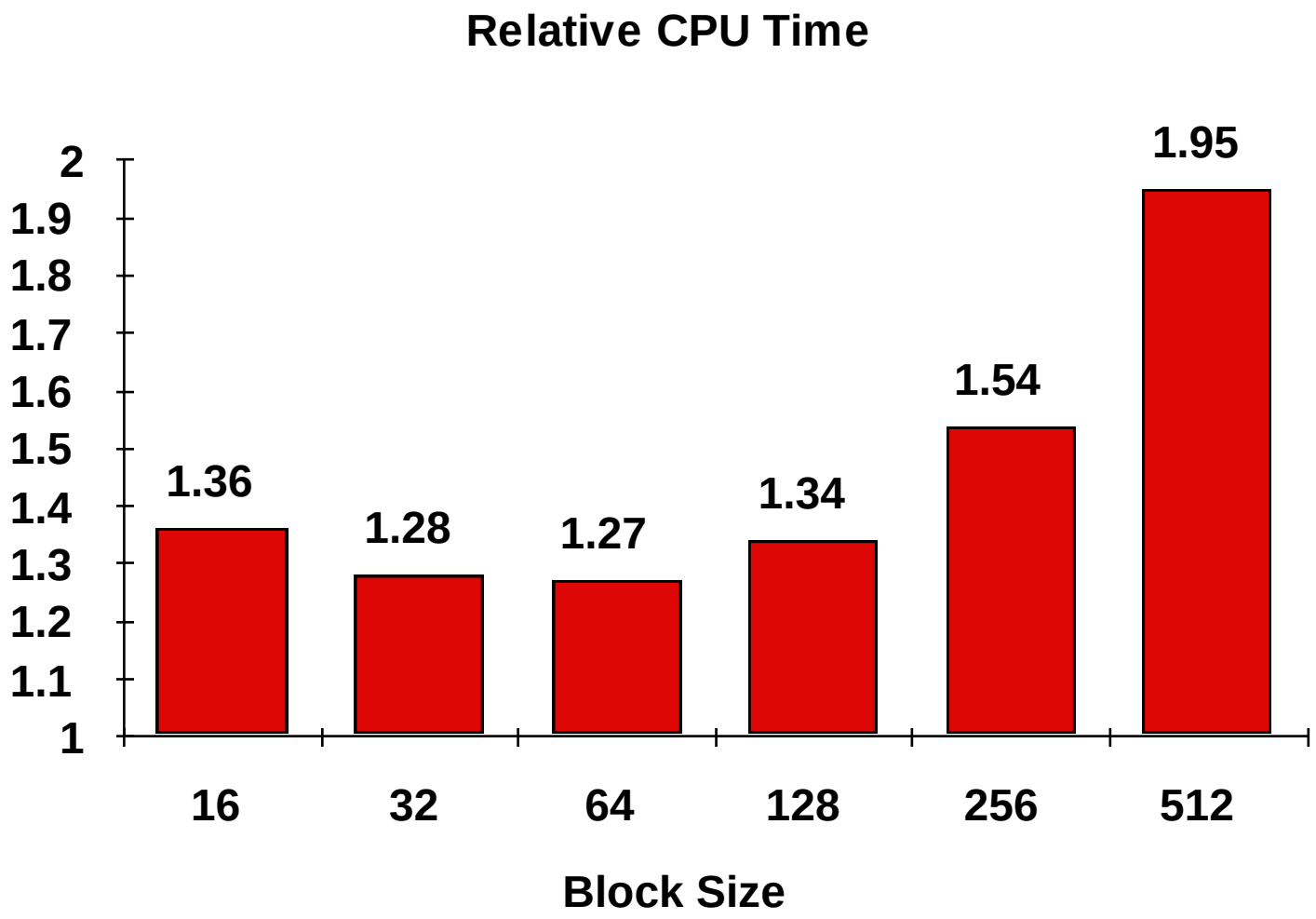
Local and Global Miss Rates

- **32 KByte L1 cache;**
- **Global miss rate close to single level cache rate provided L2 >> L1**
- **Don’t use local miss rate because it is a function of the miss rate of the first level cache.**
- **L2 not tied to clock cycle!**
- **Cost & A.M.A.T.**
- **Generally Fast Hit Times and fewer misses**
- **Since hits are few, target miss reduction**



L2 cache block size & A.M.A.T.

32KB L1, 512KB L2, 4-byte path to memory, 1 cycle to send the address, 6 cycles to access the data, and 1 word/cycle to transfer the data



Reducing Miss Penalty Summary

- **Five techniques**
  - Read priority over write on miss
  - Sub block placement
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit under Miss, Miss under Miss)
  - Second Level Cache
- **Can be applied recursively to Multilevel Caches**
  - Danger is that time to DRAM will grow with multiple levels in between
  - First attempts at L2 caches can make things worse, since increased worst case is worse

Review: Improving Cache Performance

1. **Reduce the miss rate,**
2. **Reduce the miss penalty**
3. ***Reduce the time to hit in the cache*.**

Reducing Hit Time

- **Hit time affects the CPU clock rate**
  - Even for machines that take multiple cycles to access the cache
- **Techniques**
  - Small and simple caches
  - Avoiding address translation
  - Pipelining writes
  - Small sub blocks

1. Small and simple caches

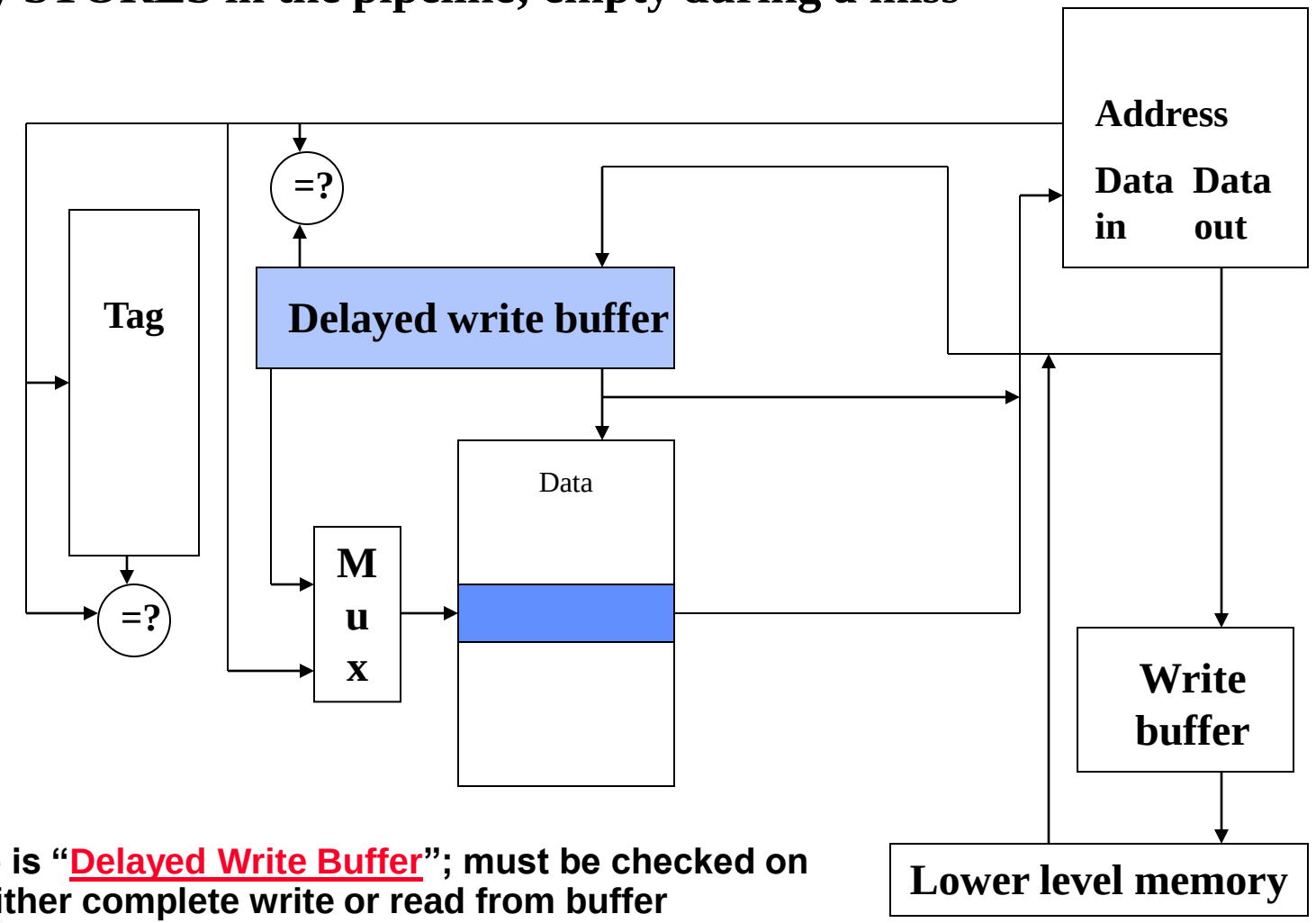
- **Why is this good?**
  - Generally, smaller hardware is faster – so a small cache should help the hit time…
  - If an L1 cache is small enough, it should fit on the same chip as the actual processing logic…
    - **Processor avoids time going off chip!**
    - **Some designs compromise and keep tags on a chip and data off chip – allows for fast tag check and >> memory capacity**
  - Direct mapping also falls under the category of "simple"
    - **Relates to point above as well – you can check tag and read data at the same time!**

2. Fast hits by Avoiding Address Translation

- **Send virtual address to cache? Called *Virtually Addressed Cache* or just *Virtual Cache* vs. *Physical Cache***
  - Every time process is switched logically must flush the cache; otherwise get false hits
    - **Cost is time to flush + "compulsory" misses from empty cache**
  - Dealing with *aliases* (sometimes called *synonyms*); Two different virtual addresses map to same physical address
  - I/O must interact with cache, so need virtual address
- **Solution to aliases**
  - HW guaranteess covers index field & direct mapped, they must be unique; called *page coloring*
- **Solution to cache flush**
  - Add *process identifier tag* that identifies process as well as address within process: can't get a hit if wrong process

3. Fast Hit Times Via Pipelined Writes

- **Pipeline Tag Check and Update Cache as separate stages; current write tag check & previous write cache update**
- **Only STORES in the pipeline; empty during a miss**



In shade is "Delayed Write Buffer"; must be checked on reads; either complete write or read from buffer

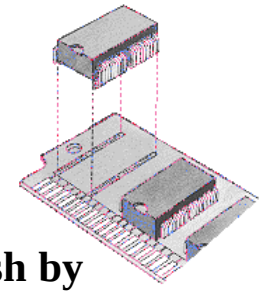
4. Fast Writes on Misses Via Small Subblocks

- **If most writes are 1 word, subblock size is 1 word, & write through then always write subblock & tag immediately**
  - *Tag match and valid bit already set*: Writing the block was proper, & nothing lost by setting valid bit on again.
  - *Tag match and valid bit not set*: The tag match means that this is the proper block; writing the data into the subblock makes it appropriate to turn the valid bit on.
  - *Tag mismatch*: This is a miss and will modify the data portion of the block. Since write-through cache, no harm was done; memory still has an up-to-date copy of the old value. Only the tag to the address of the write and the valid bits of the other subblock need be changed because the valid bit for this subblock has already been set
- **Doesn't work with write back due to last case**

Cache Optimization Summary

	Technique	*MR*	*MP*	*HT*	*Complexity*
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Subblock Placement		+	+	1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2
hit time	Small & Simple Caches	–		+	0
	Avoiding Address Translation			+	2
	Pipelining Writes			+	1

Main Memory



- **Main memory generally utilizes Dynamic RAM (DRAM), which use a single transistor to store a bit, but require a periodic data refresh by reading every row increasing cycle time.**

DRAM: Slow but high density

- **Static RAM may be used for main memory if the added expense, low density, high power consumption, and complexity is feasible (e.g. Cray Vector Supercomputers).**

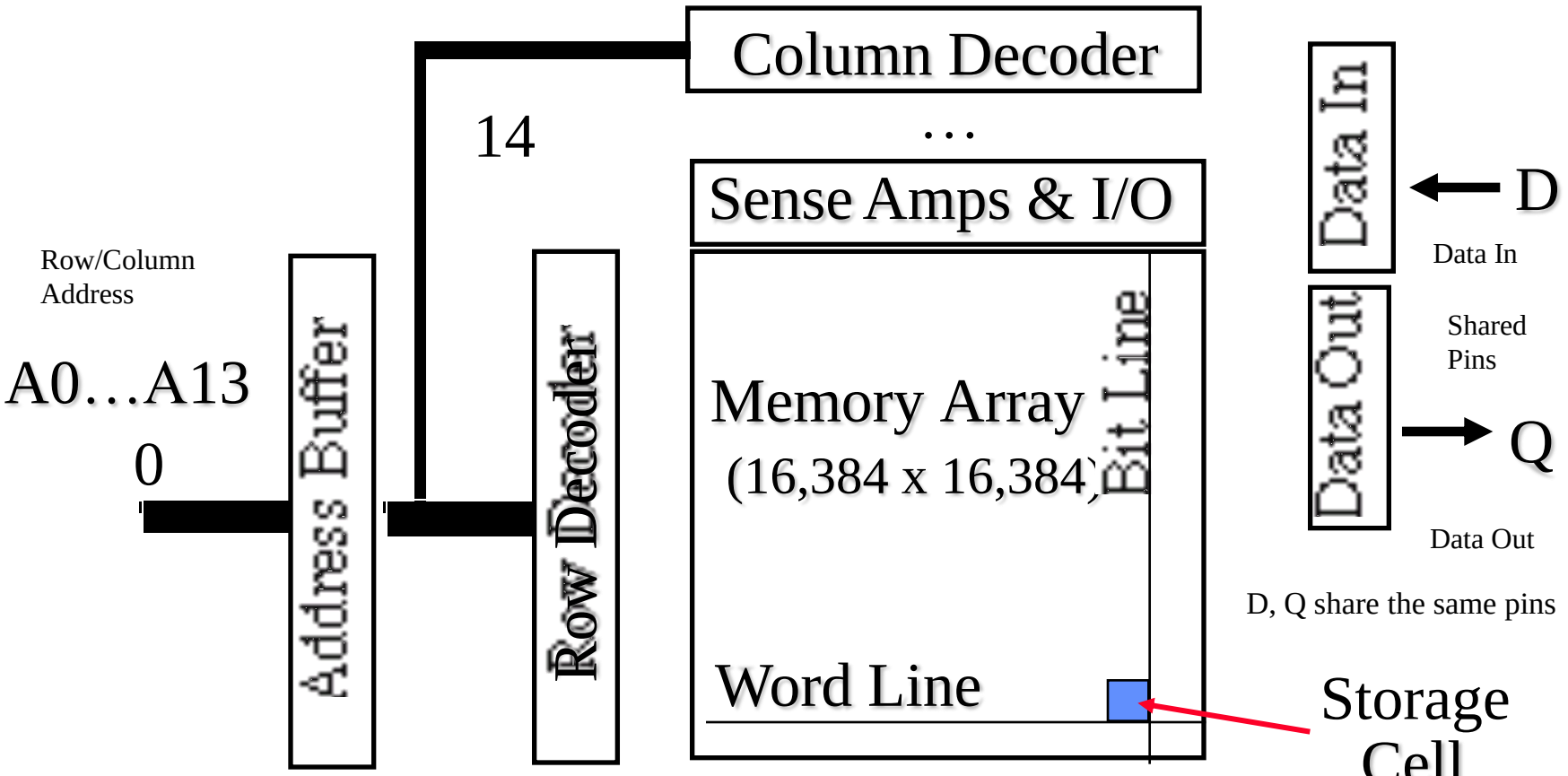
SRAM: Fast but low density

- **Main memory performance is affected by:**
  - **Memory latency: Affects cache miss penalty, M. Measured by:**
    - **Memory Access time: The time it takes between a memory access request is issued to main memory and the time the requested information is available to cache/CPU.**
    - **Memory Cycle time: The minimum time between requests to memory**

    (greater than access time in DRAM to allow address lines to be stable)
  - **Peak Memory bandwidth: The maximum sustained data transfer rate between main memory and cache/CPU.**
    - In current memory technologies (e.g Double Data Rate SDRAM) published peak memory bandwidth does not take account most of **the memory access latency**.
    - This leads to achievable realistic memory bandwidth < peak memory bandwidth

Or maximum effective memory bandwidth

Logical Dynamic RAM (DRAM) Chip Organization (16 Mbit)



Basic Steps:

Control Signals:

1 - Row Access Strobe (RAS): Low to latch row address

2- Column Address Strobe (CAS): Low to latch column address

3- Write Enable (WE) or
Output Enable (OE)

4- Wait for data to be ready

1 - Supply Row Address 2- Supply Column Address 3- Get Data

A periodic data refresh is required by reading every bit

Four Key DRAM Timing Parameters

1 • **t_{RAC}: Minimum time from RAS (Row Access Strobe) line falling (activated) to the valid data output.**
  - **Used to be quoted as the nominal speed of a DRAM chip**
  - **For a typical 64Mb DRAM t_{RAC} = 60 ns**

2 • **t_{RC}: Minimum time from the start of one row access to the start of the next (memory cycle time).**
  - **$t_{RC} = t_{RAC}$ + RAS Precharge Time**
  - **t_{RC} = 110 ns for a 64Mbit DRAM with a t_{RAC} of 60 ns**

3 • **t_{CAC}: Minimum time from CAS (Column Access Strobe) line falling to valid data output.**
  - **12 ns for a 64Mbit DRAM with a t_{RAC} of 60 ns**

4 • **t_{PC}: Minimum time from the start of one column access to the start of the next.**
  - **$t_{PC} = t_{CAC}$ + CAS Precharge Time**
  - **About 25 ns for a 64Mbit DRAM with a t_{RAC} of 60 ns**

1 - Supply Row Address 2- Supply Column Address 3- Get Data

Quest for DRAM Performance

1. **Fast Page mode**
   - Add timing signals that allow repeated accesses to row buffer without another row access time
   - Such a buffer comes naturally, as each array will buffer 1024 to 2048 bits for each access
2. **Synchronous DRAM (SDRAM)**
   - Add a clock signal to DRAM interface, so that the repeated transfers would not bear overhead to synchronize with DRAM controller
3. **Double Data Rate (DDR SDRAM)**
   - Transfer data on both the rising edge and falling edge of the DRAM clock signal $\Rightarrow$ doubling the peak data rate
   - DDR2 lowers power by dropping the voltage from 2.5 to 1.8 volts + offers higher clock rates: up to 400 MHz
   - DDR3 drops to 1.5 volts + higher clock rates: up to 800 MHz

- **Improved Bandwidth, not Latency**

DRAM Generations

Year	Size	RAS (ns)	CAS (ns)	~ RAS+ Cycle Time	Memory Type	
1980	64 Kb	150-180	75	250 ns	Page Mode	Asynchronous DRAM
1983	256 Kb	120-150	50	220 ns	Page Mode	
1986	1 Mb	100-120	25	190 ns		
1989	4 Mb	80-100	20	165 ns	Fast Page Mode	
1992	16 Mb	60-80	15	120 ns	EDO	
1996	64 Mb	50-70	12	110 ns	PC66 SDRAM	Synchronous DRAM
1998	128 Mb	50-70	10	100 ns	PC100 SDRAM	
2000	256 Mb	45-65	7	90 ns	PC133 SDRAM	
2002	512 Mb	40-60	5	80 ns	PC2700 DDR SDRAM	

8000:1 (Capacity)

15:1 (~bandwidth)

3:1 (Latency)

Peak

A major factor in cache miss penalty M

PC3200 DDR (2003)

DDR2 SDRAM (2004)

DDR3 SDRAM (2007-8?)

Basic Memory Bandwidth Improvement/Miss Penalty (M) Latency Reduction Techniques

1 **Wider Main Memory (CPU-Memory Bus):** i.e wider FSB

Memory bus width is increased to a number of words (usually up to the size of a cache block).

- **Memory bandwidth is proportional to memory bus width.**
  - **e.g Doubling the width of cache and memory doubles potential memory bandwidth available to the CPU.** e.g 128 bit (16 bytes) memory bus instead of 64 bits (8 bytes)
- **The miss penalty is reduced since fewer memory bus accesses are needed to fill a cache block on a miss.**

2 **Interleaved (Multi-Bank) Memory:**

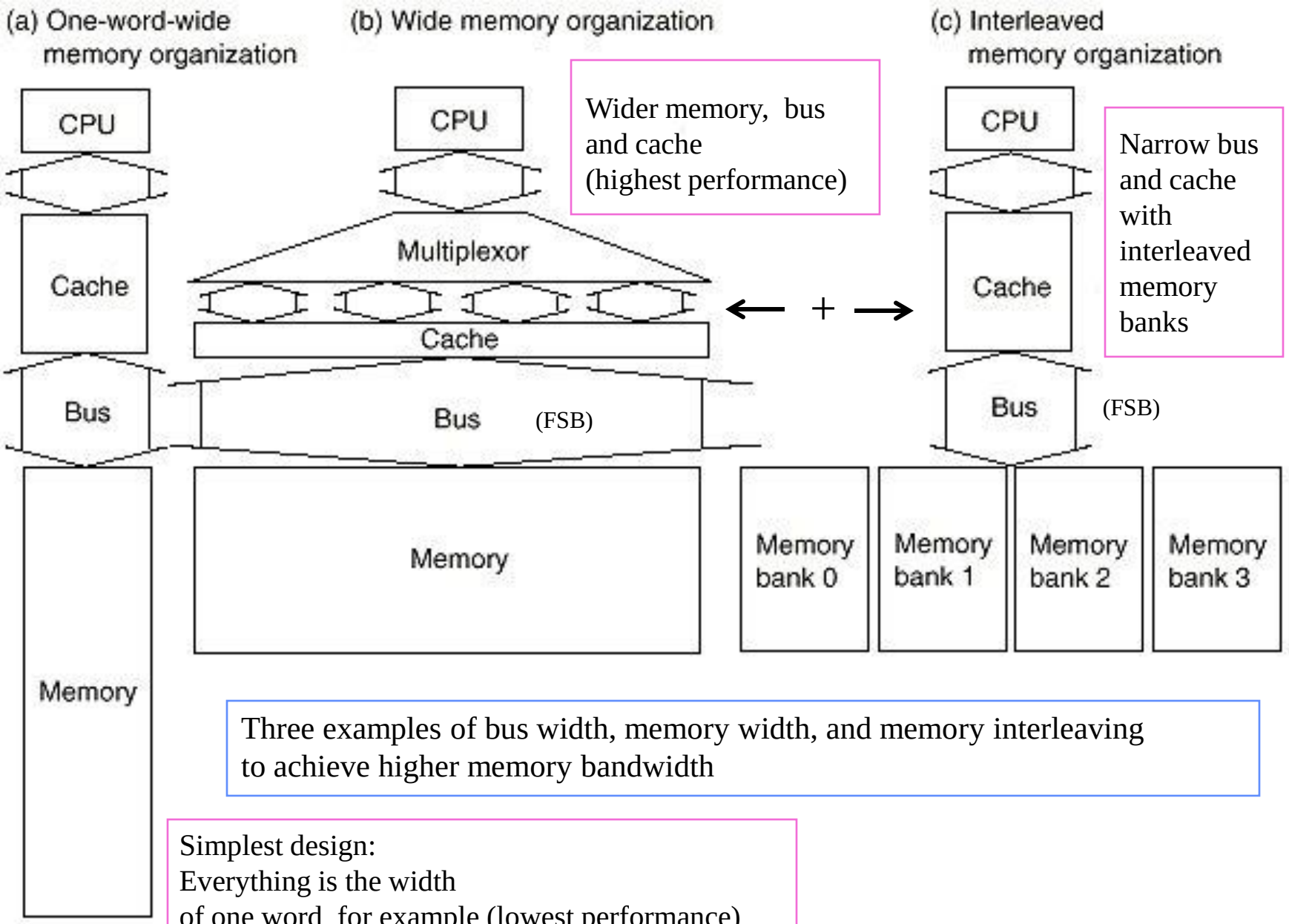
Memory is organized as a number of independent banks.

- **Multiple interleaved memory reads or writes are accomplished by sending memory addresses to several memory banks at once or pipeline access to the banks.**
- **Interleaving factor: Refers to the mapping of memory addressees to memory banks. Goal reduce bank conflicts.**

  e.g. using 4 banks (width one word), bank 0 has all words whose address is:

  (word address mod) 4 = 0

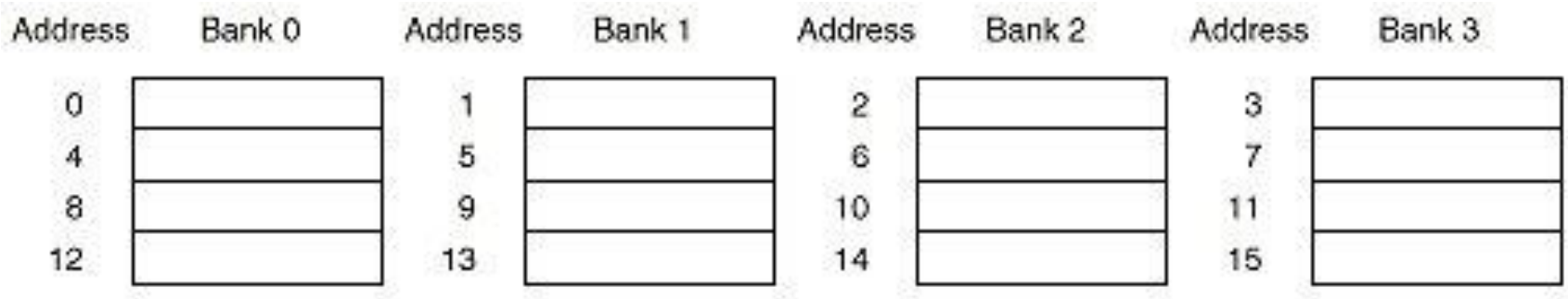
The above two techniques can also be applied to any cache level to reduce cache hit time and increase cache bandwidth.



Three examples of bus width, memory width, and memory interleaving to achieve higher memory bandwidth

Simplest design:
Everything is the width
of one word for example (lowest performance)

Front Side Bus (FSB) = System Bus = CPU-memory Bus

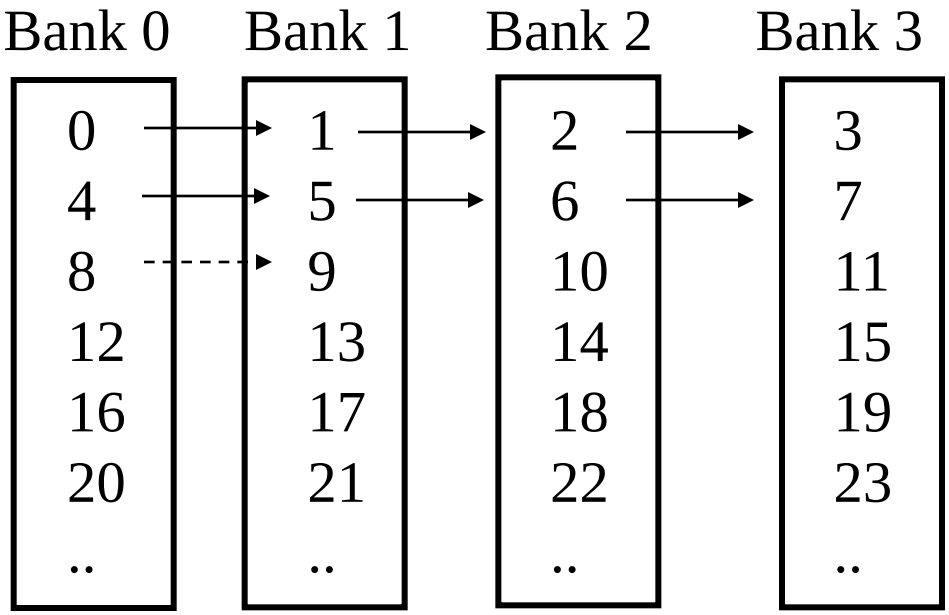


Four Way (Four Banks) Interleaved Memory

Memory Bank Number

Sequential Mapping of Memory Addresses To Memory Banks

Example



Address Within Bank

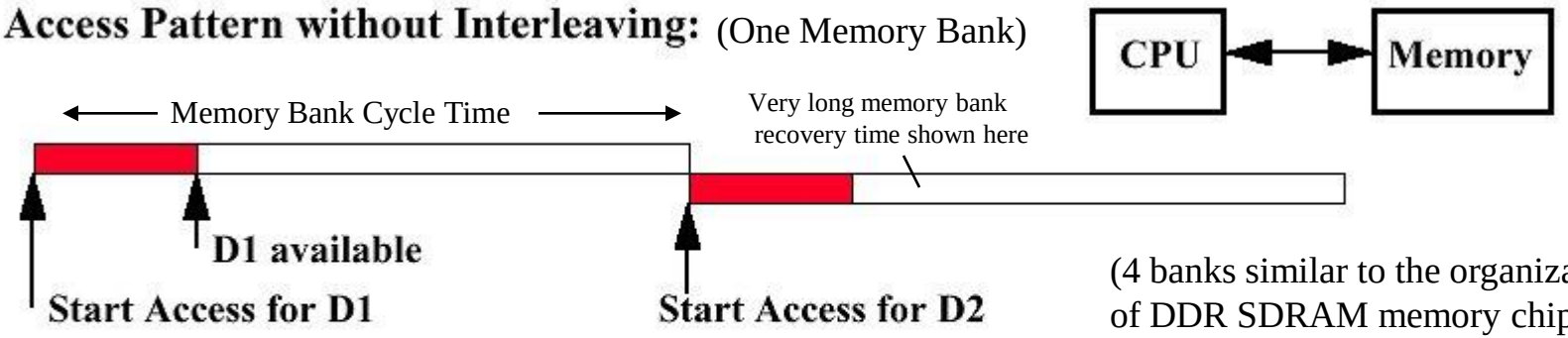
Bank Width = One Word

Bank Number = (Word Address) Mod (4)

Memory Bank Interleaving

Can be applied at: 1- DRAM chip level (e.g SDRAM, DDR) 2- DRAM module level 3- DRAM channel level

Access Pattern without Interleaving: (One Memory Bank)

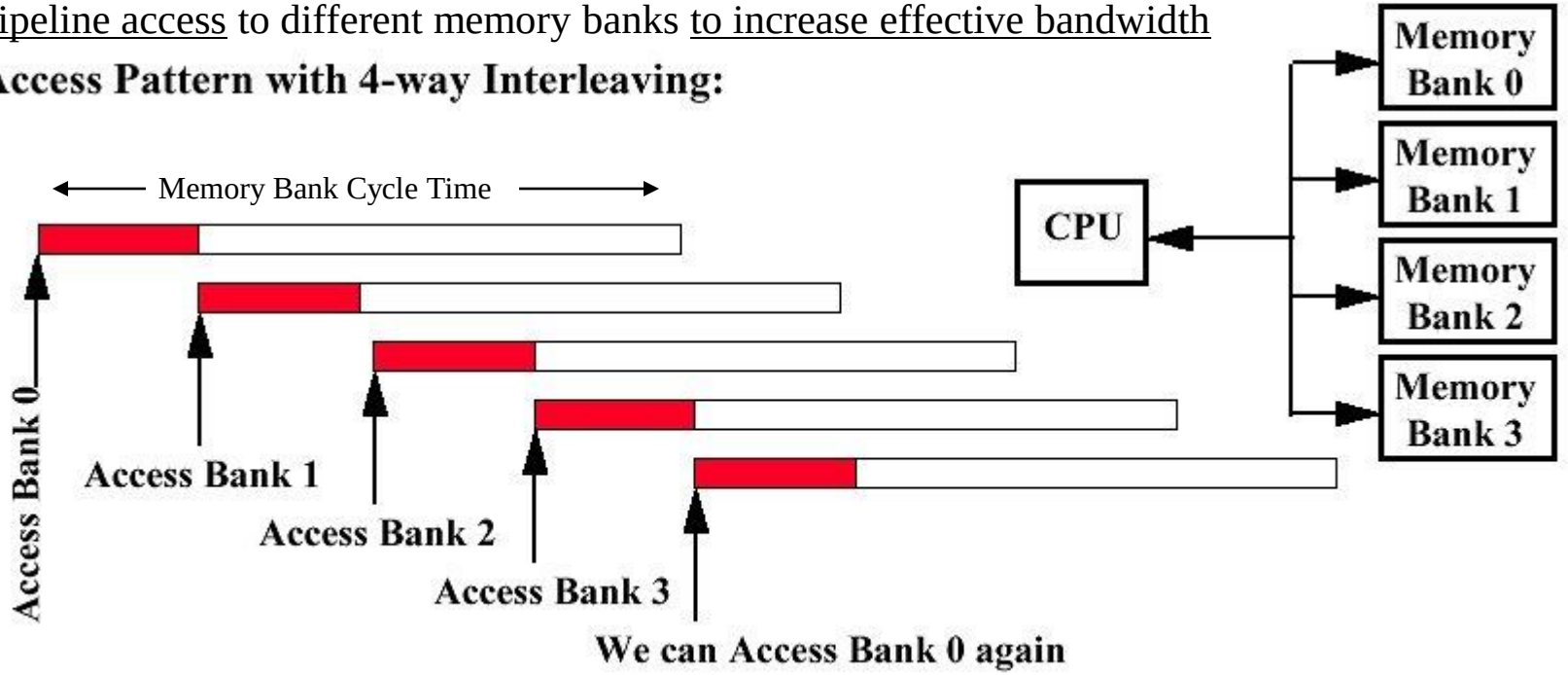


(4 banks similar to the organization of DDR SDRAM memory chips)

Also DDR2 (DDR3 increases the number to 8 banks)

Pipeline access to different memory banks to increase effective bandwidth

Access Pattern with 4-way Interleaving:



Number of banks ≥ Number of cycles to access word in a bank

Bank interleaving *does not reduce latency of accesses to the same bank*

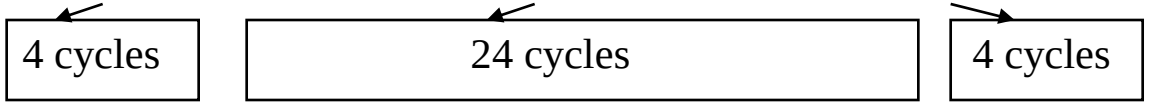
Memory Width, Interleaving: Performance Example

(i.e multiple memory banks)

Given the following system parameters with single unified cache level L_1 (ignoring write policy):

Block size= 1 word Memory bus width= 1 word Miss rate =3% M = Miss penalty = 32 cycles

(4 cycles to send address 24 cycles access time, 4 cycles to send a word to CPU)



Miss Penalty = M= 4 + 24 + 4 = 32 cycles

(Base system)

Memory access/instruction = 1.2 $CPI_{execution}$ (ignoring cache misses) = 2

Miss rate (block size = 2 word = 8 bytes) = 2% Miss rate (block size = 4 words = 16 bytes) = 1%

- **The CPI of the base machine with 1-word blocks = 2 + (1.2 x 0.03 x 32) = 3.15** (For Base system)

Increasing the block size to two words (64 bits) gives the following CPI: (miss rate = 2%)

- **32-bit bus and memory, no interleaving, M = 2 x 32 = 64 cycles CPI = 2 + (1.2 x .02 x 64) = 3.54**
- **32-bit bus and memory, interleaved, M = 4 + 24 + 8 = 36 cycles CPI = 2 + (1.2 x .02 x 36) = 2.86**
- **64-bit bus and memory, no interleaving, M = 32 cycles CPI = 2 + (1.2 x 0.02 x 32) = 2.77**

Increasing the block size to four words (128 bits); resulting CPI: (miss rate = 1%)

- **32-bit bus and memory, no interleaving , M = 4 x 32 = 128 cycles CPI = 2 + (1.2 x 0.01 x 128) = 3.54**
- **32-bit bus and memory, interleaved , M = 4 + 24 + 16 = 44 cycles CPI = 2 + (1.2 x 0.01 x 44) = 2.53**
- **64-bit bus and memory, no interleaving, M = 2 x 32 = 64 cycles CPI = 2 + (1.2 x 0.01 x 64) = 2.77**
- **64-bit bus and memory, interleaved, M = 4 + 24 + 8 = 36 cycles CPI = 2 + (1.2 x 0.01 x 36) = 2.43**
- **128-bit bus and memory, no interleaving, M = 32 cycles CPI = 2 + (1.2 x 0.01 x 32) = 2.38**

Miss Penalty = M = Number of CPU stall cycles for an access missed in cache and satisfied by main memory