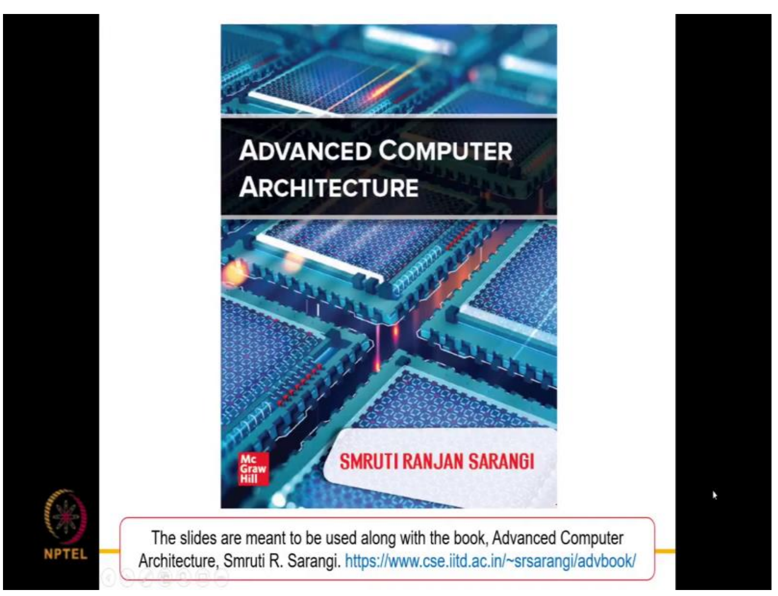
Advanced Computer Architecture

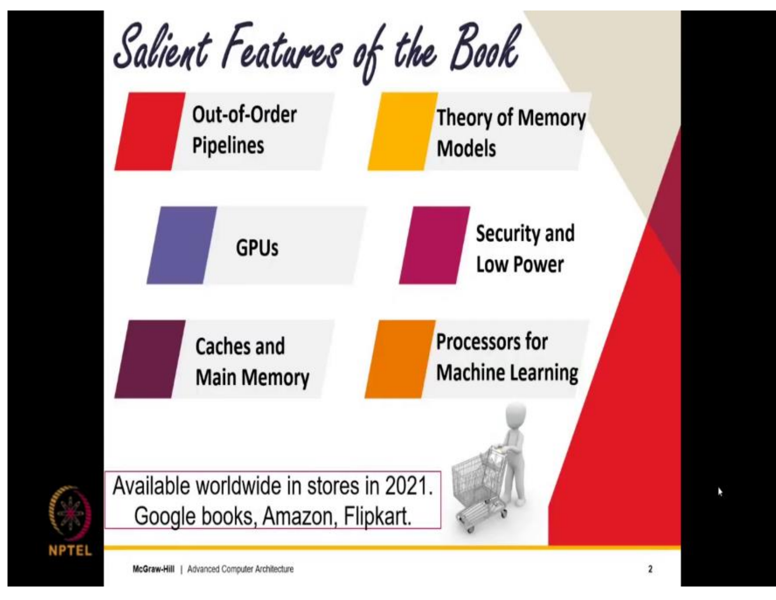
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Chapter - 07
Lecture - 24
Caches Part - VI

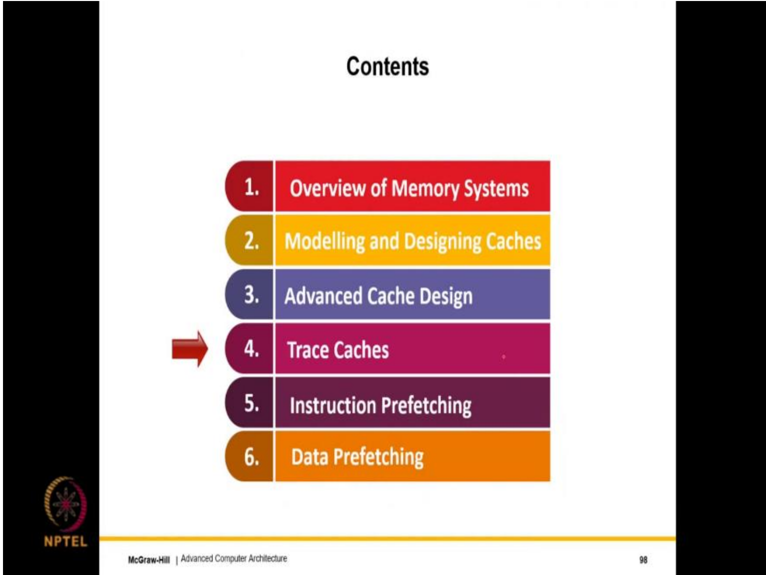
(Refer Slide Time: 00:17)



(Refer Slide Time: 00:23)

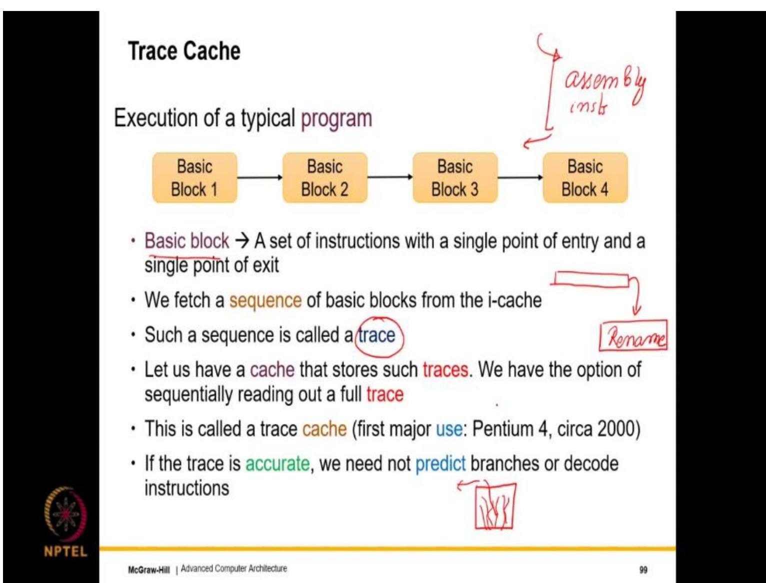


(Refer Slide Time: 00:35)



Now, we will discuss trace Caches. So, trace caches as I had discussed in the last part of the last lecture, there caches the store and entire sequence of instructions, which are a decoded sequence of instructions, such that we can totally eliminate the overhead of the decoder as well as the branch predictor.

(Refer Slide Time: 01:01)



So, how does the execution of a typical program proceed? So, we typically execute programs a basic block after a basic block. What is a basic block? A basic block is a sequence of assembly instructions that has one entry point, in the sense you can only enter

it in the beginning. And it has only one exit point, which means you can only exit it at the end; which means there is no branch that points to the middle of a basic block and there is no branch that jumps from the middle of a basic block.

So, you always enter the control from the beginning of the basic block and you always leave at the end. So, nothing happens at the middle. So, as I said, a basic block is the set of instructions, assembly instructions, machine instructions with a single entry point and a single exit point. So, we fetch a sequence of basic blocks from the i cache, such a sequence is called a trace.

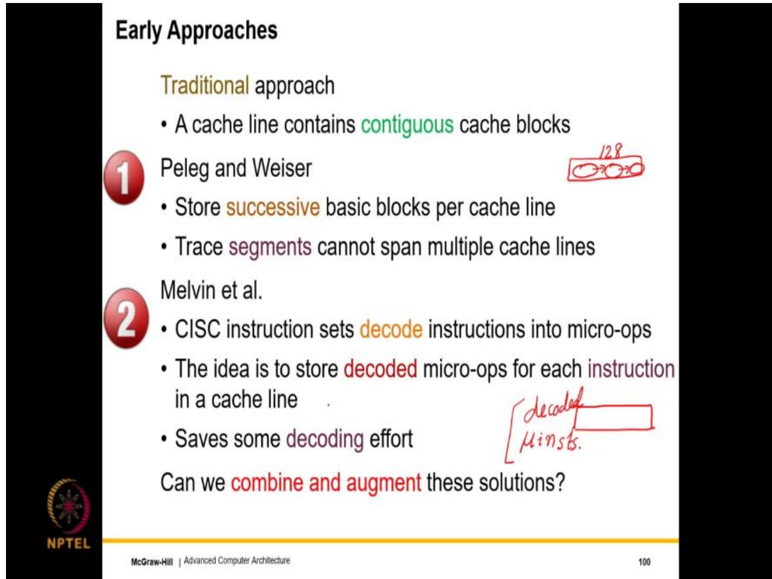
So, if we store such traces; so what are we in a sense storing? What we are in a sense storing is basically an unrolled execution; a segment of an unrolled execution and further more we can store the decoded instructions. So, then all that we need to do is, we need to take a trace which is a sequence of basic blocks, which most likely this will be the sequence based on our previous predictions and we feed this directly into the rename engine.

So, what we can do is, we can have a cache that stores such traces; we have the option of sequentially reading out a full trace. So, this is called a trace cache which Intel used for the first time with pentium 4, that is around 2000. And the trace cache is by and large very accurate and the it stores instructions in the most likely order in which they will be accessed as opposed to an i cache, which stores the most frequently used addresses.

So, we can think of this as a different method of storage right as opposed to the i cache, this is a different method of storing. And we need not predict branches or decode instructions, which as I have said is an advantage of this mechanism. So, we can think of a trace cache as like a box of noodles, where essentially every strand is a trace. So, it is a box of chow mein.

So, essentially what will do? We pick out one such strand and we slurp it out. So, the same way we read out an entire trace and give it to the pipeline.

(Refer Slide Time: 04:09)



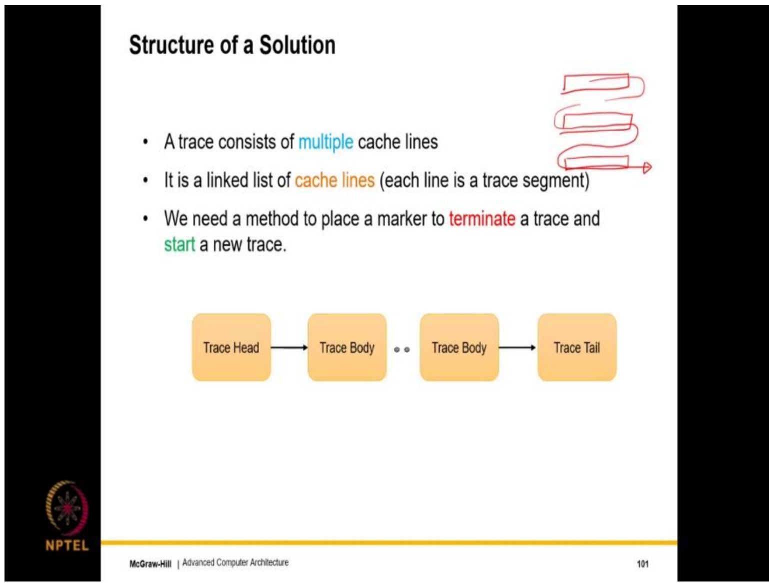
So, the trace cache was broadly speaking a fusion of two older ideas. So, what are these older ideas? So, of course, the traditional approach was that a cache, cache line contains contiguous cache blocks, but again not a very good idea. So, what Peleg and Weiser did is that, in a single cache line, they stored successive basic blocks.

So, what they would do is; let us say that if this is a cache line and let us say you consider a long line 128-bit line. So, you store one basic block, then the one that is most likely accessed after it, that is the way that they use to store. But one limitation of this approach was that, this trace segment which is a sequence of basic blocks, could not actually span multiple cache lines.

So, that became the problem to a certain extent. So, Melvin et al what they did is that, they proposed a different idea, where as we have seen in CISC instructions, we decode them into micro ops. So, the idea is to store decoded micro ops for each instruction in a cache line. So, instead of storing instructions as Peleg and Weiser were doing, in every cache line they were storing decoded micro instructions.

So, the idea here is that, we store the decoded micro instruction, such that the next time we do not have to use the decoder. So, this is like an advancement of a pre decoding, something that we had studied way back in chapter 3. So, can we combine and augment these solution? So, the answer is yes and that is how the broad idea of the trace cache was born.

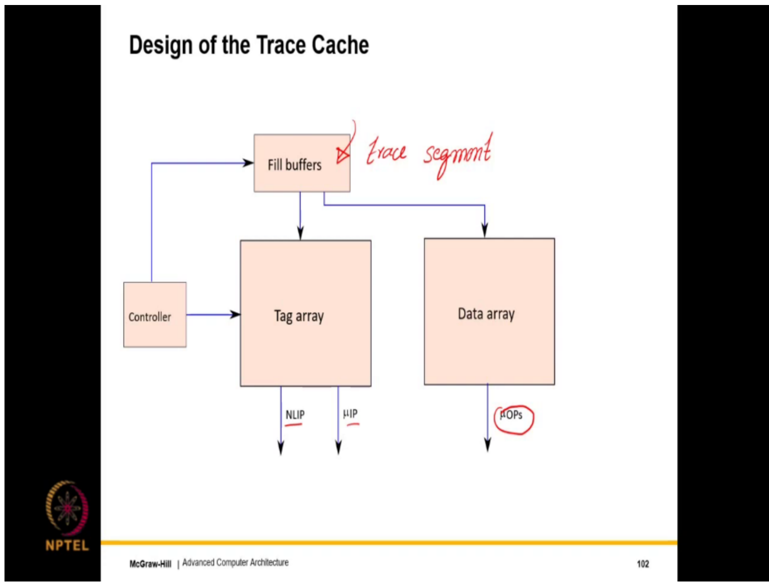
(Refer Slide Time: 06:16)



So, the structure of a solution is like this, let us consider a trace and let us assume that it consists of multiple cache lines. So, in this case, we are talking of a long trace that combines both the approaches. So, in this case, we can have multiple cache lines and we can say that a trace and was spans through all of this. So, this is like a linked list of cache lines, where each line is a trace segment, broadly speaking this is what we want to construct.

So, we need a new method to place markers, single marker actually to terminate a trace and start a trace; say any trees trace the way that we would see it, would consist of a trace head, a set of trace bodies in a trace tail. And they would essentially be a sequence of cache lines in a i cache; because we are fundamentally not changing the organization of the i cache. So, that is not changing, per say that is not changing. So, because that is not changing, we need to construct a linked list out of what we already have.

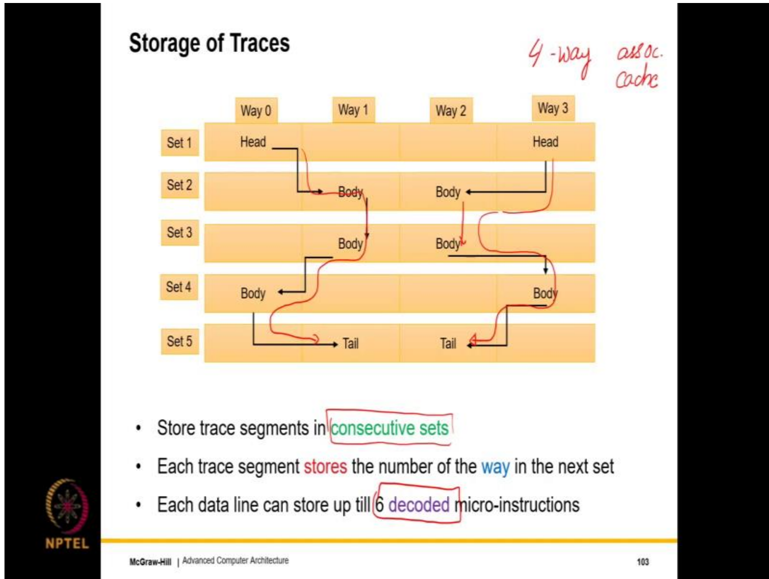
(Refer Slide Time: 07:33)



So, the basic structure of a trace cache; this is the way that it would look that, we will have the same data array and tag array. So, let us not worry about these terms for the time being. So, micro ops of course, this is something that we know, which comes out of the data array; but for the rest let us not worry about NLIP and micro IP for the time being. But we have a controller, in this case the controller is smart; because the controller is something that needs to implement the trace cache, so it has to be smart.

And we have a fill buffer that locally constructs a trace segment and once the trace segment has been constructed, the trace segment is transferred to the data array primarily, but an entry is also made in the tag array. So, this is broadly speaking an overview of the trace cache and let us get into the details. So, now, we will discuss what exactly is stored where.

(Refer Slide Time: 08:37)



So, here is the broad idea. So, pretty much every cache line stores the trace segment. So, let us divide the cache or let us visualize the cache set wise. So, let us assume that each row is a set and within each set this is a 4 way associative cache, so we have 4 lines in each set. So, in this case the 4 lines as you can see are arranged horizontally and what we can visualize over here is that, we can visualize two traces.

So, one trace starts from way 0 set 1, this is the next trace segment; then we have a sequence of body trace segments, this is how the entire trace flows and finally, we have the trace tail. On similar lines we have another trace that is stored like this; so you should had an arrow here and then it flows like this and finally, we have it over here.

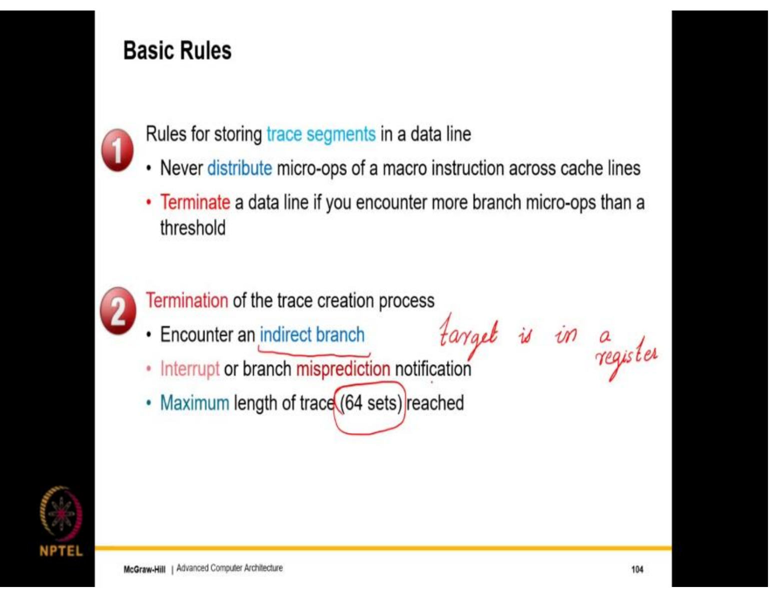
So, if we have a large number of sets and a large number of; but not a large number, but four to eight lines within the set. So, we can organize them in this fashion, but there is there are some important patterns to note over here; the first is that trace segments are stored in consecutive sets.

So, that is important, for a trace cache, this is very important that, trace segments are stored in consecutive sets; each trace segment stores the number of the way in the next set. So, basically since they are store in consecutive sets and it is not necessary to actually store the index of the next line or the next set.

So, that is not required; because if you are in set i, we know that the next segment is in trace i + 1. So, the only piece of information that we need to store over here is the number of the way in the next set and there are some limitations on the number of decoded micro instructions micro ops that can be store in each data line.

So, limit of 6 has been placed. So, this was also the so, this limit is basically; because number one a decoded micro instruction requires some space, that is point number one. And the second is that you have a limited rename, but a limited rename bandwidth, so that is why we have such a limitation, but broadly speaking this is the structure.

(Refer Slide Time: 11:26)



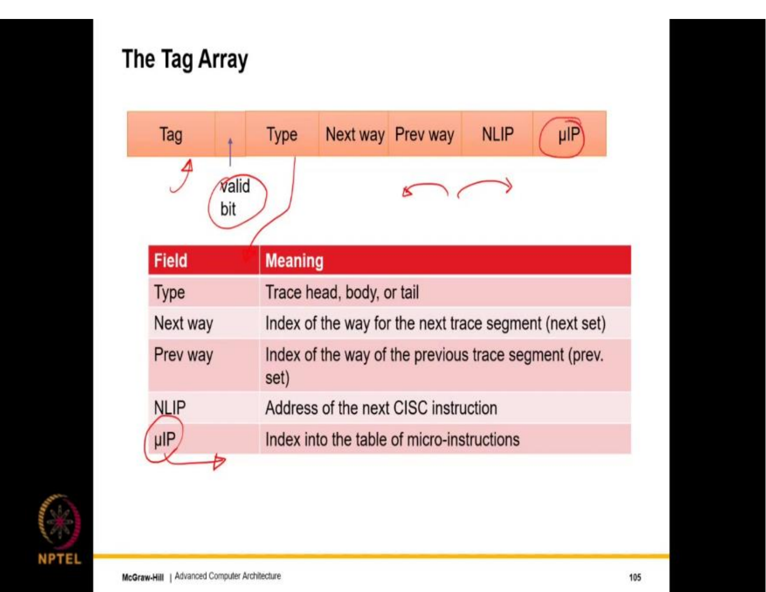
So, let us now look at the rules for storing trace segments in a data line. So, the first rule is that we never distribute micro ops of a micro instruction across cache lines. So, it is never the case that a single micro op is partitioned, a part of it is there in cache line one and a part of it is there in cache line two.

So, that is never the case; the second is that we terminate a data line, if we encounter more branch micro ops than a threshold. So, this we have already discussed, the threshold is 6 and of course, this can vary with a trace cache design; but the important point is we have a limit on the maximum number of micro ops per line and also a single micro op is never split. The other is that the trace creation process is terminated upon the following conditions.

When an encounter an indirect branch; so what is an indirect branch? It is where the target is stored in a register. So, when a target is stored in a register, we terminate the trace creation process. And so, this is primarily because any branch miss prediction either in the outcome or in the target with in any case terminate the trace and since indirect branches targets are hard to predict, it is a much better idea to terminate that trace when you encounter such an instruction.

The other is that, if there is a branch miss prediction or an interrupt comes then also it is terminated. And also there is the maximum length of a trace which is 64 sets; if this is reached, we do not lengthen the trace any more. So, this puts a maximum limit on the size of the trace, which anyway should be there because the branch prediction methods have finite accuracy, have limited accuracy. So, this anyway should be there.

(Refer Slide Time: 13:33)



So, now let us look at the tag array. So, in a tag array we have the following fields, let us look at one after the other. So, one of course is the tag, is a regular tag; then the valid bit which means whether the entry is valid or not. So, nothing no surprises in this, the next is a type. So, the type is essentially that trace segment type it can be a head, a body or a tail.

So, this information is kept over here the next is next way, which is the index of the way for the next trace segment. So, we know that three segments are stored in consecutive sets and each set contains multiple ways. So, we want to find out the index of the way in the

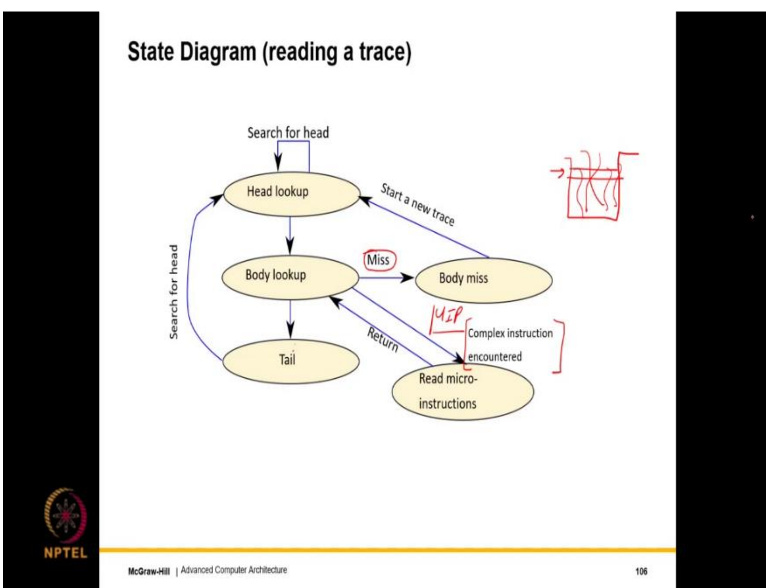
next set next set previous set. So, it is a doubly linked list, in the sense there is one pointer to the next and one pointer to the previous.

So, we have a pointer to the next way and the previous way. NLIP is the address of the next CISC instruction. So, this is required internally; because we would like to know which actual CISC instruction, the current one is and the next one is. So, also this information is kept; because we also want to have address information with the trace, because this is important, many instructions such as load effective address require it.

And we also have micro IP, mu IP, which indexes into the table of micro instructions; just in case we have these complex instructions which cannot be decoded, but we need to need to read the microcode table. So, this is an index into the table.

So, as we see that when we are looking at a practical system, which is Intel trace cache and by the way all of you are encouraged to read Intel's original patent on trace cashes. So, this has all the details. So, what is being presented here is somewhat at a high level. So, nevertheless the key point is that, we do store decoded instructions; but occasionally there is a need to index going to the table of micro instructions, hence we have this pointer.

(Refer Slide Time: 16:00)



So, let us look at a few state diagrams of reading a trace. So, we start from the head lookup state, where we are trying to find a trace head. So, we search for the head; when we find the head of the trace, we go to the body lookup state. So, in the body lookup state we keep

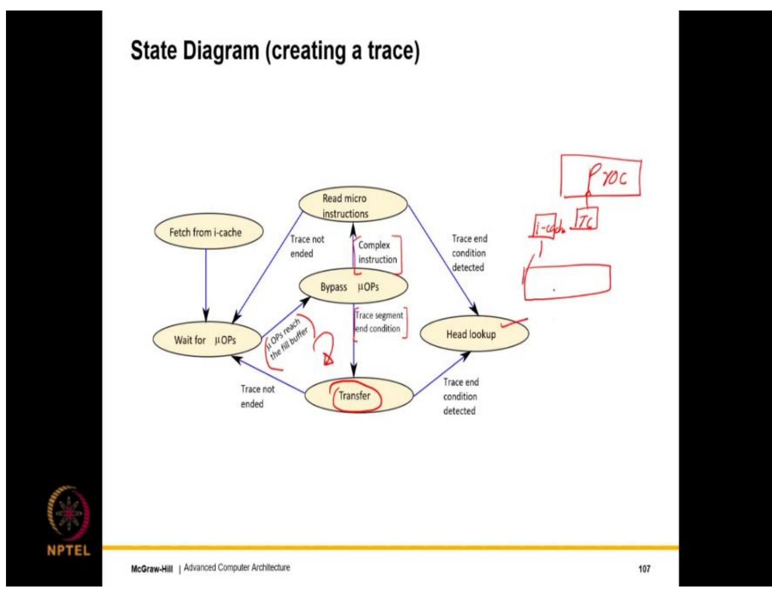
reading trace bodies. So, let us say that there is a miss, in the sense that a trace segment is aviated.

So, we are not able to find the next body of a trace, we go to the body miss state. So, again we essentially start a new trace from there; because the entire trace could not be read. However, if that is not the case, we keep on reading the trace until we reach the tail. Another exception is possible at the body lookup state, which is if we reach a complex instruction, which is something where we will not decode it, but we will rather read the microcode from a microcode table.

So, in that case we read the micro instructions and then we return, we come back to the body lookup state. So, this is where the micro IP pointer is useful and after reaching for the tail, we again search for the next head of the trace and we again read out the next trace, alright.

So, this is the simple idea as I said let us look at it as a box of chow mein, where essentially these are the traces and the way we have implemented is it is that each row is essentially a set and tracers are stored in consecutive set. So, you just store the number of the way in the next set. So, it is actually a doubly linked list, such that you can know about the current segment and the previous segment as well.

(Refer Slide Time: 17:45)



For creating a trace, so for creating a trace what we do is that, we along with a trace cache, we have a regular I cache as well. So, the processor diagram is that, we have a trace cache which we prefer; because it is faster, but we have a regular i cache as well, which is connected again to the memory system. So, we fetch instructions from the i cache, we wait for their micro ops to come and then the micro ops are temporarily buffered in the fill buffer.

So, if you would recall several slides ago; when I had showed the system diagram, I showed the fill buffer where the micro ops come and they are temporarily stored there. So, we reach the bypass micro ops stage. So, then this is where we are kind of buffering them in the fill buffer; from this state, there are two transitions.

So, let us first take a look at the normal transition, where we encounter a trace segment end condition, which is one of the conditions that I showed earlier, where let us say we have filled up the line with a sufficient number of micro ops, then we transfer it.

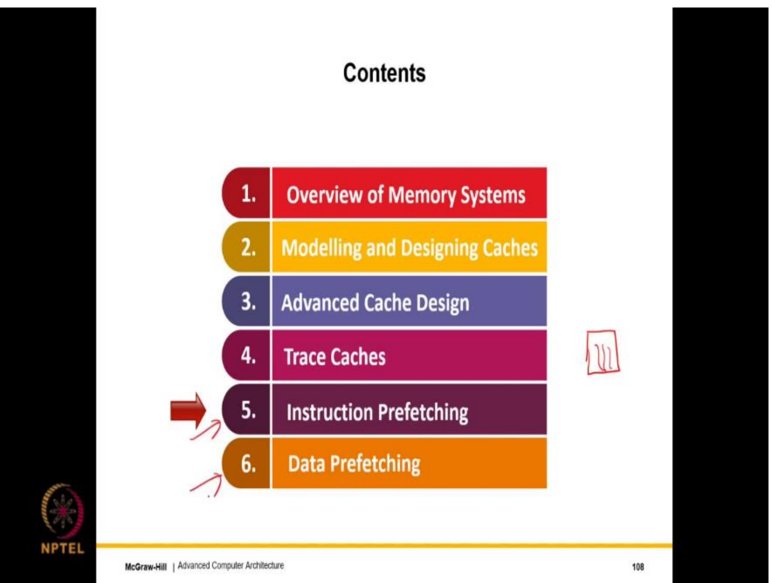
So, we transfer it to the trace cache. And if the trace is not ended, this process continues, so this loop over here continuous. The other transition from this state can be when we encounter a complex instruction. So, there what we do is that, we can have two options; either we can add indirect way to the microcode table and we can come back or we can read the micro instructions, formulate the trace segment and again come to this point.

So, this of course, would depend and intern has not released a lot of details of its patent in its patent; but what computer architecture sense would tell me, what should be done is that, we should take a look at the CISC instruction. If let us say it translates to a few micro instructions, then they can be directly put in the fill buffer and entered as a trace segment; otherwise there should be an indirect into the microcode table and we should read out the trace from there.

There are again two exceptional conditions possible in the transfer and read micro instructions state. So, in a transfer state, it could be that we detect the condition for a trace end. If that is the case, well that is fine; then we terminate the trace. So, again we search for a new head and we always try to read existing traces if they are there or we transition to the trace creation condition and that was shown on the previous slide.

Similarly, when we are reading the micro instructions, it could be that when we are reading it; when we encounter a complex instruction, we could see that we have reached the trace end condition. So, then again we end the trace and we come to this state, where we look up search for the next trace.

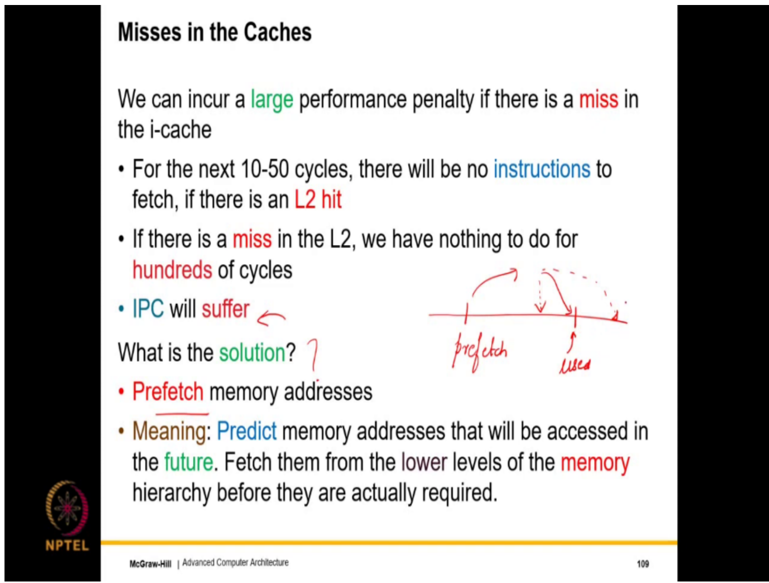
(Refer Slide Time: 20:59)



So, now we have completed trace caches. So, an important point is due here, we need to explain why did we go for trace caches in the first place; what was the need? The need basically was that we were not happy with the throughput and speed that the i cache was giving us; because after all the i cache does not, it is kind of like a dumb organization of data. So, we looked at a much smarter organization, where we stored data traces and we kind of read out these traces.

So, that was the broad idea of having a trace cache, that we have these stresses and we read out these traces. But I mean trace caches are worthy idea; but something that is much more popular it is a far more popular and used or the pre fetching techniques, where we try to guess the addresses that we will access later in the future and we try to fetch the data corresponding to those addresses, they can either be instruction addresses as you can see or regular data addresses.

(Refer Slide Time: 22:16)



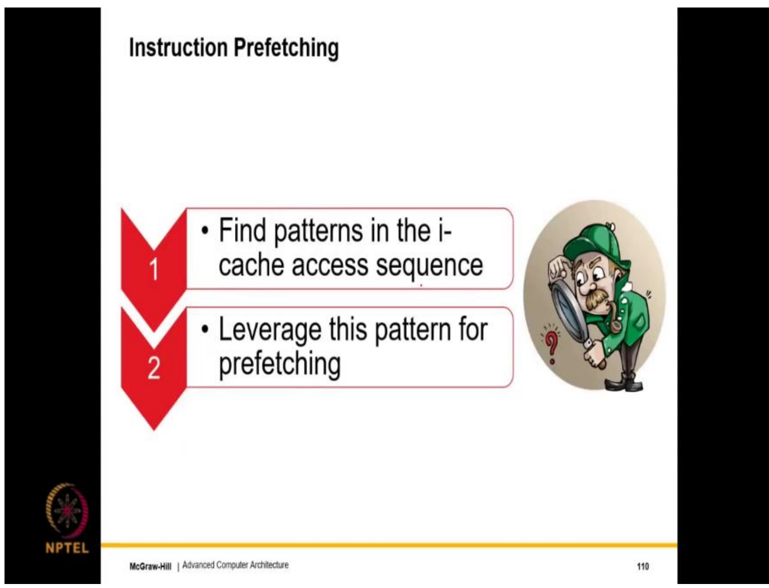
So, let us look at misses in the caches. So, if there is a miss in the i cache, it is actually a very serious issue; because the first part of the pipeline is the in order part and a miss is expensive. So, this misses cannot be hidden. So, it is the miss in the L 1 cache well then we will have to go to the L 2 cache and L 2 cash access later see as you will see in the next chapter, where we will discuss NOC; they can be reasonably large like 10 to 50 cycles, there can be large latencies.

So, because of that we would like to avoid L 1 misses as much as possible and we would definitely like to avoid L 2 misses; because an L 2 miss will again lead to hundreds of cycle, hundreds of cycles are wasted were no worth basically. IPC will suffer. What is the solution? Prefetch memory addresses, which means predict memory addresses that will be accessed in the future; fetch them from the lower levels of the memory hierarchy before they are actually required.

So, what we do is that, at some point we have a prefetch message that goes to the memory system, it comes over here. And it should come just in time, not earlier and not later; but maybe slightly earlier, before it is actually used. Because if it comes in very early, let us see if it comes in over here; it means it will displace some useful data from the cache.

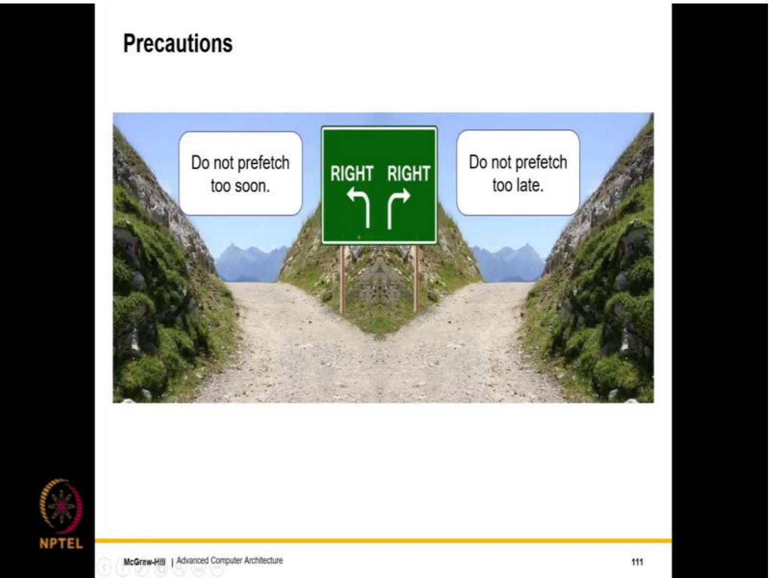
So, it will increase the number of misses, it come, if it comes in too late; it means we still have to wait for this duration. So, that is not useful. So, it should come in just before the first usage.

(Refer Slide Time: 23:59)



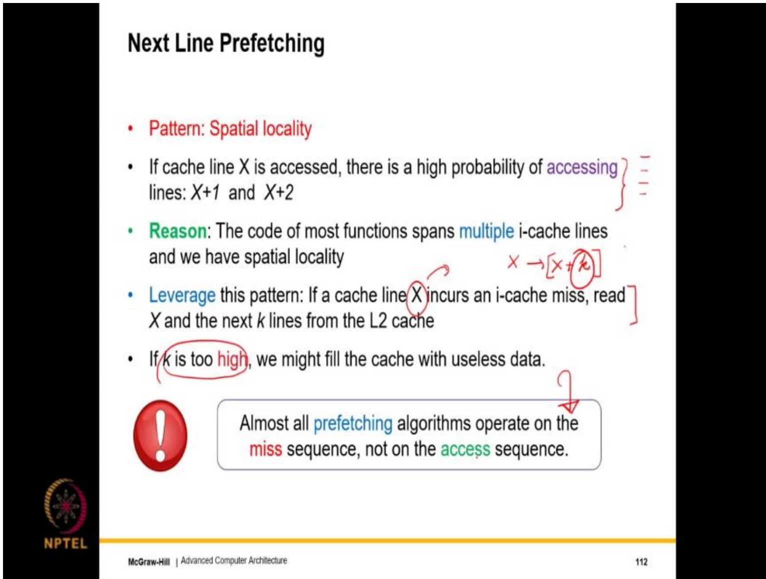
So, the broad idea is that, we find patterns in the i cache access sequence; we leverage this pattern for prefetching. So, we learn a pattern and this pattern is leveraged.

(Refer Slide Time: 24:11)



So, as I discuss what are the two precautions that we need to bear in mind that, we do not prefetch too soon and we do not prefetch too late. So, those are the two mistakes or the two bad things that can be done with prefetching, which is that we either prefetch too soon or we prefetch too late.

(Refer Slide Time: 24:34)



So, let us look at one of the most basic instruction prefetching algorithms, which a surprisingly works very well. The pattern is spatial locality, say if cache line X is accessed; there is the high probability of accessing lines [X + 1], [X + 2], [X + 3]; this is primarily because of spatial locality that, we access one line then the next, then the next, then the next and so on.

At the code of most functions, typically spans multiple cache lines and we have spatial locality. So, because we have spatial locality, this works. How do we use our leverage this pattern? Well the way that we use our leverage this pattern is that, if a cache line X incurs an i cache miss; then what we would do is that, we would read X and the next k lines from the L 2 cache.

So, then what will happen is that, we always operate on the miss sequence, not on the access sequence; because if you were operating on the access sequence, what would happen is that, our prefectcher will be active almost most of the time. So, most of the time our prefectcher will always be churning the sequence and understanding what the sequence is doing, trying to predict and that will be a lot of activity, a lot of power.

So, you always operate on the miss sequence and if let us say that the missed line X, then we prefetch line [X + 1]. This can be made smarter, because typically this might be too late. So, there are (Refer Time: 26:19) of this idea, where if let us say we miss with miss

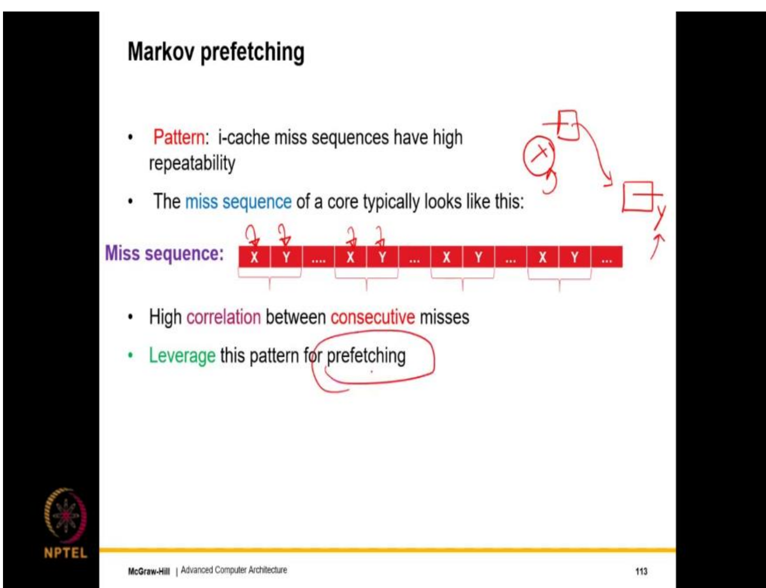
online with address X; we prefetch the line which is [X + k], in the hope that this will ultimately be accessed.

Similarly, we miss on [X + 1], we prefetch [X + k + 1] or maybe fetch multiple lines at the same time or we fetch the next k lines; there are numerous variations of this idea. So, this idea has a lot of variations and it needs to be found out from simulation studies, which one is actually the best; because in every pattern we are trying to use spatial locality, but the spatial locality across programs may differ and we need to find the best possible way of doing it.

So, there are a lot of prefectures are dynamically tune these parameters k and so on, such that we are able to fetch the correct data and the correct data also arrives at the right time. So, clearly if k is too high, we will prefetch a lot of useless data, not a good idea.

And the most important point you need to keep in mind is that, almost all prefetching algorithms operate on the miss sequence, not on the access sequence. So, the sequence is always the miss sequence; you miss on one line, you predict that, another line will miss, so you go and prefetch it.

(Refer Slide Time: 27:48)



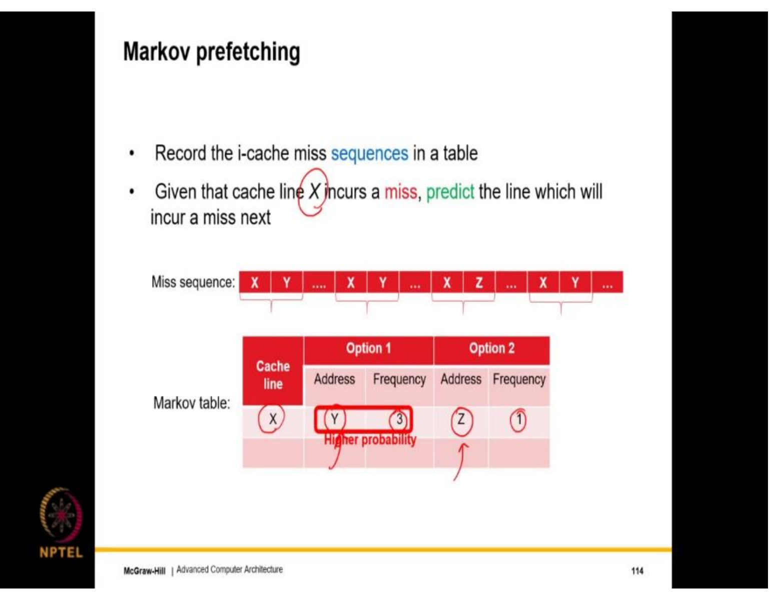
Next line prefetching is still only for the next line. So, the in a sense it is reach is limited. So, it is not really that smart a scheme, a slightly smarter scheme is Markov prefetching; in this case, we will have i cache miss sequences that are not necessarily to contiguous

lines and they will have higher repeatability. So, the typical miss sequence of a core would look like this that, if you miss on line X; most likely the next miss will be to Y, again X, then Y and X and Y need not be continuous.

So, this can be let us say one function call over here, it calls another function over here; there is one miss to a line over here, another miss to a line over there. In that case, there is high correlation between consecutive misses; but the addresses of the consecutive blocks where there is a miss, these addresses are not necessarily contiguous. So, one can be address X, other can be another address Y; but the moment that the processor sees address X, it can immediately start the process of prefetching Y, because it remembers these correlations.

So, this is similar to the idea of a Markov chain, mathematical construct that we have in probability. So, Markov chain can be used here also, used here as well for recording these correlations and for using these correlations for instruction prefetching.

(Refer Slide Time: 29:25)



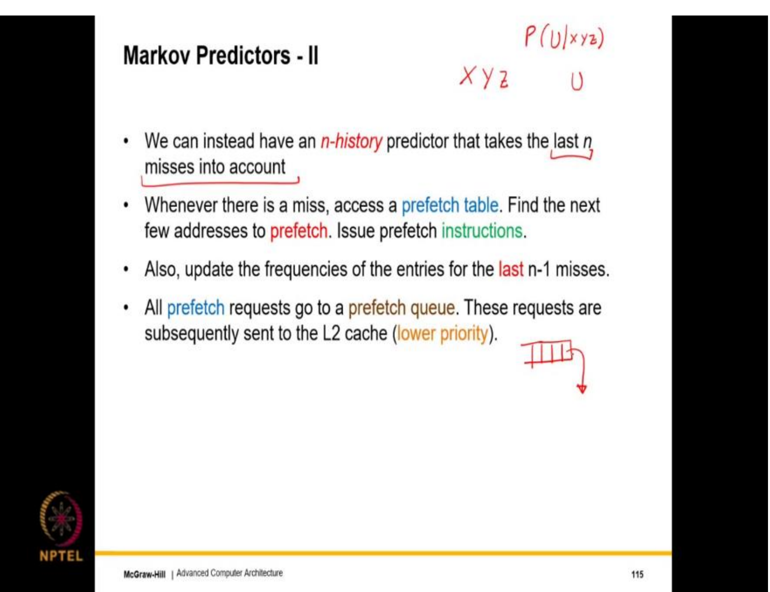
So, what we do we do? Well, we record the i cache miss sequences in a table; given that line x incurs a miss, we predict the line which will a miss next. So, what we do is, we have a Markov table, which say that look line X had a miss. So, then we will have a set of options. So, after X, the next miss was to Y, this was recorded 3 times; the next miss was to Z, this was recorded once. We could have a few more such columns and pretty much

the frequency will change, may change over time; because the behaviour of the program may change, but regardless of the fact, the table will capture this behaviour.

Now, what we can do is that, when we see X; we just need to access this table and find the entry with the largest count and prefetch that, because more likely that line we will miss next. Of course, again variations of this are possible, in the sense we can prefetch both Y and both Z possible.

So, I will not discuss the variations; but I think we are in a position where we can appreciate the pros and cons of each variation. So, I have a rectangle around the higher probability miss which is to Y, that has the higher probability, in this case; but so, these again these counts will saturate, we need to periodically decrement them. So, we are seen a similar pattern in other chapters and sections as well. So, this is what is going to be used here also.

(Refer Slide Time: 31:07)



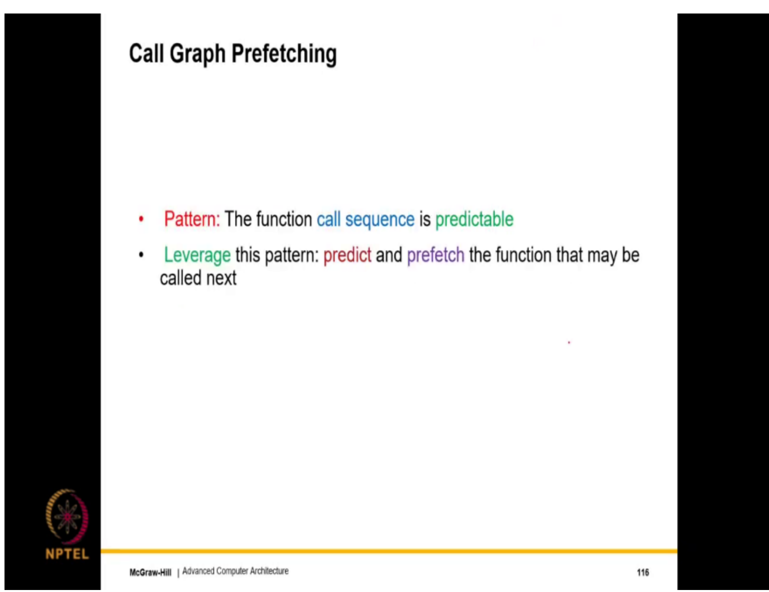
Markov predictors part II, well instead of having a single history predictor; we can have an n history predictor, that takes the last n misses into account. So, this will just complicate our table.

So, this will say that look if misses are to X, Y and Z, where X, Y and Z stand for three different addresses; then the next miss will be to line U. So, this is again we can think of n probability as we are looking at the joint distribution of this. And so, we are looking at the conditional probability that, given what is the probability of a missed line U; given that

we have seen this sequence and we will choose that line which maximizes the conditional probability.

So, wherever there is a miss, of course we access a prefetch table; we find the next few addresses to prefetch and we update the frequencies of the entries for the last (n – 1) misses. We send all the prefetch request to a prefetch queue. The prefetch queue stores all the prefetch request that need to be sent; but clearly these are low priority messages. As compared to let us say an actual miss that happens, nevertheless gradually the prefetch queue is drained; which means that requests are sent to the memory system, the requests are serviced and then they come back.

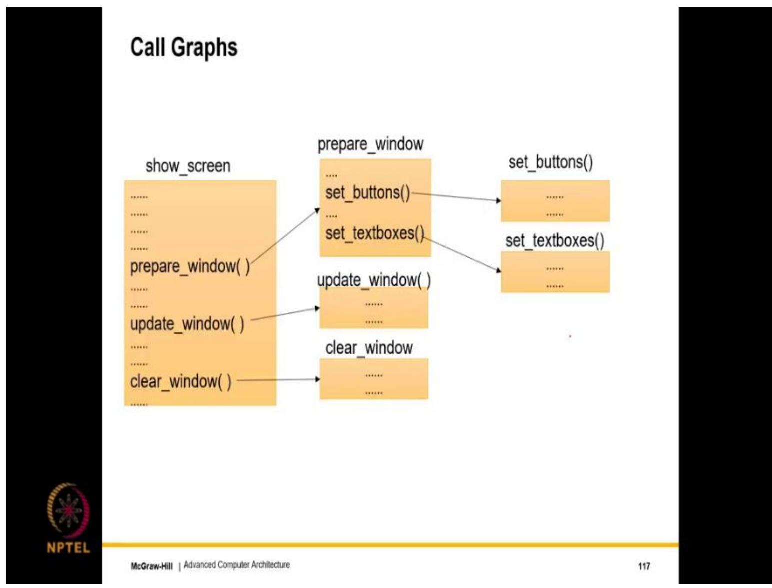
(Refer Slide Time: 32:45)



Now, let us look at a still more advanced method of instruction prefetching, which is called call graph prefetching. So, we can say that, the function call sequence is broadly predictable; the basic block sequence is also predictable, but the function call sequence is even more predictable.

So, if we operate at the granularity of functions, we might get a better prefetching outcome. So, we can leverage this pattern, we can predict and prefetch the function that we call next and then we can fetch all the basic blocks that are a part of the function. So, recall that we had studied basic blocks when we were discussing trace caches.

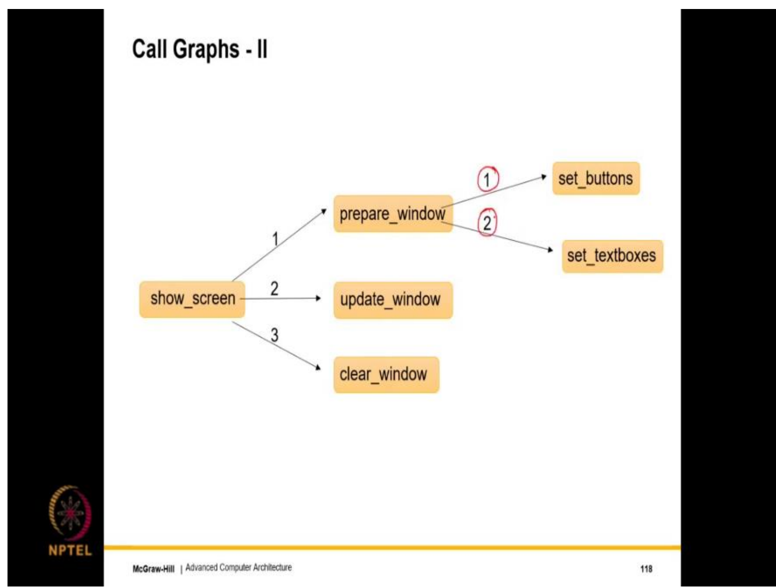
(Refer Slide Time: 33:30)



So, let me take a simple codes snippet from the code of the Linux gnome toolkit. So, what this does is that, when we are trying to show a screen; any screen in the gnome toolkit, gnome toolkit, we first call prepare window which. So, so it is not exact code, but it is kind of inspired by what happens.

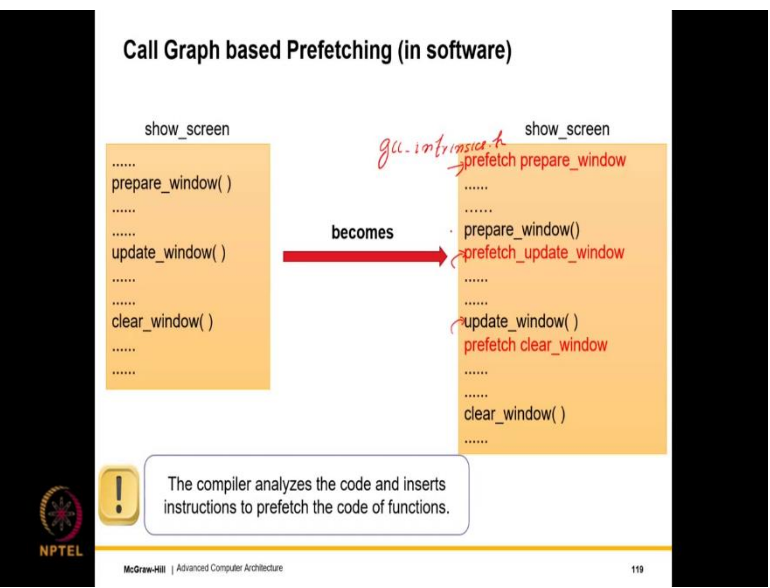
Say in prepare window, we set the buttons and text boxes, which are different functions in their own. Then we update the window with whatever recent information that we have and finally, we clear it; finally, we clear it if you want to redraw something else.

(Refer Slide Time: 34:16)



So, there is clearly a call graph, where each node is a function and each edge is an invocation of a child function. So, show screen first calls repair window, then update and then clear 1, 2 and 3 and prepare window call set buttons and calls set text boxes first 1 and then 2.

(Refer Slide Time: 34:38)

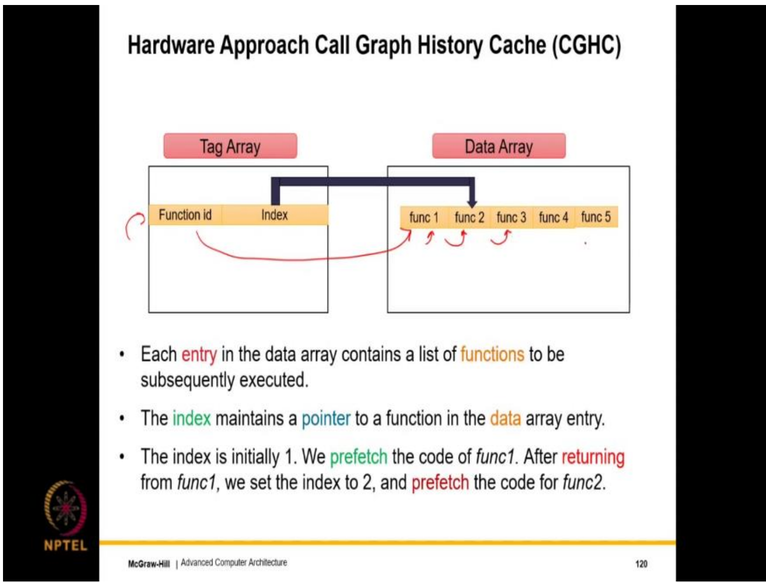


So, what we can do is that, there are two variants of call graph prefetching; one is in software and one is in hardware. So, we can do it in software first. So, this would require compiler intervention. So, what the compiler would basically do is that, it would add instructions to the code, where the instructions will go to the memory system and prefetch instructions and prefetch the data for instructions.

So, almost all instruction sets have such prefetch instructions and see you can look at gcc intrinsics dot h on Linux; see it will show you built in macros that, can be used to issue such kind of prefetch instructions. So, broadly speaking what the compiler would do is that, when we are entering the show screen function; it will prefetch the first function. After we return from the first function, it will prefetch the code of the second function and after we return from the second function, it will prefetch the code of the third function.

So, the compiler will essentially analyze the code, all the code and insert prefetch instructions for fetching the code of functions that will be invoke next. And what this is going to do is that, this would really reduce the time that we spend on misses.

(Refer Slide Time: 36:07)



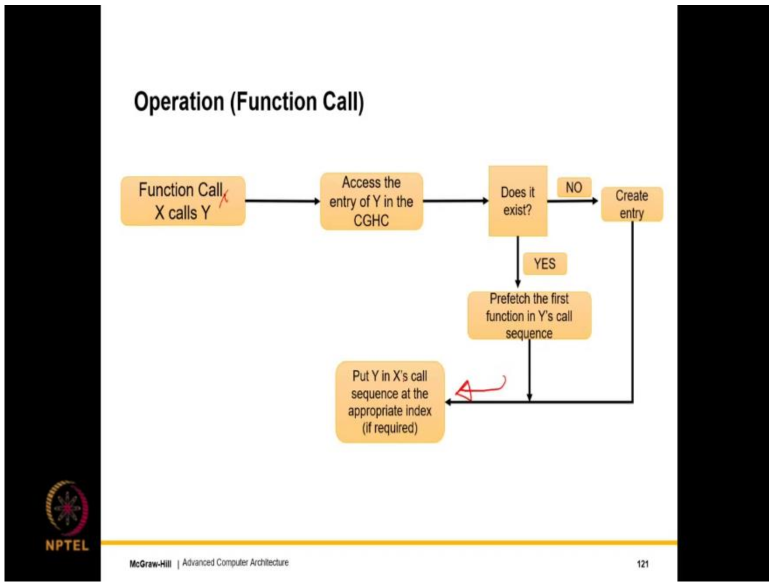
Because recall that, the argument that we have been making is that, instruction misses are far more expensive than data misses. The reason is that, if there is a miss, if there is a data miss; we will always have other instructions that can be executed, this is because of out of order execution.

And so, we will have other instructions that can be executed. So, we might not perceive the delay to that extent; but instructions since they are an in order, they enter in order part of the pipeline and remain there for a large part of their execution, essentially fetch gets stalled. So, so that affects the rest of the pipeline much more.

So, we would like to have a higher accuracy with instructions preferably. So, the key hardware that is used to realize call graph prefetching, uses a tag array and a data array. So, in the tag array we have the function id as the tag, in the sense that the function id is used to index the entry, to find the correct entry. And the function id can very well be the address of the first instruction of the function.

So, this can represent the function id and the data array basically contains the addresses of all the functions and this function is expected to call. And the index would first start with let us say the index to the func 1, when it returns the index will point over here; then when it returns index will point over here. So, the index always maintains a pointer to a function in the data array entry, that is going to execute the next. So, index will initially be 1 as I said; then 2, 3, 4 and 5.

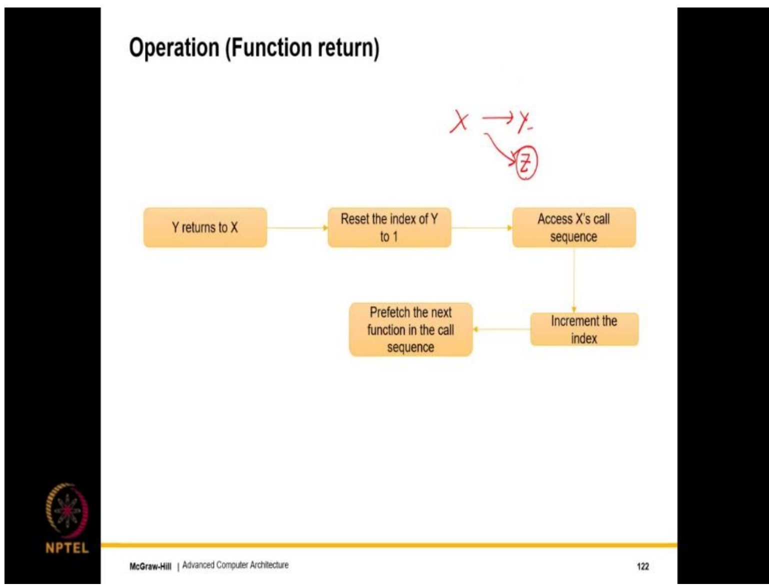
(Refer Slide Time: 38:03)



So, how does this scheme work? Well, the way that this works is a function call X calls Y; we access the entry of Y in the CGHC, which is the call graph history cache. So, does it exist or not? If it does not exist, we create an entry; if it exists what we do is, we prefetch the first function in Y's call sequence.

So, once you have entered Y, so Y will start calling some functions. So, in that we prefetch the first function. And in both cases we come to this box, where we put Y in X's call sequence or the appropriate index if required. Say if it is not already there, then we put function Y in function X's call sequence if required and we proceed.

(Refer Slide Time: 38:55)

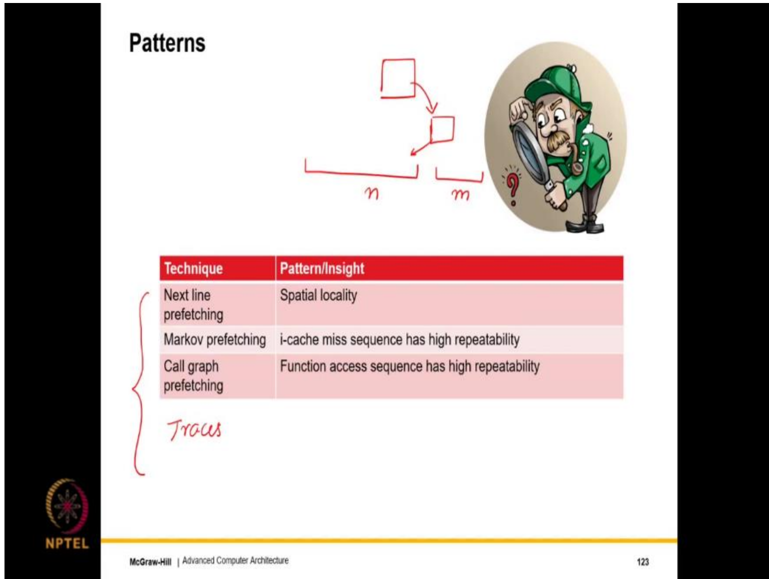


What is the operation of a function return? The operation of a function return is like this that, let us say that Y returns to X; at that point the present (Refer Time: 39:12), the present life of Y is over. So, we reset the index of Y to 1. So, the index of Y gets reset to 1; then we access X's call sequence.

So, we if we access the call sequence of function X; what we do is that, we increment the index, because let us say if X was calling Y, then after that it will call some other functions. So, we increment the index and let us say that is function Z.

So, you prefetch the next function in the call sequence. So, basically increment the index and we prefetch the code of function Z, which is what we had described two slides ago. So, this is how a call graph history cache works, where the code of one function is prefetch, then the next, then the next, then the next and so on.

(Refer Slide Time: 40:06)



The key inside in all the three schemes has been the following; the first is next line prefetching, which is a straight forward expression of spatial locality. Then we did discuss the shortcomings of just the next line or the next k lines or the kth line and we discussed the idea of Markov prefetching; in this case, the sequence has high repeatability the miss sequence, but these are not contiguous lines.

So, these are non contiguous lines like X, Y, Z, U, V, W the example that we gave. So, what we do is, we maintain the frequency of pairs; say it could also be in a n previous misses and the next miss or we can make it generic, we can make it more generic, we can say n previous misses and m subsequent misses, where n and m can both be > 1.

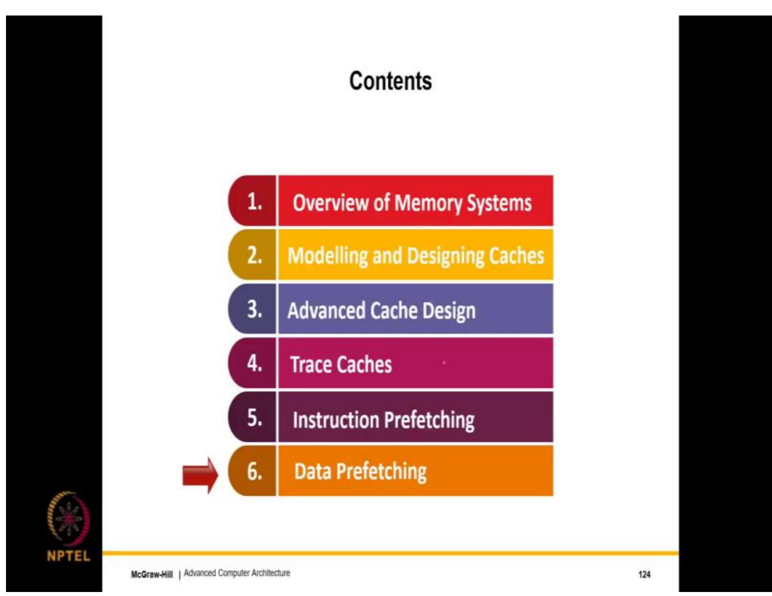
So, we can make it generic different variations of this idea are possible; but the important point over here that, nevertheless it is a; it is a Markov sequence, which follows from the idea Markov chains in probability theory.

Then we discussed the idea that, instead of having prefetching at the level of instructions or basic blocks; can we look at a higher granularity, where we fetch directly at the level of functions, that is where we started tracking function calls and then we prefech the entire functions in one go. So, unfortunately this is still not enough. So, we have seen a higher granularity, where we have moved above functions and we have looked at traces.

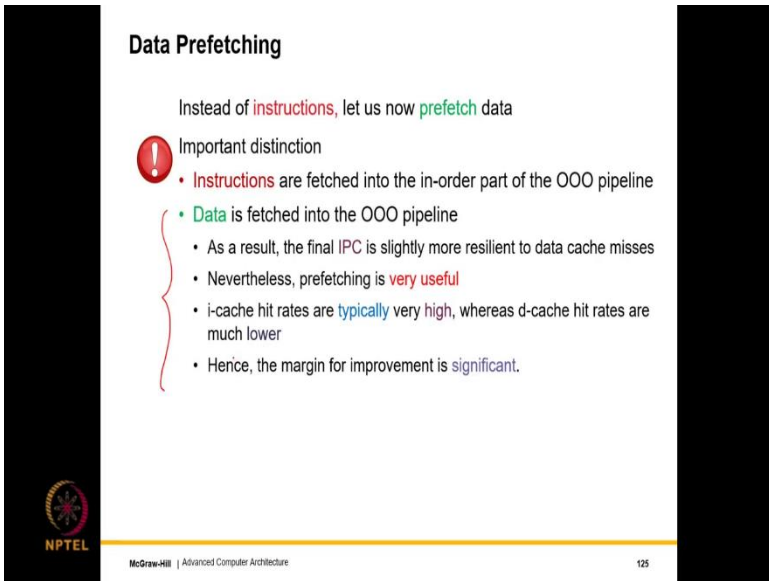
So, much of the recent work also moves to one more level above traces and it tries to look at all kinds of sophisticated ways of constructing traces and maybe it can fetch an entire page of memory or sequence of pages or a subset of a sequence of pages. So, there are lot of, there is lot of current work and lot of current interest in this area, where essentially instead of function; instead of fetching one function at a time, we fetch a group of, we prefetch a group of functions at the same time.

So, this helps increase the efficiency substantially of the entire processing system and it reduces the miss rates quite a bit. So, this is where I would like to terminate as far as instruction prefetching is concerned.

(Refer Slide Time: 42:52)



(Refer Slide Time: 42:55)



Let us now discuss data prefetching; say instead of instructions, the idea now is to prefetch data. So, the important distinction that is being made over here is that, instructions are fetched into the in order part or the out of order pipeline. So, the things are more sensitive over there; because clearly if an instruction is not fetched, we cannot fetch any instructions after that.

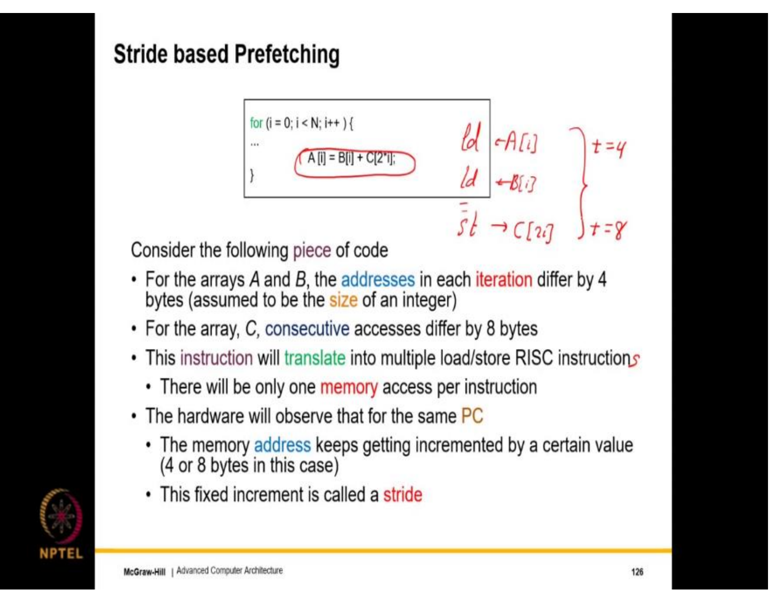
So, as a result what happens is that, there is a slowdown, but data on the other hand; because of the out of order nature of execution, the out of order pipeline is more resilient to a data miss as opposed to an instruction miss. And so, this resilience is primarily because that, if let us say a given load instruction misses; we might find sufficient, a sufficient number of independent instructions, instructions that are independent of the load, which can nevertheless execute.

And they can execute, their consumers can execute, their entire forward slice can execute and they can keep the pipeline busy. So, I cache rates are traditionally very high; that is also because of the pattern and of the code, d cache rates are in comparison lower.

So, let us say the i cache rate you would find to be as high as 90 to 95%, maybe even more than 96, 96, 97, 98%. L 1 data cache hit rates will be in 80 to 90% range the local hit rates and the local hit rates of the L 2 cache might be in the 50, 60, 70% range.

So, the margin for improvement is significant. So, the i cache improvement was more like a low margin business; but this is like a high margin business, where we can do a lot to improve the hit rate. But, of course here the sensitivity is lower than an i cache. So, the same improvement in an i cache will actually give you more in terms of performance; nevertheless, data prefetching is very important and almost all processors have an instruction cache prefetcher and a data cache prefetcher.

(Refer Slide Time: 45:23)



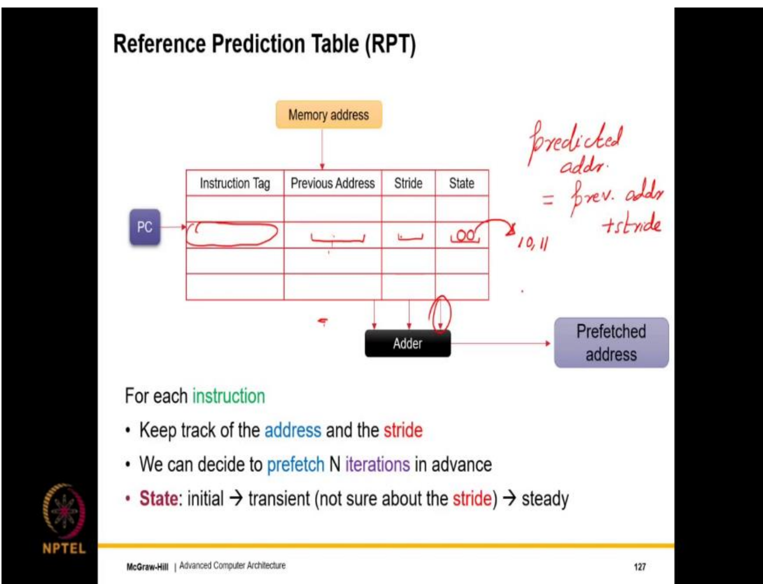
So, let us look at the simplest possible prefetching scheme, which is stride based prefetching; this is a typical example of a for loop and an array axis. So, if you consider arrays A and B; the addresses in each iteration differ by 4 bytes, where the assumption is that the size of an integer is 4 bytes.

But if we consider array C, consecutive accesses differ by 8 bites; because it is 2 into i. This instruction will translate into multiple load store RISC instructionsand there will be only one memory access per instruction.

So, what will happen is that, the hardware will always observe the same PC; but for the same PC, so let us say when we are loading A i. So, I am just writing it an in formal way when we are loading A i into some register; when you are loading B i into some register and then finally, after an ALU operation, we will store a value into C to i. So, for each of these instructions, the PC is the same. So, the PC remains constant; but every subsequent access you will find the addresses increasing by 4, the addresses increasing by 8.

So, this 4 or 8 is known as a stride and if the stride can be predicted; well then we can predict the addresses for an array access very well.

(Refer Slide Time: 46:57)



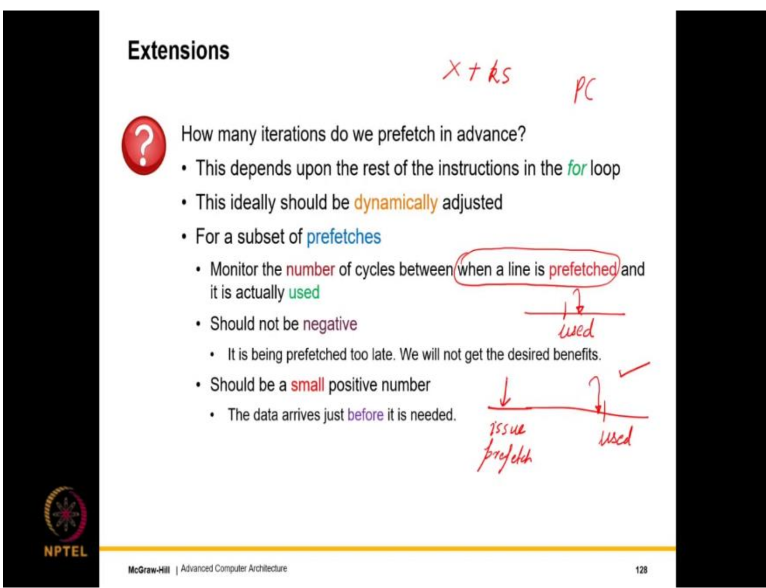
So, we typically have what is called a stride prediction table or a reference prediction table; let us just call it a stride table colloquially, where we will have the PC and the PC will map to the instruction tag. So, this we have seen before; it will have a previous address, the stride and the state. So, let us see what they mean. So, we keep track of the addresses.

So, basically for a given PC, we keep a track of the previous memory address, we subtract the current. So, whenever there is a miss right; what we do is, we subtract the miss from the previous address and if we see whether it is equal to the stride or not. So, if it is not equal to the stride, well we set the stride to this value; but let us say over two or three cycles if you find the stride remaining the same, the state here can be a saturating counter, which you can say 0 0 means that, which is not a strided access.

So, gradually when it becomes either 1 0 or 1 1, we can infer that for a given address; the access pattern does follow a stride and this will be captured by the saturating counter over here. And once we are sure that for a given address, the access pattern does indeed follow a stride; what we can do is that, the next time that we fetch the PC over here, we will know the previous address, we just add the stride to the previous address. So, the predicted address is equal to the previous address plus the stride.

So, whenever we encounter this program counter in the sense we fetch it and we see that the state is steady in the sense that we are following a strided access pattern; what we do is, well we just add. So, the state is not added into be adder, but it only determines whether we will add or not; we add the stride to the previous address and that becomes the address we need to prefetch. So, this basic piece of hardware will take care of most strided access patterns.

(Refer Slide Time: 49:17)



Few extensions; well how many iterations to we prefetch in advance? So, what we can do; so we can make this smarter. What we can do is that, look if the previous address was something X and the stride is S; so instead of just prefetching (X + S), you can fetch (X + k) S.

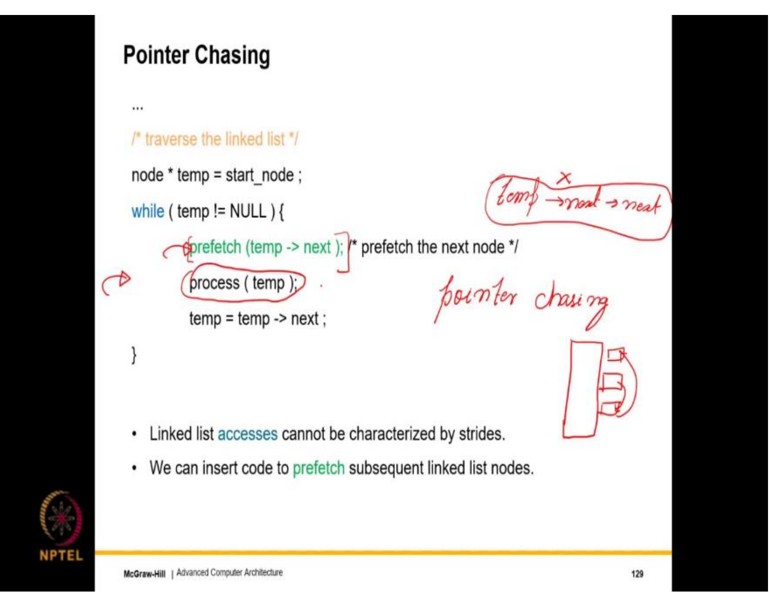
So, then what will happen is that, we can dynamically adjust the distance between when we issue the prefetch instruction and when we receive it. So, this distance can be dynamically adjusted. So, one way of scientifically doing this is, to monitor the number of cycles between when a line is prefetched and when it is used.

So, basically this and then it is clear that. So, it is not when line is being prefetched, it is when it is arriving. So, if it is negative, it is meaning that this is being prefetched too late. So, the way that we interpret this phase over here; that when line is prefetched means, when the data arrives and when it is used.

See if let us say it is used at this point of time and let us say that it arrives over here and this distance is negative. So, essentially this time minus this time; it means that the prefetch data is arriving too late. So, we will not be able to derive the desired benefits. So, the ideally should be a small positive number; which means that if let us say this is where we want to use the data, this is when the prefetch data should arrive. It should not again arrive too early; because it would displace useful data, but it should arrive just before the data is used.

So, then we will be able to do it. So, now, the question is that, if I let us say have a given PC and we know that this is a part of the loop; if I fetch the address that is going to be used in a next loop iteration, it is possible that I might reach the next loop iteration first and the data will come much later. So, what I should do is, actually I should fetch something which is maybe several iterations later. And this will ensure that this distance is just about optimum and this is when the data arrives. So, we have this scenario over here.

(Refer Slide Time: 51:48)



So, there is a fair amount; let me go back to the previous slide, there is a fair amount of research to actually adjust the time between when we actually issue the prefetch instruction and when it is actually used.

So, the point is that, this is the ideal scenario where we issue it and just before it is used, the data arrives. But we will never know when to issue the prefetch instruction; unless we

have a clear cut idea, we are actually monitoring the time, it takes from a prefetch to usage and based on that our algorithm adjusts.

So, what is the node that we have? Well, the single node that we have is how many iterations to be prefetched in advance and that can be tailored appropriately. Now, from arrays let us graduate to linked list. So, what I am showing over here is a typical code of a linked list, where we start at a start node and we just keep on iterating.

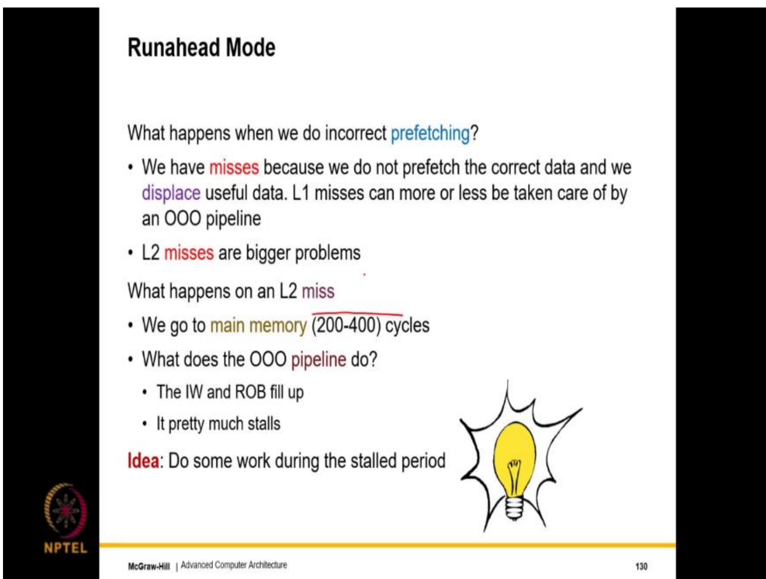
So, essentially temp = temp→ next. So, we keep on iterating. So, then we process a node, go to the next, process that and so on. So, what we can do in software is that, we can have a prefetch command; I can prefetch given a pointer to the next node, it can prefetch the entire next node itself. So, this prefetch instruction or this prefetch function in this case, can do the job; I will be it in a compiler assisted way of actually prefetching the next pointer.

So, this is also called a pointer chasing method, where we are essentially chasing the pointers of a linked list. And what we can further do is, instead of fetching the next one, we can just do this next. So, it can be the next one after that next, next; but of course, to find next, next we will have to get the data.

So, there is no shortcut for doing it. So, that is the reason, this is not a practical idea as compared to temp→ next. But the point is if we have some data structure that, just has this pointer; but does not have the data, some sort of an indexing data structure, then this can be done and we can prefetch nodes of the linked list that we are going to access in the future.

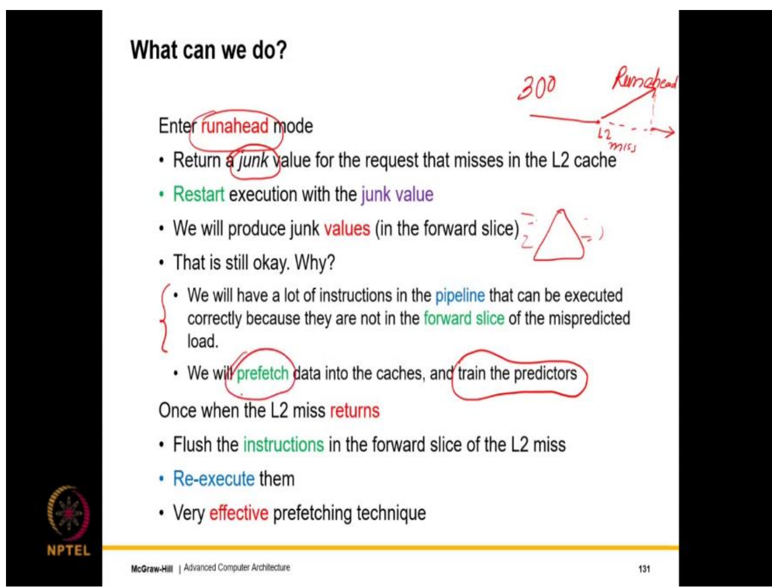
Of course, this cannot be used which strides, because linked list accesses cannot be characterized by strides; because typically in if you assume a linear memory spac, then the objects will be there everywhere. So, the linked list will basically be formed like this. So, there is no fixed strided access pattern, but with some compiler intervention and by adding this prefetch instructions, these the prefetch data can come in.

(Refer Slide Time: 54:46)



What happens when we do incorrect prefetching? Well, we have discussed this many many times; prefetch to early displace useful data, prefetch too late there is no advantage. And again a data cache less sensitive, i cache more sensitive; L 1 misses well that is less of an issue, L 2 misses more of an issue, because we go to main memory, the access time is large very large 200 to 400 cycles. So, our entire pipeline stalls; it fills up, the ROB fills up, then the work get stalled.

(Refer Slide Time: 55:24)



What can we do? Well, a new novel idea; not that novel, it is fairly old, but at least it is a different idea. We enter what is called the run ahead mode; say in this case let us say we have a miss in L 2 cache. So, clearly it will go to memory and that might take 300 cycles. So, it returns the value, which will be a junk value; it will not be a, it will not be a good value, it will be a junk value.

So, we restart execution with a junk value. So, recall the idea with replay, where we had a poison bit. So, what we do is that, any instruction that forwards the junk value to another one, that will also produce junk values. So, is entire forward slice will essentially produce junk values, similar to the idea of poison bits.

So, why are we doing this? Well, we will see in a second, but the key idea is that, if you miss in the L 2 cache the long latency miss, no problem; what we would do is that, we will simply get the data, set a bit, say it is a junk value, propagate to its forward slice.

So, we will nevertheless still have a lot of instructions in the pipeline that can be executed correctly; because they are not in the forward slice of the miss predicted load, see you will have a lot of instructions outside it. If we allow them to execute correctly, in effect what they will do is that; they will train all the predictors and also they will prefetch data. So, they will access data; accessing data means that pulling it from the lower echelons of the memory hierarchy to the upper echelons upper levels. And so, effectively even if we are running in the spatial mode.

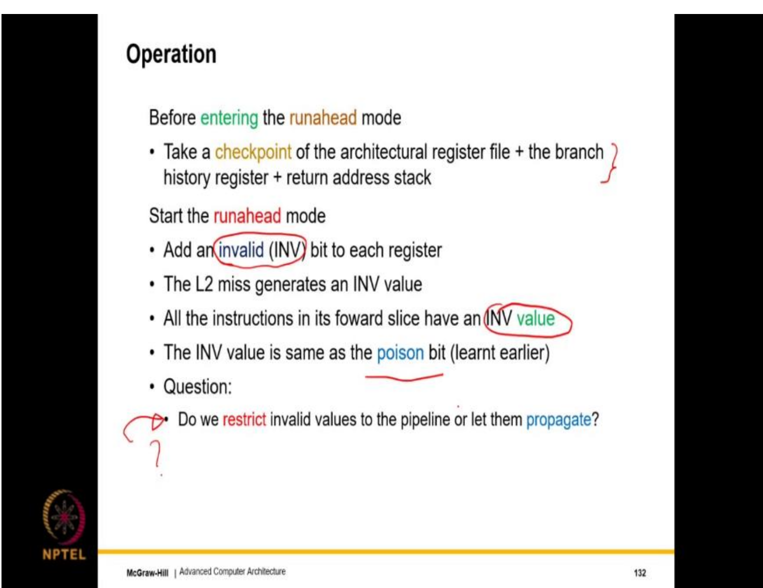
So, what do we do? We have regular execution, we encounter an L 2 miss; then what we do is, we move in a different trajectory which is a run ahead mode, until our L 2 data comes back. Till that point of time we do not do any useful work, in the sense we do not update architectural state; but we execute those instructions that are not there in the forward slice of the load that return the junk value.

This nevertheless do have an advantage and the advantage is that, they do the job of training the predictors and prefetching. Once the real data comes, we restart regular execution.

So, what do we do? We flush the instructions in the forward slice of the L 2 miss; see either we can flush the large part of the pipeline or only those instructions that are in the forward slide. We reexecute them using a replay mechanism and we resume regular

execution. So, this was regular this was run ahead, regular run ahead and regular. So, by doing this, we prefetch data and train our predictors.

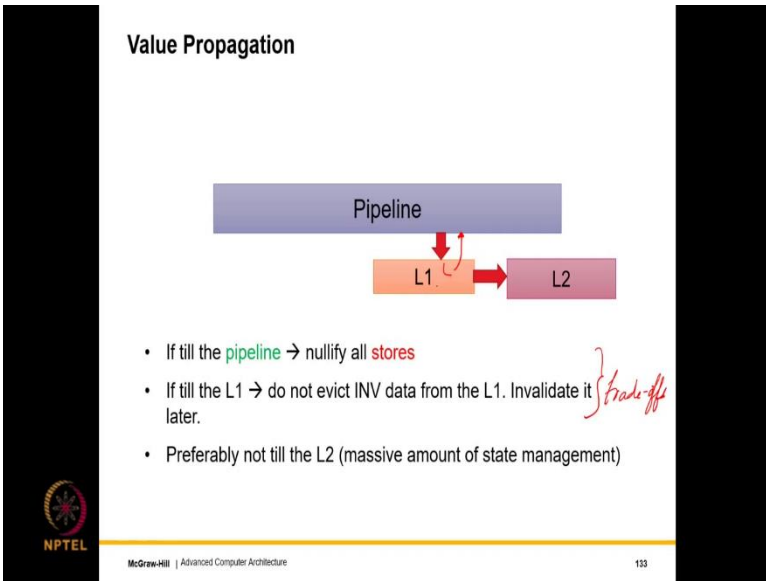
(Refer Slide Time: 58:30)



So, a few more details so, before entering the run ahead mode, we take a check point of the architectural register file, the branch history register, return address stack, such that the run ahead execution does not necessarily corrupt them. So, we do take a check point of these things, then we enter the run ahead mode. So, whenever we read a junk value, we add an invalid bit; all the instructions in the forward slices have an invalid bit, the inv value similar to the poison wit.

So, now, there is the important question that we ask that, do we restrict the invalid values to the pipeline, what if there is a store that is storing, a storing an invalid value, storing a junk value, what do we do?

(Refer Slide Time: 59:18)



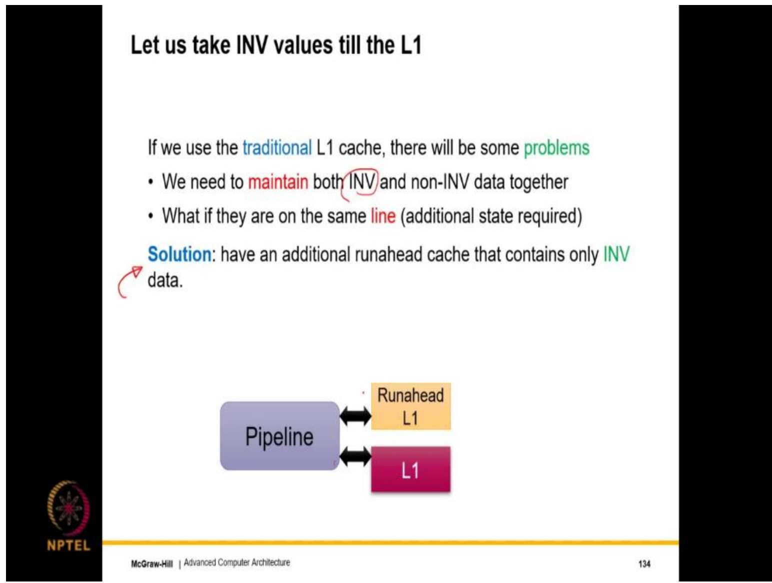
So, this is the typical architecture, where we have pipeline L 1 and L 2. So, if we restrict the changes to the pipeline; well then we nullify all the stores, we do not write anything to architectural state. If it is still the L 1, then that is ok; we can have invalid data in l and then it can be read back from here, but we do not allow any invalid data to go to L 2, because that would involve a massive amount of state management.

And once we return from the runahead mode; so we do have something called a flash clear mechanism; the details are there in the book, where all the invalid lines can just be marked to be invalid, I mean they are just not valid.

So, we can simply ignore them. So, both the options are possible, in the sense that we do not allow any right to the memory, in the pipeline; we just absorb, kill all the stores that are invalid. Or we allow some of some junk data to come to the L 1 cache, particularly if you are predicting the value of the L 2 miss and then we allow it to proceed in the hope that may be some good would have come out of it, at least and that good would be seen in the enhanced accuracies or predictors and also in prefetching.

So, both the options are possible and the tradeoffs need to be well understood. So, let me put it, the trade offs are important of which one to choose. So, there is a need for deeper thinking over here.

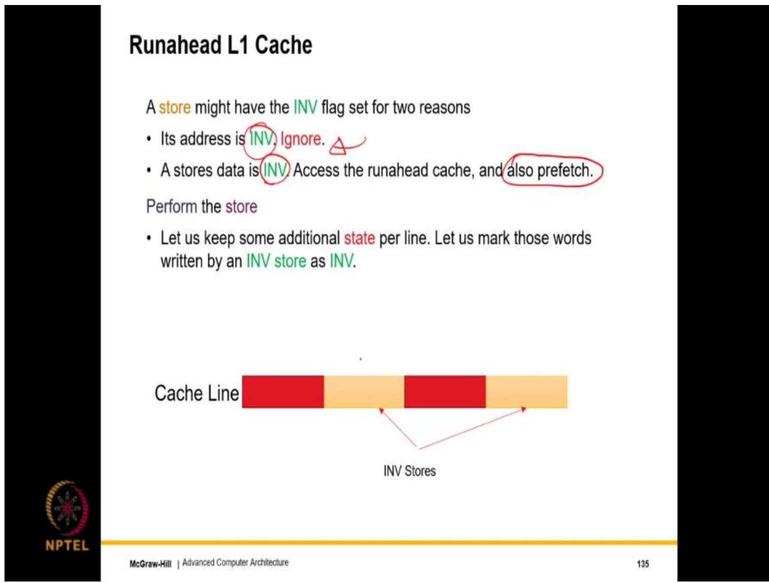
(Refer Slide Time: 60:59)



So, if we take invalid values till the L 1 cache, if you use a traditional L 1 cache, there will be some problems; we need to maintain both INV and non INV data together; they might be on the same line as well. So, it can be in a 64 byte line; 4 bytes are valid, 4 bytes are not valid.

So, one simple solution is that, do not have all of this complexity, have an additional run ahead cache that contains only invalid data, only invalid data is kept there that kind of solve your problem to a certain extent; that this is the pipeline, we will have a regular L 1 and run ahead L 1 and the run ahead L 1 will be used in run ahead mode, particularly for invalid data.

(Refer Slide Time: 61:45)

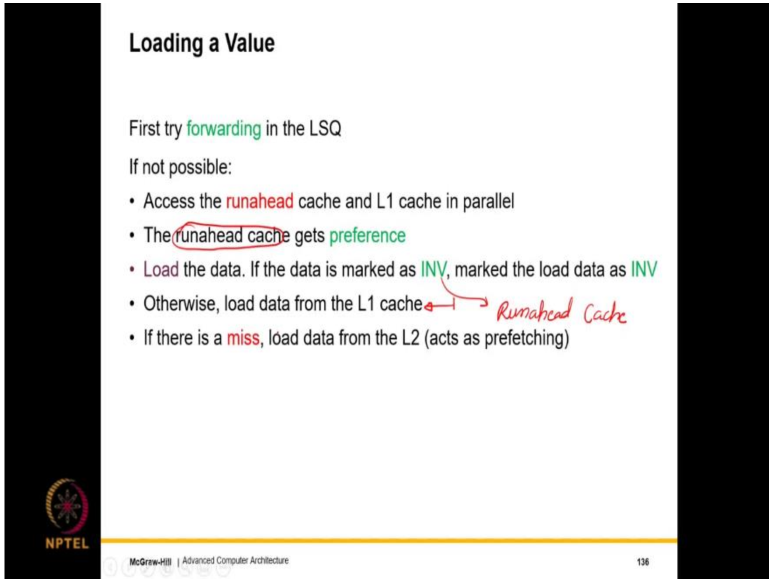


So, run ahead L 1 cache more details, a store might have the INV flag set for two reasons; its address is invalid, then you ignore. So, then it is wrong and let us ignore. A stores data is invalid, but then we should not ignore.

What we should do is, we should access the run ahead cache and we at the same time we should also prefetch; because since the address is correct, the data is not correct, at least we should prefetch. So, the job of prefetching would be to bring the block closer to the core; which means bring it to the L 1 cache, this should definitely be done. We should perform the store; we can keep additional state per line, where we can mark those invalid stores as INV.

So, in a single cache line, it is possible parts of it might be invalid, parts of it might be valid. So, then when the run ahead mode return, some state management has to be done.

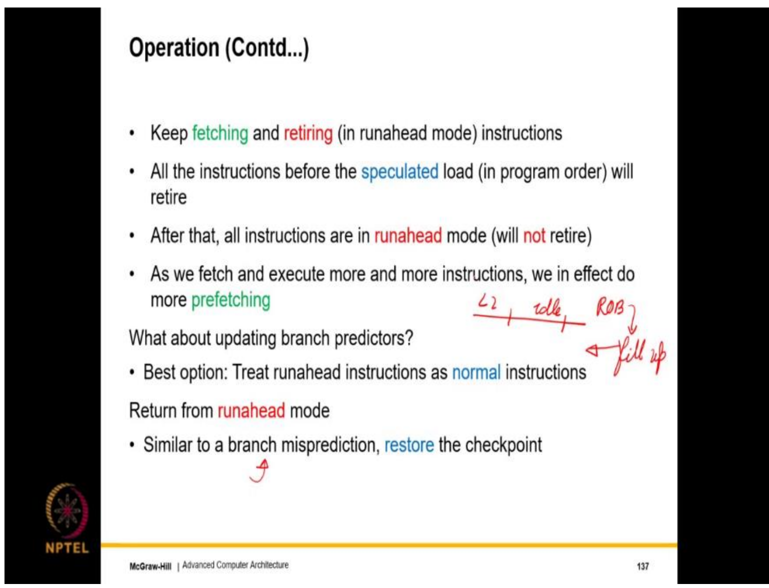
(Refer Slide Time: 62:46)



Loading a value, well loading a value will be the same first try forwarding in the LSQ; if it is not possible, we access the run ahead cache and l one cache in parallel, but of course the runahead cache gets preference. We load the data; if the data is invalid, we have marked the load data as invalid. So, otherwise we load data from the L 1 cache and if there is a miss.

So, so by the way there is an important point to put over here that, invalid data is only restricted to the runahead cache; it never enters the regular L 1 or L 2 cache, it is only restricted to that point, it is only restricted to the runahead cache. And if there is a miss, then we are the same as prefetching; we will load data from the L 2, in the sense we will bring it up.

(Refer Slide Time: 63:43)



Operation continued, we keep fetching and retiring in runahead mode instructions; all the instructions before the speculated load, the one that missed in the L 2 cache they will retire. After that all instructions enter the other runahead mode; which means they will not retire, they will not commit. As we fetch and execute more and more instructions, in a sense we are doing prefetching; along with that, we are also training the branch predictors by treating runahead instructions as normal instructions.

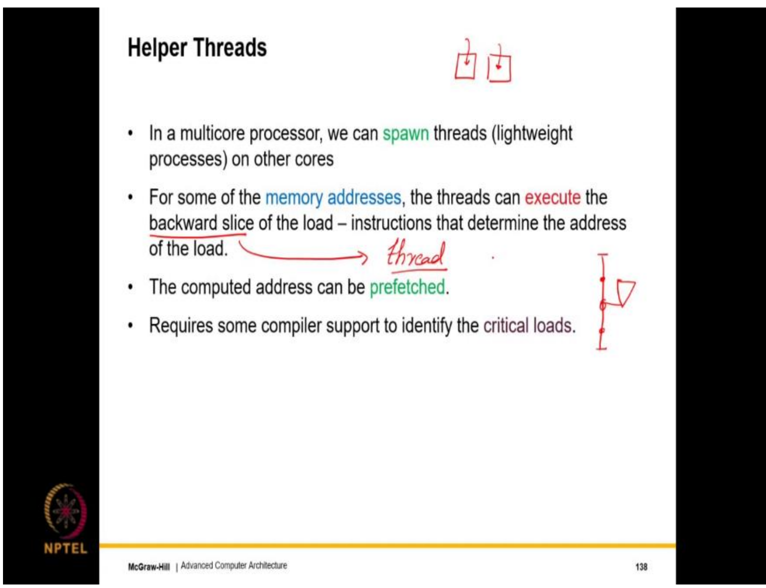
So, definitely some amount of branch prediction training is being achieved. Once we return from the runahead more, again to a branch miss prediction; we restore the check point and we start from there. So, correctness is clearly not being sacrificed over here; but the additional advantage we are getting is that by running these extra instructions, when the pipeline would have anyway idled so.

So, what is the key inside? Well, the key insight is that look; if we have a miss in L 2, after that point for the next 3-400 cycles, the pipeline will basically idle, the ROB will fill up and essentially fetch will stall and the pipeline will idle. Why not use this time to do something productive?

What is it that we can do productive? What we can do is that, we can execute a lot of instructions, which can possibly the data might be junk; but the address will be correct and also instructions that are not there in the forward slice of the miss predicted load. So, that would happen anyway and simply just keep on executing them.

So, if the addresses are correct, that would do the job of prefetching and all the data that we need will move up to the caches, which will save us a lot of time after we exit the runahead mode and entire normal node. So, that is the key basic the most important idea.

(Refer Slide Time: 65:55)



Something similar was proposed by another group called helper threads. So, here of course, we have multiple codes, not simple not a single one. So, in a multi core processor when you have multiple cores, we can spawn threads or lightweight processes and other cores. If you have one thread over here, we can spawn another spawn thread over there. So, let us assume in the program, there are some of these load instructions that are kind of prone to misses, they often miss.

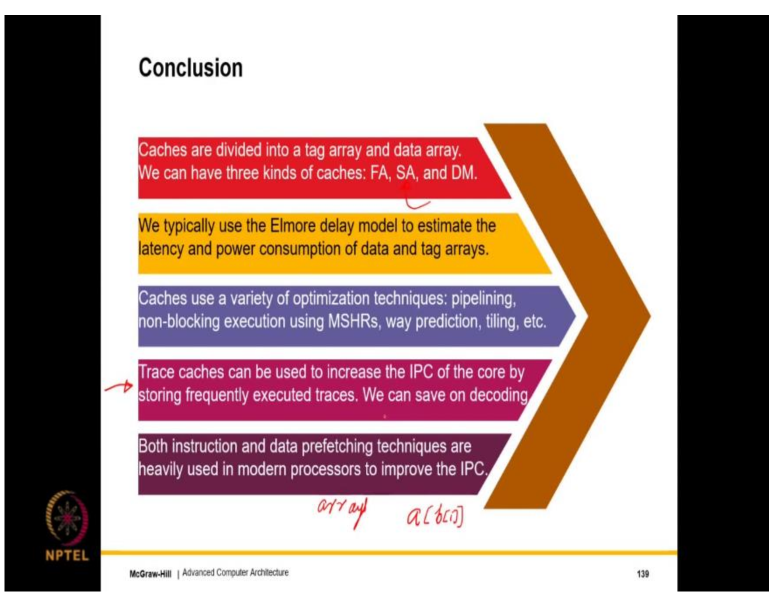
So, for this load instruction, we can store up, we can have a backward slice; the backward slice is a set of instructions that compute the address of this load, this backward slice can be moved to a separate thread on a separate core. So, we can move the instructions and their data, this can be moved. The separate thread can then execute the backward slice in parallel. So, there is no correctness or performance issue over here, that is happening separately; the computer address can then be prefetched.

So, we can it is kind of tag some of these critical loads might require compiler support or might require a profiling run, which is an earlier run to identify the loads that were very hard to predict; their addresses were very hard to predict, for each of them we can store this small backward, where we can just compute the address and prefetch it.

So, essentially for every load we kind of run a small nano program to compute it is address and it is prefetched in parallel.So, the idea of helper threads is pretty popular; I am at least not aware of any processor that directly uses either runahead execution or helper threads, but it is a pretty promising idea and different of thoughts of it have influence different design decisions over the years.

And they might become far more much more popular in the coming years and the other thing is that, many of the tricks that are there in current processors, these are actually not disclosed to the public. So, they may very well be there, it is just that we do not know.

(Refer Slide Time: 68:10)



Finally, the conclusion so, this is the slide we all like to see, this means it ends a chapter. So, let us quickly conclude and be done. Caches are divided into a tag array and a data array; we discussed the three most important kinds of caches, fully associative, set associative and direct mapped, where set associative is by far the most common, because it has the benefits of both the designs.

We use the Elmore delay model to estimate the latency and power consumption of large structures within the caches, wire, sense amplifiers, data arrays, tag arrays. Caches use a variety of optimization techniques, pipe lining, nonblocking execution, way prediction tiling etcetera. They have the limitations in terms of their miss rate.

So, trace caches are used, it is kind of like a smart cache that stores an entire trace of decoded instructions, such that we can bypass both the predictors as well as the decoders. And finally, regardless of trace caches, we do need an instruction prefetecher and a data prefetcher; for that prefetch data as to arrive at the right time which is just before it is used, we should not displace useful data. And instruction prefetching is sensitive in the sense we expect high hit rates.

Data prefetching for L 2 at least the hit rates are low, but again there is a huge margin for improvement. And we basically need to predict the instructions in the future and fetch them. For many access patterns like arrays and so on it is easy, but for linked list then many other irregular data structures; even these doubly indirect arrays, something like this, these are hard, these are irregular data accesses.

So, these are hard to prefetch, but that sudden done; there are lot of these pointer chasing techniques, where we try to traverse the sequence of pointers, either small program does it or the compiler adds instructions to do it or we looked at the runahead more, we looked at helper threads. Fair, we somehow traverse such data structures and prefetch, that is the broad idea. So, now, we have finished chapter 7, that is that should give us a big smile; the next chapter would be the chapter on the network on chip NOCs.