

Computer Organization and Architecture: A Pedagogical Aspect
Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 26
Associative and Multi-Level Caches

In this lecture we continue our discussion on cache memory organization.

(Refer Slide Time: 00:30)

Line Size Vs. Miss Rate

- Larger lines \implies Higher spatial locality \implies Lower miss rates
- Miss rates eventually grow when line size becomes a significant fraction of cache size
 - No. of lines in cache reduce \implies higher competition for lines
 - A line is bumped out of cache before many of its words are accessed
 - Spatial locality among words of the line reduces
- Larger lines \implies Higher miss penalty
 - Miss penalty – time required to fetch a line from the next lower level of the memory hierarchy and load it into cache.

Computer Organization and Architecture 46

We start by discussing the effect of line size on cache miss rates. So, larger the size of cache lines, higher becomes a spatial locality and therefore, lower becomes the miss rates. We discussed about spatial locality. In the last lecture what is spatial locality, the words that have been discussed that have been accessed recently the words nearby them are highly probable to be accessed in the near future. So, when cache lines becomes larger. So, there is a higher probability that words that are nearby will be accessed. So, we get more words which are nearby and therefore, cache miss rates reduce; however, when the cache lines grow from say 8 kb to 16 kb to 32 kb and so on.

When it grows up after a certain time, miss rates eventually grow instead of reducing. Miss rates grow when the line size becomes a significant fraction of the cache size, why does this happen, because the number of cache lines reduced when the size of one cache

line increases, given a fixed size of cache, the number of lines in the cache reduce. So, we have lower number of lines and therefore, higher competition among memory blocks for these lines.

Therefore, cache performance reduce and therefore, a line is bumped out of cache before many of its words are accessed. So, we wanted to increase cache line size, because we wanted to get more words which are nearby words that have been accessed recently; however, before those words which have now come within the vicinity of due to a larger cache size, can actually be accessed, they are bumped out why, because number of lines in cache are now much lower. So, therefore, effectively spatial locality among words of the line reduces.

Now larger lines also means a higher miss penalty and what is the definition of a miss penalty, it is the time required to fetch a line from the next lower level of memory, let us say main memory and cache. For example, here it will be main memory, we can have multiple levels of cache as well. Therefore, in general I have written it as lower level of memory in the hierarchy and load it into cache. So, miss penalty is the time required to fetch a line from the next lower level of memory hierarchy and load it into cache

(Refer Slide Time: 03:32)

Line Size Vs. Miss Rate

- **Time to fetch a line has two parts**
 - Latency to the first word
 - Transfer time for the rest of the block from the lower level memory
- **Larger line size → Higher transfer time**
 - Improvement in miss rate also reduces with very large lines
 - Thus, cache performance reduces when lines are too large
- **Line size can be increased to some extent with better block transfer techniques**

Computer Organization and Architecture 47

Now this time to fetch this miss penalty has two parts to it; one the latency to fetch the first word from the memory into the cache and the transfer time for the rest of the block from lower level memory to the cache. Now larger cache line size, larger cache line size

means higher transfer time, because you have to transfer more data from the memory into cache. So, improvement in miss rate also reduces with large size caches. Now along with this higher transfer time improvement in miss rates also reduces as we said that with significantly large size cache caches, when they become comparable to the size of the cache line size.

Then the miss rates also, miss rates also increase that is improvement in miss rate reduces. Thus cache performance reduces when line sizes are too large; however, this higher transfer time if we can reduce. So, one aspect of due to which the performance reduces can be checked if the, if the transfer time be reduced. So, line size can be increased to some extent with better block transfer techniques and here we will discuss two block transfer techniques; one is early restart that is resume execution, as soon as requested word is returned instead of waiting for the entire block.

(Refer Slide Time: 05:00)

Improved Transfer Techniques

- **Early Restart – resume execution as soon as requested word is returned instead of waiting for entire block**
 - Works best for instruction accesses because they are largely sequential
 - If the transfer rate is ~1 word per clock cycle, the memory system may be able to deliver new instruction words to the processor just in time
 - Less effective for data caches because data word requests are typically less predictable
 - Request for an word from a different cache line before completion of transfer, is high
 - Processor stall if data cache is inaccessible due to an ongoing transfer
- **Critical word first - a more advanced technique**
 - Transfer requested data first
 - Then transfer rest of the block starting with the address after the requested word and wrapping around to the beginning of the block.

Computer Organization and Architecture 48

So, in general what happens is, that when we have requested a word from cache and that word is not in cache. We have a cache miss and the entire block is first fetched from main memory or lower level of memory into cache, and then execution starts early restart, says that that instead of doing this resume execution, as soon as the requested word is returned instead of waiting for the entire block, it works best for instruction accesses, because they are largely sequential. So, if the transfer rate is 1 word per clock

cycle, the memory system may be able to deliver new instruction words to the processor just in time.

So, what does it say, if the transfer rate of the, transfer rate from the main memory into cache is about 1 word per cycle per clock cycle then what happens, the processor gets a new instruction every clock cycle. So, so once we have got the word that we need sup. So, that is the, what is the word that we needed the instruction that we wanted to execute in case of instruction access Now the next instruction is the next word in cache in that block. So, we have not waited to get the next word. However, once the processor executes the current instruction, the next instruction is available, because it we have said that it takes about 1 cycle to load 1 word from the memory into cache. So, therefore, we get the of instruction words just in time as they are required.

So, they become, this early restart mechanism could be very effective in case of instruction accesses. However, they are less effective for data caches, because data word requests are typically less predictable, why, because we require sequential instructions one after other one after the other, data accesses may not be such sequential request for a data word from a different cache line before completion of before completion of transferred is high.

So, therefore, in case of early restart what have we said that, we said that we will get the data word in case of data access will get the data word will not wait for the entire block to be transferred from main memory into cache and instead one whenever we get the required data word due to which the miss has occurred we will we will immediately start execution.

Now if we do this, for data word there is a high probability that we will request an word from a different cache line before completion of transfer, before completion of transfer and this will this what will happen, if this happen, if this happens what is the problem and the problem is that, there is one transfer which is currently going on from the memory into cache. Now for the second transfer have to startm, the first transfer has to complete. So, there will be a lot of memory stalls. So, processor stalls, if data cache is inaccessible due to an ongoing transfer, this is the problem.

The next mechanism of to improve transfer rates is the critical word first, this is a more advanced technique early restarts states that we will start getting words of the block from

the start of the block. However, once we get the data word that that due to which cause the miss, we will immediately start execution critical word first says that do not start your transfer from the start of the block, you start the transfer from the data which caused the miss.

So, you first get the data which caused the miss from the memory and start execution and then transfer the rest of the block. So, transfer request data first then transfer rest of the block starting from the address after the requested work and wrapping around to the beginning of the block before proceeding further or we will first quickly discuss about how cache misses are typically handled.

(Refer Slide Time: 09:50)

Handling Cache Misses

- **Handling instruction misses (similar technique for data misses)**
 - If instruction access results in a miss, instruction register content is invalid
 - Address of instruction – content of PC – 4 (PC is incremented in clock cycle)
- **Steps for handling misses**
 - Send value of PC – 4 to memory
 - Instruct main memory to read and wait (pipeline stall) for main memory to complete transfer
 - Write the cache entry, putting data from memory into the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on
 - Restart instruction execution at the first step, which will refetch instruction, this time finding it in cache

Computer Organization and Architecture 49

So, we will talk about how to handle instruction cache misses. A very similar technique is used for handling data cache misses. So, if instruction access results in a miss. So, what has happened, I wanted an instruction, I floated, I floated the PC with an address handling instruction misses. So, next we will talk about how to handle cache misses, we will specifically talk about how to handle instructions misses, a very similar technique will be used for handling data misses.

So, when we do not find the instruction the next instruction; that is to be executed by the processor in cache and instruction miss results and then the instruction register content becomes invalid. So, what is the address of the instruction, then that has to be fetched from memory, because it is not in cache, it is the content of PC minus 4, because

in the first cycle of, in the first cycle of execution PC is incremented. So, so the value of PC is 4 plus the instruction for a 32 bit computer is 4 plus the instruction that I want from memory. So, the address of the instruction actually is content of PC minus 4.

So, what are the steps for getting this instruction from the main memory, the value of PC minus fold is first sent to the memory, then we instruct main memory to read and wait pipeline stall. So, we instruct main memory to read and then the processor waits; that is that; that means, pipeline stalls happen. So, what is the meaning of a pipeline stall, temporarily the all the data and all the contents are frozen and the processor does nothing, it just waits until the data or the instruction is obtained. So, instruction, the processor instructs the main memory to read and then it itself waits for the main memory to complete transfer, then in during the transfer we write into the cache entry, what do we do?.

We put data from memory into the data portion of the entry and the remaining tag field is then populated into the tag and in the tag part of the entry. So, write into the tag part of the entry and we turn the valid bit on. So, once both the tag and the data has been put into cache we get the va. We put the valid bit on and say that the content of cache is now valid. Then we restart instruction execution at the first step of the pipeline. This will re fetch the instruction and this time it will find it in cache. So, this is how it will handle a miss ha handling writes is a bit more complicated than this.

(Refer Slide Time: 13:17)

Handling Writes

- **Inconsistency between cache and memory**
 - Suppose on a store instruction, we wrote the data only into the data cache without changing memory
 - Thus, memory and cache will have different values after the write – *inconsistency*
- **Write-through** – A scheme that keeps cache and memory consistent by always writing the data to both memory and cache
 - On a write miss
 - Fetch block from memory into cache
 - Overwrite word that caused the miss into cache line
 - Also write the word to main memory using full address

Computer Organization and Architecture 50

So, how do we specifically handle write, we before starting as to what how to handle writes, we first talk about what is in consisting what, what is the meaning of inconsistency between cache and memory. Suppose on a store instruction we wrote the data only into the data cache without changing memory. Thus memory and cache will have different values after the write and this is what we term as inconsistency.

So, during a store we have written the data into cache, but this corresponding data, we have not. This updated data which is there in cache has not been populated into the main memory or the lower level memory. So, therefore, the lower level memory block and the cache line data corresponding to this word becomes inconsistent. Now we have two types of write policies; the first is write through.

In the write through it is a scheme that keeps cache and memory consistent by always writing data to both memory and cache. So, the inconsistency that we were talking about before does not happen here, because whenever we write into cache we also write into memory. So, on a write miss what happens, we fetch the block from memory into cache first, we overwrite the word that cause the miss into the cache line and we then also write the word to main memory using the full address.

So, now, every write also effects a memory write as well, every write into the cache is also a write into the memory why? Because we are keeping everything consistent if the write is a miss, it is not there in cache, then the cache block is first fetched from memory into cache, then the word which caused the miss is overwritten into that line and then that word is also written back into the corresponding memory block.

(Refer Slide Time: 15:40)

Write-through (WT)

- **Often results in inefficient performance**
 - Every write causes data to be written to main memory
 - Memory writes take a long time, at least ~100 clock cycles
 - If CPI without cache misses was 1.0 and if 10% instructions are stores,
 - **Effective CPI becomes: $1.0 + 100 * 10\% = 11$**
 - Performance reduction by a factor of 10
- **One solution is to use a write buffer**
 - A queue that holds the data while it is waiting to be written to memory
 - Execution can continue after writing the data to cache and write buffer
 - After completing the write to memory, write buffer entry is freed

Computer Organization and Architecture 51

This policy has significant overheads and therefore, results in inefficient performance, why is that so? Every write causes data to be written into main memory. So, memory writes as we know takes a long time, at least say 100 clock cycles. So, if the CPI without cache misses was 1.0 and if 10 percent of the instructions are stores then the effective CPI becomes 1 plus 100 into 10 percent, which is 11. So, therefore, the performance reduction happens by a factor of 10.

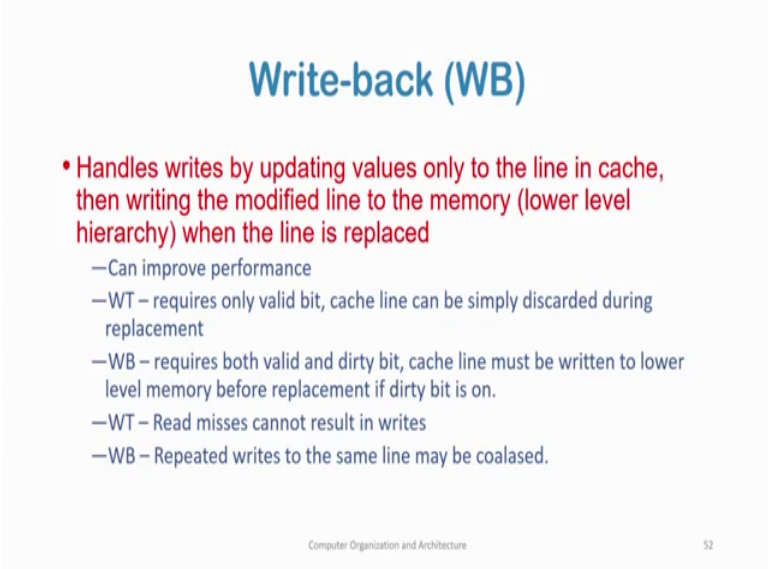
So, here one is is the CPI without any write misses ok, without any writes in it, because all write we will go into memory. So, without writes the CPI is 1 for every write I incur 100 cycles, because I go into memory and write the word also into memory. If I have 10 percent store instructions, each of these instructions will cost me 100 cycles and therefore, if for 10 percent instructions the, with 10 percent store instructions the effective CPI becomes 11 and the performance degrades by a factor of ten.

So, one solution to circumvent this problem is to use a write buffer. So, what is the write buffer? a write buffer is basically a queue that holds the data while it is waiting to be written into memory. So, execution can continue after writing the data into the cache and write buffer. So, what happens here? We are not writing every write into memory and then proceeding again. Once we have written into the write buffer we proceed with the execution and do not wait for that word being written into memory ok.

So, after completing the write to memory the write buffer entry will be freed; however, memory stalls will still happen, meaning that after a write. So, during a write what happens, I am writing it into cache and also into the write buffer and once the write buffer is completes the write into the memory that entry from the write buffer is removed. However, if there are too many writes in a burst from the memory what happens the write buffer gets full, because write buffers are typically of the size of 2 to 4 words.

In any case if the if the rate at which the processor writes into the cache is higher than the rate at which the transfer can be made into memory, whatever the size of the write buffer we will still have a stall of the processor, because the write buffer will be full; however, typically that does not happen writes are on an average scattered and therefore, it has been seen that the write buffer sizes of about 2 to 4 words seem to be sufficient in the typical case.

(Refer Slide Time: 19:12)



Write-back (WB)

- Handles writes by updating values only to the line in cache, then writing the modified line to the memory (lower level hierarchy) when the line is replaced
 - Can improve performance
 - WT – requires only valid bit, cache line can be simply discarded during replacement
 - WB – requires both valid and dirty bit, cache line must be written to lower level memory before replacement if dirty bit is on.
 - WT – Read misses cannot result in writes
 - WB – Repeated writes to the same line may be coalased.

Computer Organization and Architecture 52

The other policy; so we talked about the write through policy in which whenever I write, I write both into the cache and the memory or as a slight deviation we write into the cache as well as a write buffer. Write back works differently, it handles writes by updating values only to the line in cache and then writing the modified line to the memory; that that is lower in the hierarchy in the main memory or maybe the, may be a

lower level of cache when the line is replaced. So, on a write we only write into the first level cache.

We do not bother about lower level memories at all, when this cache has to be replaced, this cache line when it has to be replaced, I see I see that the data in cache has been modified and therefore, is inconsistent with the memory, at that time I write back whatever is there in the cache line into the memory first, and then replace that line. We do not replace before we have we have we have transferred a modified line in cache into the memory.

Now write back caches as we understand can improve cache performance why is that so, because if I have say repeated writes into the same block, it can be call as together. If the same word is written again and again it can be call as together ok. So, we and for many write we finally, transferred it once together. So, this is why write back is typically better than write through caches; however, implementation of write back caches are more difficult as we will see. So, for write through caches we require only one valid bit. The cache line can simply be discarded during replacement.

So, why is that so. In a write through cache we are ascertain that the cache and the memory is always consistent. So, when we need to replace a cache line with another block of memory, then we just need to discard this cache line, because whatever is there in the cache is that up to date information is also there in the memory. For write back caches; however, we require 2 bits; one a valid bit and also another a dirty bit. So, the valid bit just tells that the that the word in memory is valid is, is correct. The dirty bit says that the cache line, the dirty bit says that whether the data after it has been brought from memory has been modified or not.

So, if a word has been modified after it has been brought from memory, then this dirty bit will be 1, this is required for write back caches, because I am not writing back every time I am write into the cache. So, I am writing to the cache and I am not updating that to the memory, but I am setting the dirty bit on. So, it requires both valid and dirty bit. Cache line must be written to lower level memory before replacement if the dirty bit is on. For write through caches read misses, read misses cannot result in writes ok.

Now when I have a read miss, what is what is the problem. I have a read missing cache; that means, somebody else. So, some other cache line is now being held in the cache. So,

whatever I want the block of memory I want is not in the cache, the cache line is filled with some other memory block. So, therefore, I need to replace this line. So, when I have a write through cache, I just discard this line from cache, everything is consistent and I write whichever, and I just transfer whichever block is necessary from memory into cache. For write back caches, this is not possible as we have seen.

(Refer Slide Time: 23:41)

Write Allocate Vs. No Write Allocate

- Consider write misses in a WT cache
- Write allocate – the block is fetched from memory and then the appropriate block is overwritten
- No write allocate – Do not allocate new cache line; directly update the appropriate portion of the block in memory
 - Use case: Say when OS initializes an entire page of memory
- Efficiently implementing stores in WB cache is more difficult
 - On a write miss, WB scheme must first check if the data in cache is modified
 - If so, the modified data must first be written to lower level memory
 - No problem for WT caches as cache is always consistent with main memory

Computer Organization and Architecture 53

Now, for write through caches we can either have write allocate or no write allocate policy. So, consider write misses in a write through cache. So, in a write allocate policy. So, we are considering write misses in a write through cache. So, in a write allocate policy, the block is fetched from memory and then the appropriate block is overwritten. in a no write allocate policy we do not allocate new cache line, we directly update the appropriate portion of the block in memory.

So, what has happened here, in a write through cache we do not have a write hit; that means, the data that I want to write into cache; that data is not present in cache that particular cache line holds somebody else, some other memory block, so therefore, I need to replace. So, in a allocate policy I first replace from memory into the cache line write and then I override. So, the block is fetched from memory and then the appropriate block is, appropriate data is overwritten into both the cache and the memory block.

In a low write allocate policy, I do not bring in and bring in the new cache line, I directly write the word that is to be written directly into the memory. So, I do not bring I bro, I do

not bring from memory the cache line. So, when is this advantageous a typical use case is that, say when the OS initializes an entire page of memory. So, it has an entire page of memory, it will fill it with zeros. So, a full block of memory is going to be filled with zeros.

If we bring that whole memory block into cache, I have to write each word into cache and then I have to also write each word into the memory and this is an overhead. Instead of that in a no write allocate policy, I do not bring the cache line at all and. What do I do is, I directly write those words directly into memory, not bringing the memory block into cache. So, this helps save overheads. Efficiently implementing stores in write back cache is more difficult. For a write through cache it is much simpler.

On a write miss what will happen. I will try to write my data into cache, if that cache line is not the one into which I should be writing, the memory block is different. Then at most what will happen, I will write in eh on a wrong word and then I will see that this is a write miss and I would replace the cache and maybe write again. This is not a problem because the memory is always consistent with the cache. In a write back cache as we said the cache is not always consistent with the memory.

So, if I write without seeing anything, whether the data in the cache is modified, I have the problem of or the risk, I have the risk of overwriting some data which has not been actually written into memory. So, I have the risk that this write. So, I have not checked whether the data in the cache has been modified or not. So, this word, the dirty bit for the word is on or not. If the dirty bit for the word is on, this means that this means that this word has not been written into memory ok. If the dirty bit for the word is on this word has not been written into memory.

Now if I do not check that and write my data into a write back cache, I have the risk of overwriting inconsistent data. So, therefore, in write back caches, I first need to check, whether this is the cache miss, whether this is a write miss or not. If this is a write miss; that means, the cache line that I am writing into is not the one corresponding to the memory block in which I should write, then I should first make the memory and seem.

Ah cache consistent, then replace the cache line and then write it into the proper cache line ok. So, therefore, implementing stores efficiently in a write back cache is more difficult compared to a write through cache.

(Refer Slide Time: 28:49)

Measuring Cache Performance

- **Components of CPU time**
 - Program execution cycles (includes cache hit time)
 - Memory stall cycle (mainly from cache miss)
- **With simplifying assumptions**
 - Memory stall cycle = (Memory accesses / Program) × Miss rate × Miss penalty
 - Memory stall cycle = (Instructions / Program) × (Misses / Instruction) × Miss penalty
- **Average memory access time = Hit time + Miss rate × Miss penalty**

Computer Organization and Architecture 54

Now we will take a few examples for measuring cache performance. We will do some small sums to evaluate the performance of cache. To start with that, we will talk about the components of CPU time. We will first do a simplifying assumption, saying that CPU time has just two components; one is the program execution time and this we will assume to include cache hit time, as we will say that cache hit time is part of the processor execution processor executing, processor execution of instructions hm.

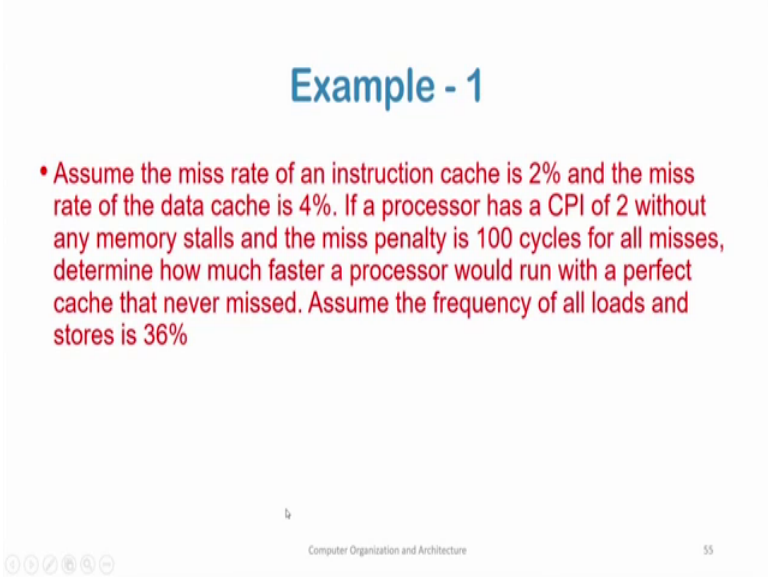
We will not take hit time separately and along with that there will be memory stall time, memory stall cycles mainly due to cache misses. So, the CPU time required to execute a program will be composed of two components; one is the program execution cycles that will include cache hit time and memory stall cycles, this will include cache misses. Now with the simplify with simplifying assumptions, we will say that memory stall time is equal to memory accesses per program, number of memory accesses per program into miss rate into miss penalty. So, miss rate is how many miss rate or miss ratio. So, how many out of the total number of memory accesses what percentage results in misses in the cache.

So, why we are saying that this is a simplifying assumption. See that we said that reads are typically lowered in cost with respect to writes, but we have assumed here that for all memory accesses miss penalty is same. So, in this is the simplifying assumption that we have done and we can also. In other words we can also say memory stall cycles as

number of instructions per program into number of misses per instruction into miss penalty.

So, this is the other way of saying how much is the memory stall cycle and the total CPU time of the program will be program execution cycles per memory stall cycle. Now if we consider a hit time, if we take hit time into consideration, then the average memory access time will be given by hit time plus miss read into miss penalty. So, we will take four examples, the first of which is as follows.

(Refer Slide Time: 31:31)



Example - 1

- 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%

Computer Organization and Architecture 55

Assume the miss rate of an instruction cache is 2 percent and the miss rate of the data cache is 4 percent. So, miss rate of the instruction cache is 2 percent and miss rate of that on the data cache is 4 percent. Here that therefore, we are assuming that we have a split cache and not a unified cache, we have separate instruction caches and separate, we have separate instruction cache and separate data cache. If a processor has a CPI of 2 without any memory stalls right.

So, for a perfect, the perfect CPI without stalls memory stalls and the processor executes absolutely smoothly, then the CPI is 2. So, cycles per instruction is 2, I require 2 cycles to execute one instruction, 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. So, assume the frequency of all loads and stores is 36 percent. So, we have assumed here that loads and stored that is reads and writes have the same penalty

and the frequency of loads and stores is 36 percent out of the total number of instructions executed.

(Refer Slide Time: 33:03)

Example - 1

* 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%

- Let I be the total number of instructions in the program
- No. of memory stall cycles due to instruction access misses = $I \times 2\% \times 100 = 2 \times I$
- No. of memory stall cycles due to data access misses = $I \times 36\% \times 4\% \times 100 = 1.44 \times I$
- Total no. of memory stall cycles = $2 \times I + 1.44 \times I = 3.44 \times I$
- Thus, total no. of cycles per instruction including memory stalls = $2 + 3.44 = 5.44$
- Therefore, (CPU time with stalls / CPU time with perfect cache) = $5.44 / 2 = 2.72$

Computer Organization and Architecture 60

So, we proceed by saying that let I be the total number of instructions in a program. So, the number of memory stall cycles due to instruction access misses. So, I am trying to access instruction I will miss, how many times, we said that 2 percent the miss, the catch is 2 percent. So, miss rate of an instruction cache is 2 percent. So, 2 percent of all instructions result in misses. So, I into 2 percent and for each such, each such miss the miss penalty is 100 cycles. So, So, what is the total number of memory stall cycles due to instruction access misses I into 2 percent in 200 equals to 2.0 into I . Number of memory stall cycles due to data access misses, we said that 36 percent are data accesses, out of the total number of instructions.

So, I is the total number of instructions out of which 36 percent are data accesses, out of these data accesses 4 percent are misses and the all such misses take 100 cycles. So, the memory stall cycles due to data access misses becomes 1.44 into I . So, the total number of memory stalls composing both instruction access misses and data access misses become 2 into I plus 1.44 into I equals to 3.44 into I .

Thus the total number of cycles per instruction including memory stalls become 2 plus 3.44 equals to 5.44 per instruction. So, for the perfect CPU, when there are no memory stalls perfect processing, we have 2 and we have additional 3.44 per instruction. So, $5.44 / 2 = 2.72$ we have

memory stalls. So, the total number of cycles per instruction become 2 plus 3.4 equals 5.44. So, therefore, CPU time with stalls divided by CPU time with perfect cache is 5.44 by 2 equals to 2.72, this is what was required.

So, the question was how much faster a processor would run with a perfect cache that never missed. So, how much faster would it run? it would run 2.72 times faster than a than a processor which misses at this rate, in as shown in this example.

(Refer Slide Time: 35:54)

Example - 2

- We have now made the processor faster and the CPI has reduced to 1 from 2.

— Thus, total no. of cycles per instruction including memory stalls = $1 + 3.44 = 5.44$
— The system with perfect cache will be $(4.44 / 1 =) 4.44$ times faster
— Percentage of time spent on memory stalls will rise from $(3.44 / 5.44 =) 63\%$ to $(3.44 / 4.44 =) 73\%$

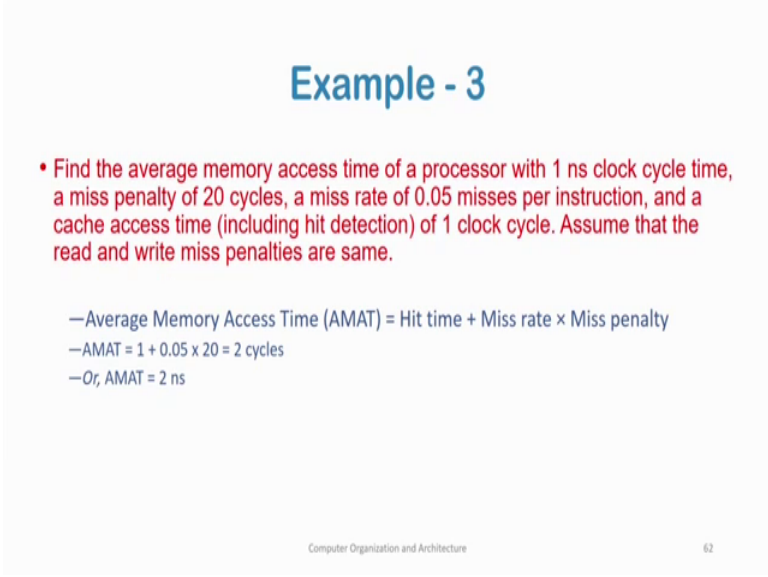
Computer Organization and Architecture 61

Example 2 we the same instruction will now see what happens when we make the CPU faster, we will see we will see that the effect of memory stalls become more pronounced. So, we have now made the processor faster and the CPU has, CPI has been reduced from 1 to 2. So, reduce to 1 from 2. So, therefore, now I take only 1 cycle per instructions. Previously I took 2 cycles per instruction. So, the total number of cycles per instruction including memory stalls will now become 1 plus 3.44 equals to 4.44 ok.

So, there is a mistake in this slide, this 5.44 will be 4.44 in the first line. The system with perfect cache will therefore, be 4.44 by 1 equals to 4.44 times faster now. So, previously thus the system with perfect cache was 2.72 times faster. Now the system with perfect cache will be 4.44 times faster. So, therefore, the effect of memory stalls have become more so pronounced in this case. The percentage of time spent on memory stalls will rise from 63 percent; that is 3.44 by 5.44.

So, the percentage of time spent on memory stalls. So, 3.44 out of the total of 5.44, I wait for memory, I wait on memory stalls right. So, 63 percent of the time I wait on memory stalls. Now when the CPU, when the CPI becomes faster, when the machine becomes faster the CPI reduces, then the effect of memory stall becomes more pronounced. So, now, this ratio increases to 3.44 divided by 4.44 and which is 73 percent of the total time. So, now in the in the faster process processor I will wait 73 percent time on memory stalls.

(Refer Slide Time: 38:15)



Example - 3

- Find the average memory access time of a processor with 1 ns clock cycle time, a miss penalty of 20 cycles, a miss rate of 0.05 misses per instruction, and a cache access time (including hit detection) of 1 clock cycle. Assume that the read and write miss penalties are same.

—Average Memory Access Time (AMAT) = Hit time + Miss rate × Miss penalty
—AMAT = 1 + 0.05 × 20 = 2 cycles
—Or, AMAT = 2 ns

Computer Organization and Architecture 62

Now, we come to the third example, what we have to do is the following. Find the average memory access time of a processor, average memory access time of a processor with 1 nanosecond clock cycle time a miss penalty of 20 cycles a miss rate of 0.05 misses per instruction and a cache access time including hit detection of 1 clock cycle. So, we assume that read and write miss penalties are same again in this case. So, as we said before the average memory access time which we also called AMAT in short is hit time plus miss rate into miss penalty.

So, we said that the hit time is 1 clock cycle. My clock cycle length is 1 nano second and my hit time is 1 clock cycle, cache access time including hit detection is 1 clock cycle and our my miss rate is 0.05 so. So, now, out of 100 instructions, out of 100 instructions only 5 instructions miss and each such miss causes a penalty of 20 cycles. So, the total miss penalty becomes 2 cycles which is 2 nanoseconds. So, then sorry the average

memory access time AMAT becomes 2 cycles or ah; that means, 2 nanoseconds, because 1 cycle is 1 nanosecond in length the last example, we will take is the following.

(Refer Slide Time: 39:59)

Example - 4

- The memory access time is 1 nanosecond for a read operation with a hit in cache, 5 nanoseconds for a read operation with a miss in cache, 2 nanoseconds for a write operation with a hit in cache and 10 nanoseconds for a write operation with a miss in cache. Execution of a sequence of instructions involves 100 instruction fetch operations, 60 memory operand read operations and 40 memory operand write operations. The cache hit-ratio is 0.9. The average memory access time (in nanoseconds) in executing the sequence of instructions is _____

— Time taken for 100 instruction fetch operations (fetch = read) = $100 \times ((0.9 \times 1) + (0.1 \times 5)) = 140 \text{ ns}$
— // 1 corresponds to time taken for read when there is cache hit
— // 5 corresponds to time taken for read when there is cache miss
— // 0.9 is the cache hit rate

Computer Organization and Architecture 64

The memory access time is 1 nanosecond for our read operation with a hit in the cache. So, for a read operation when I have a hit in the cache, I take 1 nanosecond memory access time. Memory access time is 5 nanoseconds for a read operation with a miss in the cache. Memory access time is 2 nanoseconds for the write operation with a hit in the cache and memory access time is 10 nanoseconds for a write operation with a miss in the cache.

Execution of a sequence of instructions involves 100 instruction fetch cycles, 60 memory operands of memory operand read operations and 40 memory operand write operations. So, instruction fetch cycles are how many 100, memory operand read operations are 60 and memory operand write operations are 40. The cache hit ratio is 0.9. So; that means, 90 percent of the times I have a cache hit, 10 percent of the times I have a cache miss and in those times I have to go into the memory.

So, now, I have to find out the average memory access time in nanoseconds in executing the sequence of instructions. So, the time taken for 100 instruction fetch cycles. So, each instruction fetch cycle is a read write. So, for 100 instruction fetch cycles I have 100 into 0.9 into 1. So, one is the, one is the time required to get the get a memory get an

instruction when there is a cache hit ok. So, and 5 is the time required when there is a cache miss.

So, the total time that will be required for 100 instructions is given by 100 into 0.9 into 1 plus 0.1 into 5. So, 1 minus 0.9 is the miss rate, for the cache and this gives 140 nanoseconds. So, one corresponds to time taken for read when there is a cache hit, 5 corresponds to time taken for read when there is a cache miss and 0.9 is the cache hit rate. So, 1 minus 0.9 which is 0.1 is the cache miss rate.

(Refer Slide Time: 42:47)

Example - 4

- The memory access time is 1 nanosecond for a read operation with a hit in cache, 5 nanoseconds for a read operation with a miss in cache, 2 nanoseconds for a write operation with a hit in cache and 10 nanoseconds for a write operation with a miss in cache. Execution of a sequence of instructions involves 100 instruction fetch operations, 60 memory operand read operations and 40 memory operand write operations. The cache hit-ratio is 0.9. The average memory access time (in nanoseconds) in executing the sequence of instructions is _____

– Time taken for 100 instruction fetch operations (fetch = read) = $100 \times ((0.9 \times 1) + (0.1 \times 5)) = 140 \text{ ns}$
– Time taken for 60 memory operand read operations = $60 \times ((0.9 \times 1) + (0.1 \times 5)) = 84 \text{ ns}$
– Time taken for 40 memory operand write operations = $40 \times ((0.9 \times 2) + (0.1 \times 10)) = 112 \text{ ns}$
– // 2 corresponds to time taken for write when there is cache hit
– // 10 corresponds to time taken for write when there is cache miss

Computer Organization and Architecture 66

Now similarly time taken for 60 memory operand read operations also becomes 60 memory operand reads, 0.9 is a hit rate cache hit rate, one is the read, one is the read time, 1 nanosecond, 0.1 is the miss rate and 5 is the miss time. So, total of 84 nanoseconds for 60 memory operand read operations. So, 100 instruction fetch operations. So, instruction read operations take 140 nanoseconds, 60 memory operand read operations take 84 nanoseconds. Time taken for the 40 memory operand write operations will be a bit different.

So, it will be 40 0.9 is the hit rate in cache, 2 is the, 2 is what 2 nanoseconds is the time taken for a write operation when there is a hit. So, 2 nanoseconds it is the time taken for a write operation when there is a hit in cache. So, 40 into 0.9 into 2 nanoseconds plus 40 into 0.1 into 10 nanoseconds, 10 nanoseconds is the time taken when there is a miss in cache.

So, therefore, the total time taken for 40 memory operand write operations become 100 and 12 nanoseconds, 2 nanoseconds 2 a 2 2 2 corresponds to time taken for write when there is a cache hit, 10 corresponds to time taken to write when there is a cache miss.

(Refer Slide Time: 44:32)

Example - 4

- The memory access time is 1 nanosecond for a read operation with a hit in cache, 5 nanoseconds for a read operation with a miss in cache, 2 nanoseconds for a write operation with a hit in cache and 10 nanoseconds for a write operation with a miss in cache. Execution of a sequence of instructions involves 100 instruction fetch operations, 60 memory operand read operations and 40 memory operand write operations. The cache hit-ratio is 0.9. The average memory access time (in nanoseconds) in executing the sequence of instructions is _____

— Time taken for 100 instruction fetch operations (fetch = read) = $100 * ((0.9 * 1) + (0.1 * 5)) = 140 \text{ ns}$

— Time taken for 60 memory operand read operations = $60 * ((0.9 * 1) + (0.1 * 5)) = 84 \text{ ns}$

— Time taken for 40 memory operand write operations = $40 * ((0.9 * 2) + (0.1 * 10)) = 112 \text{ ns}$

— So, total memory access time taken for the (100 + 60 + 40 =) 200 memory operations

- = 140 + 84 + 112 ns = 336 ns

— Average memory access time (in nanoseconds) in executing the sequence of instructions is:

- $336 \text{ ns} / 200 = 1.68 \text{ ns}$

Computer Organization and Architecture 68

So, total memory access time taken for the 100 plus 60 plus 40. So, 200 total memory operations. So, what is the total memory access time for these 200 memory operations; 140 plus 84 plus 100 and 12 nanoseconds equals to 336 nanoseconds. So, average memory access time in executing the sequence of instructions is 336 nanoseconds divided by 200 equals to 1.68 nanoseconds. So, this is the average memory access time, this is this is the total memory access time over 200 instructions. So, the average memory access time becomes 336 divided by 200 and this is 1.68 nanoseconds for each memory access time on average.

With this we come to the end of this lecture.