Computer Architecture

Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

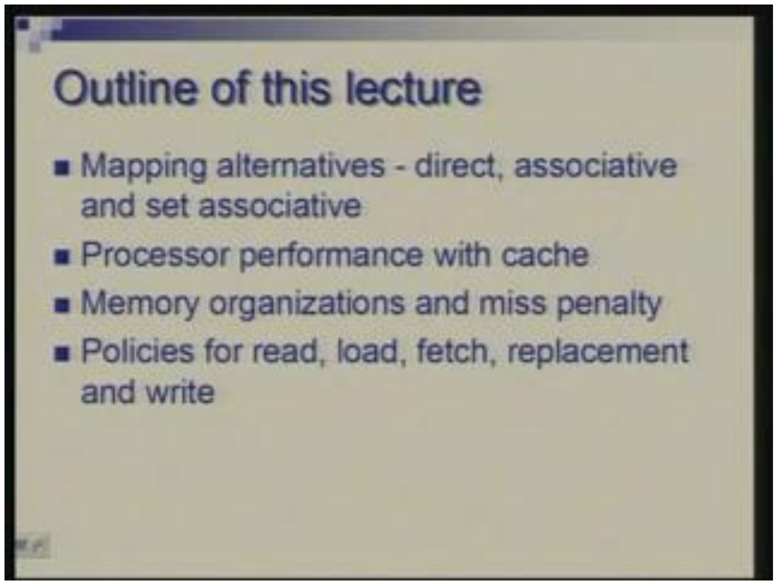
Lecture - 29

Memory Hierarchy: Cache Organization

In the last lecture we discuss the basic principles of memory hierarchy and we noticed that the underlying idea is that there is locality of reference and you can bring data or instructions from lower level memory to a higher level memory and reuse it several times so thereby you gain in terms of the speed of the higher level memory and the effects of getting capacity of the lower level memory.

Today we will specifically focus our attention on cache memory which is the closest to the processor and lies in between the main memory and the processor. So I briefly mentioned about the mapping alternatives. We will elaborate further on those and then we will start looking at the performance issue.

(Refer Slide Time: 00:01:44)

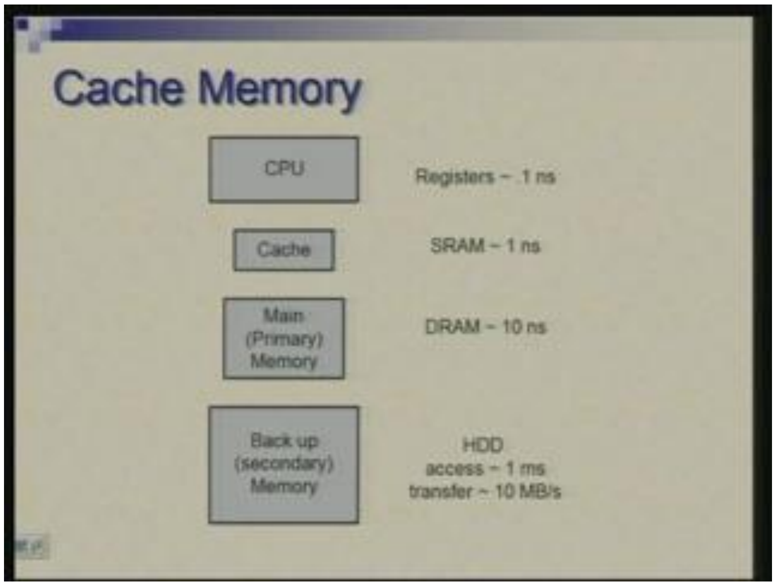


Outline of this lecture

- Mapping alternatives - direct, associative and set associative
- Processor performance with cache
- Memory organizations and miss penalty
- Policies for read, load, fetch, replacement and write

So how does the processor performance or the time to execute a program changes in light of cache accesses, in light of memory hierarchy particularly cache and the main memory. We will see the influence of choosing different memory organizations on the miss penalty. It is the additional time one has to spend when a miss occurs. And we will look at different alternative which exist at various stages: read stage, load stage and so on. So these different alternatives have again different cost and performance implications.

(Refer Slide Time: 00:02:32)



Cache Memory

CPU	Registers ~ .1 ns
Cache	SRAM ~ 1 ns
Main (Primary) Memory	DRAM ~ 10 ns
Back up (secondary) Memory	HDD access ~ 1 ms transfer ~ 10 MB/s

So we are talking of typically three levels of memory. In many cases they are more. You have cache which is closer to CPU; Main memory is the reference point so there is a faster memory closer to CPU called cache and a slower memory but giving large capacity which is beyond the main memory. Typically you have a vast variation in terms of their access times (Refer Slide Time: 3:08).

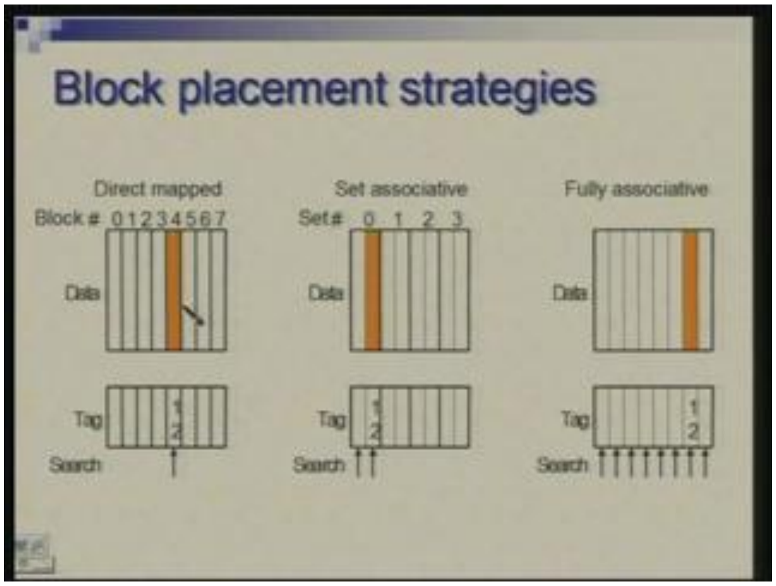
For example, the CPU registers with current technology require only a fraction of nanosecond to access; so of the order of let us say 0.1 nanosecond; these are the figures which change every year so we are looking at these as more as a ball park figure and I am not talking of exact values. Cache which is made of SRAM or the Static RAM is of the order of 1 nanosecond. Now this could be less or more. particularly if cache gets integrated with processor which it does in most cases the value could be lower but there are cache memories off the chip also then it could be even little higher.

The main memory which is basically DRAM or Dynamic RAM technology is typically an order of magnitude slower as compared to cache and the backup memory or the secondary memory which is built around hard disc drive this has much slower..... you can see there are several orders of 5 or 6 orders of magnitude slower but here there are two different kind of timings which are involved: One is time to reach a point where your data exists; you are trying to access some particular information on a disk to reach that point on the disk because it is a moving medium it may take of the order of a millisecond or a couple of milliseconds. But once you reach a point the transfer takes place faster and the rate of transfer could be of that order at some 10 megabytes per second on that order.

So any decision you take about organizing these levels one has to bear in mind that kind of order of magnitude of difference between various levels of technologies which exists.

The first question we came across was how to map the addresses of the main memory on the cache. So, given a data or instruction which resides in some particular address in the main memory where should you place it in the cache and the simplest solution was to take the memory address, modulo the cache size and you get the location where it gets positioned in the cache; so there is a fixed position.

(Refer Slide Time: 00:05:35)

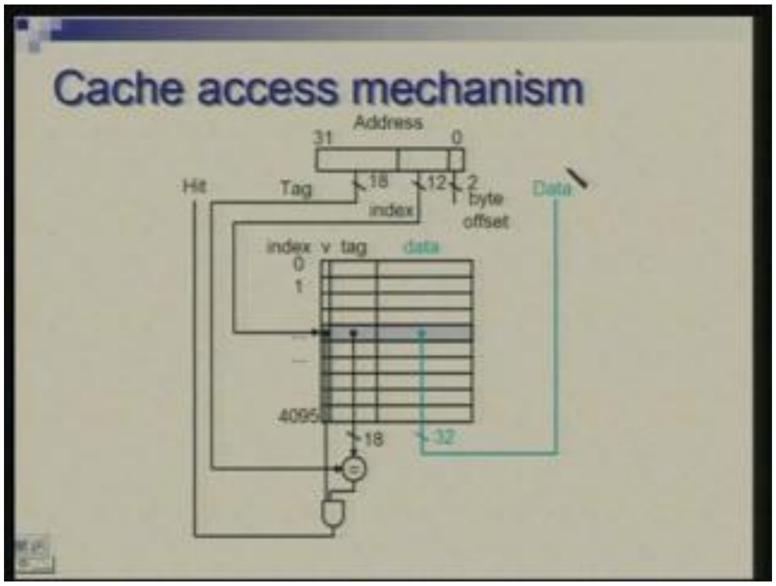


Block placement strategies

The other possibility was that allow anything to be placed anywhere and there is a cost implication of doing so although it gives you maximum flexibility. So typically most often what you have is something which is in between the two which is called set associative. So, given any address you have a range of locations in the cache where it can possibly exist.

What it means is that a set of locations in the memory or set of addresses in the main memory will compete for a small pool but they are not mapped into a single one. So let us say if sixteen locations in main memory compete for four locations in cache still you have little flexibility and at least four of them can reside at any time. So, as you go from direct map to fully associative (Refer Slide Time: 6:35) you get increased flexibility. The degree of associativity and set associative case gives the size of each set. More the degree typically you talk of 2 4 8 16 but generally not more than that; more is the degree of associativity more is the flexibility and eventually it shows up in performance.

(Refer Slide Time: 00:07:04)



Cache access mechanism

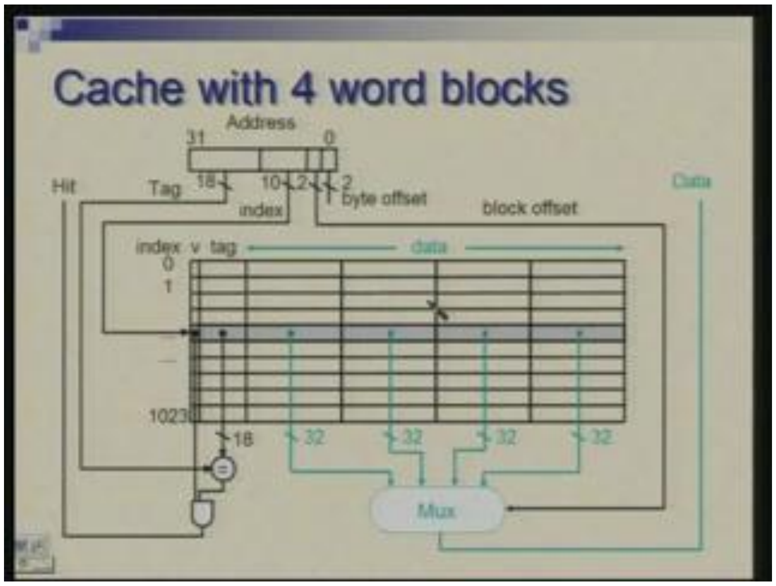
I have redrawn the diagrams of hardware structure for these and I have tried to have organization show for cache access with direct access mechanism (Refer Slide Time: 7:222) with one block size or larger block size and associative access. So let us look at it one by one.

In all these I have kept the cache sizes as same and you could see that the field sizes in the address will vary accordingly. So now I am assuming a 4k words cache size and main memory address we are assuming is a 32-bit byte address. So you have 4 Gigabytes and here you have 4k word which means 16k bytes; we are talking of 16k byte cache memory and in direct access case this gets addressed by a 12-bit field here; 12-bit field will select one word and you can actually get 32 words of data from here. So the access will require that you index into one row here, look at the tag; match the tag with the tag field of this. So, different main memory addresses which can map to this cache address this cache location (Refer Slide Time: 8:38) would differ in terms of tag and the word which is sitting here will be identified by the tag value it is holding here. So you match the tag and if tag matches then it is a hit; if tag does not match it is a miss. So, if hit occurs you can read this otherwise you ignore this and do something else.

There is an additional bit which we call as valid bit which indicates whether something has been placed here or not. Because initially you may have an empty cache and let us say as the program begins initially you get lot of misses and the cache will get filled up; after that after it gets filled up, it will be a matter of replacement, you get new words they replace the old word and so on. So you are not only matching the tag you are also looking at the valid bit and if both conditions are met, only then you consider it as a hit.

Now let us extend the size of the block.

(Refer Slide Time: 00:09:40)



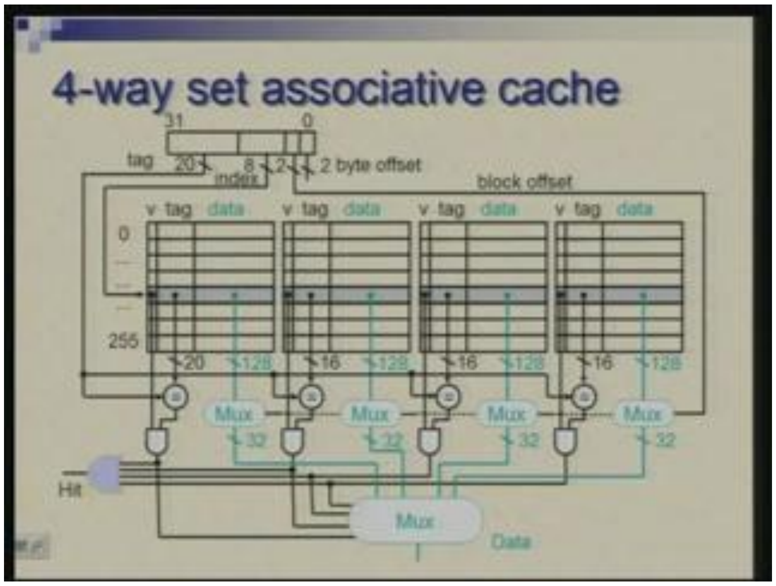
Cache with 4 word blocks

In this diagram, effectively each block is a unit of transfer sorry each word is a unit of transfer, each word is also an accessible or addressable unit here. But now we talk of a block which is let us say 4 words in this case or 16 bytes. So the same capacity of cache would involve now 1k blocks because each block is 4 words. And you can imagine these 4 words of a block to be laid out horizontally. Logically they are forming part of one row and they carry a single tag. so the mapping here is in terms of one block at a time and as misses occur, as you need data or instructions you will typically get one block from the memory with the hope that you are capturing some spatial locality. So, if you are referring to this word it is likely that soon enough you will refer to this word and then this word (Refer Slide Time: 11:00).

As we would see later, that in terms of transfer of data between memory and cache it is more economical or better in performance to move some set of words, not a single word; it is advantageous to transfer blocks of data. So, once you have a miss you might do little more work and reap the advantages later on. So in this case the index field in the address reduces to 10 bits and we will still have same 18-bit tags which need to be matched. So, given an address we will find either its entire neighborhood that entire block there or entire block absent. So 2 bits in the address could be used to select one word out of these four words which you pick up from the cache.

Actually another name for the block is line, cache; you also talk of cache line which means the same thing as block and why it is called a line is because logically you imagine this as one row.

(Refer Slide Time: 00:12:44)



Now let us look at set associative organization. So we are considering in particular 4-way set associative organization. That means each block will have four possibilities; it can appear anywhere within a set. So each horizontal line here forms what we call as a set and there are now 256 sets so again factor 1 by 4 so each sets get indexed by 8 bits. And you would notice that to compensate for that the tag size has become larger. So it is now the identity of a particular block in the set which is now 20 bits because the index has reduced so 2 bits go to the tag. The reason for this..... another way of looking at this is that, now there are actually more words which can actually get placed here; four times as many words which can get placed here as compared to the previous situations. So I am assuming..... the size have shrunk here now; this is one block which means 4 words or 128 bits and the whole thing can be thought of as organization with the four blocks, there are four quadrants here.

So, given an index you are selecting, you are accessing this entire row corresponding to a set and within each a tag matching will be done in parallel. So you are reading out tags, matching with this tag field of the address and match may occur anywhere, any of these matches and therefore we have an OR gate so there is also a valid bit which is being looked at. So, if any of these show a match there is a hit, if none of them shows a match it is a miss.

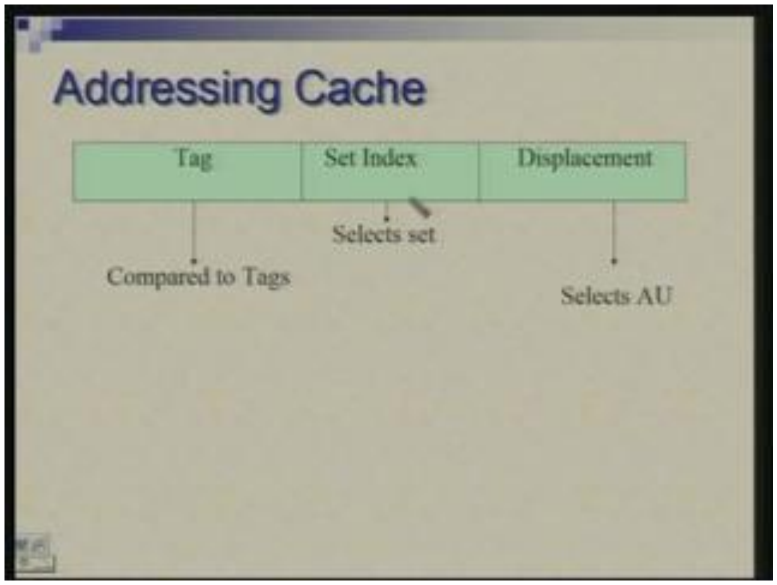
Now, depending upon where a match occurs we would need to pick up a word. But before that what we are doing is, out of the blocks we are picking up the required word. There is a multiplexer with 4 words coming in and 1 word going out (Refer Slide Time: 15:14) each of these and these multiplexers are being controlled by this 2-bit field which we consider as a block offset. So these multiplexers smaller ones are actually same as what I showed here.

And then there is another round of selection done here. So these match signals will enable one of these and that could be finally taken as a data. So you would notice that there is replication of this matching hardware which is an extra thing; there are more bits which you are spending on tags. So these are the overheads of increasing the degree of associativity and the advantages as I mentioned earlier that you are likely to reduce the miss rate. The reason being that, if degree of associativity is low there are more conflicts and therefore some of the blocks which you may like to use later on may get thrown out.

[Conversation between student and Professor: .... any question about these three organizations ? 3? oh that is a typographic error. Actually all are..... yeah this is a mistake (Refer Slide Time: 16:39); all are 20-bit; this is carried from some other diagram. Yeah please correct this; these are all 20-bit tags uniformly.

[Student: noise.............(16:57)] well, depend... now.... we are assuming here that you are accessing one word although it is..... I am showing it as a byte address memory but out of this you get one word out or you send one word in. Actually I am showing the read operation; write operation would be similar. So the unit of transfer between this memory system and the processor is one word. If the requirement is of a byte or a half word then out of that you will pick up and if there are misalignments again then multiple accesses get made. So, at the moment I am talking of reading and writing a word with this memory; does that answer your question? Any other question?
Okay.]

(Refer Slide Time: 00:18:01)

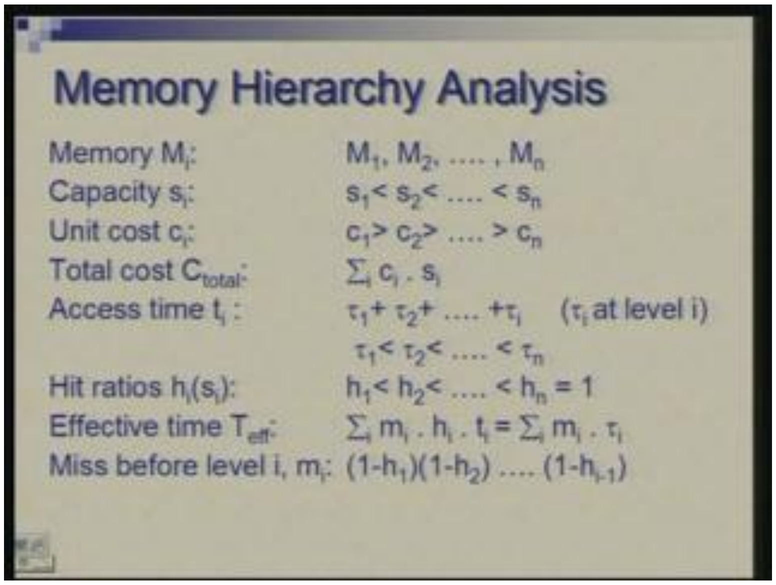


Addressing Cache

So basically just to summarize; we have main memory address divided into effectively three parts: there is one part which we call as tags; one part is set index and other part is some displacement within a block. So the tag gets compared to the tag stored in the cache. Set index is used to select a set and this displacement which could again be two parts: a word within a block and then byte within a word although I have shown it as a single field; this selects an addressable unit; whatever is an addressable unit it could be byte or it could be word it is selected with the help of these bits (Refer Slide Time: 18:36).

Now let us again look at the performance issue. We, in general, try to characterize the performance of a hierarchy n level hierarchy in terms of the time once spent at each level and also the total cost in terms of the capacity or the size at each level.

(Refer Slide Time: 00:18:49)

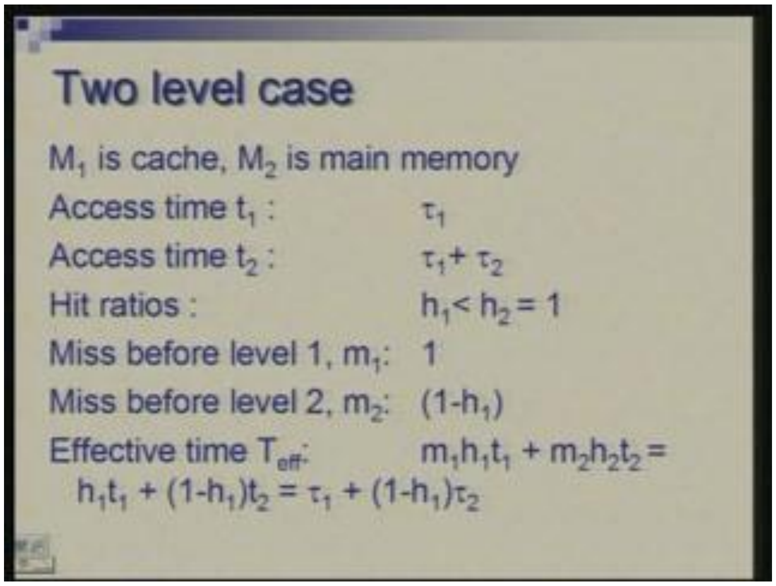


Memory Hierarchy Analysis

Memory M_i:	$M_1, M_2, \ldots, M_n$
Capacity s_i:	$s_1 < s_2 < \ldots < s_n$
Unit cost c_i:	$c_1 > c_2 > \ldots > c_n$
Total cost C_{total}:	$\sum_i c_i \cdot s_i$
Access time t_i :	$\tau_1 + \tau_2 + \ldots + \tau_i$ (τ_i at level i)
	$\tau_1 < \tau_2 < \ldots < \tau_n$
Hit ratios $h_i(s_i)$:	$h_1 < h_2 < \ldots < h_n = 1$
Effective time T_{eff}:	$\sum_i m_i \cdot h_i \cdot t_i = \sum_i m_i \cdot \tau_i$
Miss before level i, m_i:	$(1-h_1)(1-h_2) \ldots (1-h_{i-1})$

So it is a size which we said will influence the cost as well as the performance and therefore by playing around with the capacity or the sizes you could achieve a suitable combination of effective time and the total cost. Now let us simplify this and visualize this into context of two levels.

(Refer Slide Time: 00:19:29)



Two level case

M_1 is cache, M_2 is main memory

Access time t_1 :	τ_1
Access time t_2 :	$\tau_1 + \tau_2$
Hit ratios :	$h_1 < h_2 = 1$
Miss before level 1, m_1:	1
Miss before level 2, m_2:	$(1-h_1)$
Effective time T_{eff}:	$m_1h_1t_1 + m_2h_2t_2 =$

$h_1t_1 + (1-h_1)t_2 = \tau_1 + (1-h_1)\tau_2$

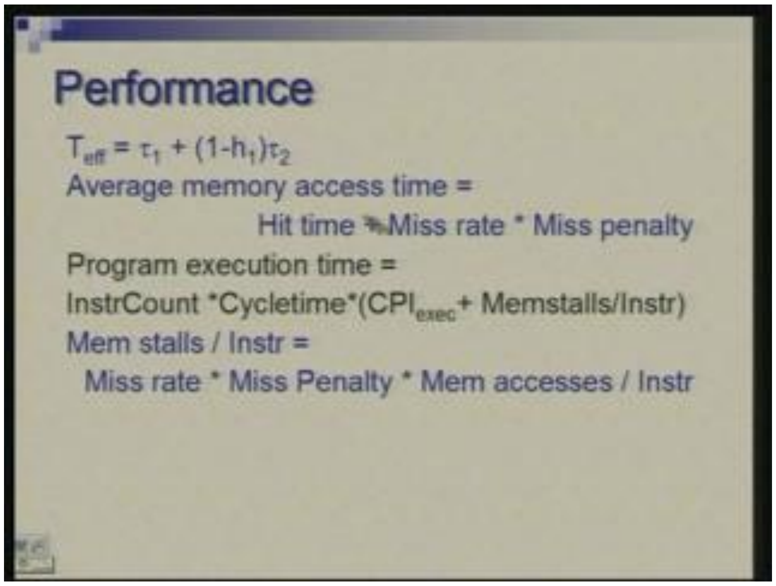
Suppose you are talking of now, a simple case, two levels, m1 and m2; m1 is cache, m2 is main memory. The access time is t1 that means if you were to get data from cache only if there was a hit then as we formulated earlier it is just tau 1 time we encounter. But if we get the data at next level the time t2 is tau1 plus tau2 because we would have made an attempt at cache level failed and then made an access at the main memory level.

So the hit ratios or hit probabilities are h1 and h2 where h2 is 1. we are assuming that there is no further level beyond main memory so whatever you are looking for has to be ultimately found at the last level which is main memory in this case so h2 is 1. The miss before level 1 m1 is always 1 because there is no 0 level so before that its miss we have to start from level 1 only and m2 is simply 1 minus h1.

Now, in terms of these we can express T effective as this was the formula: sigma mi hi ti so m1 h1 t1 plus m2 h2 t2. Now, putting these values we get this as h1 t1 plus 1 minus h1 t2. If you replace t1 t2 by tau values what you will get is tau1 plus 1 minus h1 tau2 so either way you can look up on this. So intuitively you could see this is very straightforward that if there is a hit at the cache the time spent is t1, if there is a miss the time spent is t2. Or one could argue this way that tau1 is the time you in any case spend whether there is a hit or miss. The additional time is spent if there is a miss and that additional time is tau2.

So in more common terms this is often written as average memory access time is the hit time, the time is spent if there is a hit plus miss rate which is 1 minus h1 into miss penalty that is tau2.

(Refer Slide Time: 00:21:41)



Performance

$T_{eff} = \tau_1 + (1-h_1)\tau_2$

Average memory access time =

Hit time + Miss rate * Miss penalty

Program execution time =

InstrCount *Cycletime*(CPI_{exec}+ Memstalls/Instr)

Mem stalls / Instr =

Miss rate * Miss Penalty * Mem accesses / Instr

The term penalty is that this is the punishment you get if you are not able to find things in the cache. So now our ultimate interest is in program execution time; that given a program which executes certain number of instructions how much time it takes and that is what we had mentioned as the ultimate measure of performance. So this will be instruction count * cycle time *..... typically we talk of only CPI but now we say CPI plus the extra cycles which are called memory stall cycles because what happens is that when there is a miss you hold back the CPU; similar to the data hazard you introduce stalls. So you do nothing but waiting for a few cycles till data comes from main memory and is supplied. So the cycle you spend in actually executing the instruction...... normally if there is no miss plus the additional cycle you spend when there is a miss.

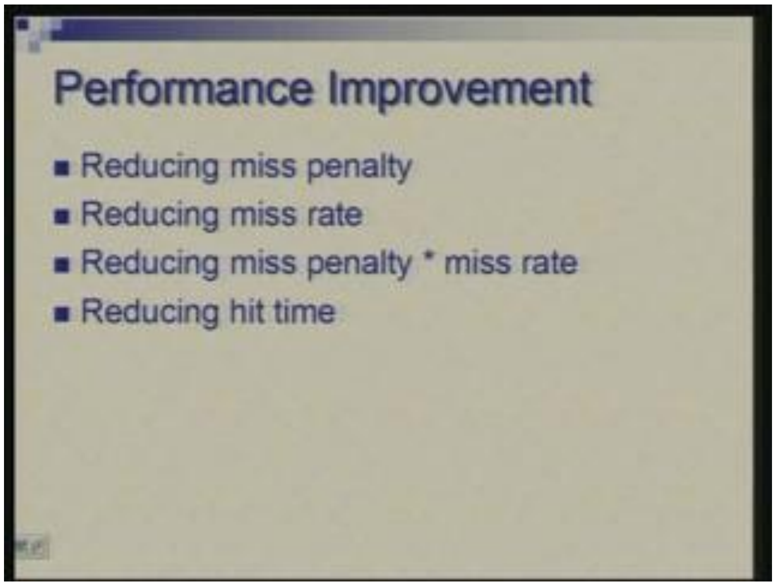
And this factor memory stalls per instruction is a miss rate multiplied by miss penalty multiplied by memory accesses per instruction. So now, memory access per instruction would depend upon the kind of instruction you have and how often various instructions are executed.

So, in our architecture, we have assumed that, each instruction is a one word instruction, so one access is made to fetch that instruction and load and store are instructions which make another access to memory. So memory accesses are either 1 or 2 in different instructions and with appropriate weightage if you find the average you will get 1. something as average number of memory access per instruction. So miss rate and miss penalty are same thing. So when miss occurs this many often (Refer Slide Time: 24:09) the miss occurs every time you incur so many extra cycles and this miss rate has to be seen in conjunction with how many times you make memory access per instruction. So this product gives you memory stalls or the extra cycle you spend per instruction because of misses.

So now in view of this if you were to improve performance from cache organization point of view what is it you could do? So basically you could improve on any of these factors. Actually in the second one, hit time is sort of subsumed in this; hit time is actually subsumed in this. So if there is no miss we have assumed for example that you can access memory in one cycle so that is actually taken into account while counting the cycles of an instruction.

So, overall performance can be improved if you reduce one or more of these things. Or if you do a change which reduces some, but increases something else, then you have to ensure that advantage is more than the loss. So there are different techniques which will either try to reduce miss penalty or reduce miss rate or reduce this product; in the process there may be increasing one but on the whole reduces a product or they could also try to reduce hit time. So we have to keep this at the back of our mind that when you are comparing two or more alternative what does it do.

(Refer Slide Time: 00:25:44)

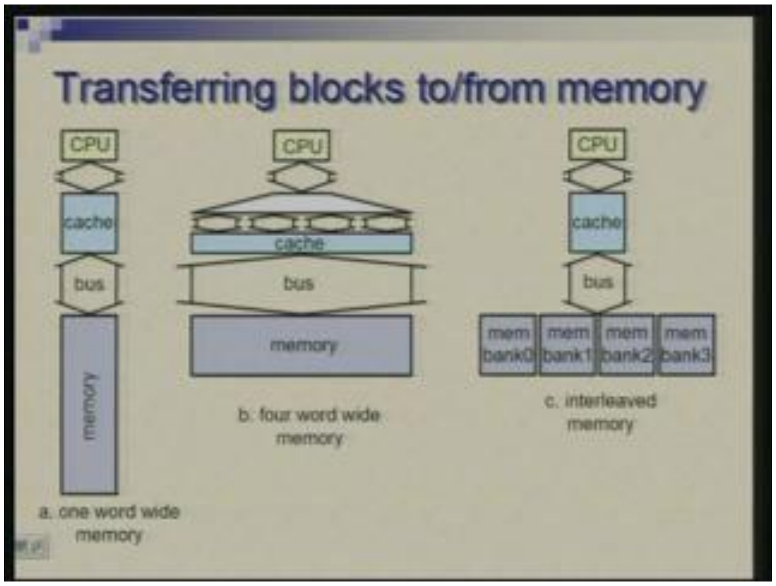


Performance Improvement

- Reducing miss penalty
- Reducing miss rate
- Reducing miss penalty * miss rate
- Reducing hit time

So the miss penalty is dependent upon how you transfer data between memory and cache and we looked at three possibilities that is you have one word wide memory and one word wide bus; other extreme was this one (Refer Slide Time: 26:27) that you have memory as wide as the block and the bus is also equally wide.

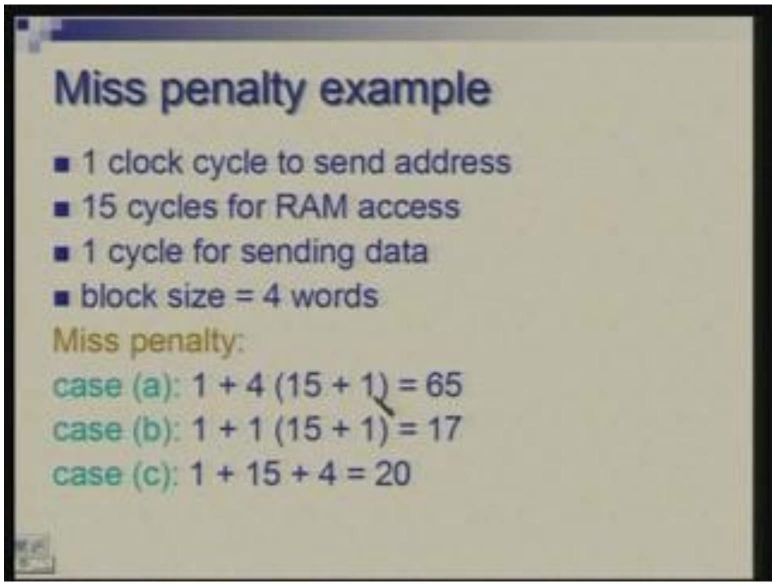
(Refer Slide Time: 00:26:17)



Transferring blocks to/from memory

So this is the fastest possibility but very expensive therefore there is alternative arrangement which is interleaved memory. Well, it is not necessary that the width of the memory here or the degree of interleaving should be same as the block size. Suppose block size is 16 words even if you have 4-way interleaving or 4 word wide memory it will still give us some advantage. So, larger the width here or larger the degree of interleaving better it would be. So let us quantify this to some extent for some cases.

(Refer Slide Time: 27:07)



Miss penalty example

- 1 clock cycle to send address
- 15 cycles for RAM access
- 1 cycle for sending data
- block size = 4 words

Miss penalty:

case (a): 1 + 4 (15 + 1) = 65

case (b): 1 + 1 (15 + 1) = 17

case (c): 1 + 15 + 4 = 20

As an example, suppose you have one clock cycle which is required to send the address to the memory 15 cycles are required for RAM to have access internally. That means after address RAM take some time before it gives data so let us say that is 15 cycles. Now these are CPU cycles; one memory transaction maybe different or the bus cycles maybe different but here we assuming that bus also takes one cycle for sending data. Though the two may not necessarily match, in this example, we are taking figures like this. Also, let us assume that block size is 4 words. So, in the first organization where you had 1 word wide memory and 1 word wide bus you will send address once and then go through four transactions, 4 cycles every time spending 15 cycles for access and 1 cycle for transferring data. So, total number of cycles or miss penalty in these terms of number of CPU cycles will be 65.

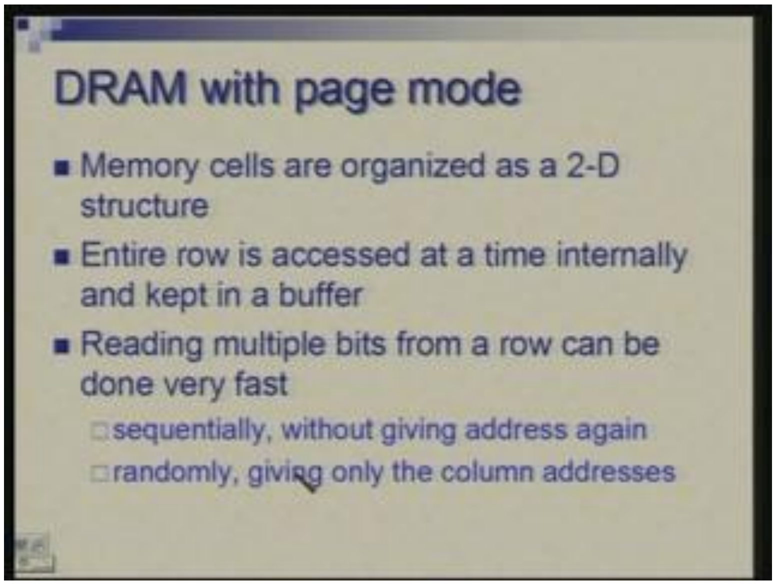
In case (b) this 4 factor reduces to 1. In one shot you will access 4 words, transfer all the 4 words and therefore you spend only 17 cycles.

In the last case, you will make only one access to the memory; all four modules or all 4 banks of memory would be ready with their words within 15 cycles but then you will take 4 cycles in the bus to send them to the cache so 1 plus 15 plus 4 gives you 20.

Now we can also imagine intermediate situation. Suppose in case (b) the memory and bus were 2 words wide then what we will have here is a factor of 2, we get 2 words at a time and we do it twice so this will be 33 which is somewhere in between the two. And if degree of interleaving was 2 here then what you would do is in 15 cycles you will get 2 words out, spend 2 cycles in sending those so 15 plus 2 17 and that 17 has to be repeated twice. So 17 into 2 34 and 1 so it will be 35.

One thing I would like you to notice is that what you are achieving here is closer to this rather than that and in the intermediate case also this is 35 and that is 33 so the difference is not so much but the cost is much lower.

(Refer Slide Time: 00:29:58)



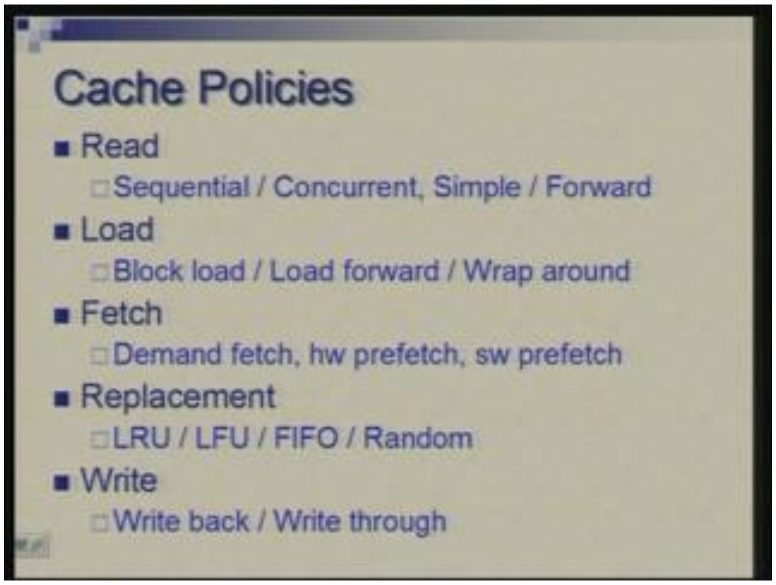
DRAM with page mode

- Memory cells are organized as a 2-D structure
- Entire row is accessed at a time internally and kept in a buffer
- Reading multiple bits from a row can be done very fast
  - sequentially, without giving address again
  - randomly, giving only the column addresses

Here I assume that you physically have two or four or more different memory banks where addresses are interleaved. But in DRAM in Dynamic RAM what happens is that internally it has a structure which gives you similar effect that is called page mode in a Dynamic RAM. In a DRAM the storage cells are organized actually as a two dimensional structure in row and columns and it is organizes such that the addresses has two parts row address and column address so you give the row address an entire row is accessed and kept in a buffer from that buffer you can pick out things faster so time to access one word out of the row which has been accessed and kept in a buffer is much smaller. So effectively what you do is you can think of this as a page which we have accessed and if you are referring to a block of data which lies within this page, you can make faster access.

So first word takes longer but subsequent words come out faster. So reading multiple bits from a row can be done very fast. It can either be done sequentially that is you do not give even column address repeatedly, you give one column address and then consecutive bits can be read out or you can also randomly read bits out of this page or the row by giving only column address, row address is given only once.

(Refer Slide Time: 00:31:45)

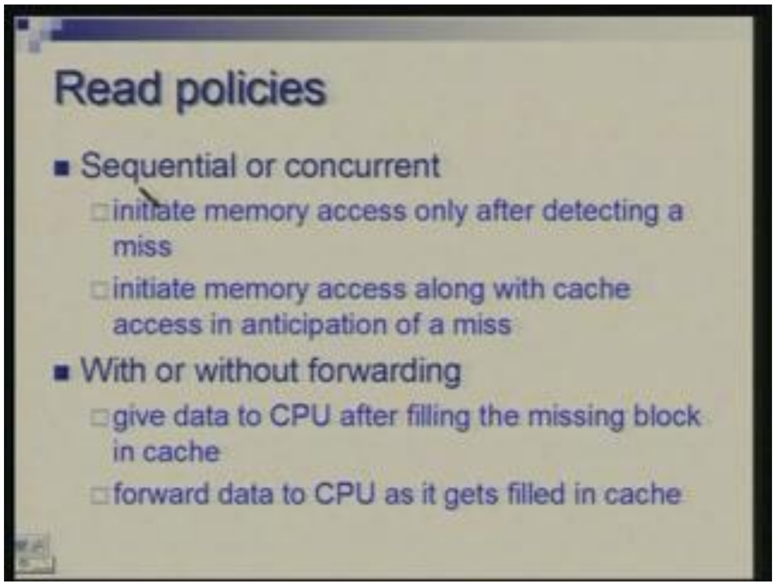


Cache Policies

- Read
  - Sequential / Concurrent, Simple / Forward
- Load
  - Block load / Load forward / Wrap around
- Fetch
  - Demand fetch, hw prefetch, sw prefetch
- Replacement
  - LRU / LFU / FIFO / Random
- Write
  - Write back / Write through

Now this is what we have discussed is the basic operation, within this there are lots of variation which are possible and they have their own implications on performance. so we will talk of variety of ways in which you can do read that means initiate data transfer between memory and cache, load means how the data gets filled up in the cache, fetch means when you initiate a transfer, replacement means when a new block comes how do you choose which block to go away and what happens when there is a write hit or a write miss so that we have not dealt with and there are possibilities there also. So let us look at each of these one by one.

(Refer Slide Time: 00:32:46)



Read policies

- Sequential or concurrent
  - initiate memory access only after detecting a miss
  - initiate memory access along with cache access in anticipation of a miss
- With or without forwarding
  - give data to CPU after filling the missing block in cache
  - forward data to CPU as it gets filled in cache

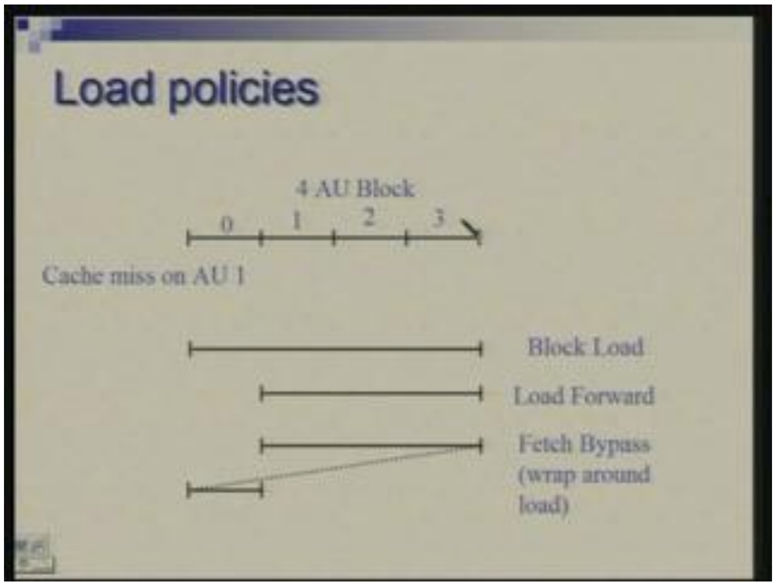
In reading, one variation is that, when you are making an access to the cache, at the same time you might initiate an access to main memory in anticipation. So if cache shows you a hit then you can abort that, if it shows a miss then you can continue so you would gain initial one or two cycle which are required for cache hit so that is called sequential or concurrent read.

Secondly, when you have the data brought to the cache, one approach could be at first you fill up the entire block in the cache and then out of that block you give one word to the CPU for which it had halted so CPU gets stalled for one word. Suppose you are doing load word but because of miss you are getting the entire word the entire block from the main memory. So if these two are sequentialized, that is, first you fill up the cache and then transfer data to processor then it takes longer. What you can do is that as the required word out of the block is getting filled into the cache you can also transfer that to the processor so there is kind of forwarding path can be created so that the data straight comes from the memory to processor. That is with or without forwarding.

[Conversation between student and Professor: 34:23].... it could be typically one cycle two cycles and ideal would be one cycle read, one cycle write but it could be more, multiple cycles and then again things vary whether when you have multiple level of cache so typically first level of cache would like to have synchronized with the processor but subsequent levels may not necessarily be.

Now the question is also of how do you load different words which form a block into the cache. Suppose you are talking of a four word block 0 1 2 3 then the most simple thought would be to bring these into the cache in that order.

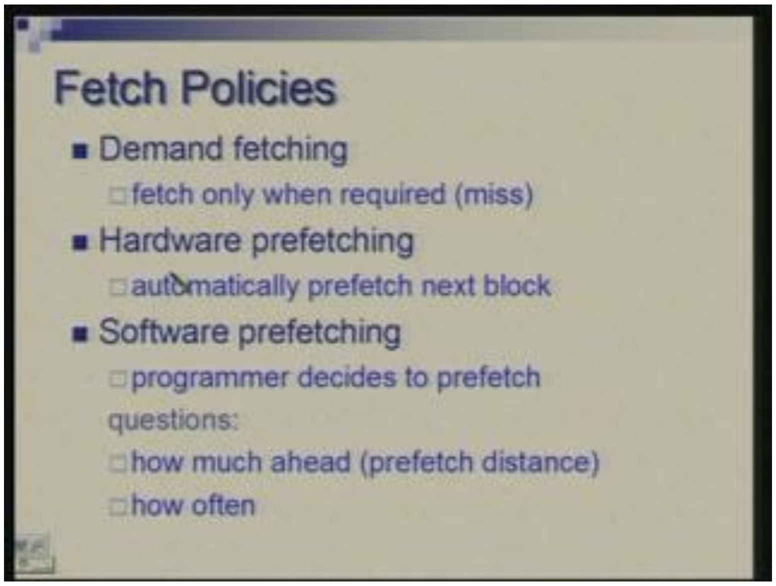
(Refer Slide Time: 00:35:00)



Load policies

But one might say that if the miss occurred for word number 1 within that block then why not start with this. So particularly when you are forwarding the data to CPU it may be advantageous to fill up 1 2 3 in that order. You might just leave the words before it unfilled and the valid bits would have to be now one per word and not one per block. So you can keep those words invalid and these are valid. if you require those then you can fill up them or alternatively you could start in a Round Robin fashion, start with 1, go to 2, go to 3 and then fill up, come back and fill up 0; so in a wraparound fashion you can load all of them but start at the one which you need most immediately.

(Refer Slide Time: 00:36:12)



Fetch Policies

- Demand fetching
  - fetch only when required (miss)
- Hardware prefetching
  - automatically prefetch next block
- Software prefetching
  - programmer decides to prefetch

  questions:
  - how much ahead (prefetch distance)
  - how often

Fetching means when do you start the transfer. Typically what you do is called demand fetching that only when you encounter a miss then you fetch the data so that is demand fetching. But you can again anticipate things and do it before time; ahead of time you can fetch and this could be this prefetch which means getting the information ahead of time in anticipation this could be initiated by the hardware or by software so there could be hardware prefetching or software prefetching.

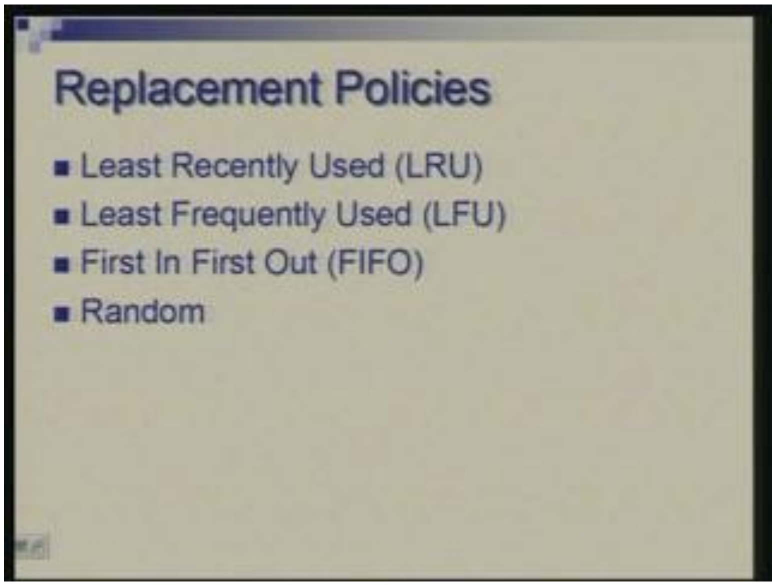
One simple mechanism used for hardware prefetch is that when you are getting one block get another block just the once which is following it hoping that you are going to have sequential references or it could be software driven where a programmer may be able to guess more clearly what is required ahead of time. So one has to have special instructio which will actually cause a miss an artificial miss and then force hardware to get a block ahead of time.

So now the open question here is how much ahead can you initiate a prefetch by software. If you do it too much ahead you might do a superfluous work because it maybe that you are thinking that in your program flow you will go a certain way but if you are doing too much ahead may be there is a branch point in between and you may go elsewhere and you may not use what you anticipated. So there could be wasteful transfer and you may basically load the memory processor bus.

Other question is how often you do it; do you do it always or do you do it sometime; again it is a matter of tradeoff and judgment.

Replacement issue comes when a new block has to be brought and an older one has to be thrown off. In a direct map cache there is no choice, a new block comes and it has to go to a fixed place so it is trivial. But when there is some degree of associativity, you need to find out which, suppose it is a 4-way associative then which of the four you need to replace so that is decided by what is called replacement policy.

(Refer Slide Time: 00:38:58)



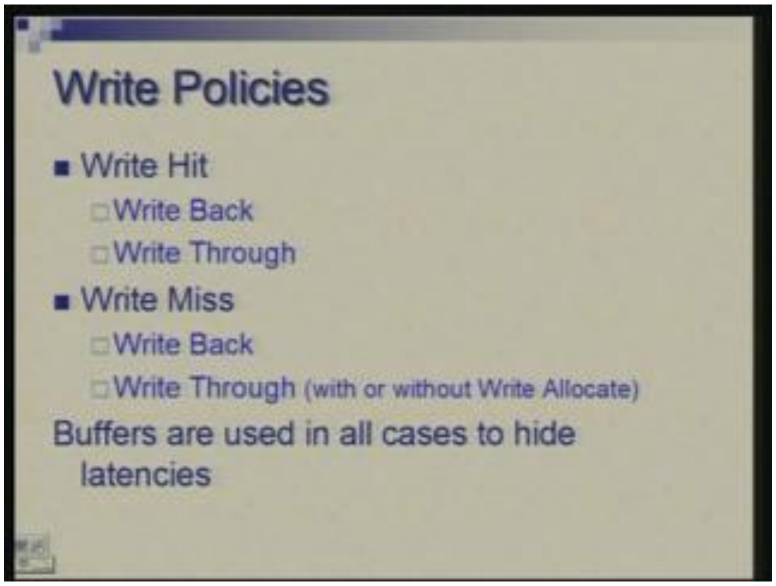
Replacement Policies

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- First In First Out (FIFO)
- Random

And the most commonly used policy is LRU or Least Recently Used so you need to keep track of which of the four blocks was most recently used which was least recently used and the one which is least recently used is the one which is replaced.

Alternative strategies are Least Frequently Used. Instead of counting when was it used last you keep track of how often it was used from the time it was brought. Or you could use a simply FIFO approach one which is brought in earliest gets thrown out or you may do it randomly of course that means you do not think just but arbitrarily replace something. All these are some kind of heuristics. What has been found in practice is LRU is most suitable.

(Refer Slide Time: 00:40:00)



Write Policies

- Write Hit
  - Write Back
  - Write Through
- Write Miss
  - Write Back
  - Write Through (with or without Write Allocate)

Buffers are used in all cases to hide latencies

Now, this is very important and this has very profound influence on the performance. There two kinds of write policies which are followed. We have so far focused our discussion more in terms of read that is you are trying to get data. What happens when you are trying to write. You can follow two approaches: one is called write back, the other is called write through.

In write back what you do is you keep on writing in the cache and in write through you directly write in the main memory. Now, first let us assume that there is write hit. These choices come when you have write hit or a write miss in both cases. First let us examine write hit case. Write hit means that the address where you want to write that block is present in the cache. In a write back case you will update that, you will write into cache you will not write into main memory and as a consequence main memory is not up to date.

When do you update the main memory?
It is when this block which you had modified is about to be thrown away. Before this gets thrown away you must make sure that main memory is updated. So, to do it judiciously what you need to do is, with each block you need to maintain a bit which is called a clean bit or a dirty bit. So you may get some block only for reading and you may keep it clean, but moment you write you call it dirty and when at the time of replacement if the block which is being replace is dirty then you update the main memory otherwise you do not have to take that trouble.

In write through what you will do is you will actually write in both; you will write in cache, in parallel you will also initiate write in main memory. In case of write miss write back arrangement would require that first you get the block from main memory into the cache serve the miss first and then write in cache only; whereas in write through you have a choice of getting the block or not getting the block. Those two differences are called write through with write allocate and write thorough without write allocate. That means, now imagine that there is a write miss which means that the block where you are trying to write is not there in the cache, so one possibility is that first you get the data from the memory into the cache and since it is write through you write there in the cache as well as in the main memory. The other possibility is that if the data is not there in the cache it does not matter, you simply go and write in the main memory.

Now the problem with this is that you will save the time although you will save the time of getting data from the cache but you are likely to encounter more misses; whereas in case of write back you will definitely bring a block and then you can simply keep on writing in cache because writing into main memory would be more time consuming.

In any case what is typically done is that you do not write into main memory directly actually, you write into buffers. So, if you are following write back policy, the buffer will accommodate one or more blocks because you are writing one block at a time whereas in write through you are writing one word at a time and therefore the buffer would accommodate a couple of words. So what CPU does is that it will write either a block or the word as the case is into the buffer which may typically be possible within a single cycle and then data could get transferred from this buffer to the main memory in its own due course. This could cause problems.

For example, suppose you are writing something which is still sitting in the buffer and very soon you require to read it. So now data is sitting in the buffer it has not reach main memory and you want to read, so from where do you read; you have to have a mechanism which will either let you wait till this gets written, till the buffer gets cleared or it should be possible to look at the buffer; figure out if the data you wanted to read is waiting somewhere in the buffer and read from the buffer itself so those complications arise. The moment you think of any such architectural improvement which tries to improve the performance you have to often look at these complications or side effects which arise and have to make sure that the meaning of the program does not go wrong.
the processor should do the job correctly so all those checks have to be made.

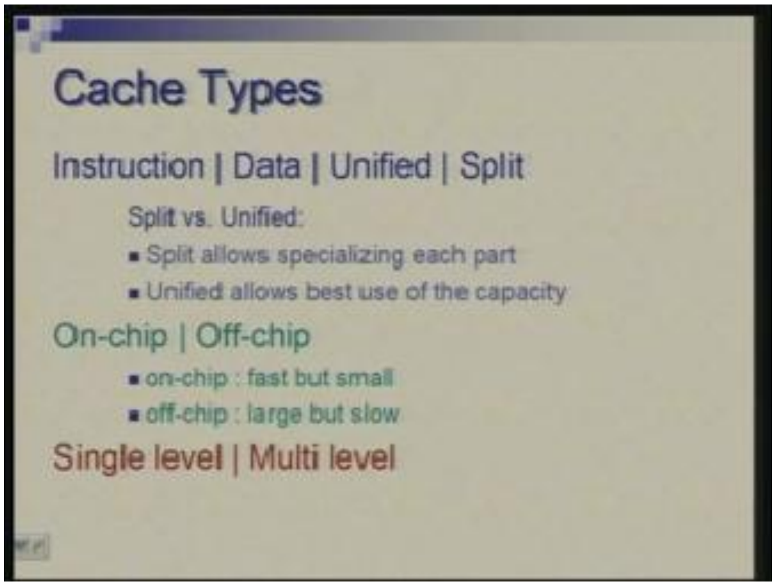
[Conversation between student and Professor: .... 45:24 Yeah, writing into main memory means that what was there in the main memory will gets replaced by something new. Was that you were asking? Yeah, actually, well, if you think why would you write in memory, I mean, whatever you have attempted to write in the memory let us say that is getting written in the cache. But if there was no subsequent use of that it was useless, you would read that and print or do something so the last thing has to be reading of what you have written. if the last let us say last statement is an assignment to some variable and that variable has not been read subsequently then it does not matter whether you update this in the cache update this in the main memory or not. So, updation of main memory should actually be forced by reads eventually.

If you have read and taken action then it does not matter if it goes to main memory or not, you are terminating the program anyway.]

[Conversation between student and Professor: .... 46:58] yeah, see, the reason for transferring blocks is that it is more efficient. so your question is that suppose in one block only one word was modified yes what you are saying is right; technically it is possible, what you can, this dirty flag has to be then with every word and not with the block. So if you are able to if you are willing to maintain one bit with every word which is indicating whether it is dirty or not then you can take the trouble of writing only those many words. But the experience with locality says that typically you would have dirtied many words; it will be more often the case that in a block it is less likely that only one word is dirty, you would have dirtied many more words. So statistically it turns out that it will be better to maintain the set block level only; although there could be situations where this is overdone.]

Finally, let me talk of couple of variations in the overall cache organization as such. You could think of either a cache being there just for instructions or a cache being there for just the data or separate cache for instruction and data which is called basically split cache.

(Refer Slide Time: 48:53)



Cache Types

Instruction | Data | Unified | Split

Split vs. Unified:

- Split allows specializing each part
- Unified allows best use of the capacity

On-chip | Off-chip

- on-chip : fast but small
- off-chip : large but slow

Single level | Multi level

Unified cache means that the cache is for instruction as well as data. The pros and cons of split and unified cache are not very difficult to see. Split cache allows specializing of each part because data may have some behavior, cache may have some behaviors or some policy may suit instruction cache, a different set of policies may suit data cache and you can choose these two separately if you have two separate caches; in unified you have to have a common policy.

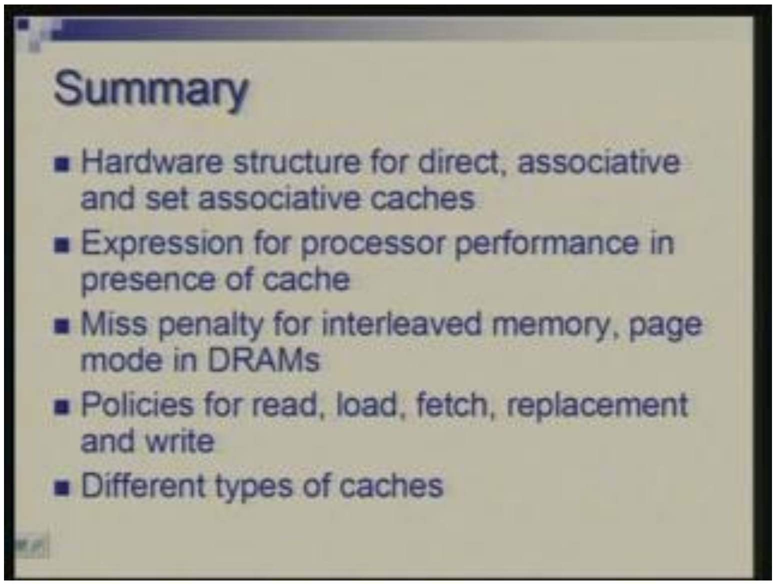
Split cache is also….not written here but it also gives you parallelism. If you have unified cache then there could be..... in a pipelined implementation there would be the source clash. Two separate caches means that you can be simultaneously accessing data and instruction. On the other hand, a unified cache will allow best use of the capacity.

Suppose you can afford let us say 16k bytes on the whole; if you were to split you may have to make a decision, 8 byte 8 kilo bytes for this, 8 kilo bytes for that and at some point of time in the program execution your requirement maybe more for data less for instruction and at some other point more requirement for one than the other. So if you are not splitting then you are using entire 16 kilo bytes and the instruction data will share these as and when to the extent necessary.

So, unified cache utilizes a capacity better. But in case of split you may find that some time you are short of cache words and instruction, some time you are short in data, some time you are surplus in one and sometimes you are surplus in another. So that is the variation from one point of view. Typically in multi-level caches the first level is invariably split cache and second level is most often a unified cache. Then you can talk of on-chip cache or off-chip cache; cache which is integrated with the processor on the same chip, once you integrate then you are constrained by the chip area of the processor and there may be constraint on the size but there are advantages of speed.

On-chip cache is typically fast but small, off-chip cache can be larger but relatively slower. And in many systems you often have multi-level cache so you can have one level, two level, three level and in more levels you have more expensive it becomes but it gives you...... since there are orders of magnitude difference you can position multiple caches which are differing slightly in the performance and on the whole you are filling up the spectrum nicely.

(Refer Slide Time: 00:52:09)



Summary

- Hardware structure for direct, associative and set associative caches
- Expression for processor performance in presence of cache
- Miss penalty for interleaved memory, page mode in DRAMs
- Policies for read, load, fetch, replacement and write
- Different types of caches

So I will close with the summary. We began with looking at the structure of how we map the addresses from main memory to cache; we looked at direct mapping, associative mapping and something in between which is set associative mapping. We looked at the expression for performance both for memory, access time on the average and the total number of cycles or total time spent for executing a program. We looked at the relationship on the miss penalty depending upon the DRAM organization and we also noticed that page more in DRAMs serves similar purposes as interleaving. We looked at some variety of policies for reading, loading, fetching, replacement and writing and looked at different possibilities from the point of view of overall organization.

Thank you.