HPC
Unit-1

Classes of Computers

1) Personal Mobile Devices
→ Minimize Memory (Memory & Cost)
→ Application: Web based & media-oriented

2) Desktop computing
→ High performance microprocessing with cost reduction
→ Used by individuals
→ Low Cost

3) Servers
→ Computer used by multiple users to execute their programs
→ Characteristics:
* Availability
* Scalability
* Efficient throughput

4) Clusters / Warehouse-Scale computers
→ Group of PCs / Servers that are connected together by LAN & they run on their own OS
→ Price - Performance & Power

5) Embedded computers:
→ Computer inside another computer which is application specific.
→ Should be cheap

Classes of Parallelism

1) Data - level Parallelism:
→ Execution of same function across different elements of the dataset

2) Task level Parallelism:
Simultaneous execution of multiple & different functions on different or same dataset.

Computer H/W exploits these 2 applications in 4 ways

1) Instruction level Parallelism: (ILP)
Exploits DLP & uses pipelining & speculative execution at medium levels.

2) Vector Architecture & GPU: (VA)
Exploits DLP by applying single instruction to multiple data in parallel

3) Thread level Parallelism: (TLP)
Exploits DLP/TLP in tightly coupled memories that allows interaction among parallel threads

4) Request level Parallelism: (RLP)
exploits parallelism specified among largely decoupled tasks specified by the programmer or OS

Michael Flynn classes of Parallelism & Parallel Architectures:

1) Single Instruction, Single Data Stream:
→ A std seq compiler exploiting ILP (Ex: Von Neumann)

2) Single Instruction, Multiple Data Stream:
→ Same instruction executed by multiple processors using different data stream
→ Exploits DLP

3) Multiple Instruction, Single Data Stream:
→ No commercial application

4) Multiple Instruction, Multiple Data Stream:
→ Each processor fetches its own instruction & runs it.
→ Exploits TLP.
→ Tightly coupled MIMD exploits TLP & loosely coupled exploits RLP (Ex: VLIW)

Defining Computer Architecture:

Quick Review of ISA:

1) Class of ISA:

a) Register-memory ISAs-

→ which can access memory as part of many instruction.

b) Load-Store ISAs-

→ which can access memory only with load/Store instruction

2) Memory Addressing

a) Byte Addressable

b) Word addressable

3) Addressing Modes:

→ For RISC V/MIPS:

* Register
* Immediate (for const)
* displacement, where a const offset is added to register to form memory address

→ For 80x86:

* no registers
* 2 registers (based indexed with displacement)
* 2 registers where one is multiplied by size of operand in bytes

4) Types & Size of operands:

5) Operations

* Data transfer
* Arithematic & logical
* control & F.P

6) Control flow instructions:

* MIPS & RISC V conditional branches check for contents of the register but ARM / 80x86 check for test condition code bits.
* ARM & MIPS place the return address in a register while RISC V & 80x86 place it in a stack of memory.

7) Encoding an ISA:

→ fixed length: (ARM & MIPS use 32 bit long instru)

* Simplifies decoding
* PC updating is easier.

→ variable length: (80x86 takes 1-18 bits, RISC also varies)

* takes lesser space in memory.

Requirements to be considered while designing a new computer

1) Functional Requirement

* Different Applications use different types.

2) Level of S/W compatibility:

* Application SW determines what will the computer be used for.

3) OS requirements:

* Necessary features to support chosen OS

4) Standards:

* FP, I/O interfaces, OS, Networks, programming languages

Dependability:

→ Probability that system o/p is correct even though there is a fault.

→ Service level Agreements: guarantee a company gives to their customers for their service if it fails, it goes to service interruption.

→ Module Reliability:

* measure of the continuous service accomplishment from a reference initial instant.
* Reliability Measure: (Mean Time To Failure)
  Defines how long continuously service is accomplished on an avg between the periods of service interruption.
* Rate of Failures: 1/MTTF. rate of failure, generally measured in per billion hours of operation.
* Service Interruption: Mean Time To Repair (MTTR)
* Mean Time Between Failures (MTBF): MTTF + MTTR
* Module Availability: measure of accomplishment w.r.t alternation b/n accomplishment & interruption.
* Mean Time To Repair (MTTR): once service is interrupted, how long it takes to repair & restore the state of accomplishment

---

Example Assume a disk subsystem with the following components and MTTF:

- 10 disks, each rated at 1,000,000-hour MTTF
- 1 ATA controller, 500,000-hour MTTF
- 1 power supply, 200,000-hour MTTF
- 1 fan, 200,000-hour MTTF
- 1 ATA cable, 1,000,000-hour MTTF

Using the simplifying assumptions that the lifetimes are exponentially distributed and that failures are independent, compute the MTTF of the system as a whole.

$$\text{failure rate}_{\text{system}} = 10 \times \frac{1}{10^5} + \frac{1}{5\times10^4} + \frac{1}{2\times10^4} + \frac{1}{2\times10^4} + \frac{1}{10^5}$$

$$\Rightarrow \frac{10+2+5+5+1}{10^5} \Rightarrow \frac{23}{10^5} \Rightarrow \frac{23\,000}{10^8}$$

$$\Rightarrow 23{,}000 \text{ FIT}$$

$$\text{MTTF: inverse of failure rate} = \frac{10^8}{23000} \Rightarrow 43{,}500 \text{ hours}$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{execution time}_{\text{old}}}{\text{execution time}_{\text{new}}} \Rightarrow \frac{1}{(1-\text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}$$

Quantitative Principles of computer Design:

1) Take advantage of parallelism.

2) Principle of locality:

Predict with reasonable accuracy about what data will be used in the future by looking at the past. There are 2 types:

a) temporal locality: If an item was referenced then it will tend to be referenced again in a short period of time.

b) spatial locality: If a memory was referenced, then it is highly likely that the memory surrounding it will also be referenced.

3) Focus on the common case:

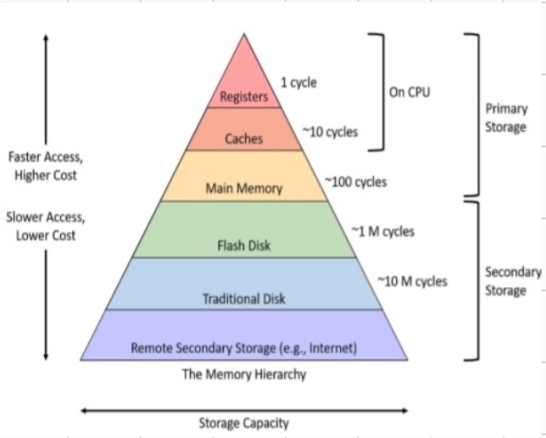
enhancing the common case is better than enhancing a rare case. Frequent case is much simpler & easier to enhance

Computer Architecture

1) Hierarchy of Memories:

→ faster memories are closer to the processor.

→ Goal is to present the user with as much memory as available, making it faster & cheaper.

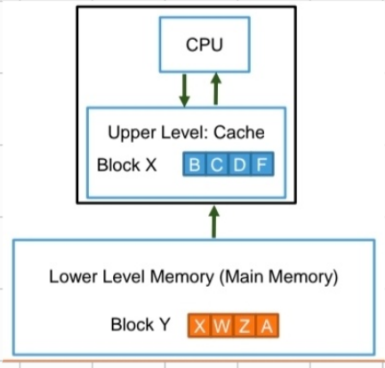


2) Basics of Cache:

→ makes the main memory faster to the processor than it really is.

→ small & fast, inserted b/n the memory & processor

→ Primary cache: internally present with processor, Secondary cache: External to processor.



→ Some terminologies:

* Block: minimum unit of data to move between levels.
* Hit: Data requested is in some block of upper level.
* Hit rate: fraction of memory accesses that are hits (i.e. found at upper level).
* Hit time: Time to determine if the access is indeed a hit + time to access & deliver the data from upper level to CPU
* Miss: Data requested is in some lower level block.
* Miss Rate: fraction of memory accesses that are not hits. Miss rate = (1 - hit rate)
* Miss Penalty: time taken to replace a block from upper level with a corresponding lower level one plus time taken to move it to the processor from there.

Direct Mapped Cache:

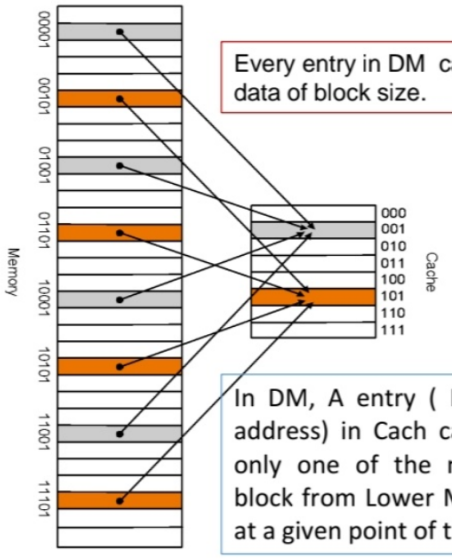
→ Each main Memory address maps to exactly one cache address.

→ From diagram, main mem block 0, 8, 16, 24 are mapped to cache block 0 & blocks 1, 9, 17, 25 map to cache block 1 & so on.

→ To compute the mapping: (block address) modulo (Number of blocks in the cache).

→ TAG field: contains address info to identify whether the associated block in cache corresponds to requested word.

→ TAG bits: Address line from processor − (No. of address lines for index (n) + No. of address lines for block interim of words (m) + byte offset)



Valid bit: Indicates whether associated block contains valid data or not.

Example : Let us say Main Memory is 32 words and Cache is 8 word size. Assume Address lines from processor 5 bits and word of byte size.

Index tells the block where to go in the cache, Tag tells which block is present in that cache.

$n = \log_2(8) \Rightarrow 3$ bits

$m = 0$

byte offset = 0 (word is of byte size).

TAG bits = $5 - (3+0+0) \Rightarrow 2$

Let us consider a 32 bit addressing with Direct Map Cache with Cache of 1K blocks with each block of word size (1K x 4 = 4096 bytes=4KiB).
Show the Cache memory arrangement indicating

a) Number address line bits for Index, block offset and byte offset
b) Find the size of the TAG field
c) Find the total size of cache memory in bits for each index
d) Find total size of memory in Cache in terms of bits.
e) If the address coming from processor is 0x2E101A20, Identify Indexing address?

Answer:

a) Index (n) = 1024 bytes $\Rightarrow 2^{10} \Rightarrow$ 10 bits

block offset (m) = 0 (one word size)

offset = 4 bytes $\Rightarrow 2^2 \Rightarrow$ 2 bits

b) TAG Field size: $32 - (10+0+2)$

$\Rightarrow 20$ bits

c) total size of cache memory: TAG field + data (1 word) + valid bit (1 bit)

20 + 32 + 1

=> 53 bits

d) total size of cache => 1024 x 53

Size of Cache in bits = Number of blocks in Cache x Size of cache memory in bits for each index

=> 54,272 bits

e) 0x2E101A20 => 0010 1110 0001 0000 0001 1010 0010 0000

1010 0010 00 = index (648), 00 = offset

A Direct Mapped Cache with **128 blocks**, each **with 16 bytes**, Find the Number of bits required to represent **TAG filed, Index Field, block offset and byte field**. For a byte address 0x8200 find the mapped Index value in cache. Assume 32 bit addressing.

128 blocks, ∴ index: $\log_2(128) \Rightarrow 7$

16 bytes for each block, 1 word is 4 bytes

hence 4 words

$m = \log_2(4) \Rightarrow 2$, byte offset = $\log_2(4) \Rightarrow 2$

∴ TAG = 32 − (7+2+2) => 21 bits

TAG =21	m bits = 7	k=2	b=2

Consider a Cache with 64 blocks with 16 bytes in each block. Find the block number to which does the byte address 1200 mapped in Direct Mapped Cache?

block address = $\frac{\text{byte address}}{\text{no. of bytes in each block}}$

$\Rightarrow {}^{1200}/_{16} \Rightarrow 75$

only 64 cache blocks are available but total is 75, hence 0->63 is used, 64 onwards it starts from 0 again

Mapped block in cache = Block addr mod no. of blocks in cache

=> 75 Mod 64

=> 11

→ If a set is full & a new block needs to come in, which block needs to be kicked out?

soln: Least Recently Used (LRU): Block that hasn't been used in a long time will be replaced.

Handling Cache Miss:

→ If an instruction access results in a miss, then the content of the instruction register is invalid. To get proper instruction to the cache, we must be able to inform the lower level in memory hierarchy to perform read.

1. Since the PC is incremented in the first clock cycle of execution, the address of the instruction that generates 'miss' = PC − 4
2. Once we have the address, we need to instruct the main memory to perform read.
3. Write the cache entry, putting the data from memory in the data portion of entry, writing the upper (n-m) bits of the address into the tag field & turning the valid bit on.
4. Restart the instruction execution at first step, which will refetch the instruction, this time finding it in the cache.

Handling writes:

→ Imagine a scenario where we write a value to cache but memory is not updated.

→ Now imagine, we are writing data to same index (different TAG bits). This causes the data which was already in that index to be replaced & data is lost forever.

→ If we try to access the data of prev TAG bit in cache (same index), then it is lost because it is overwritten & not stored in memory.

→ we do not want this to happen, so there are 2 methods to prevent this:

1) Write through:

→ Simplest way is to update both lower level (memory) & higher level (cache).

→ In this way data won't be replaced.

→ It wouldn't provide very good performance. Causes every data to be written to main memory which takes a lot of clock cycles.

Handling Write through using Write Buffer:

→ It is the queue that holds data while data is waiting to be written in the memory.

→ This is done because accessing cache is faster than accessing memory when done simultaneously. There will be a bottleneck if a buffer wasn't present.

→ If the buffer is also full, then the processor stalls.

2) Write Back:

→ When a write occurs, new value is written only to the cache block.

→ Only when another value needs to be written to the same index, will it get updated to main mem

→ Additional bit called as dirty bit needs to be added to indicate whether value of main memory has been updated or not. (bit 1 indicates cache is updated but memory is not, bit 0 indicates both are updated.

→ Much faster than write through but harder to implement.

* Write Misses:

1) Write Allocate:

→ When a write miss occurs, the missing block is brought from the main memory to the cache first.

→ If you do not end up reading it, the block will still remain in cache.

→ It acts just like read miss.

2) No Write Allocate :

→ Here to modify the block, we do not need to bring the block all the way to the cache.

→ Instead, it's modified in the lower level memory itself.

→ Block only enters cache if specified.

→ Generally write-back cache's use write allocate hoping that subsequent writes to that block will be captured by caches

→ Write-through cache's use no-write allocate. Even though there are subsequent writes to that block, it will go through lower level memory, so no point.

Assume a ~~fully associative~~ Direct Mapped, write-back cache with many cache entries that starts empty. Below is a sequence of five memory operations (the address is in square brackets):

```
Write Mem[100];
Write Mem[100];
Read  Mem[200];
Write Mem[200];
Write Mem[100].
```

What are the number of hits and misses when using no-write allocate versus write allocate?

Soln:

No Write Allocate:

Operation	Hit/Miss	Action
Write Mem[100]	Write Miss	Write into Main Mem [100]. Cache Not Updated.
Write Mem[100]	Write Miss	Write into Main Mem [100]. Cache Not Updated.
Read Mem[200]	Read Miss	Reads from Main Mem[200] into cache & then reads.
Write Mem[200]	Write Hit	Writes into cache Mem[200]
Write Mem[100]	Write Miss	Write into Main Mem [100]. Cache Not Updated.

← Data directly updated in main memory

← Data is bought to cache because of R operation

← since we got it to cache during R operation

Write Allocate:

Operation	Hit/Miss	Action
Write Mem[100]	Write Miss	Reads from Main Mem[100] into cache & then writes.
Write Mem[100]	Write Hit	Write into Cache Mem [100]
Read Mem[200]	Read Miss	Reads from Main Mem[200] into cache & then reads.
Write Mem[200]	Write Hit	Writes into cache Mem[200]
Write Mem[100]	Write Hit	Write into Cache Mem [100]

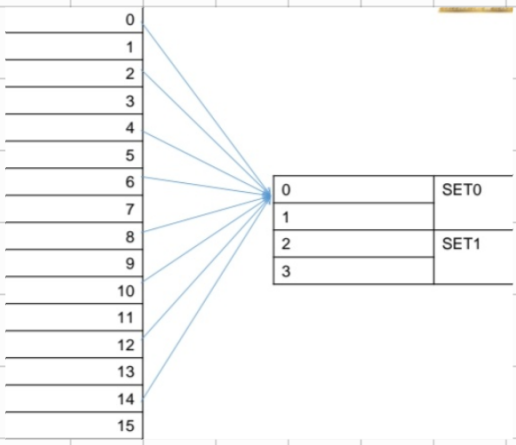
← Initially cache is empty, so miss but after this, data is updated to cache.

← Now it's in cache.

← Data not in cache but after R operation, its updated to cache

Measuring & Improving Cache Performance:

1) Set Associative Mapping



→ Fixed number of blocks as sets in cache where each block of main memory can be placed.

→ A set-associative cache with n locations for a block is called an n-way set associative cache.

→ (block number) modulo (no. of sets in cache)

→ A fully associative cache with m entries is simply an m-way set associative cache; It has one set with m blocks & an entry can reside in any block within that set.

→ Better than direct mapped because 2 or more data can be in the same index simultaneously without constantly kicking each other out.

Assume there are three small caches, each consisting of four one-word blocks. One cache is fully associative, a second is two-way set associative, and the third is direct-mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, and 8(Memory is word addressable)

Soln:

Block Address	Cache Set
0	0 mod 4 => 0
6	6 mod 4 => 2
8	8 mod 4 => 0

1) Direct Mapped

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

2) 2 way Set associative

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

In the end, 'least Recently Used' methodology was used. How in 4th block, mem[8] was replaced by mem[6] & mem[0] replaced by mem[8] in 5th step.

3) Fully associative

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

→ It is clear that Miss rate is high for direct mapping & low in fully & set associati -ve. But HIT time increases as cache block can be anywhere.

Possible Associativity structure for 8 block cache:

→ Size of tags versus set associativity:

* Increasing associativity requires:
  * More comparators
  * More TAG bits per cache block as No. of SETs ↓es by a factor of 2.

Assuming a cache of 4096 blocks, a 4-word block size, and a 32-bit address, find the total number of sets and the total number of tag bits for caches that are direct mapped, two-way and four-way set associative, and fully associative

Direct Mapped:

$n = \log_2(4096) \Rightarrow 12$ bits

$m = \log_2(4) \Rightarrow 2$ bits

TAG = $32 - (12+2+2) \Rightarrow 16$

∴ Total no. of TAGs = 16×4096

$\Rightarrow$ 66K bits

2 Way Set Associative:

2048 blocks, 2 sets

$n = \log_2(2048) \Rightarrow 11$

$m = \log_2(4) \Rightarrow 2$

∴ TAG bits = $32-(11+2+2)$

$\Rightarrow$ 17 bits

∴ Total TAG bits = 17×4096

$\Rightarrow$ 115K bits

4 Way Set Associative:

1024 blocks, 4 sets

$n = \log_2(1024) \Rightarrow 10$

$m = \log_2(4) \Rightarrow 2$

TAG bits = $32 - (10+2+2) \Rightarrow 18$

∴ Total TAG = 4096×18

$\Rightarrow$ 74K bits

Fully Associative:

1 block, 4096 sets

$n = 0$

$m = \log_2(4) \Rightarrow 2$

∴ TAG bits = $32 - (0+2+2)$

$\Rightarrow 28$ bits

Total TAG = 28×4096

$\Rightarrow$ 115K bits

→ by increasing associativity, no. of bits on index ↓es by 1, no. of bits in TAG ↑es by 1 & sets ↓se by a factor of 2.

2) Choosing which block to replace:

→ Algorithms used are:

1) Least Recently Used:

→ The block replaced is the one that hasn't been used in a long time.

→ For a 2 way set associative cache:

LRU	Way1			Way0		
	V	TAG	Data	V	TAG	Data

LRU bit = 1; least recently referenced in a set is way 1 block

LRU bit = 0; least recently referenced in a set is way 0 block

A 64 bit computer has a 128 KB 8 way associativity mapped cache. The cache has 128 sets and a line is 16 word. How many TAG bits does each address require?

Soln:

total address size: 64 bits

block offset: $\log_2(16) \Rightarrow 4$ bits

128×1024

128×8

index (n) = $\log_2(128) \Rightarrow 7$ bits

↑ total no. of blocks

For 64 bit computer, 1 word = 8 bytes

$\therefore$ offset = $\log_2(8) \Rightarrow 3$

$\therefore$ TAG = $64 - (7+4+3) \Rightarrow 50$ bits

1. A computer system uses 16-bit memory addresses. It has a 2K-byte cache organized in a direct-mapped manner with 64 bytes per cache block. Assume that the size of each memory word is 1 byte.
   a. Calculate the number of bits in each of the Tag, Block, and Word fields of the memory address.
   b. When a program is executed, the processor reads data sequentially from the following word

Addresses: 128, 144, 2176, 2180, 128, 2176. All the above addresses are shown in decimal values. Assume that the cache is initially empty. For each of the above addresses, indicate whether the cache access will result in a hit or a miss.

2. Repeat Problem # 2, if the cache is organized as a 2-way set-associative cache that uses the LRU replacement algorithm.

a) $m = \log_2(64) \Rightarrow 6$ bits

$n = \log_2[(2 \times 1024)/64] \Rightarrow 5$ bits

$\therefore$ TAG = $16 - (6+5+0) \Rightarrow 5$ bits

b) no. of blocks = $2048/64 \Rightarrow 32$

$\therefore$

Block address	Cache set
128	$(128/64) \bmod 32 \Rightarrow 2$
144	$(144/64) \bmod 32 \Rightarrow 2$
2176	$(2176/64) \bmod 32 \Rightarrow 2$
2180	$(2180/64) \bmod 32 \Rightarrow 2$

c) 2 way set associative

Addr of mem block addr	Hit/miss	Set 1	Set 2
128	Miss	128	
144	Hit	128/144	
2176	Miss	128/144	2176
2180	Hit	128/144	2176/2180
128	Hit	128/144	2176/2180
2176	Hit	128/144	2176/2180

Direct Mapped:

Everything is a miss

Cache Performance Review:

CPU clock cycle = Instn count (IC) × CPI

CPU exec time = CPU clk cycle × clk cycle time

→ CPU time can be divided into:

1) clock cycles taken to execute the program.

2) Clock cycles taken by processor waiting for the memory system.

→ CPU execution time = (CPU clk cycles + mem stall cycles) × clock cycle time.

→ Memory stall clock cycle = (Read stall cycle + Write stall cycle)

⇒ No. of misses × Miss penalty (cost per miss)

⇒ $IC \times \frac{\text{Misses}}{\text{Instruction}} \times$ Miss penalty

$$\Rightarrow IC \times \frac{\text{Memory Access}}{\text{Instruction}} \times \text{Miss Rate} \times \text{Miss penalty}$$

Assume we have a computer where the cycles per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?

Case 1: Always a hit

CPU execution time = (CPU clock cycles + Memory stalls) × Clock cycle

$\Rightarrow (IC \times CPI + 0) \times \text{Clk cycle}$

$\Rightarrow IC \times 1.0 \times \text{Clk cycle}$

Case 2: hits & misses:

Memory stall cycles $\Rightarrow IC \times \frac{\text{memory access}}{\text{instruction}} \times \text{Miss rate} \times \text{Miss penalty}$

$\Rightarrow IC \times (\underbrace{1 + 0.5}_{\text{instn fetch + data access}}) \times 0.02 \times 25$

$\Rightarrow 0.75\ IC$

$\text{CPU execution time}_{\text{real}} \Rightarrow (IC \times 1.0 + IC \times 0.75) \times \text{Clk cycle}$

$\Rightarrow 1.75\ IC \times \text{Clk cycle}$

$\therefore$ Computer with no cache misses is 1.75 times faster.

b) Number of Misses per instruction:

$$\frac{\text{Misses}}{\text{Instruction}} \Rightarrow \text{Miss rate} \times \frac{\text{Memory Access}}{\text{Instruction}}$$

$\Rightarrow 0.02 \times (1.5) \Rightarrow 0.03$

Memory Stall cycles:

→ coming from Read as well as write.

$$\text{Read Stall Cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read Miss Rate} \times \text{Read Miss Penalty}$$

$$\text{Write Stall Cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write Miss Rate} \times \text{Write Miss Penalty}\right) + \text{Write Buffer Stalls}$$

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls, and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.

Soln: Instruction miss cycles: $2\% \times 100 \Rightarrow 2$

Data miss cycles: $0.36 \times 4\% \times 100 \Rightarrow 1.44$
(only l/s causes the stall)

$\therefore CPI_{\text{stall}} = 2 + 2 + 1.44 \Rightarrow 5.44$

$\therefore$ Speedup = $\frac{\text{CPI stall}}{\text{CPI no stall}} \Rightarrow \frac{5.44}{2} \Rightarrow 2.72$

Split Cache:

→ Memory Hierarchy consisting of 2 independent caches running in parallel, one handling data & the other handling instructions.

→ Very useful when the memory access unit & instruction fetch unit are physically located away from each other.

→ Allows instruction & data access to occur parallely without contention.

Reducing Miss Penalty using Multilevel Caches:

→ This second level cache is normally on the same chip & is used when a miss occurs in the primary cache.

→ If data is present in the secondary level cache, then the miss penalty for the first level cache will be the access time for 2nd level cache, which is lesser than access time from main memory.

→ If data not present in primary & secondary memory, longer miss penalty occurs.

Performance of Multilevel Caches
Suppose we have a processor with a base CPI of 1.0, assuming all references hit in the primary cache, and a clock rate of 4 GHz. Assume a main memory access time of 100 ns, including all the miss handling.
Suppose the miss rate at the primary cache is 2%. How much faster will the processor be if we add a secondary cache that has a 5ns access time for either a hit or a miss and is large enough to reduce the miss rate of main memory to 0.5%?

clk time = $1/4\text{GHz} \Rightarrow 0.25\text{ns}$

miss penalty for main mem = $100\text{n}/0.25\text{n} \Rightarrow 400\text{cycles}$

miss penalty for secondary mem = $5\text{n}/0.25\text{n} \Rightarrow 20\text{ cycles}$.

$\text{CPI}_{1\text{cache}} = 1 + 2\% \times 400 \Rightarrow 9\text{ CPI}$

$\text{CPI}_{2\text{cache}} = 1 + 2\% \times 20 + 0.5\% \times 400 \Rightarrow 3.4\text{CPI}$

Speedup = $9/3.4 \Rightarrow$ 2.6 times

Average Memory Access Time (AMAT)

→ Average time taken to access memory considering both hits & misses and frequency for different access.

→ AMAT = Time for a hit + Miss rate × Miss penalty.

i) A computer has a single cache (off-chip) with a 2 ns hit time and a 98% hit rate. Main memory has a 40 ns access time. What is the computer's effective access time?

ii) **If we add an on-chip cache with a 0.5 ns hit time and a 94% hit rate, what is the computer's effective access time? How much of a speedup does the on-chip cache give the computer?**

i) Miss rate = 2%

$\therefore AMAT_{L2} = 2n + 2\% \times 40ns$

$\Rightarrow 2.8ns$

ii) L_1 : on chip cache

$AMAT_{L1} = HT_{L1} + MR_{L1} \times MP_{L1}$

$\Rightarrow HT_{L1} + MR_{L1} \times (AMAT_{L2}) \Rightarrow 0.5n + 6\% \times (2.8n)$

$\Rightarrow 0.668n$

$\therefore$ speedup $= \frac{2.8n}{0.668n} \Rightarrow 4.19$ times.

Consider an Intel processor p4 with 32 KB unified L1- cache. The miss rate for the cache is 18% and hit time is 2 cycles. The processor also has 16MB on chip L2 cache with Hit time of 18 cycles. If data is not found in L2 cache request is made to 4 GB main memory with access time of 200 cycles. On average 8.12 cycles are required to process a memory request. How often main memory is referenced?

$AMAT = HT_{L1} + MR_{L1} \times (HT_{L2} + MR_{L2} \times MP_{L2})$

$8.12 = 2 + 18\% \times (18 + MR_{L2} \times 200)$

$200 MR_{L2} + 18 = 34$

$\therefore MR_{L2} = 8\%$

Memory reference freq $= MR_{L1} \times MR_{L2}$

$\Rightarrow 0.18 \times 0.08$

$\Rightarrow 0.0144$

Classification of Cache Misses:

1) Compulsory Cache Miss:

→ The very first access to a block cannot be in cache, so the block must be brought to the cache.

→ Size of cache doesn't matter, it is always a miss if cache is empty initially.

2) Capacity Cache Misses:

→ Assume entire fully associative cache is full & there has been a miss.

→ Bring the block from main memory & do block replacement based on LRU. Hence old block is replaced with new one.

→ Now if processor refers to old block in cache, it will be a miss as that block has been replaced.

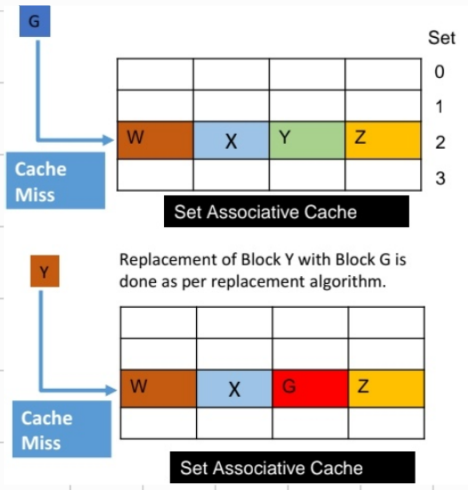
3) Conflict Cache Misses:

→ In set associative, blocks are fully mapped to a particular SET. Assume SET2 is full & holds valid blocks.

→ Assume processor generates a block to SET2 & it misses.

→ This block is bought from lower level & replacement is done via LRU.

→ Now if processor requests to evict replaced block, it will be a Miss as it's not there in the cache anymore.



Cache Optimization techniques exploit techniques to reduce AMAT. From the above equation, it is clear that AMAT can be reduced by

1. **Reducing the Miss Rate (MR)**
   - ✓ Larger block size: Reduces Compulsory Misses but effect on other Cs to be checked
   - ✓ Larger cache size: Capacity Misses but effect on other Cs to be checked
   - ✓ Higher Associativity: Conflict Misses but effect on other Cs to be checked
2. **Reducing the Miss penalty (MP)**
   - ✓ Multi Level Caches
   - ✓ Giving read priority **over writes**
3. **Reducing Hit Time (HT) in the cache**
   - ✓ Avoiding **address translation when indexing the clock.**(Especially in systems with virtual memory support where the processor generates a virtual address and that virtual address needs to be translated to a physical address. This translation is time-consuming. If we can eliminate translation time then HT will be reduced.

Techniques to Reduce Miss Rate:

Cache Parameter	Cold Misses	Capacity Misses	Conflict Misses	Overall Misses
Increased Capacity	No Effect	Decreases	May Decrease	May Decrease
Increased Block Size	Decrease	May Increase	May Increase	Varies
Increased Associativity	No Effect	No effect	May Decrease	May Decrease

- Increasing **associativity does not effect the block size.**
- We know that the **only way to reduce compulsory Misses is Increasing the Block size.**
- As a result this optimization **does not have any effect on Cold Misses.**

→ Increasing block size reduces miss rate while ↑ing miss penalty & greater associativity comes at the cost of ↑ed time.

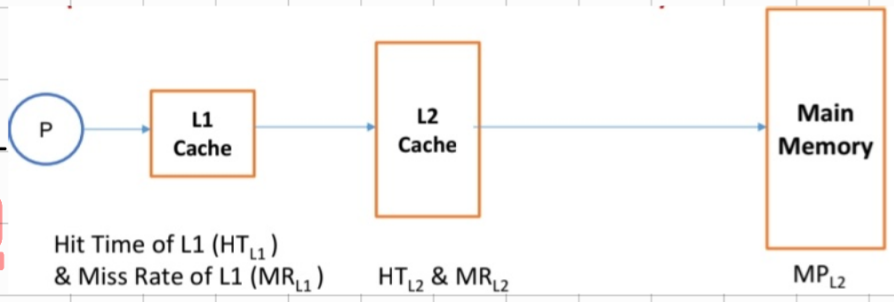
→ Reducing miss rate is as good as ↓ing cold, capacity & conflict misses

Multilevel Caches to Reduce Miss penalty

→ $AMAT = HT_{L1} + MR_{L1} \times MP_{L1}$

where $MP_{L1} = HT_{L2} + MR_{L2} \times MP_{L2}$

$MR = \frac{\text{No. of misses in cache}}{\text{total no. of memory access in cache}}$



→ global miss rate: Number of misses in cache level divided by total no. of times processor has accessed the memory.

→ global & local miss rate of L1 cache is the same as all memory generated by processor will be seen by L1 but that's not the case with L2.

Suppose that in **1000 memory references** there are 40 misses in the first-level cache and 20 misses in the second-level cache. **What are the various miss rates?** Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. **What is the average memory access time and average stall cycles per instruction?** Ignore the impact of writes.

Soln:

MR_{L1} (local/global) = $40/1000 \Rightarrow 4\%$

MR_{L2} (local) = $20/40 \Rightarrow 50\%$

MR_{L2} (global) = $20/1000 \Rightarrow 2\%$

$AMAT = HT_{L1} + MR_{L1} \times (HT_{L2} + MR_{L2} \times MP_{L2})$

$\Rightarrow 1 + 4\% \times (10 + 50\% \times 200)$

$\Rightarrow 5.4$ clk cycles

Avg stall cycles per instrn:

no. of instruction = $\frac{\text{total no. of mem access}}{\text{Mem reference per instrn}} \Rightarrow \frac{1000}{1.5} \Rightarrow 667$

Mem stall cycle per instruction = $\frac{\text{Mem Access}}{\text{Instruction}} \times MR \times MP$

$\Rightarrow (AMAT - HT_{L1}) \times \frac{\text{No. of References}}{\text{instruction}}$

$(5.4 - 1) \times 1.5 \Rightarrow 6.6$ clk cycles

Or

$(1.5) \times MR_{L1} \times MP_{L1}$

$\Rightarrow 1.5 \times 0.04 \times (10 + 0.5 \times 200)$

$\Rightarrow 6.6$

→ Speed of L1 cache affects clk rate of processor

→ Speed of L2 cache affects MP of L1 cache

Giving Periority to Read over Write:

→ Read operations are performance critical because processor has to wait for data to be available to continue further.

→ Write operations can just move on ahead with subsequent operations.

→ write buffer may complicate things because they might hold new value of location needed on a read miss.

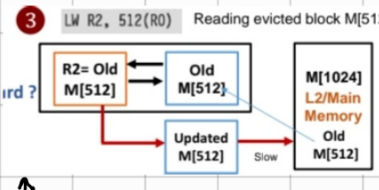
Read After Write (RAW) hazard:

→ Assume direct mapped, write through cache that maps 512 & 1024 to same block & a 4 word write buffer that is not checked on a Read miss.

↳ CPU updates M[512] by updating in write buffer cuz main mem is slow.

Read Miss and Updated M[512] from L1 is evicted and Still Buffer is holding updated value of M[512].

↳ Now CPU wants to store M[1024], so kicks out M[512] but M[512]'s updated value is still in buffer.



Read Miss as there is Read request to evicted block M[512]. **This is RAW hazard.**

↳ Now if we try to access M[512], its value is still not updated, so its a hazard.

→ To overcome this:

* wait until buffer is empty
* check content of write buffer with that particular address. If there is no conflict, then send it to lower level memory. If there is conflict, then write buffer supplies data to the processor.

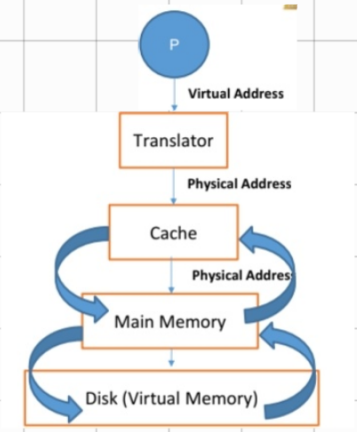
Avoiding Address Translation during indexing of cache to reduce HT

→ Processor might generate addresses that may not be physically present in the main memory.

→ Assume Address BUS size of 32 bits ≈ 4GB, but main memory is only 1GB. This indicates that the address is not physically present. They are called as virtual address

→ These virtual addresses needs to be translated to physical addresses in order to access cache or main memory.

Virtual Memory:



→ Part of system's secondary storage space (HDD/SSD) but acts as though its part of primary storage space.

→ Virtual Memory Space: Memory space addressable by the program & is defined in ISA.

Paging Concept:

→ virtual memory is organized in fixed size blocks called as pages.

ex: A page of 4KiB means page of offset $\log_2(4K) \Rightarrow 12$.

→ Physically indexed & Physically Tagged:

* virtual address is translated to physical address & then using physical indexing into cache & comparing TAGs is done.
* ↑es the hit time.

→ Virtual Caches:

* TAGs & indexing into cache is done through virtual address.
* eliminates address translation time from a cache hit, but no pg protection.

→ Virtually Indexed & Physically Tagged:

* pg offset doesn't change during translation
* ▪ **if the physical Index of the cache is part of page offset then** we can index the cache by using virtual index extracted from virtual page offset directly.
  ▪ This is possible only when Page size ≥ #Sets x Block size of cache.

Example: Let us say page size is 8KB i.e., Page offset will be 13 bits size
If Cache is 32KB, 4 way, with Block size of 64B then
Number of blocks =32K/64B= 512 and Number of sets will be 512/4 =128.
Hence #Sets x Block size of cache = Cache size / ways =32K/4 =8KB