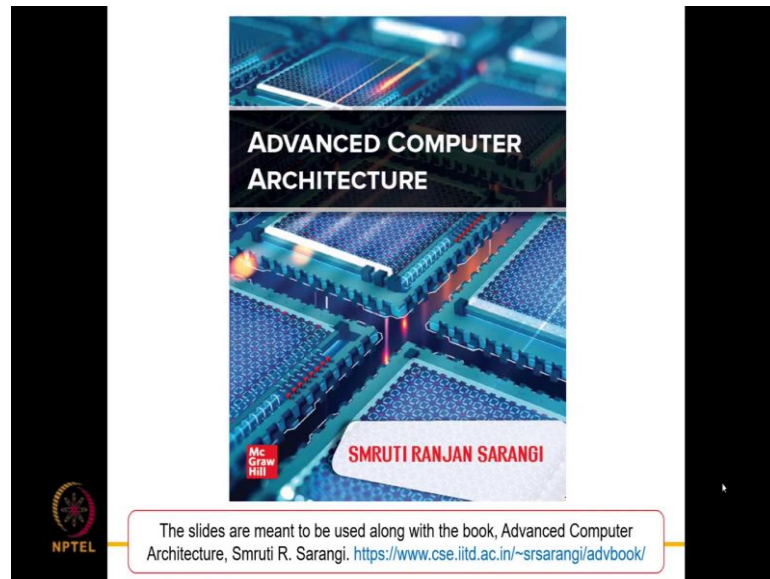


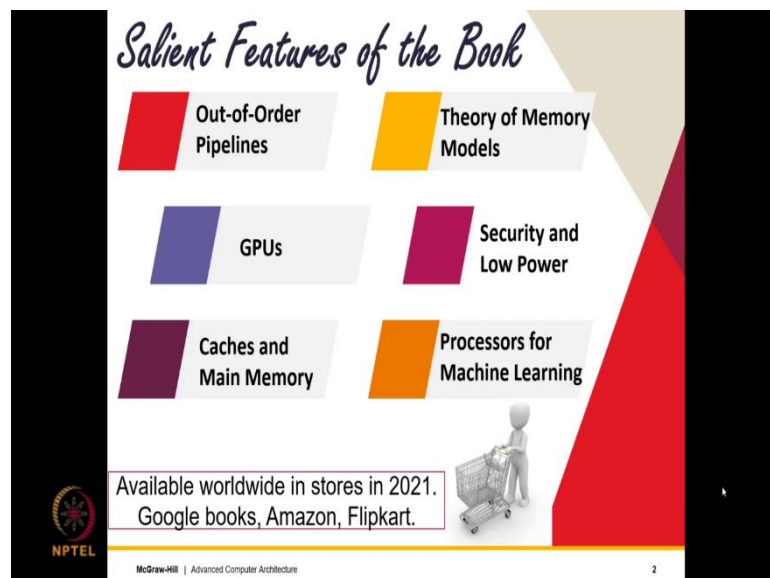
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 12
Alternative Approaches to Issue and Commit Part – I

(Refer Slide Time: 00:17)



(Refer Slide Time: 00:23)

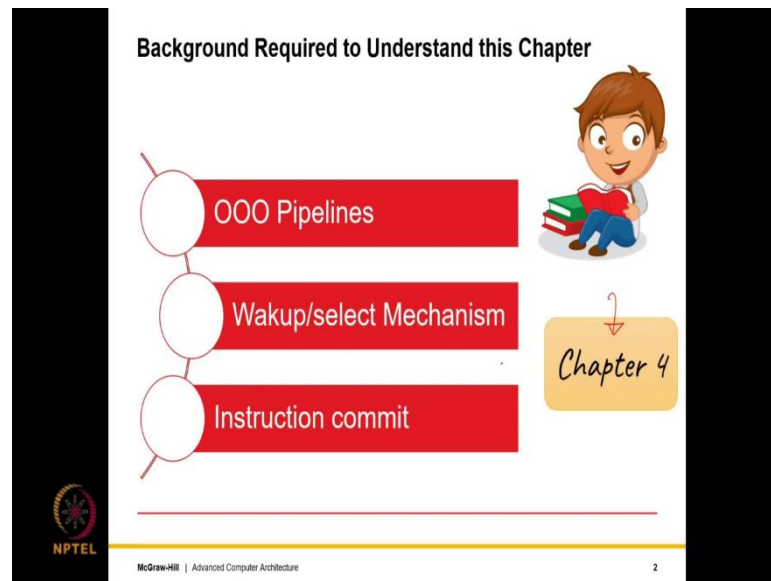


Welcome to chapter 5. In this chapter, we will discuss Alternative Approaches to the Issue and Commit stages. So, issue and commit stages, the way we have described in chapter 4, this has kind of been described in a very simplistic manner. Say modern out of order

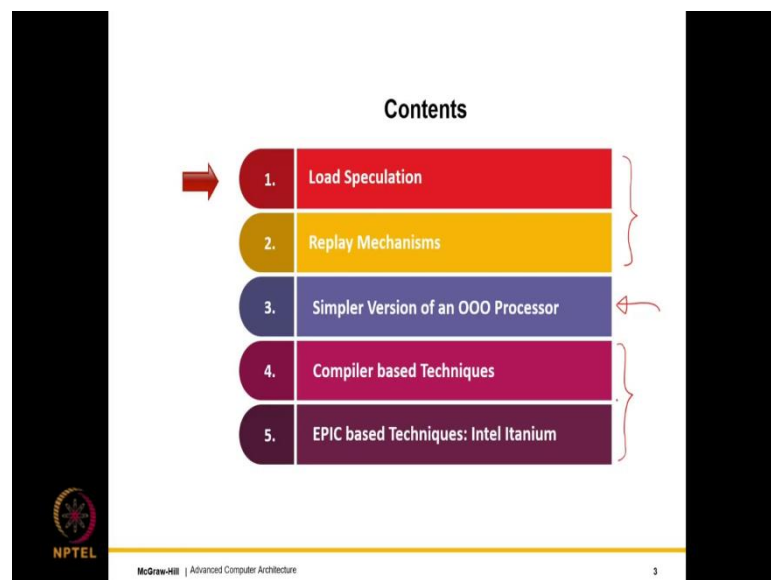
processor is way more complicated than the basic description that has been provided in chapter 4.

Nevertheless, the aim here is to basically look at some of the corner cases and then, move on to a simpler design which of course, is less efficient and then discuss.

(Refer Slide Time: 01:14)



(Refer Slide Time: 01:15)



So, I have the schedule over here and so, then we discuss first we complicate our design, then we simplify it and discuss something which is less efficient and finally, we discuss compiler based techniques where of course, we discuss a different paradigm, where we

So, in this chapter, we do require some background and the background is of course chapter 4 on the issue execute and commit stages. So, we will start with out of order pipelines. This is something that is required. So, the, for the background; then of course, we need to know the wakeup select mechanism in great detail and finally, precise exceptions on the instruction commit. So, the requirements for this chapter are these three things which you will get in chapter 4.




Aggressive Speculation

Branch prediction is one form of speculation

- If we detect that a branch has been **mispredicted**
- Solution:** **flush** the pipeline

This is not the **only** form of speculation

- Another very common type: **load latency speculation** or **value speculation**
- Assume that a load will hit in the **cache**
- Speculatively** wakeup instructions
- Later on if this is not the case: **DO SOMETHING**



NPTEL

McGraw-Hill |

4

So, whatever we do is just with registers and wires and with nothing else. So, with that we implemented a host of stuff. So, the biggest attraction in that sense, in the previous chapter which was chapter 4 was branch prediction which is one form of speculation, where if we detect that a branch has been mispredicted, we flush the pipeline. So, we flush the pipeline and when do we do that?

We wait for the mispredicted branch to reach the head of the reorder buffer, at that point we detect it and we flush the pipeline. This is not the only form of speculation; we have other kinds of speculation as well, which we will discuss in this chapter. So, these other kinds of speculation are low latency speculation and value speculation, which we will discuss in this lecture.

So, the idea here is simple with low latency speculation it is that we have different levels of memory. We have an L 1 cache, L 2 cache, main memory and so on. So, the we have a memory hierarchy basically. So, the pipeline is connected to the uppermost level of memory which is fast, so which we have assumed take 1 or 2 cycles, but that does not contain all the memory locations.

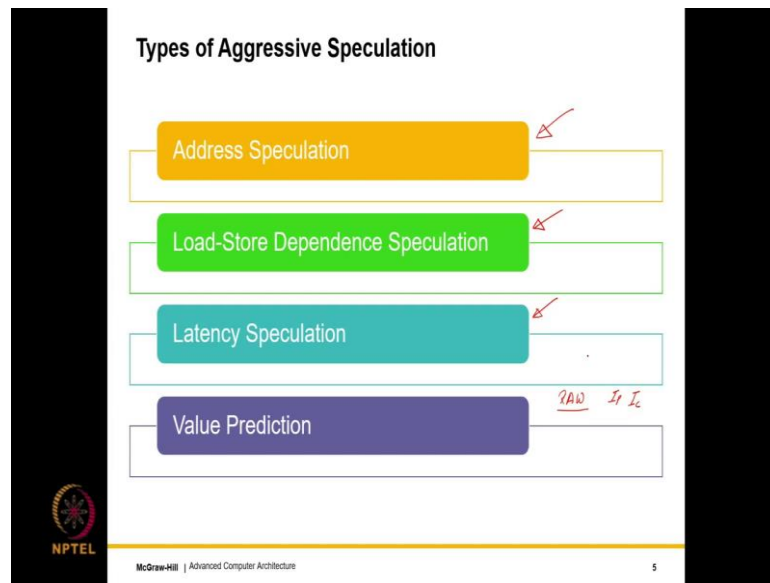
So, then we have an L 2 cache which contains a superset. So, we can make a guess that we will find the value in the L 1 cache; but if we do not find the value, we will have to go to the L 2 which takes more time. But from the point of view of the pipeline when we do an early broadcast, we can as we shall see in some cases, in many cases rather we can make a guess that we will find the load in the L 1 cache and proceed. But what if our guess is wrong? Well, we will see what needs to be done. Similarly, we can also predict values.

So, recall that for read after write dependencies which were true dependencies, we have not done anything. We said that look, it is a read after write dependency, we have a producer, we have a consumer; the producer produces the value, the consumer consumes the value, we have nothing to do that was our stand. But if we have a producing instruction over here, let us say I p which produces something that the consumer instruction consumes.

So, if this is let us say a memory location or a register, we can have a value predictor if it is possible to predict the value. In this case, we can run the producer and consumer in parallel and the dependency is broken; of course, if the value prediction is wrong. So, in this case is speculation because we not only predict, but we also move forward. So, that is speculation.

Say this speculation is wrong, then of course, there is trouble; we need to do something. So, what do we need to do? Well, we will see, but the problem at hand is that in an aggressive processor, we do a lot of guesswork. Based on that, we wake up our instructions and we execute them. Sometimes the guesswork turns out to be wrong and in that case, we need to do something that is the crux of our discussion.

(Refer Slide Time: 06:21)



What are the types of aggressive speculation? Well, there are many many types, but these are clearly the most common. One is address speculation, where we try to predict the address of a load and we dispatch the load early to the memory system such that all the instructions that are dependent on the load can be woken up. This is slightly more specific, where we predict the dependencies between loads and stores.

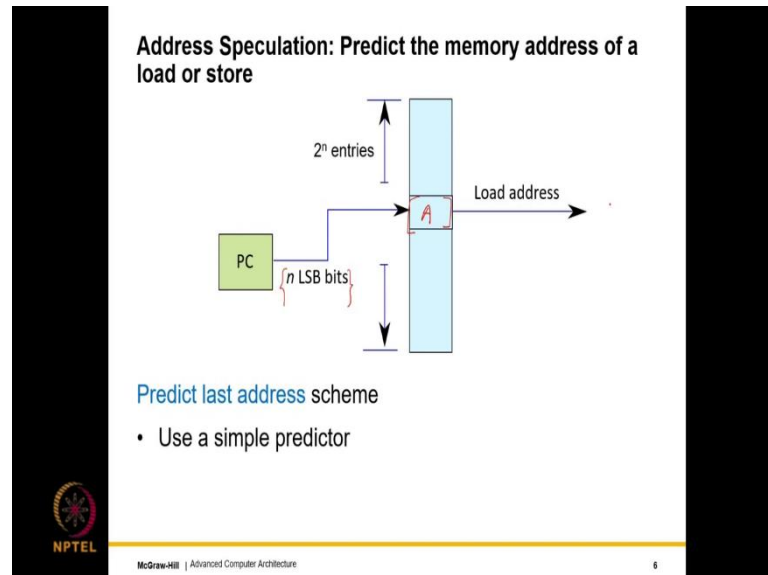
That allows us to do many things, we can send loads early, we can forward data between stores and loads by passing unresolved stores. Next is latency speculation, where we assume that a given access will hit in the either in the load store queue or in the data cache. A hit means that we will get the value. So, a memory access will take 1 cycle or 2 cycles; in the sense, we know the latency.

Then, we have a value prediction which of course is the hardest, but it is the only way of working around read after write dependencies. So, there is no way of actually solving read after write dependencies, this problem. The only way that we can actually still get a high I pc in code that has such dependencies, if we somehow make the producer and the consumer instructions run in parallel.

This is possible if let us say the value that the consumer instruction reads, that value is predicted and on the basis of that we proceed. So, this is known as a speculative execution and modern processors do embody many many kinds of speculation like this. But mind

you, with any kind of speculation, there is a chance of a mistake. In that case, something needs to be done.

(Refer Slide Time: 08:18)



So, let us look at our address speculation. So, we have seen this before. So, this diagram over here, we had seen something very similar, while we were predicting the outcome of branches. So, the predictor is generic. So, we had discussed this there also that is a generic structure of a predictor. So, let us say for a given memory address if you want to predict the address of the load or a store, we can use exactly the same structure where we use the n least significant bits.

We use it to access a table of 2^n entries, wherever there is a hit, if let us say there is a match over here, then there will be an address over here and this, we will use as the load address can be the store address as well. But since loads are on the critical path, we typically do the prediction for loads. So, this is a very simple scheme, where you predict the last address and unlike branch predictors, we cannot use saturated counters over here.

Because this problem is not amenable to saturated counters. So, it is not fundamentally a counting problem. We are storing an address; so, in this case, we store the last address. So, as we had discussed the branch predictor gives the generic structure of a predictor which of course can be used in many many different cases scenarios.

(Refer Slide Time: 09:48)

Stride based Address Pattern

C code

```
int sum = 0, arr[10];
for (i=0; i < 10; i++){
    sum += arr[i];
}
```

Assembly code

```
// Let us assume that the base address of arr is in r0
mov r1, 0 // i = 0
mov r2, 0 // sum = 0

loop: cmp r1, 10 // compare i with 10
      beq .exit // if (r1 == 10) exit
      ld r3, [r0] // load arr[i] to r3
      add r2, r2, r3 // sum += arr[i]
      add r0, r0, 4 // increment the memory address
      add r1, r1, 1 // increment the loop index
      b .loop
```

NPTEL

McGraw-Hill | Advanced Computer Architecture

7

So, let us now look at a different kind of predictor called as stride based address predictor. So, in this case, let us take look at the C code first. The C code is doing something that is rather simple; the C code is just taking an array of 10 elements and adding all the 10 elements and producing a sum.

So, what we have over here we is that we have $\text{sum} += \text{arr}(i)$; where i is varying from 0 to 9. So, we are adding the first 10 elements. So, that is what we are getting, the sum of the first. See if I were to do it in assembly, I can convert it to simple disk assembly, where I might I map i to $r1$, sum to $r2$ and the base address of the array r is stored in $r0$.

So, what I do is that I compare $r1$ with 10 every cycle. If the comparison fails, if there is no equality, if there is inequality, then I load the value of the base of the array to $r3$ and then I update the sum over here and then, what I do is I increment the memory address which is the base address of the array over here, I increment the loop index and then, I jump back over here. So, this line is what I want to draw your attention to first. In this case, we increment the base address by 4.

Why 4? Because we are assuming that the size of an integer is 4 bytes, so, that is why we are incrementing it by 4. Now, let us consider this load instruction which is the most important. So, it is so important that I am erasing the ink on the slide. Whenever I do that, it should be very important.

So, here what I am doing is that I am loading a value from the address store in register r0, this line should be read along with this line, where we are incrementing the value of r0 by 4. So, the address stored over here is getting incremented by 4 every iteration. So, we can think of this as a stride based pattern, where if I have an address predictor over here.

Then, the address predictor will have to explicitly take into account the fact that every time we access this instruction, the address is getting incremented by 4. And 4 in this case, it is stride.

(Refer Slide Time: 12:52)

Predicting the Stride

Last address (A)	Stride (S)	Pattern (P)
A		1 → 0

Salvaging Counter

- Last address (A): The memory address **computed** the last time the instruction with this PC was **executed**.
- A stride-based access **pattern** is followed if:
current address – last address = S
 $A' - A = S$
- Then we **set** the pattern bit, P
- Alternatively, if P is **set**, we **predict** the next address to be
 - $A + S$

NPTEL
McGraw-Hill | Advanced Computer Architecture

So, what do we do? Well, we use a different stride based predictor because in this case, the last address based predictor will not be useful. So, we need to use a different kind of predictor called a stride-based predictor. So, the last address, let it be A. Let the stride be S and the pattern be P. So, what we have is that we have a similar table as over here, which of course is addressed with the n LSB bits of the program counter.

Each entry looks like this, where we have the address, the stride and the pattern. So, in each entry, what we store is the memory address computed the last time, along with that, along with the memory address computed the last time, the this instruction with this PC was executed, we store to additional pieces of information.

So, we store what is the stride, which means historically what the memory address has been getting incremented with and whether we follow a stride based pattern or not. So, let

us see for the sake of prediction, if let us say the pattern bit is 1 which means that we can conclude that we follow a stride-based addressing pattern we can predict $A + S$ as the next address.

When we actually compute the address of this load instruction, so let us say the computed address is A' . So, we subtract $(A' - A)$ and we equate that to the stride. So, I should use a double equal to because it is clear it is an equality and not an assignment.

So, we pretty much equate that $(A' - A) == S$. See if a stride based addressing pattern is indeed being followed, this will be equal to the stride. So, we update the last address from A , we update it to A' , the stride of course remains the same and the pattern bit is set to 1, if it is not already 1.

But let us say that this stride based addressing is not being followed, then we have a choice. Say if this is a 1 bit from $1 \rightarrow 0$ or this can also be a saturating counter. So, it is important to understand that a saturating counter is a generic mechanism which is not just limited to branches right; clearly not. So, for example, if I were to consider the code that is shown here in slide number 7, again I want to draw your attention so I am erasing ink on the slide.

(Refer Slide Time: 15:57)

Stride based Address Pattern

C code

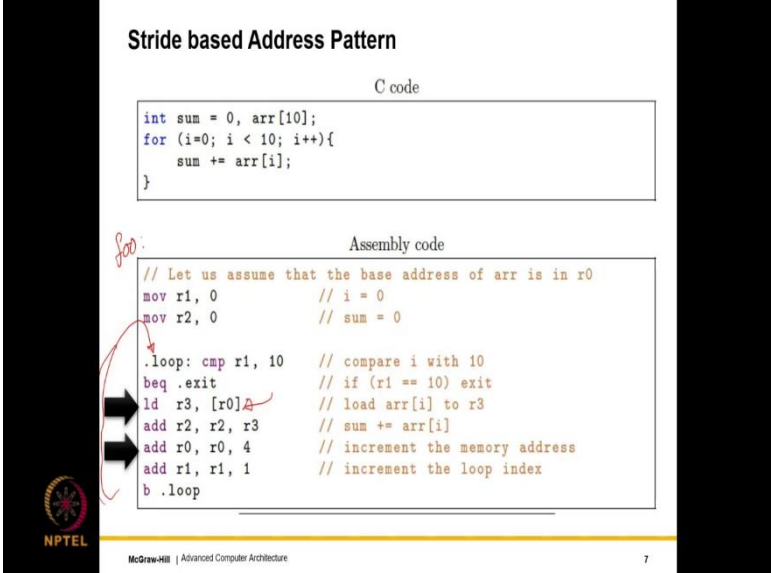
```
int sum = 0, arr[10];
for (i=0; i < 10; i++){
    sum += arr[i];
}
```

Assembly code

```
// Let us assume that the base address of arr is in r0
mov r1, 0      // i = 0
mov r2, 0      // sum = 0

.loop: cmp r1, 10 // compare i with 10
beq .exit      // if (r1 == 10) exit
ld r3, [r0]    // load arr[i] to r3
add r2, r2, r3 // sum += arr[i]
add r0, r0, 4   // increment the memory address
add r1, r1, 1   // increment the loop index
b .loop
```

foo:



So, let us say that this was part of some high level function foo. We will find a stride-based addressing pattern over here. So, this is something that we will find. But after we exit the

function and we come back again, so let us say we exit and a long time later again we come back, we will find that for the first axis the stride-based pattern will not hold.

But that does not mean that the behavior has changed. This is essentially a temporary aberration. If we use a saturating counter, then what we can do is we can ignore that and we can still continue to predict on the basis that it is indeed stride-based and the rest of the predictions will turn out to be correct.

So, that is why whenever we store something about the past history, we should think not once, but twice, but thrice that what exactly is the pattern that is being followed and how much of hysteresis do you want to give it.


(Refer Slide Time: 17:01)

Predicting the Stride

Last address (A)	Stride (S)	Pattern (P)

$A + S$ \rightarrow Sat. Cntr

- Last address (A): The memory address **computed** the last time the instruction with this PC was **executed**.
- A stride-based access **pattern** is followed if: $(A' - A) = S$
current address – last address = S
- Then we **set** the pattern bit, P
- Alternatively, if P is **set**, we **predict** the next address to be
 - $A + S$



NPTEL

McGraw-Hill | Advanced Computer Architecture

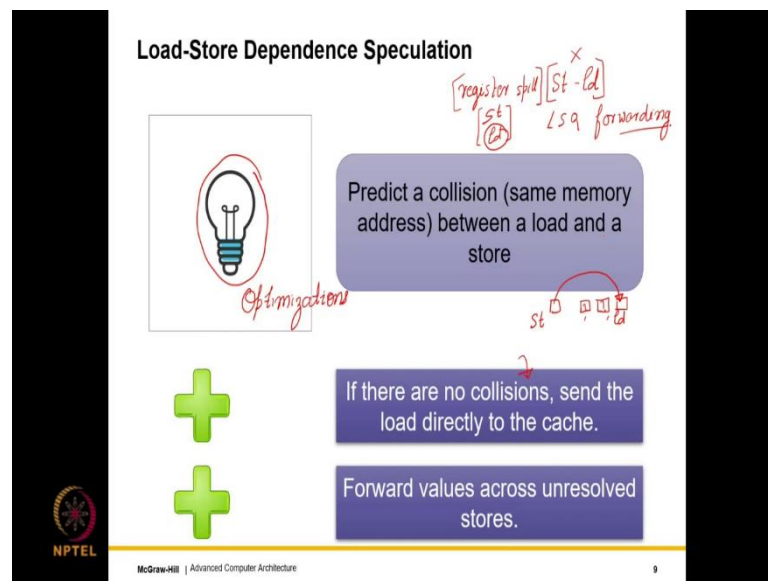
8

If we want to tolerate these occasional anomalies, this should be a saturating counter that is number 1. And let us see if the stride changes, then we should see whether we should just treat it as an occasional anomaly and not change the stride or let us say there is a persistent change in the stride, then of course, the value of the stride S should be changed over here.

Say much of this depends on actual simulation and actual engineering, but the key idea is that we store these three pieces of information and the prediction if a stride based pattern is followed is always $A + S$ and whenever we resolve the address, we always compute $(A' - A)$, where A' is the resolved address and compare it with the stride.

If it is equal, well that is great we can increase the confidence of P. If it is not equal, we can do several things. Much of that depends upon whether P is the saturating counter are not. An amount of hysteresis that we want to give it and so, there is no hard and fast rule on how much of hysteresis needs to be given. So, typically, what happens is that we run hundreds, thousands of simulations on the programs that people are expected to use and we look at the behavior that these programs are following.

(Refer Slide Time: 18:28)



So, we have discussed address speculation now. Let us now discuss the load store dependence speculation. So, what has been seen is that if a given memory address, let us see it has a load. If there is a collision; collision means same memory address with a previous store in the pipeline.

This tends to be in a sense a predictable steady state behavior and these collisions in a certain sense can be predicted, which means that if I have a load, I can predict with a reasonably high accuracy that look the value will actually come from a previous store that is there in the pipeline. In the sense, an LSQ forwarding will give the value and the value will not really come from the data cache. Why is this the case?

Well, we will have to look at the nature of the code. Say a lot of code is essentially this register spill code, which leads to this behavior that when we spill a register because we run out of registers and then, we execute a function and restore it. In that case, what

happens is that well we have a store and then, we have a load and as I said we can either spill because we run out or we spill it because of a function call regardless of the reason.

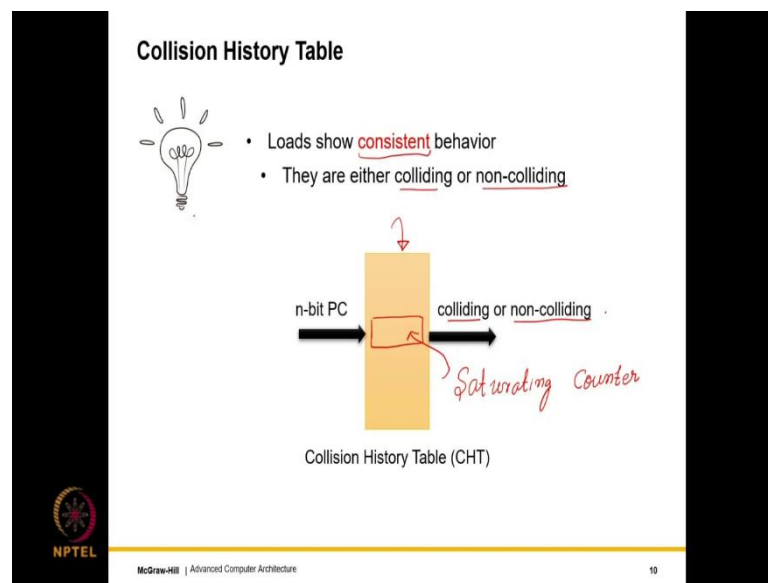
We always have the store load pairs and since, they are not that far apart, they will still be in the pipeline. So, moment, we see this load, we can say with high confidence that look there is a previous store that will supply value to it and there are many other examples of coding patterns, mainly with respect to function calls or with respect to loading data and so on, where such a pattern holds.

See if you can predict these collisions with good enough accuracy. So, let us say if we predict that there are no collisions, then we do not have to wait for unresolved stores. We just send the load directly to the cache, we get the value and we wake up the consumers or let us say if you predict that there is a very high probability of a collision between a store and a later load, even if there are unresolved stores in the middle, we need not wait for them.

Because if you have predicted that look this store and this load are going to collide and nobody else, then we directly forward the value; do not wait for these and the load proceeds. So, this of course, is highly dependent upon the way that we write programs; but given that most assembly programs are written roughly in a similar manner and this pattern holds across language, across compilers, across processors.

That is why processor design should be designed in such a way to leverage this pattern. So, there is an important philosophical insight over here. Almost all architecture and optimization, so in a sense, based on their learning based optimizations. So, they look at the way that programmers actually write their code and on the basis of that, they decide what kind of hardware will best take advantage of these patterns. If there were no patterns, no architectural optimization would actually work ok.

(Refer Slide Time: 21:58)



So, how do we find collisions? Nothing, we have a standard template that we have set up in our good old branch prediction days. So, given that we have learned that we just keep on using it. So, some loads do show a consistent behavior, they can be predicted as colliding or non-colliding. So, little bit of idea whenever you have these bulbs that is when we have an insight.

So, we have a collision history table CHT which takes an n-bit PC n-bit program counter and then, well, it points to an entry same as a branch predictor we predict whether its colliding or non-colliding. And here, you can have a single bit or you can have a saturating counter; no problem at all. So, we can have a saturating counter which can take care of those occasional anomalies. How many bits to have? Well, a lot of those are outcomes outcrops of a simulation related study.

(Refer Slide Time: 23:13)

Using the CHT

- When we **compute** the address of a load
- We **access** the CHT
- If it is predicted to be **colliding**
 - Wait for all prior stores to be resolved
- Else (*non-colliding*)
 - Send the load to the d-cache →
- Once the address is resolved
- Update** the CHT, recover the state (if necessary)

We can augment it with the store→load distance (**D**). A load waits till there are less than D entries before it in the LSQ.

Decoding Resolution

forwarded

Op formation

d-cache

decode

< D

Op formation

NPTEL

McGraw-Hill | Advanced Computer Architecture

11

Let us now discuss how to use the CHT. So, the CHT is used like this that we can use it at two points. We can do the prediction itself at two points; the first is at the time of decoding which of course I am not showing over here, it is not can be done, also it can be done at the time that we resolve the address of the load. So, it can be done at two points.

So, I am showing here that when we compute the address of a load, we access the CHT. So, this is of course at the time of resolution, but also it can be done at the time of decoding. So, in this case, it does not matter; but we will show other cases, where it actually does. So, if the load is predicted to be colliding, what it basically means is that in the load store queue, there is a store which has the same address with the load.

See it does not mean that there are no intervening stores in the middle that would not have the same address, but there is at least one. So, what do we need to do? Well, we need to wait for all the prior stores in the load store queue to be resolved to get resolved. Once they get resolved, what needs to be done is two things.

Either we get this forwarded value that is option 1, well then there is no problem at all or we send the value to the data cache. If it is predicted to be non-colliding which means that there is no forwarding that is going to happen most likely of course, then we send the load directly to the data cache and we are done.

Once the address is resolved, so basically the address of let us say the store in consideration is resolved. So, if you do not know which one, then pretty much once the addresses of all the stores before the load are resolved and let us say all of them its known the entire state of load store queue before the load is known the store queue actually not the LSQ, but the store queue before the load is known, we are in a position to see whether there is a collision or not right.

See, any of these addresses match, then there is a collision; else not. So, in this case, we update the CHT with this information and let us say that there was a collision, but we speculated fast it, there is a need to recover the state as simple as that. So, this essentially allows us to send some loads which are non-colliding to the data; to the data cache L1 cache without waiting for unresolved stores before it to get resolved. So, that wait time we avoid.

So, which means we can send loads early, we can make up their consumers. So, this gives us more I pc. It gives us more parallelism and more I pc. This can be augmented with more information which is where when we predict becomes important. So, let us say that we do this prediction at decode time.

So, what we can augment it? Augment it with is we can augment it with the store to load distance which basically means that if there is a forwarding, how many entries separate the store and the load. Howsoever, it is measured because we have separate load and store queues.

See, it can be the number of stores before it, number of loads before it or a sum, it does not matter. For a load, it will typically be the number of stores before it. So, which will basically essentially be this distance, where these are all stores. Say if this distance is D , what we do is that a load waits still there are less than D entries before it in the LSQ. So, of course, this needs to be interpreted contextually.

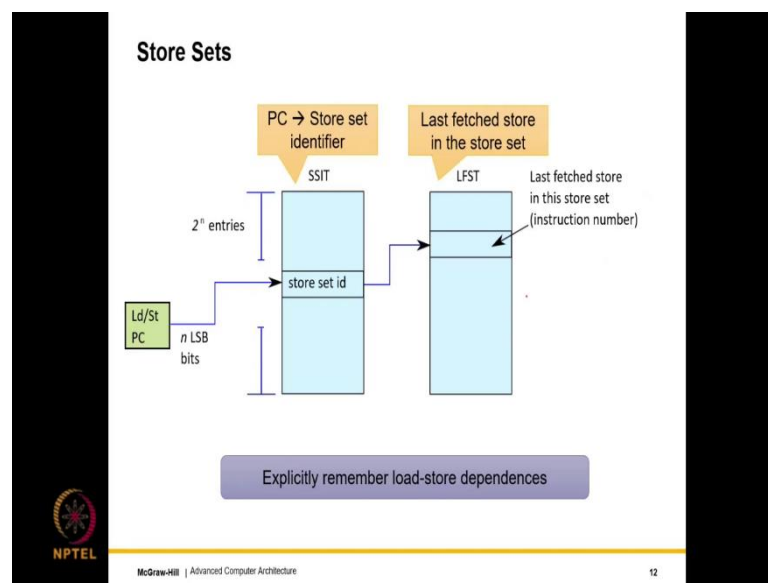
See in the context of this description, it is the number of entries before it in the store queue. So, the load basically just waits still the number of entries in the store queue are till they are $< D$ which means until they are greater than equal to D nothing; but the moment it goes below D and no forwarding has happened. You are kind of sure that no more forwarding will happen because this is what we have predicted.

So, load at this point can be dispatched to the data cache. It can be sent to the data cache because what we predicted is that we predicted a collision, but we predicted something more also. What we predicted is that the store to load distance can be measured in different ways, but let us measure it for the time being with the number of stores before the load.

So, this was predicted as D and the moment that the distance between a load and the earliest store in the store queue, this falls below D. You will find that this prediction pretty much does not hold which means that there are no more matching stores, there are no matching stores most likely of course. So, then the load can directly be sent to the data cache.

So, this is an optimization no doubt. It uses more information which is the distance D. Whenever, we use more information, we use more space. So, we expect a better prediction, a more accurate prediction, a more informative prediction. In this case, the prediction indeed gives us more information and this information can be used productively.

(Refer Slide Time: 29:04)



Now, let us come to another idea which is one of the most sophisticated ideas in this space. So, this is a far more refined proposal so to speak. So, as compared to the basic CHT, this mechanism is known as store sets. So, we start out in the same manner. So, always our predictors will keep on looking the same, a right pretty much or basic branch predictor design.

So, here our explicit aim is to remember both store dependencies and pretty much for a load, we should be able to say that which stores are kind of part of what is its stored set. For a store set is basically the set of all the stores that can forward their value to a load, pretty much like that store set is a set of stores.

So, here is what we do. So, I will describe this kind of part by part because it is a kind of an intricate scheme, but once understood, it will not appear to be that complicated. So, given a load or store, we start in a conventional manner, where we extract the end least significant bits.

We use it to access 2^n entries table. So, this table previously was giving us direct prediction information, in this case it will not give us. It will instead give us a pointer to one more table. So, what is the first table? The first table is the store set identifier table SSIT. What we read over here is for a load or store we read its store set id.

So, for every load in the simple design, we will have one store set that is associated with it and that will have a unique id. So, we will discuss slightly later how that unique id is given, but let us assume it has a unique id. So, then, we will figure out this by reading this table. Similarly, every store is part of one store set. So, this of course, in the initial proposal, it was one store set; later on, it was extended to multiple store sets.

But in this slide set, we are only discussing the simple idea which is a single store set. Say every store also if you read, then you will have a pointer to its store set id which is essentially the store sets that comprises that does not compile comprise; but that contains this store.

So, what we do is we use the store set id regardless of whether it is a load or store, we use it to access the LFST table which is one more table. So, if let us say this is a 7-bit id, then this table will have $(2)^7$ entries or 128 entries, each entry stores the last fetched store in the store set.

So, as I said a store set is basically a set of stores and it will tell us that look a certain store which is there in a pipeline is the latest in this store set which means it will most likely forward the value. So, how do we identify an instruction? Well, we provide each instruction with a unique instruction number which pretty much if you read the book, it pretty much is a number which is twice the size of the reorder buffer.

So, it can be a circular counter which can be I mean actually more than twice the size of the reorder buffer such that if there is one instruction before it or after it, you will never have the same instruction repeating; it is more than twice the size.

So, then if we have a cyclic counter like that, we can generate a unique instruction number for every instruction in the pipeline, such that at one point of time there should be no repetition. So, basically if let us say in the pipeline, all instructions can have a unique number and furthermore, if there is one instruction over here.

So, it will first start at the bottom of the ROV and gradually work its way up the ROV. Since, lifetime we should never see two instructions with the same number that are actually different. So, having an instruction number that varies between let us say 0 and any number that $> 2 \times \text{ROV size}$, this to a certain extent solves this.

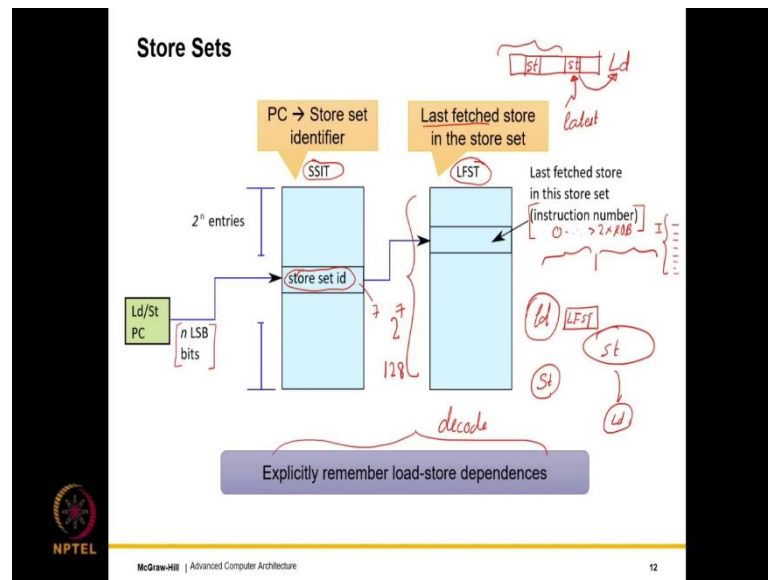
So, the long and short of this story is that every instruction is allotted a unique instruction number and in the LFST table, we say that look in this store set what is the instruction number of the latest store that has been fetched. So, this when so this check mind view is done at decode time not at resolution time.

So, the load basically knows that if I am the load, there are a lot of instructions that have been fetched before me. Out of these some of these stores are in the store set; what is a store set? The set of all the stores that can potentially forward their value to the load out of this store is the latest. It is a part of my store set and it is the latest which means this is the store that will most likely forward the value to me.

The rest are important in the sense they are in my store set, but I do not care. I only care about the idea of the latest store because that is what is going to forward a value to me and that to me is the one that I should be concerned about and this is the last fetch store in the store set and how do I identify it?

Well, as I said, we have a scheme of providing a unique instruction number to each instruction and this is recorded in LFST given the store set. So, load basically reaches the LFST via single level indirect and it finds the latest store id and it stores it. So, every load basically if of course, it has a store set associated with it has the idea of the latest fetched store LFST which it will use.

(Refer Slide Time: 35:43)



How will it use? Well, so, this is kind of a loaded text of the slide, but let us go slow. So, what we have said is that for every load, we have an associated store set. Stores that have forwarded values to it in the past are members of the store set and a store in this case is a part of a single store set. But as I said, there are extensions to this idea where this rule is broken, but we are not considering that.

So, what do we do for loads and stores? Well, for loads and stores of course we do different things. There is a first is that we read the store set id which is exactly what I am showing over here, this is step 1. The second is we get the instruction number of the latest store in the store set from the LFST, of course subject to the fact that it exists which is step 2.


The load waits for store S to get resolved. So, the load basically locates the store S in the store queue or we can say that in the store queue whenever the load is resolved, it probe the store queue to find if a store with that instruction number is there. So, let us say this is the load which got resolved.

So, once it does, it finds the actually this can be done before resolution also, the moment we enter a load, we if we know the LFST the latest store in the store set that this load is a part of, then the load can simply probe the store queue and find the entry in this that is the latest fetch store.

So, then we can do several things; one is that if this has not been resolved, we wait for this to be resolved and we can check the addresses. The other is we do not have to check for the address, the moment we know what is to be stored, we can do a direct forwarding, even before the address of the load has been computed. So, several things are possible.

So, but in all cases, we can ignore all the instructions between the store and the load because with very high confidence, we have predicted that there is a dependency between the store and load. So, let me explain this once again because this is a reasonably complex concept. So, should be explained in different ways till you were able to understand it rather thoroughly.

(Refer Slide Time: 38:32)



Basic Idea

- For every **load**, we have an associated store set
- Stores that have **forwarded** values to it in the past
- A **store** may be a part of a single store set

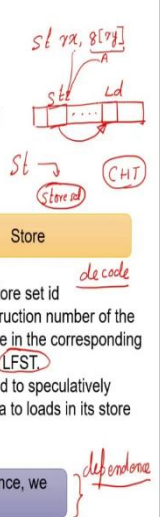
Load

1. **Read** the store set id
2. Get the instruction number of the **latest store (S)** from the LFST
3. The load **waits** for store S to get resolved and then receives the forwarded data.

Store

1. **Read** the store set id
2. **Set** the instruction number of the current store in the corresponding entry of the **LFST**.
3. Can be used to speculatively **forward** data to loads in its store set.

Whenever we detect a load-store dependence, we update the SSIT and LFST



McGraw-Hill | Advanced Computer Architecture
13

So, the idea if I were to explain it in a different way would be that let us consider the conceptual view of the load store queue. So, let us see that we add a load at this point at decode time and at that point, we read the SSIT and LFST and we are sure that there is a store before it in the LSQ, where we clearly have a store set based match.

So, what can be done in this case is that if we really want to aggressively speculate, when the value that will be stored which is roughly also the time when it will get resolved because let us look at a store. A store instruction will be let us say of this type that store rx which is some offset let us say it is 8 ry.

So, what we will do is we will read rx, we will read ry both from the register file roughly at the same time and then, we will add 8 to ry that will give us the address of the store and we will write both the values which is the resolved address as well as the value that needs to be stored at the same time.

Why at the same time? Well, because if we do it a different time that will necessarily increase the number of ports that we require in LSQ. So, in terms of implementation it will be hard. So, at the time of resolution, we will write both. See the store has been resolved, then well we can directly take the forwarded data, even before the load has been resolved and continue.

So, this will give us in a sense a certain speed up and we do not have to wait for the load to be resolved and this as far as we are concerned is a performance enhancing optimization. So, this is clearly a very powerful thing because what we are doing is we are ignoring all the stores and loads, where loads can be ignored; but all the stores in between and even if they are unresolved we are saying that we do not care.

For a store what we do is we read the store set id and then, when are we accessing the store? Again, let us say write after decode we are accessing the SSIT. So, this store is clearly the latest instruction in the store set. So, we update the LFST with the instruction number of the current store and subsequently, we can speculatively forward data to loads that are there in its store set.

Well, I would not say loads are there in the store set; but I would say that loads that are supplied value stores in that store set and so, this is how we can in a sense manage these fine grained dependencies and also, increase the ILP using this technique. When and where do we update the table? Well, the details are there in the book in chapter 5.

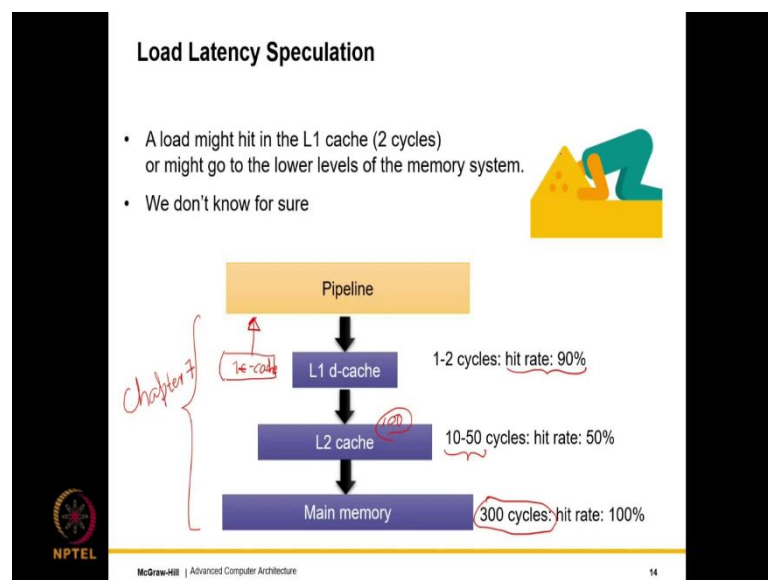
But clearly, when we detect a load store dependence, then if a store set has already been allotted, we simply add the store set, we simply add the load to the store set or we just record that for this given load, this is the store set. And let us say if the store is not a part of the store set, then we add the store to the store set.

We get rid of its previous mapping and we add it to the store set and let us say the store set itself is not there, let me create a new store set and we update the SSIT, SSIT such that both this store and this load point to the same store set. So, this bookkeeping is always

done when we detect the dependence, we need to do this bookkeeping with the SSIT and LFST; but that said and done, this is clearly a very powerful optimization its clearly more effective than a CHT.

Because CHT just predicts a collision. In some cases, it can predict the collision and also have distance information. But again, that is not that accurate because the distance can vary; code can vary, behavior of the code can vary. But this is a far more accurate method of predicting which stores will collide with which loads, then we can explicitly wait for those stores and forward values and so on.

(Refer Slide Time: 43:02)




Now, let us come to the next which is load latency speculation. So, we do have a memory hierarchy much of this will be discussed in chapter 7. Hence, I really do not want to broach this topic right now. But all that I can say is that look, we have a hierarchy of memories for performance reasons of course, so there is an instruction cache which I am not showing this instruction cache provides instructions to the pipeline.

The data cache is fast we have assumed a 1 to 2 cycles, it takes to access. It has a very high hit rate of 90% for all the accesses that go below. So, this 50% is essentially a local hit rate; in the sense that if 100 accesses come request come to the L 2 cache, only 50% of those are satisfied. This is a far slower structure, it takes 10 to 50 cycles and of course, if we do not find it in the L 2 cache, we go to the main memory, where we will find everything.

Again, we will break this assumption, but let us assume it holds for the time being and this has a very long access time 300 to 400 cycles very long. So, we do not know for sure, where we will find the data for a given load right. So, we do not know. So, that is the reason, I buried my head in the sand.

(Refer Slide Time: 44:36)

Make a guess




"I bet you can't guess what it is."

- For load instructions, **predict** if it will hit in the data cache or not. If it will, do an early broadcast.
- Design a **hit-miss** predictor. Same idea as branch predictors.

Load PC (saturating counter)

Aggressive Speculation



NPTEL

McGraw-Hill | Advanced Computer Architecture

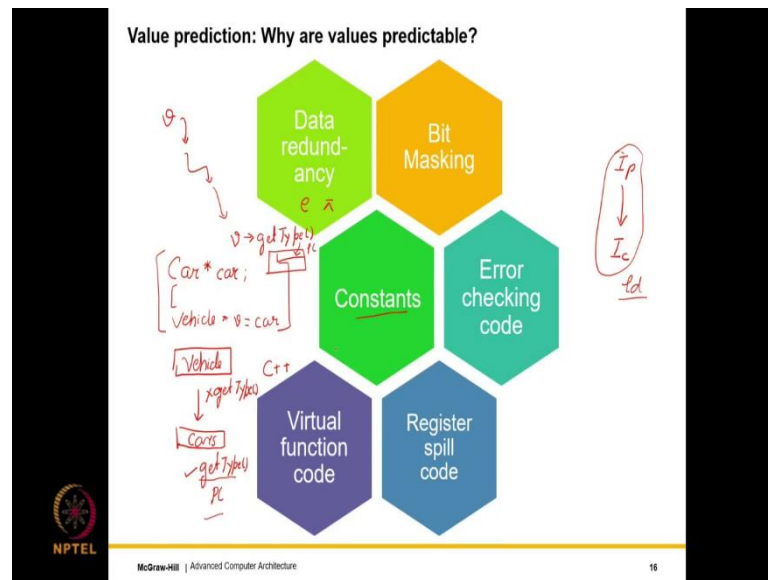
15

But you can always make a guess. So, we love making guesses that is why we call this sub chapter aggressive speculation, where we guess as much as we can. In the hope of getting that last drop of ILP, it is like searching for water in the desert, we guess speculate as much as possible such that whatever we can extract in terms of slightly higher instruction level parallelism, we do that. So, for load instructions, we predict if it will hit in the data cache or not. If it will, we do an early broadcast.

And so, let us say that if let us say a small predictor says it will hit in the data cache, well no problem. We broadcast, we wake up the consumers. If the prediction is wrong, we need to do something. We will see that in the next section. We can design a hit miss predictor for a load given the load PC, we can see whether it will hit or miss; the idea is very similar to a branch predictor.

So, we will not discuss the same thing over and over again. Here, also we can use simple bits or we can use saturating counters. So, everything depends upon the nature of the workloads. So, a lot of architectural simulation has to be done, but the basic pattern is the same.

(Refer Slide Time: 45:58)



So, you know this by now. What else can we predict? Well, it turns out we can predict values as well. Say value prediction as we have discussed is the only way to break a read after write dependency between a producer and a consumer, the only way is if we can somehow predict the values that the consumer will use and clearly one of the slowest instructions in this category, one of the biggest culprits in this category is a load instruction.

So, the load instructions value is something that we can predict. So, why are values predictable? Well, many times, we use this data redundancy. So, use the same values over and over again like let us say it is a scientific program, we use the values of e and π over and over again. Bit masking many a times even if we are considering long values, we are only interested in a few bits. So, those bits in a sense are predictable.

Many times, we are using constants in our code. Many times, we have checking code, where we almost never have errors; but a large part of the code is the same, a lot of values that it uses the same. Virtual functions; well, virtual function is an important virtual functions are important things in C++.

So, what they basically say. So, my aim is clearly not to introduce an object oriented language here, but clearly some introduction might be due that look let us say I have a vehicle class. Say it defines everything about a vehicle and let us see in this you have get type.

Say vehicle will have a type and then, we have a subclass of vehicle let us say cars. So, cars will again have a definition for the get type function. With the get type for a car will be a car, but for the vehicle it might be undefined. So, we can always have this. So, this is the key of any object oriented program.

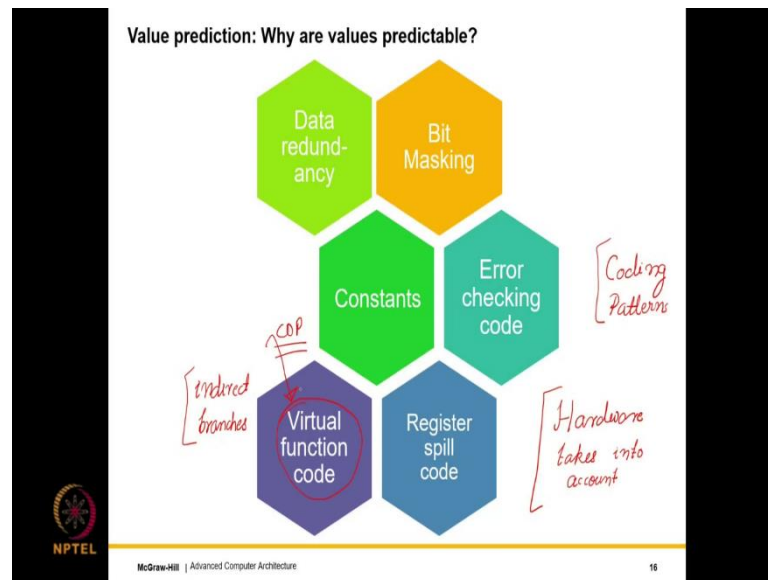
We can always have a pointer to a car like let us say in a car star. So, as is the convention, car starts with the proper car star with a capital letter and a variable with a small letter. So, let us have a car and let us say that we have initialize the properties of the car. So, then, we can have a pointer to a vehicle and set this equal to car.

Now, what do we do is that let us say we sent this pointer v across a set of functions; does not matter. So, let us say we send this pointer far away and then, we call the get type function. So, mind you, we are calling get type on a pointer to vehicle. But what will magically happen is that the car will also would have defined the get type function, what will magically happen is that instead of this function being called, actually this function of the car will be called because this vehicle is actually a car.

Even though, in the entire code, we are using vehicle star; but this function will get called and this is a virtual function and the way it actually does happen is that we have a small table with each object, which essentially says that look for get type it just has a pointer to the PC and in this case, it is this PC.

So, all of this code for virtual functions is predictable; similarly, register spill in which we have already seen. So, if we take a look at all of this, I would suggest the viewers to just take a look at this, these are all coding patterns.

(Refer Slide Time: 50:06)



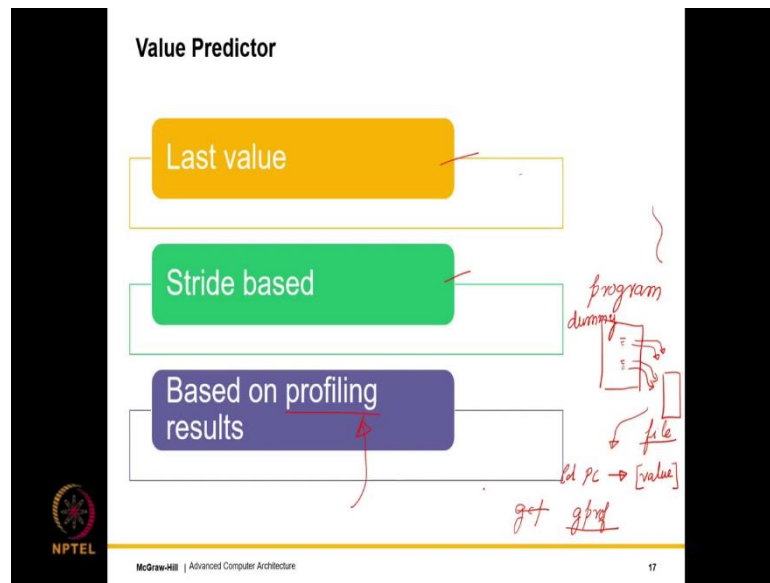
So, for example, before object oriented code came along, virtual functions were not there; but once that came along and once let us say there was a big need for register spill in, hardware always has to take into account software patterns and virtual functions are clearly a big example of that once programmers started writing programs in a certain manner, hardware had to take that into account and then, hardware had to be redesigned to in a sense ensure that C++ programs run quickly.

So, that this the market for C++ programs which was not there and let us say the mid-80s, by the late 90s had become huge. In fact, even now, most of the proper software is written in C++ and other dot net languages; but C++ is still big very big.

So, this is where hardware designers have to step up and ensure that their processors could actually run these programs quickly and then, they realize that virtual functions are a very important object oriented programming concept and this needs to be supported. One of the key ways of supporting virtual functions is of course having support for function pointers, that is one in hardware which means that we need support for indirect branches, where the target is not hardwired, but it is stored in the register.

So, this is undoubtedly required, but other than that, they also realize that many of the values that are used here are predictable. So, you need a value predictor.

(Refer Slide Time: 51:48)



So, value predictors pretty much follow the same pattern that we have been discussing up till now, which is either predict the last value or let us say the value increases with the constant stride predicted on the basis of a stride or you do it on the basis of profiling. What is profiling? Well profiling is like this that we take the program, we give it a set of representative inputs.

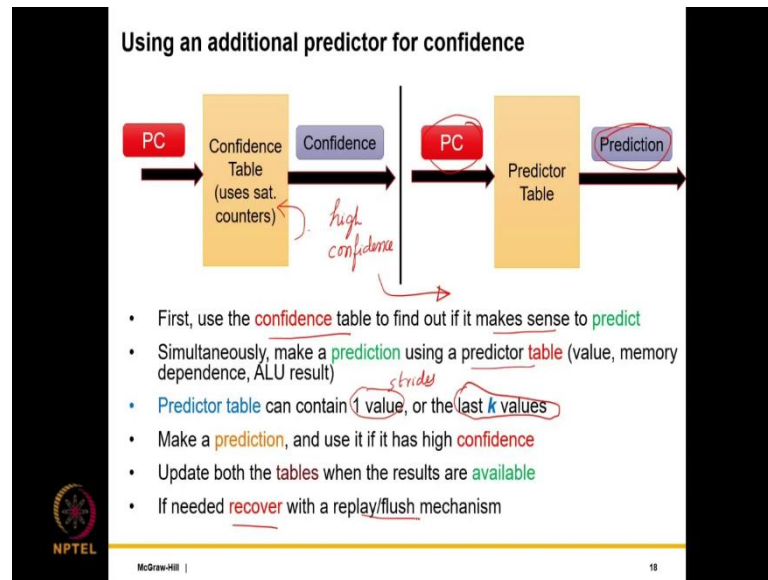
So, give it a set of dummy inputs or representative inputs or whatsoever you want to call it and you have a lot of instructions that are embedded in your profile program; but they will not be a part of your final program, which will essentially write the results of values and write which direction the program is taking, write down the outcomes of branches a host of information to a file a regular file.

So, this file can then be analyzed and then, important pieces of this information can then be used to actually kind of pre-program the hardware such that let us say for example, you can say that look for this PC, it most often takes this value. So, let us say if it is a load PC, it most often reads this value from the memory system.

So, all of this information will come via profiling and all of this information in a certain sense can be fed to the hardware via special instructions. If the hardware has a value predictor and instructions have been exposed to software to program it, see in that sense, you will see that your accuracy will improve substantially. So, I would advise all the viewers to take a look at the program.

What does g prof do? I will not tell you now, but you take a look at it, you will get to know what it does. So, it is something on these lines along profiling, but what it does you explore on your own.

(Refer Slide Time: 54:03)



This is the last slide in this section, where we discussed the use of confidence bits. So, as you have seen we have predicted a lot of things, but of course, that prediction should we predict all the time because many times unlike branches, where we are forced to predict; in many cases, we need not predict. So, we can have a separate confidence tables which will predict the confidence in a sense that how strong will the prediction be and this can use saturated counters.

So, we can see that certain things can be predicted with high confidence. If historically, they have been very predictable; otherwise, we say that they are low confidence and then, of course, we will have a predictor table as we have seen we enter the PC, we get the prediction.

So, we first use the confidence table to find out if it makes sense to predict, simultaneously, we also predict. The predicted table can use you know a single value, can use strides, we can have complicated predicted tables that use the last k values and try to predict on the basis of that.

So, these things can be arbitrarily complicated, but the question is that whether we should use the predictive value or not, we can use the separate confidence predictor to actually tell us and this is an important augmentation to our predictive prediction mechanisms because we do not want to unnecessarily predict low confidence stuff and just lose cycles, that would be really bad.

Once the results are available, we update both the confidence as well as the predictor table and of course, if the results are available, we also know whether we made a mistake or not. So, we recover with a replay flush mechanism.

Next we will discuss replay mechanisms, where we show how we recover from these misprediction, misspeculator, misspeculation related faults.