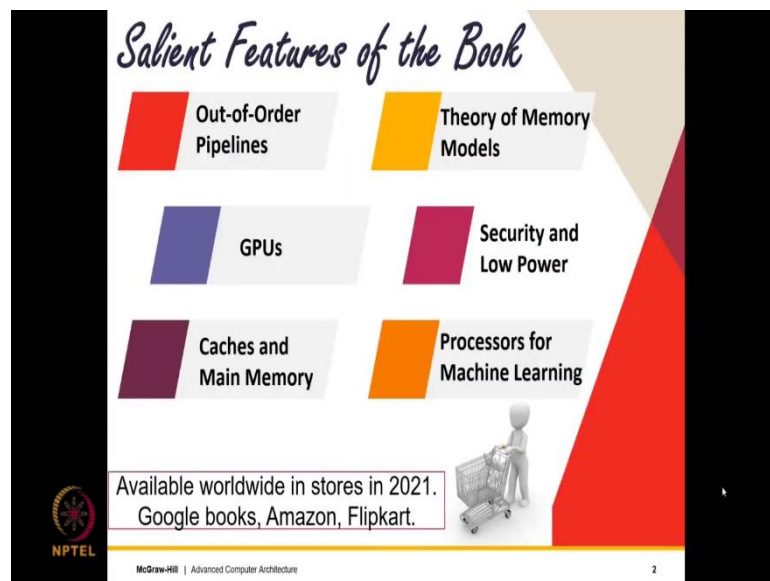


**Advanced Computer Architecture**  
**Prof. Smruti R. Sarangi**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**

**Lecture - 23**  
**Caches**  
**Part - V**

(Refer Slide Time: 00:24)



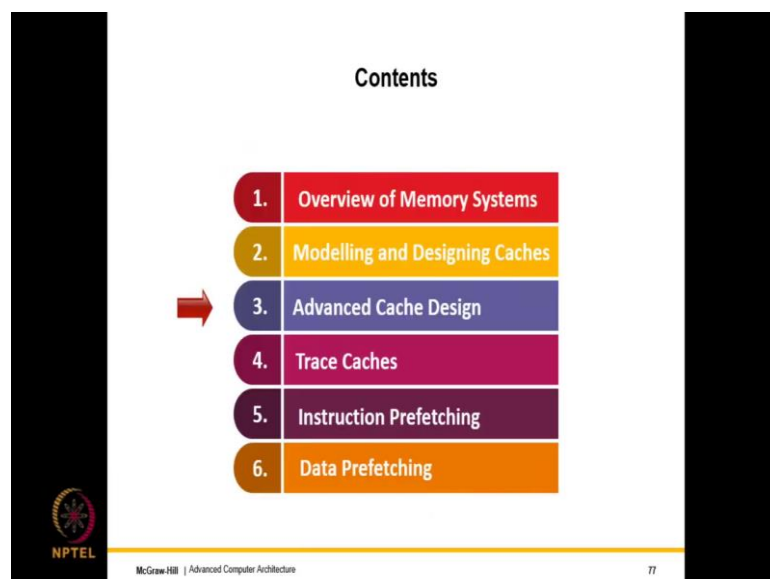
*Salient Features of the Book*

- Out-of-Order Pipelines
- Theory of Memory Models
- GPUs
- Security and Low Power
- Caches and Main Memory
- Processors for Machine Learning

Available worldwide in stores in 2021.  
Google books, Amazon, Flipkart.

NPTEL  
McGraw-Hill | Advanced Computer Architecture 2

(Refer Slide Time: 00:36)



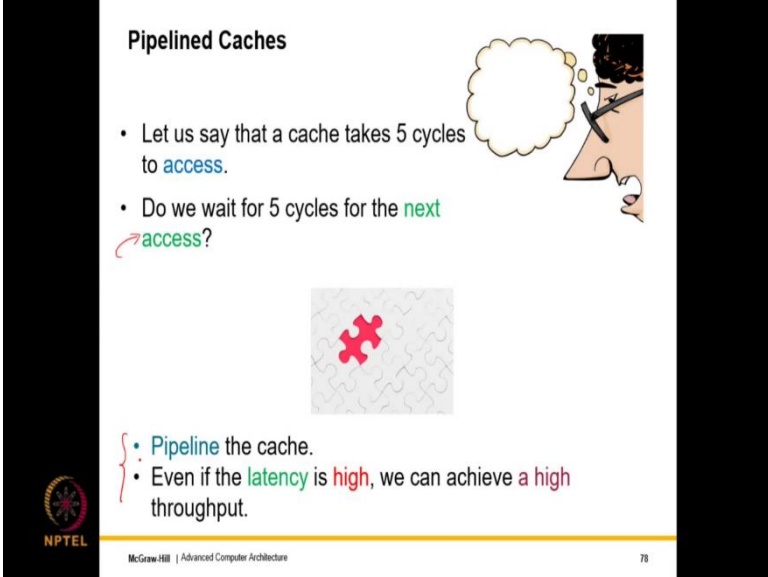
**Contents**

1. Overview of Memory Systems
2. Modelling and Designing Caches
3. Advanced Cache Design
4. Trace Caches
5. Instruction Prefetching
6. Data Prefetching

NPTEL  
McGraw-Hill | Advanced Computer Architecture 77


Let us now discuss some advanced aspects of cache design. So, these will definitely be deeper design choices as compared to some of the design choices that we have seen particularly in the first section. Where we discussed methods to improve the performance and efficiency of a cache such as critical word fetch early we start and others like a victim cache, but these ideas are slightly more advanced.


(Refer Slide Time: 01:11)



**Pipelined Caches**

- Let us say that a cache takes 5 cycles to **access**.
- Do we wait for 5 cycles for the **next** **access**?





- **Pipeline** the cache.
- Even if the **latency** is **high**, we can achieve a **high** throughput.

NPTEL  
McGraw-Hill | Advanced Computer Architecture 78

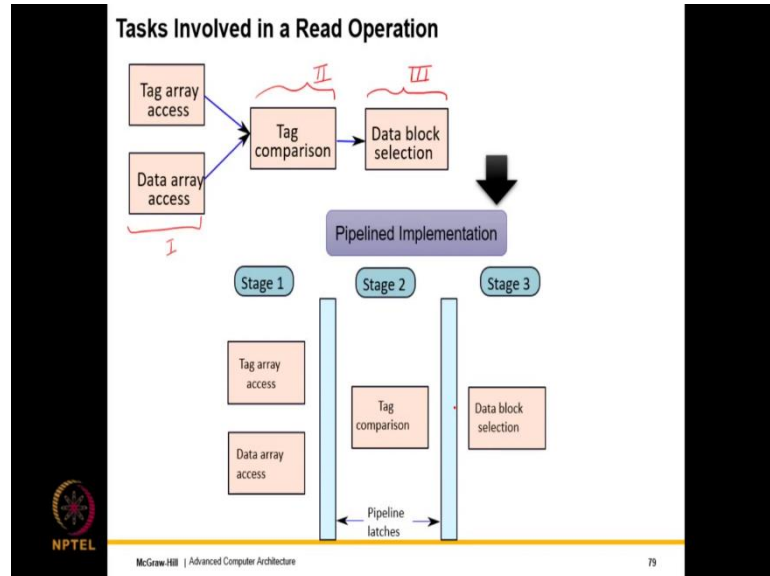
So, let us start with asking a question let us say that a cache access takes 5 cycles which can be the case particularly if we have a large L1 cache and we have a reasonably aggressive clock cycle time. So, in that case it is very much possible that we can have a cache that takes 3, 4, 5 cycles to access and definitely a larger cache such as L2 cache can take even more time. So, the critical question over here is that do we wait for 5 cycles for the next access, so the answer is no.

So, we have already answered this question when we discussed in order pipelines and clearly the answer here is no it is not going to be the case. what we will instead do is we will pipeline the cache, which means that at least we can accept a new request if not every cycle then once every 1 or 2 cycles the latency can be high as we have seen, but what pipelining gives us is the is that it gives us high throughput.

So, this is an important point to bear in mind that even if the latency is high that is not desirable, but nevertheless if we pipeline the cache at least we can ensure that our

throughput is high and so many applications that are throughput dependent will at least be benefited from this design.

(Refer Slide Time: 02:56)



So, let us look at a simple idea of how to pipeline, look at the tasks that are involved in a read operation. So, in a read operation what happens that we have a parallel tag and data array access.

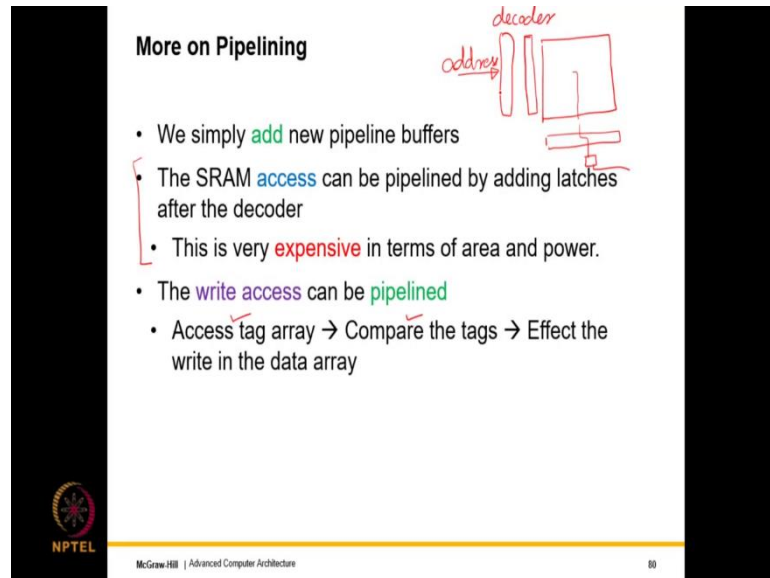
So, we have already discussed this and this was a very beneficial design choice the reason being that of course, we do waste a little bit of power, but at least we get the k data blocks in a k way set associative cache in the same cycle and once we know which tag matches we can then choose the appropriate data block. So, this is of course, the first part let us call it stage I then stage II is when we compare the tag.

So, the comparison is also slow process because in comparing what do we do? We basically take two tags and we subtract one from the other. So, that is one way of comparing the other is that we just compare the bits and then we compute an XOR and if let us say any bit pair differs then the values are not the same.

So, regardless of how we do it this will take one full sub stage and finally, once if there is a match if we find a match then we select the data block. So, what we can do is we can take the simple design and we can pipeline it. So, pipelining does not require any additional

circuitry all that we need to do is we need to insert a pipeline latch between adjacent stages a tag array data array access tag comparison and data block.

(Refer Slide Time: 05:07)



**More on Pipelining**

- We simply **add** new pipeline buffers
- The SRAM **access** can be pipelined by adding latches after the decoder
- This is very **expensive** in terms of area and power.
- The **write access** can be **pipelined**
- Access tag array → Compare the tags → Effect the write in the data array

The diagram shows a red box labeled 'decoder' with an 'address' input. Below it is a schematic of a decoder circuit with a pipeline latch.

NPTEL  
McGraw Hill | Advanced Computer Architecture 80

So, something similar can be done for a write operation. So, what we would do is that for a write operation we will access the tag array then we compare the tags and we affect the right in the data array. So, of course, if we have a read and write to the same address some forwarding will be required.

So, that forwarding logic needs to be added. So, let us now see what else I have on this slide. So, what we can do is that we can further add more pipeline stages. So, one where the idea in this spaces to pipeline the SRAM access? So, if you would recall in the SRAM access we have this huge row decoder input is the address and then of course, this sets the appropriate word line then we go to the SRAM array.

We read whatever the is that needs to be read and by that time we are expected to have configured the column mux de mux. finally, the sense amplifiers sense amplifiers read the data and the data comes out. So, we can in principle pipeline the decode process in the sense that we can add a huge pipeline latch over here where the decoded address. So, the decode lines would be sort of temporarily stored in that latch for one cycle.

So, of course, here the latch has to be very large write roughly the size of the decoder because storing one bit is not easy if let us say we want fast access. So, it will be it will

have to be one d flip d flip flop per bit and this does cause issues in terms of a high area foot print and also it adds to the capacitive delays of the word lines and so on. So, because of that pipelining the SRAM access any further is not advisable.

(Refer Slide Time: 07:29)

**Non-Blocking Caches**

- Let us say that there is a **cache miss**. *(Handwritten: B)*
- Do later cache **access** stall?
- Cache **accesses** need not be **blocked**. We simply need to record all accesses to the **missed** block. *(Handwritten: HW Data Structure)*

**MSHR** Miss Status Holding Register

*(Handwritten: ↗)*

NPTEL McGraw-Hill | Advanced Computer Architecture 81

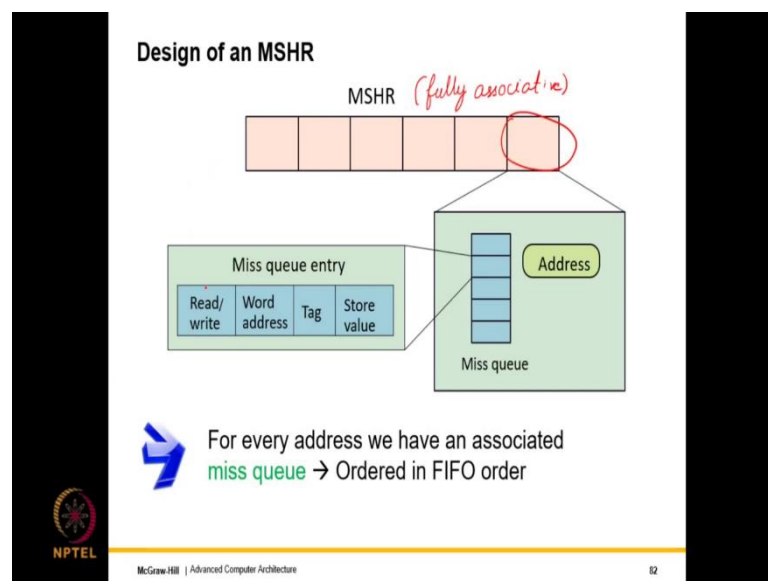
So, what do we do then? What we do then is that we look at other kinds of optimizations which brings us to a second optimization this is another idea which is called a non blocking cache. So, let us say that we have a cache miss for a certain block B see if there are later cache accesses, what happens?

See if there are different if there are two different blocks well they can go through, but let us see if there are two different memory words within this block. So, what do we do to they actually stall. So, what we do is that accesses to other addresses need not be blocked that is the key idea.

That accesses to other blocks need not be blocked, but let us say that for the missed block what we need is we need to simply store or record all the accesses that are being made to different words of the miss block such that when the miss araised on a lower level all of them can be serviced in one go. So, to do that what we do is we add a new data structure. So, this is not a data structure in a data structure since it is a hardware structure. So, let us call it a hardware data structure.

So, let me define something, but this is of course, pure hardware its no software. so, but I love the word data structure because it uses similar concepts, but I do agree that this is kind of an abuse of the term. So, let us call it a hardware structure. So, even though I would have been internally motivated to say hardware data structure, but in the interest of keeping things consistent and simple let us call it a hardware structure. This is an MSHR miss status holding register which has the rather intricate structures which we will see next.

(Refer Slide Time: 09:33)

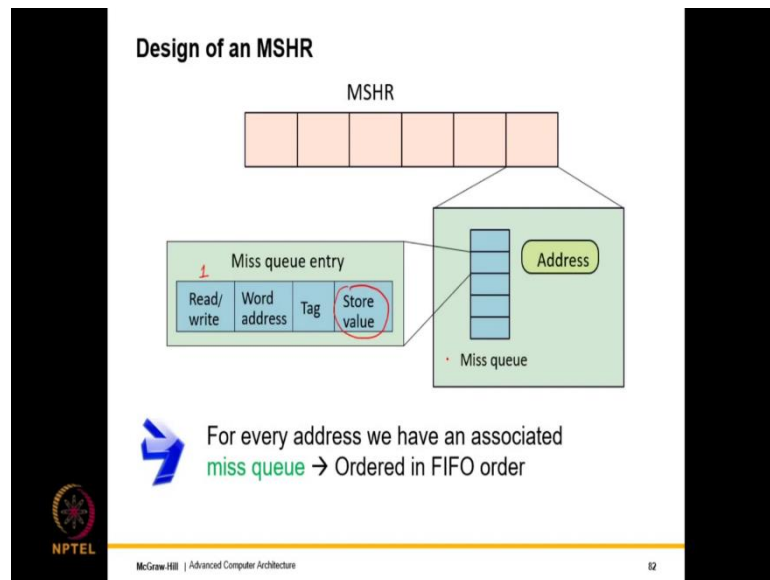


So, the MSHR is associated with a cache. So, it keeps track of the misses of the cache each entry of the MSHR has two fields if you look at it from the top level the first field is the address. So, this is the block address that is the first field and the second field is a miss queue.

So, whenever we have a miss or let us say whenever we have a read or write access the MSHR is always checked. So, the block address is looked up very quickly and the MSHRs are typically very small structures. So, they have maybe 4, 8 or 16 entries. So, this structure can be very well fully associative as well it can be.

There is no harm in that and further more each entry has a miss queue. So, the miss queue each entry of the miss queue is called a miss queue entry its fields are as follows, the first field is read or write. So, this field refers to whether the particular access is a read or write.

(Refer Slide Time: 10:51)



So, this will be a one-bit field next we have the word address this is the address of the specific word within the block which led to a miss. So, this is the word address the next is the tag. So, the tag is mostly useful for a load instruction. So, in a load instruction this tag and also this is useful for the L 1 cache.

So, this is basically the tag of the instruction the tag of the register right the physical register idea of the register that caused a miss. Why is this needed? Well this is needed because when the value of the load comes back from the lower level we can immediately send the load value to the processor and along with this the tag will go because the tag will indicate which load it is.

And the final field is a store value which is the value that needs to be stored. So, this again is useful for stores. So, for every address what we do? So, for every address meaning a block address we have an associated miss queue for all the miss queue entries are stored in first and first out order in FIFO order and so that is applied one after the other in FIFO order after the block comes from the lower level.

(Refer Slide Time: 12:21)

**Operation of an MSHR**

- When there is a **miss** in a cache, an entry is **created** in its MSHR. If there are no **free** entries, **[the request blocks]**
- We also **create** an entry in the miss queue. *↑ does not happen frequently*
- This is a **primary miss**.
- If at the same time, another miss arrives, it will be called a **[secondary miss]**
- If it is a **write**, we just append it to the tail of the miss queue.
- If it is a read, we **search** for earlier writes in the miss queue to the same set of bytes. If we **find** an entry, the value is **forwarded**. Else, we **enqueue** the entry. *↓*

NPTEL  
McGraw Hill | Advanced Computer Architecture 83

So, a little bit of detail about the operation of an MSHR. So, when there is a miss in a cache an entry is created in its corresponding MSHR see if there are no free entries. So, assume that the MSHR is full and we cannot create a new entry then of course, the request blocks and so then from a non blocking cache the cache becomes the blocking cache, but of course, the MSHR should be large enough such that this particular scenario does not happen or let us say it does not happen frequently.

So, what we also do is we also create an entry in the miss queue which will be the first entry, this is known as a primary miss. So, once a primary miss is there in the miss queue what we need to do is that we need to add additional entries if there are subsequent misses before the data actually arrives. So, these misses will be known as secondary misses. So, if a secondary miss is the right what we do is we just append it to the tail of the miss queue.

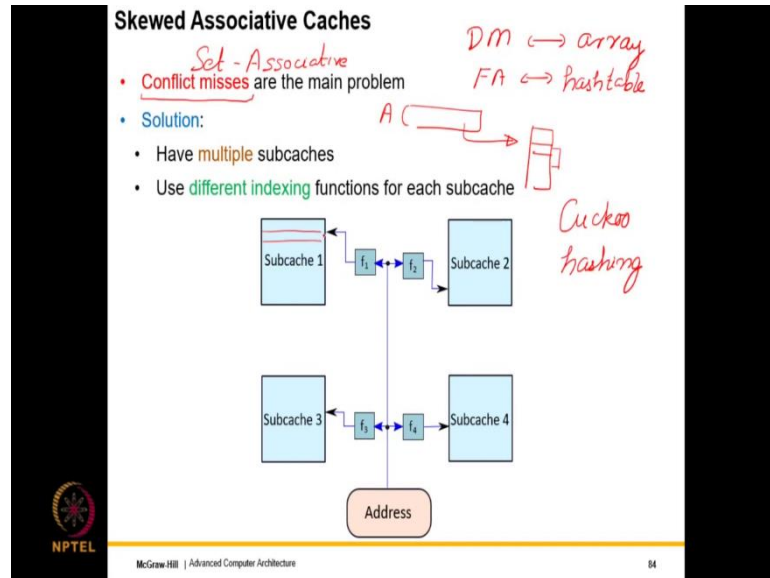
So, basically what we do is that if we assume its a queue. So, then we just add a new entry to the tail of the miss queue and the write is remains there. If it is a read similar to a load store queue is search for all earlier writes in the miss queue to the same set of bytes if we find an entry then there is no reason for us to queue the read what we can do is we can just forward the value exactly the same way we would do it in a load store queue and the processor can resume.

So, in this case, the operation of the ls queue and the MSHR is the same else otherwise we enqueue the entry. So, we add the entry to a queue and again it remains in the queue until



the entire block comes from the lower level. So, what was new in this slide the forwarding aspect was new in the sense that the MSHR has a forwarding capability similar to an ls queue. So, this aspect was made.

(Refer Slide Time: 14:53)



Now, let us come to the third idea. So, the third idea is a new kind of a cache design which again if you can see it again comes from a simple data structure concept. So, after all a cache, what is a cache? So, a cache a direct mapped cache is like an array. So, its data structure analogue will be an array.

So, consider a direct map cache this the analogue for a direct map cache will be an array where we know the exact index we go there we see the value is there or not if it is there we access it else we declare a miss. Even a set associative cache is somewhat like an array, but a fully associative cache is like a hash table. So, in this case what we do is that well a given entry can be present anywhere.

So, we use a CAM array a content addressable memory array where we can search which entry has the particular address and if you find it then at least from the top it does appear to work as a hash table in the sense that we use a baseline CAM array which again addresses it by its content.

So, this so here again the broad big broad idea is rather different we have enqueue associative caches. So, this the direct competitor for a queue associative cache is a regular

set associative cache. Which is a combination of an array and hash table or a hash table implemented on top of an array where essentially we are pointed towards a set of locations and in let us say it is a k by set associative cache then we are pointed to k locations we go there and we then see if the addresses there or not.

And of course, conflict misses cause a problem in the sense that let us say if it is a 4 way set associative cache, but there are 5 blocks that map to those 4 ways and continuously one of them will get evicted and there will be a timing penalty. So, what we do is instead of a regular hashing scheme for a regular hashing scheme is that we take an address we extract some bits out of it you might further mangle the bits and then we use it to access an array some location and then we do as we do in a set of associative case.

So, this is kind of an extension of the basic hash table idea even though some of you may not entirely agree with me in the sense that you say that you might say. That look in a hash table we only search for a few locations search in a few locations.

But in a fully associative cache we actually search all the locations I agree with you the comparison is not 100% accurate, but still a hash table searches on the basis of content this is something that a cam array does and that is why I have created this comparison even though there can be arguments against this comparison. But broadly speaking a set associative cache does embody the spirit of a hash table implemented on an array.

There is another way of hashing which is known as cuckoo hashing. So, in cuckoo hashing what we do is that instead of using 1 hash function we actually use multiple hash functions. So, what we can do is we can divide a cache into 4 sub caches given an address we can compute 4 different functions on it  $f_1$ ,  $f_2$ ,  $f_3$  and  $f_4$ . So, they would essentially produce different locations within the cache. So, if you can see all of these locations are all different.

(Refer Slide Time: 19:33)

**Basic Insight**

Consider two addresses

- If  $f_1(A_1) = f_1(A_2)$  most likely  $f_2(A_1) \neq f_2(A_2)$
- If they conflict in one subcache most likely they will not conflict in another subcache
- Reduce conflict misses

**Overheads**

- These functions need to be computed  $f_1, f_2, f_3, f_4$
- Replacements are tricky

NPTEL  
McGraw Hill | Advanced Computer Architecture 85

So, there is a beautiful property which I will come to next. The beautiful properties considered two addresses again they are block addresses in the context of this discussion. So, let us say that they are colliding in sub cache 1 in the sense  $f_1(A_1) = f_1(A_2)$ . So, this indicates a conflict miss with most likely or with a high probability in the second sub cache they will not collide in the sense they will map to different locations  $f_2(A_1) \neq f_2(A_2)$ . So, they will map to different locations.

So, the key important insight over here is that if they conflict in one sub cache most likely they will not conflict over there. So, what we do is that if already let us say block with address ( $A_1$ ) is there in one sub cache then what we do is we go to the other one and there will not have.

So, there will have another location and we put our block over there. So, this really reduces the chances of conflict misses and this is an extension of the basic cuckoo hashing idea because as I said what is architecture it is essentially a hardware wrapper over what is fundamentally a data structure concept. So, the key important question over here is that we need to compute these functions  $f_1, f_2, f_3$  and  $f_4$ . Such that of course, this property holds that if there is collision in 1 sub cache we can go somewhere else.

(Refer Slide Time: 21:23)

**Replacements**

*Regular hashing* → Cuckoo search

Assume that we want to add the block  $B$  at address  $A$

- We will first check if  $f_1(A) \dots f_4(A)$  are free
- If not, we evict the block  $B'$  with address  $A'$  stored at  $f_1(A)$ . We insert block  $B$  in the first subcache at  $f_1(A)$ .
- We try to accommodate  $B'$  at  $f_2(A')$  or  $f_3(A')$  or  $f_4(A')$
- If none of these locations are free, we choose one of the locations randomly. Let's say, it is  $f_2(A')$ . We insert  $B'$  there by evicting block  $B''$ . Again we try to accommodate  $B''$ .
- We repeat this process a few times before finally giving up and evicting a block.

NPTEL  
McGraw Hill | Advanced Computer Architecture

Replacements are tricky we will see in a second y. So, we need to do something called a cuckoo search. So, assume we want to add the block  $B$  at address  $A$ . So, we will first check the 4 locations  $f_1(A)$ ,  $f_2(A)$ ,  $f_3(A)$  and  $f_4(A)$  to check if they are free.

If none of them is free will find we put the block over there and in this case accessing in this case accessing the skewed associative cache does require more power in the sense the 4 functions need to be computed and then we need to access the different sub caches to see where exactly is the block because it could be in it could be there in any sub cache. So in this case while inserting an entry if any of these entries is free then we insert the block over there.

But let us assume the worst case that none of these entries are free. So, what we do is we evict the block  $B'$  with address  $A'$  that is let us say stored at  $f_1(A)$ . So, then we insert the block  $B$  in the first sub cache at  $f_1(A)$ . So, now, the idea is that since  $B$  and  $B'$  had a conflict miss in sub cache 1 they will not have conflict misses in other sub caches which means that even the likelihood of you know have we not use this what would have happened is that pretty much all 4 of them would have been like the same set. So, there would have been conflict misses for  $B'$  and the other sub caches as well.

But because we use different mapping functions what we can do is we can try to accommodate  $B'$  at these locations recall that block  $B'$  the address of block  $B'$  is address  $A'$ . So, we can try to accommodate it at these locations  $f_1(A')$ ,  $f_3(A')$  or  $f_4(A')$ .

So, what did we do? Well since we did not find a free location we essentially tossed block B' out from sub cache 1 and we put in block B. Now the question is what do we do with B'? So, B dash again will map to 3 other locations in the 3 sub caches.

Which are  $f_2(A')$  I mean it might be better to  $f_2(A')$ ,  $f_3(A')$  and  $f_4(A')$  these locations are most likely not same as  $f_2(A)$  which we know to be full,  $f_3(A')$  is most likely not the same as  $f_3(A)$  which we again know to be full which is and  $f_4(A')$  is again not the same as  $f_4(A)$  again with high likelihood and  $f_4(A)$  is something we known to be full. So, these locations are other locations and it is very much possible that one of these locations may be free.

Or we can randomly choose one if that is not free we can look at the other two, but is very much possible that one of these locations may be free and block B' can be put there. So, let us assume again the worst case that these locations are not free well we can do the same we can put in B' in one of these locations toss out the block B'' which was over there and try repeating the process because B'' will again with high likelihood map to another set of different locations.

And once it maps to another set of different locations ultimately after may be a few rounds we will find some free locations. So, this method is called cuckoo search. So, what I would advise all of you is that you read up regular hashing. So, regular hashing again has two variants one is called chaining and the other is called linear probing.

So, we are not really doing changing over here. What we are doing is more like linear probing which is similar to a set associative kind of access scheme, but the other access scheme which is a slightly different paradigm in the world of algorithms and data structures is cuckoo search which is exactly something that I describe.

If you are not able to understand what I describe what I would like to advise you is kindly take a look at the cuckoo hashing and cuckoo search algorithms and then you come back. So, its an interesting thing of why do we say why do we say it is a cuckoo search and why not some other bird?

So, the interesting story is that the cuckoo is actually a very clever bird. So, what the cuckoo does is that it lays its eggs in a crows nest and a crows eggs and a cuckoo's eggs look more or less the same. So, the unbelongs to the crow actually helps it incubate and

hatch the eggs of the cuckoo and once the baby cuckoos are born then their mother comes and they are collected and they fly away.

So, we are in a sense playing a similar trick over here where we are having different hashing functions and we toss around blocks and so on. So, this is pretty much the history of this algorithm which of course is found an implementation in the skewed associative cache scheme. So, as I said we can do this tossing around several times, but of course, if the entire cache is full will never find space, but we can do it a few times and if we do not find free space then we can evict the block to the lower level. So, that option always exists.

(Refer Slide Time: 27:39)

**Way Prediction**

- Default mechanism in set-associative caches
- Read all the tags and blocks in a set (**simultaneously**)
- Compare the tag part of the address with each of the tags

• We do more work than what is required. *power++*

*accurate k-ways*

- Predict the way in which the tag will be found
- Access that way first
- If the tags don't match, access the rest of the ways
- Fast and power-efficient

NPTEL  
McGraw Hill | Advanced Computer Architecture 87

Now, we come to our fourth mechanism called way prediction. So, way prediction is rather interesting in the sense it takes us very close to the actual implementation very close to a designer's point of view. So, what is the default mechanism in let us say set associative caches? We read all the tags and blocks in the set simultaneously. Again why do we do that? We do it for performance reasons we compare the tag part of the address with each of the tags.

So, you pretty much do more work than what is required, whenever that is the case we end up burning more power and what is required. So, let us propose a brilliant idea over here the idea is that we predict the way in which the tag will be found see if there are k ways in a set we try to predict we say that look maybe in the second way the tag will be found we access that way first.

So, bear in mind that we do not read all the tags at the same time we do not do that we predict in which way we will find we access that way first. If the tags do not match we access the rest of the ways and then we declare a regular hit or miss. So, the advantage over here is that the additional energy that is spent in reading all the tags and blocks in a set and that to simultaneously that energy is not spent. So, we can save on energy subject to the fact that our predictor is accurate enough.

So, that is an important insight over here that if a predictor is accurate then what we can do is that we can have a more power efficient version of a set associative cache. So, the key important point here is to design an accurate predictor to tell us in which way we will have a hit.

(Refer Slide Time: 29:54)

**Way Prediction Techniques**

If the address is **available** use it, else use some other piece of information.

**Approach 1**

- **Predict** based on the program counter
- **Use** a branch predictor-like mechanism
- **Problems:** Memory addresses for an instruction might keep changing

*PC → [blocks] way*

**Approach 2**

Use an estimate of the memory address to predict the way

NPTEL | McGraw-Hill | Advanced Computer Architecture | 88

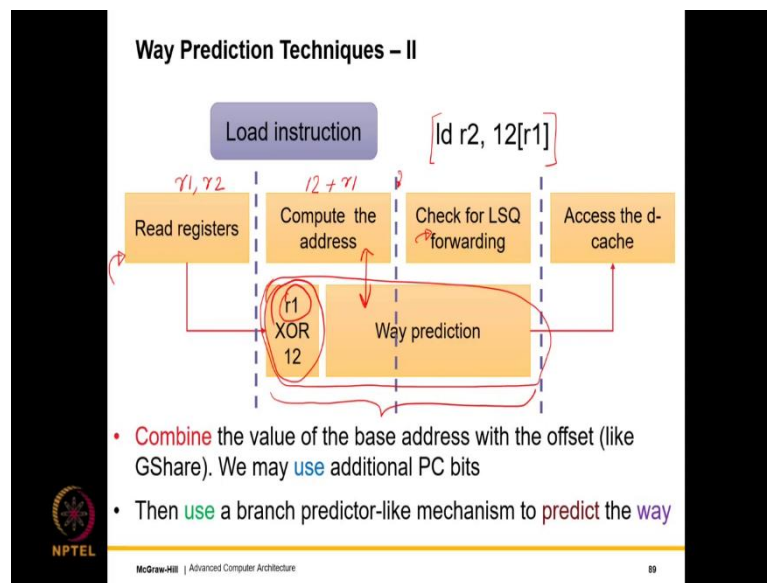
Let us now discuss the methods of way prediction. So, the basic idea is that if the memory address is available then we use it because after all the memory address will be the best predictor on in almost all cases for the way that you would expect to find it in. However, there are pipelines where the memory address may not be available when we initiate way prediction at that point we need to proceed on the basis of some other information, some other piece of information we need to proceed on the basis of that.

So, I outlining 2 approaches. So, in 1 approach we predict based on the program counter. So, we use a mechanism that is very similar to branch predictors that uses the PC address to make a prediction. So, in this case, we have a predictor which is extremely similar and

we provided the PC it predicts and what we get out is the way, a major shortcoming of this approach is that for the same program counter the addresses keep on changing.

So, the addresses are a memory addresses keep on changing the ways will also keep on changing and the predictability of this source is kind of low. So, what we do is when we do not have the address we use an estimate of the memory address the estimate is computed using a different mechanism.

(Refer Slide Time: 31:33)



The mechanism is as follows let me explain with an example. So, let us assume the load instruction is of this type load into register r2, 12 (r1). So, then what we do is that we can divide this into 4 stages. So, let me first look at the stages of the regular pipeline and then I will talk about the additional stages.

So, one stage is where we read the registers. So, in this case we read the registers r1 and r2. So, these are the source registers we read them then the next stage we compute the address. So, in this case we add 12 to the contents of r1, now we have the memory address with us. So, in this case we check for low store queue forwarding we can additionally have another stage to convert it from convert the virtual address to the physical address. So, this depends upon what we store in a load store queue.

So, if within the same thread different virtual addresses can map to the same physical address then we need to store physical addresses in a load store queue, but otherwise, if



we are guaranteed that different virtual addresses will point to different physical addresses. We can use the virtual address to address the load store queue and in parallel we can do the translation.

So, I am not showing that because I will show an optimization to this process few slides later, but in any case one cycle is definitely needed for load store queue forwarding and then we access the d cache. And this accessing the d cache itself can take multiple cycles. So, let us now look at an alternative timing where our way prediction a time it takes to do way prediction is actually closer to 2 cycles than to 1 cycle. So, there what we will see is that if we have the address at this point of time we cannot afford 2 cycles to do the way prediction and then access the d cache. So, that will not be possible.

So, what we should instead do is that we should compute an approximation of the memory address. So, the approximation can be computed like this that we take the contents of the base registers which in this case is r1 this is the base address. And we compute an exclusive or with offset which is 12.

So, this can be done quickly I mean this is much faster than addition and so this can be done independently bit wise. So, after that the rest of the time can be spent on doing way prediction and the way prediction will use similar tables as we use branch predictors. So, the rest of the time which will be roughly 2 cycles. So, we are treating this as a composite 2 cycle stage. So, this can be used to come up with the way when we access the d cache.

So, of course alternative organizations are also possible. So, there will be some questions at the back of the chapter which also looks at different ways of organizing the timing, but if we take a look at this particular example what we can immediately conclude is that we do not have enough time after the address to do the way prediction.

So, it needs to happen in parallel that is why we need an estimate where we use the XOR function like the g share predictor to provide us an estimate of the address. So, let us say if from the l s q forwarding stage we decide that we need to access the d cache then the way can come from here we first access the predicted way which is fast if let us say we do not find the data there then we access the rest of the ways.

(Refer Slide Time: 35:59)

**Loop Tiling**

How do we run large loops that access large arrays on a system with caches?

How is a 2D array stored?

- Row-major order: Store the first row, then the second, then the third and so on.
- Column-major order: first column, second column, and so on

The slide includes two diagrams. The first diagram shows a 2D array with arrows pointing downwards, indicating row-major storage. The second diagram shows a 2D array with arrows pointing to the right, indicating column-major storage. A vertical stack of boxes on the right side of the slide represents a 1D array of bytes.

NPTEL  
McGraw-Hill | Advanced Computer Architecture 90

Now, let us come to a next optimization after way prediction which is loop tiling. So, loops are very important in code primarily because the program spends most of its time executing loops. So, most of the time of a program is spent in loops and particularly if we have large loops that access large arrays on that are stored in memory on a system with caches running such large loops becomes a problem.

So, we need to first answer a question how is a 2D array stored. So, there are two ways of storing a 2D array one is a row major order and other is a column major order well after all an array can be 2D, but the memory the way that we see it is basically a 1D array of bytes with increasing addresses.

So, for a 2D array what we can do is we can store the first row then the second row third row and so on. So, many programming languages use the row major order, but what is kind of more common I would say slightly more common is a column major order in this case we store the first column second column third column and so on.

So, that is primarily the way that data is stored and so, what happens is that if let us say an array stored in row major order and then we rewrite it column wise, it will cause a lot of cache misses because essentially there is no locality. So, locality is spatial locality is not there, but if an array is stored in column major order we access it in column major order. Then of course, we will have good performance because we can leverage the effects of spatial locality.

(Refer Slide Time: 38:00)

**Basic Matrix Multiplication**

**Matrix Multiplication**  $A \times B = C$

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    sum = 0;
    for (k=0; k<N; k++)
      sum += A[i][k] * B[k][j];
    C[i][j] = sum;
  }
}
```

*Handwritten annotations:*  
- A red box highlights the innermost loop: `for (k=0; k<N; k++)`  
- A red circle highlights the addition operation: `sum += A[i][k] * B[k][j];`  
- A red arrow points from the `sum +=` line to the `C[i][j] = sum;` line.  
- Red text labels: "row major" (pointing to the innermost loop), "column major" (pointing to the middle loop), and "cache-friendly" (pointing to the overall structure).  
- A red box with an exclamation mark is placed below the code.

**!** Not suitable when arrays are large, and caches are comparatively much smaller.

NPTEL  
McGraw-Hill | Advanced Computer Architecture 91

So, let us consider matrix multiplication for example, so this is a standard code for matrix multiplication where we are multiplying 2 N X N matrices the final matrix is also N X N. So, is  $A \times B = C$ . So, the final as you can see the final result is being written here and this loop over here basically computes does the multiply step.

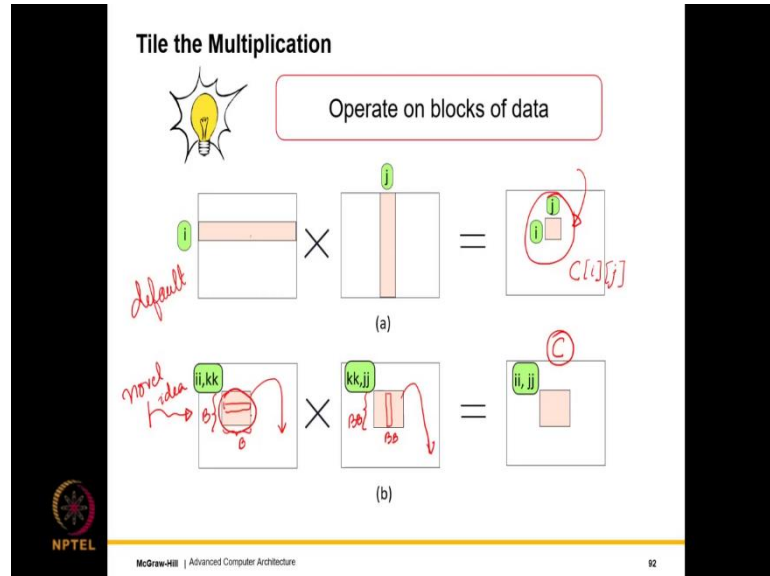
So, what we do in a matrix multiply step is we take a row multiply it with the corresponding column. So, we take the  $i$ th row here multiply it with the  $j$ th column it is a pair wise multiplication and addition that is the reason we have the  $+=$  over here. And finally, after the pair wise multiplications and additions the final sum is stored over here.

So, when arrays are large let us say they are very large they are much larger than the size of the cache. So, this will actually lead to a very inefficient traversal. So, to speak because just take a look at this array. So, in this array we are essentially traversing the array in row major order array A in row major order, but here what we are doing is we are choosing a column and we are traversing that array in column major order.

So, we are clearly not taking advantage of spatial locality of course some programming languages can be smart and some compilers can be smart. In the sense that they will store this array in row major in this array in column major, but typically that is not the case and the second is that even if they were it is possible that is single row is much larger than the size of the cache which can be the case with large data sets. So, there is a need to convert

this code into a more cache friendly version. So, the method to do that is called loop tiling which we will discuss.

(Refer Slide Time: 40:14)



So, this is also called blocking. So, we will see why it is called blocks, but let us look at the idea. So, what is the idea well the idea here is that let us first look at the default scheme. In a default scheme we take the  $i$ th row and a  $j$ th column we multiply the elements pair wise and then we add them and the final result is stored over here.

So, then the other thing is that instead of the other idea which is in our sense novel idea which is row tiling is that we do not consider an entire row or entire column instead what we do is we consider a small block of data. So, the small block of data can start at the coordinate  $ii, kk$  and then basically the size of the block might be  $B$  rows on this side and  $B$  columns on this side and we consider a similar block which again starts from here. So, what we do is that we take one small row over here we take one small column over here we multiply them.

So, we will produce a set of partial sums and those set of partial sums can then be used to basically. So, those are anyway part of the matrix multiplication. So, those partial sums will be used to update the final sums, which is the final sum is being referred to as  $C[i][j]$ . So, they will be a use to update the values of  $C[i][j]$  in the final matrix.

So, we will have to consider many many such small blocks in each block we perform a regular matrix multiplication which means we take a row take a column multiply them, but unlike the previous default approach where after one iteration we finally, have the value of  $C[i][j]$  in this case we will not have rather we will have partial values for all the cells within a block, but they are stored and this process is repeated until all the pairs that need to be multiplied or done.

One small advantage of doing this is that if let us say the size of this block is small the entire block can fit in the cache. Similarly, this entire block can fit in the cache, so all of these traversals even if because of the block effect. So, even if let us say spatial locality is lower, but at least it is guaranteed that all of the accesses will hit in the cache and so, basically this will be a very fast access.

(Refer Slide Time: 43:08)

**Tiled loops**

```

/* Iterate through the tiles */
for (ii = 0; ii < N; ii += b) {
  for (jj = 0; jj < N; jj += b) {
    for (kk = 0; kk < N; kk += b) {
      /* iterate within a tile */
      for (i = ii; i < (ii + b); i++) {
        for (j = jj; j < (jj + b); j++) {
          for (k = kk; k < (kk + b); k++) {
            C[i][j] += A[i][k] * B[k][j];
          }
        }
      }
    }
  }
}

```

$N^3$

Fits within the cache

3  $(N^3)$

NPTEL

McGraw-Hill | Advanced Computer Architecture

93

Now, let me show the code for tiling. So, the key idea is that we have 6 loops instead of 3 loops the 3 loops at the top which is the iterators ii, jj and kk they traverse the arrays at units of the tile size. So, we are of course, making an assumption over here that  $\frac{N}{B}$  otherwise of course, we can add some dummy elements or we can do a little bit of algebra to take care of it, but let us assume for the sake of simplicity that  $\frac{N}{B}$ .

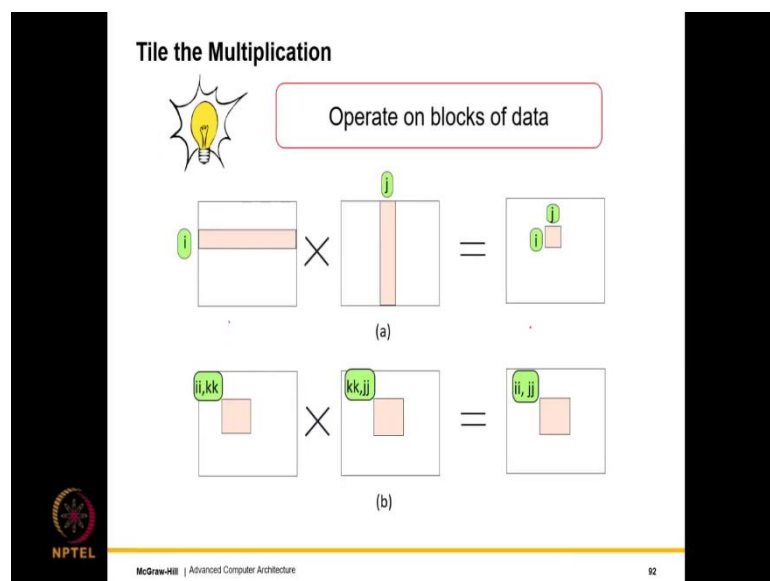
So, we divide the matrices basically into small tiles as you have seen over here that they are being divided into small tiles and each tile each  $p \times p$  tile is assumed to be small enough

such that it fits within the cache. So, what we do is that we are multiplying 2 matrices over here and whenever we multiply we consider the  $i$ th row over here and the  $j$ th column over here where  $k$  is the iterator.

So, in this case instead of the  $i$ th we consider a set of rows which is basically the tile size over here. So, we consider  $B$  rows and similarly for the other matrix what we do is we consider a set of columns. So, basically within each row we do not consider the entire row, but we consider a part of it. And similarly within the set of columns we do not consider the entire column, but we consider a part of it. So, that is given by this third parameter which is  $kk$ . So, this for us is the tile that we consider and we perform all the multiplications within the tile.

So, to perform all the multiplications within the tile, what we do is that we iterate within the tile. So, we have 3 loops over here. So, we perform an iteration within the tile. So, we iterate from the locations  $ii$  to  $(ii + b)$ . So,  $ii$  is considered the starting location of the tile.

(Refer Slide Time: 45:36)



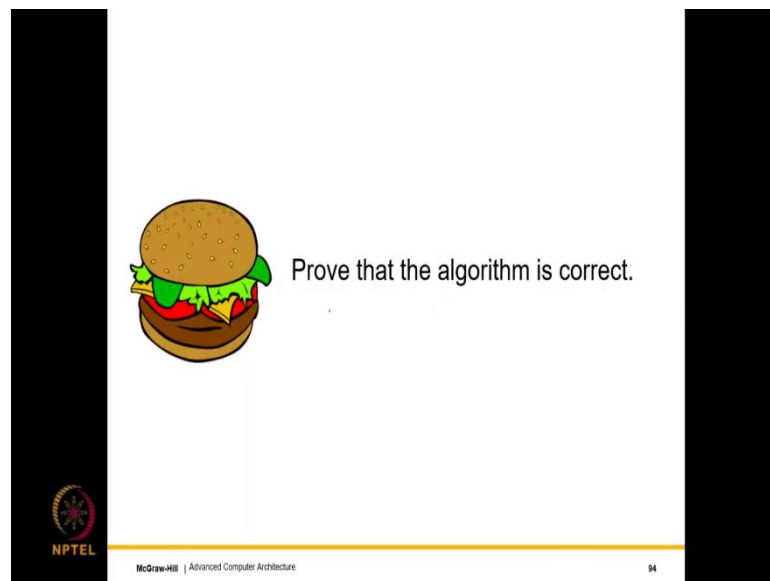
So, this is also shown over here where the starting location of the first tile is  $ii$   $kk$  and other one is  $kk$   $jj$ . So, all that we need to do is we need to iterate within this tile and we need to perform the multiplication. So, just look at the 3 iterators we are iterating from  $ii$  to  $(ii + b)$   $jj$  to  $(jj + b)$  and  $kk$  to  $(kk + b)$ . So, once that is done what we do well once that is done we have the exact in indices  $ii$  and  $k$  we perform the multiplication.

So, of course, not the  $+=$  sign. So, the multiplication over here is just adding to the partial sum such as  $C[i][j]$ . So, once all the additions are done you can see  $C[i][j]$  is complete and that would happen once we consider all the relevant tiles. So, it is in fact, quite easy to prove that this multiplication is doing exactly the same thing as matrix multiplication and. So, this is something that can be proven and its. So, if you think about it let us say that  $\frac{N}{B}$ .

So, the number of iterations of the top 3 loops are essentially  $(N)^3$  by  $(b)^3$  and the number of iterations of the bottom 3 for one given combination of this is essentially  $(b)^3 b * b * b$ . So, if I multiply both we still have the  $(N)^3$  multiplications and no multiplication happens twice.

So, and then you can also verify what exactly we are adding and how we are computing  $C[i][j]$ . So, using this it is possible to prove that both the algorithms are identical and of course, proving it more formally is left as an exercise for the viewer.

(Refer Slide Time: 47:45)



Well burger question over here is that proof that the algorithm is correct. So, this can be done we need to just prove that the same set of multiplications are done. No extra multiplication is done and the results are computed correctly.

(Refer Slide Time: 48:03)

### VIPT (Virtually Indexed Physically Tagged) Caches

• We need to **spend** an additional cycle translating the address after we have **computed** it.

• What if we don't need any **translation** to access the correct set in the L1 cache?

**Possible**

NPTEL  
McGraw-Hill | Advanced Computer Architecture 95

Now, let us come to the question that we had kind of left unanswered when we discussed way prediction. So, what we had discussed then is that any internal address is a virtual address which needs to be converted to a physical address. So, we need to compute the address its needs to be translated and the translated address needs to be used to access the d cache. So, the key insight is that we need to spend an additional cycle translating the address after it has been computed which is this cycle.

So, this cycle is essentially at least one cycle of delay because here we access the TLB between the address compute phase and a d cache accessing, but of course, to improve performance after computing the address we also access the load store queue. So, that can be that phase can be overlapped with this phase subject to the fact that we store virtual addresses in the load store queue, but storing virtual addresses in load store queue is tricky it does have correctness issues which were discussed in a previous slide.

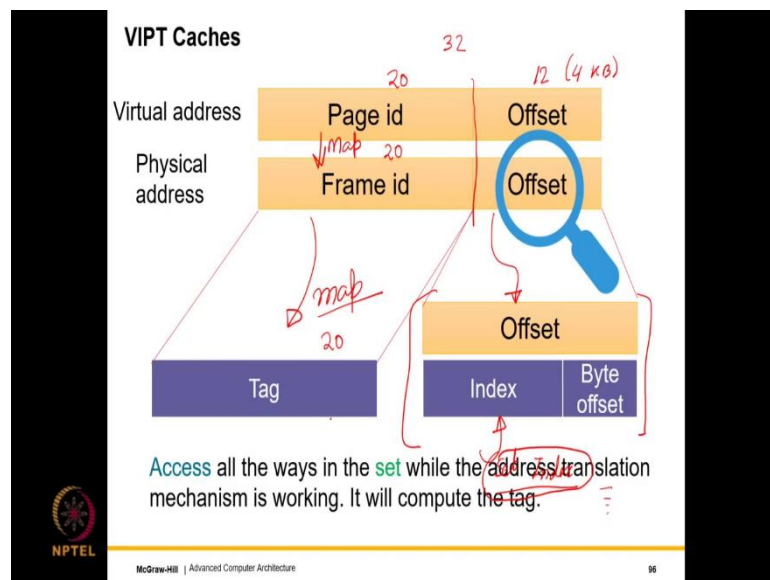
So, now the question that we ask is what if we do not need any translation to access the correct set in the L1 cache. So, what if in the l one cache we actually do not need any translation well does it mean that we send the virtual address to L1 cache that will cause a problem the reason it will cause a problem is that the next time that another thread runs on the same core another process runs on the same core thread or process actually if they are using the same virtual address there will be a problem. So, there will be a problem.



So, let us discuss this in some detail, but we are seeing that it is possible, but well the idea is that if it is there is a different thread or process that uses a different virtual address space if it is using the same virtual address space there is no problem, but we can always have a thread from a different process that is that has a different virtual address space.

So, unnecessarily if we use virtual address to access the cache we get a wrong answer. So, the overlap problem will come up. So, now, we see that there is a fundamental trade off here between performance and correctness we would of course, not want to compromise on correctness, but we would also like to use the virtual address in accessing the d cache. Now, let us see how to do it in some cases we can get away with it.

(Refer Slide Time: 50:53)



So, where can we get away let us understand this process in some detail see if I have a 32 bit or 64-bit memory address I would typically break it into a page id and an offset, where the offset is typically 12 bits because the size of a page is normally 4 kilo bytes. If I were to consider a 32-bit address space the page id would be 20 bits and then the frame id will also be 20 bits, but of course, it can be different depending upon the size of the physical memory.

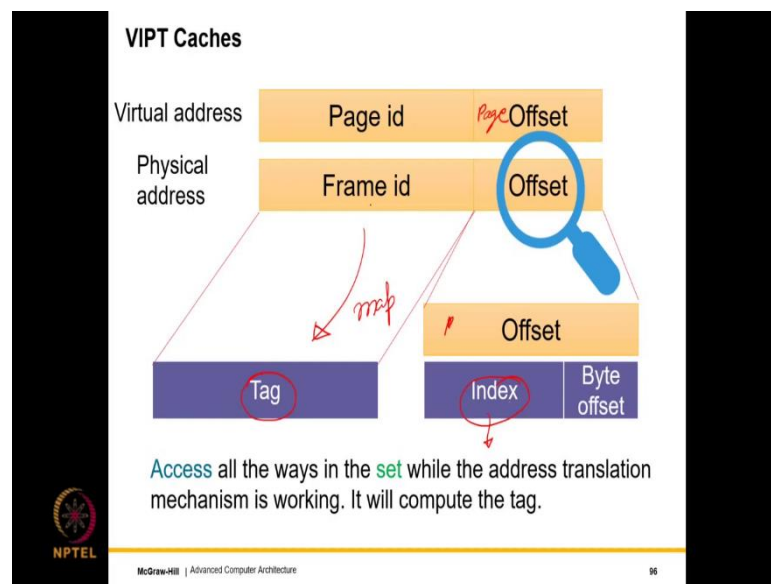
But let us say that the maximum size is limited to 20 bits say if I were to look at the offset in some detail I will see that there is a little bit of hidden magic in here. So, what I can do is that if I take my cache address if let us say these 20 bits of the frame id correspond to

the tag and the offset corresponds to the byte offset within the block and the set index then the offset which will actually not change.

So, basically the offset is not dependent upon the virtual to physical mapping. So, essentially this part is something that does not change. Whereas, this part is the one that is actually mapped we can use the offset to find the set. So, basically to so the tag of course, will come from the mapping process, but what we can do is we can use the offset to at least access the correct set within the cache and readout all the data blocks and tags within the set, simultaneously we can do the mapping.

So, let me explain this in some more detail because this is very very important mechanism. So, what is the what am I trying to say well what I am trying to say is that in any virtual to physical translation process there is a part of the address known as the page offset.

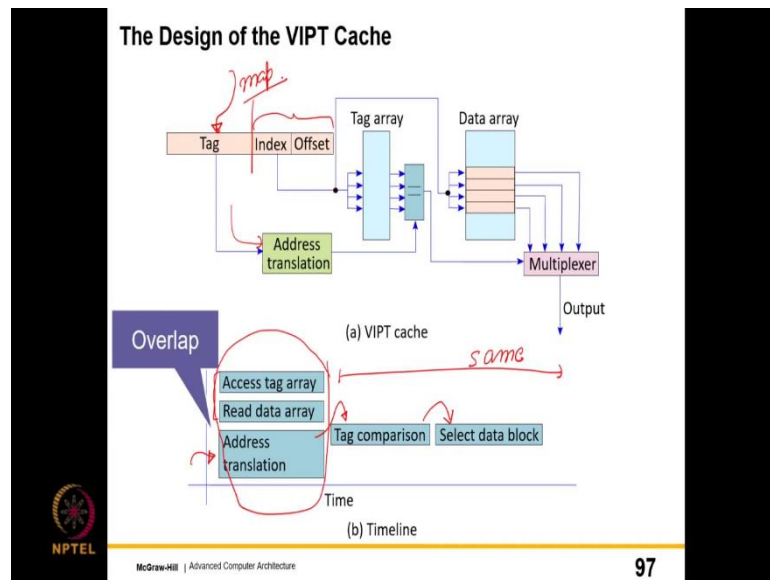
(Refer Slide Time: 53:00)



I should call this is the page offset and I should call this is the byte offset over here. So, the page offset remains constant if my addressing scheme of the cache is such, that the tag comes from the mapping phase and the rest is a part of the offset then the set index is not dependent on the mapping.

So, the key important point is that the index which is the set index is not a function of the mapping and then the brilliant idea is that I can use this index to access all the ways in the set read my tag and data blocks and at the same time also do the mapping.

(Refer Slide Time: 53:44)



So, what is it that I am doing? Well what I am doing is that my tag will go to the address translation module, but my index can come directly from the page offset this can be used to read all the tags of the tag array as well as read all the blocks from the data array in parallel some. Essentially, what am I doing I am accessing the tag array I am reading the data array in parallel I am doing the address translation. So, overlapping all of these within the same time window after that I can perform the tag comparison because I will already have the tag.

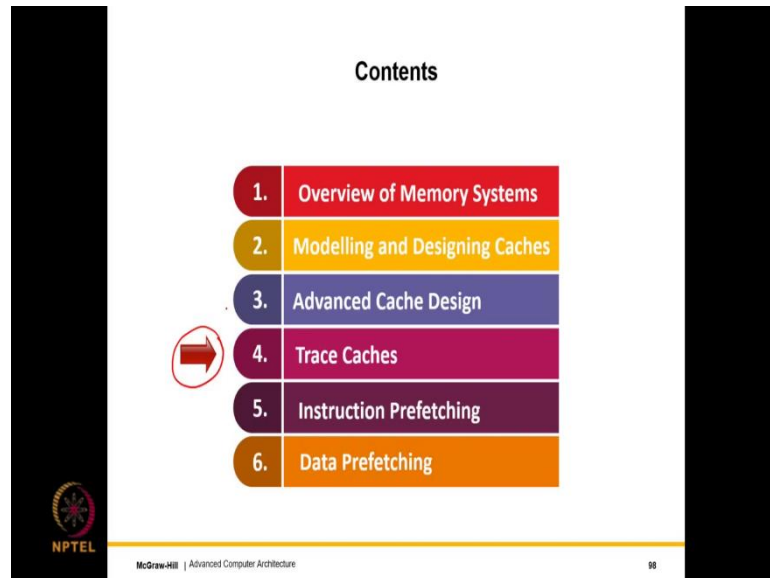
So, I am essentially saving a cycle over here that instead of this process being sequential for a translate the address first and then access the cache, I am deliberately introducing an overlap over here to give me that extra to reduce that extra 1 cycle then I do the tag comparison and I select the appropriate data block.

So, this part is the same what is actually different is the earlier part where I was able to save one cycle by creating an overlap over here and the reason that I could create this beautiful overlap is basically because the tag part of the access actually came from the mapping fields.

So, this is not a generic mechanism because this depends on the size of the cache the size of the set index and so on. So, that is the reason for an L2 cache this is not a very reasonable idea, but if you read the book and you will see a working example given you will find that

for most typical L1 cache sizes this is a very reasonable idea that is why many commercial processors actually use this mechanism.

(Refer Slide Time: 55:35)



Contents	
1.	Overview of Memory Systems
2.	Modelling and Designing Caches
3.	Advanced Cache Design
4.	Trace Caches
5.	Instruction Prefetching
6.	Data Prefetching

NPTEL

McGraw-Hill | Advanced Computer Architecture

98

So, next we will discuss a novel idea called trace caches. So, where in trace caches the idea is that can be totally side line the branch predictor and the decode unit and just store an unrolled trace of instructions that can be directly fed to the remaining stage. So, can this be directly done? So, we will look at that next.