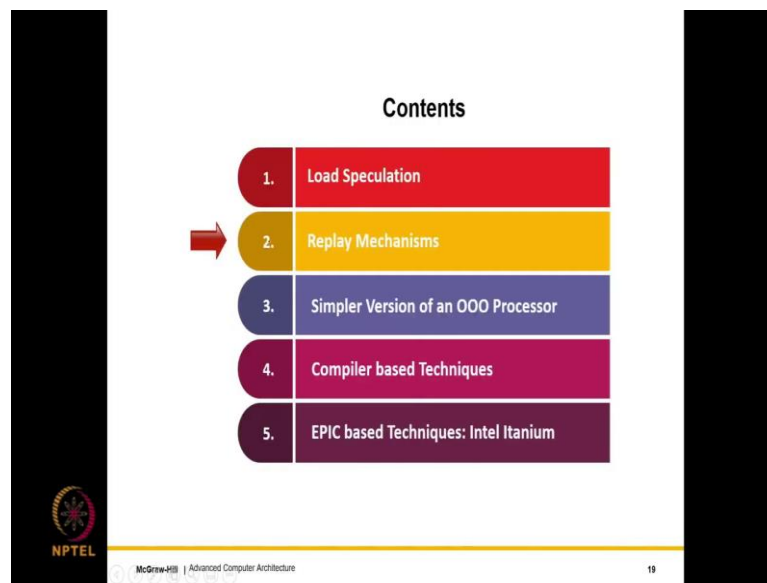


Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Module - 05
Lecture - 13
Alternative Approaches to Issue and Commit Part-II

(Refer Slide Time: 00:36)



Contents	
1.	Load Speculation
2.	Replay Mechanisms
3.	Simpler Version of an OOO Processor
4.	Compiler based Techniques
5.	EPIC based Techniques: Intel Itanium

NPTEL

McGraw-Hill | Advanced Computer Architecture

19

In this lecture we will discuss replay mechanisms. Replay mechanisms are extremely important when we have aggressive speculation. Recall that aggressive speculation was predicting load values, addresses, load store dependencies and all other forms of data and addresses which are typically not predicted.

What I mean is typically not predicted in a simple pipeline, but it is necessary to predict them in an aggressive out of order pipeline. So, sometimes in this prediction we will make a mistake and that is where replay mechanisms are required.

(Refer Slide Time: 01:18)

Replay

flush → restore and clean up
→ wasted work [extra energy]

- Flushing the **pipeline** for every misspeculation is not a wise thing
- Instead, **flush** a **part of the pipeline** (or only those instructions that have gotten a wrong **value**) (*re-execute a subset*)
- **Replay** those **instructions** once again (after let's say the **load** completes its execution)
- When the instructions are being replayed, they are guaranteed to use the **correct** value of the load
- Identify and replay the **forward slice** →

NPTEL

McGraw-Hill | Advanced Computer Architecture

20

Well, one option we have for doing a replay which is to flush the entire pipeline. However, flushing the pipeline for every single miss speculation is not particularly a very wise idea; the reason being that it is flushing the pipeline is very expensive for two reasons.

So, what are the overheads of doing a pipeline flush? So, the first reason is that we need to clean up the state and we need to restore the state from where we are flushing. So, essentially the restore and cleanup actions take time they are not instantaneous. So, they take several cycles. But this again is not that much as compared to the sheer amount of wasted work because what would happen is that we would have executed a lot of instructions and we will simply waste we simply stand to waste some work.

So, something that we would have gained had the speculation been correct the entire work will be wasted it needs to be redone and a corollary of this of course, is extra power consumption extra power or energy consumption. Well, extra power is not clear because we are flushing and restarting, but definitely it is extra energy consumption which is an issue. So, what we can do is instead of flushing the entire pipeline we can flush a part of the pipeline.

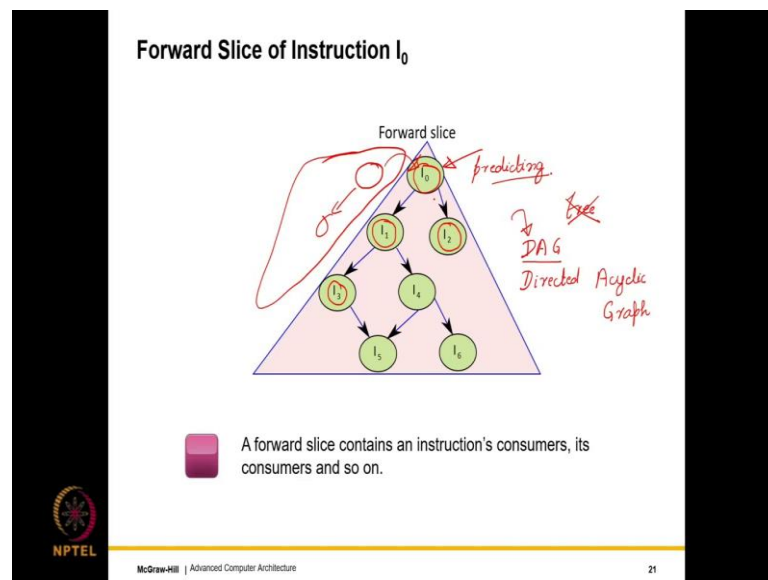
So, recall that this is a novel idea we have not discussed this before, for flushing the entire pipeline we did discuss the reorder buffer, but for flushing a part of the pipeline this has not been discussed before. So, what we do is that we somehow track all those

instructions that have possibly used the wrong value and we re-execute only them. So, we will revisit this notion several times, but essentially we execute a subset of instructions in the pipeline we re-execute a subset that is the crux of the discussion.

So, we replay those instructions once again after let us say the load completes its execution. When the instructions are being replayed in this case we are guaranteed to use the correct speculated value. So, let us say we are predicting the value of the load. So, when instructions are being replayed they are guaranteed to use the correct speculated value not the correct they are guaranty to use the correct value of the load because we detect a misspeculation when we realize that whatever we were guessing was not correct.

So, at this stage what we do is that of course, we initiate a replay we will see how we also wait for the correct value and once when that comes we restart. So, the next time we will not have a need to speculate. So, what we want to do to kind of save wasted work and also not to avoid a full scale pipeline flush is we need to locate what is called the forward slice of an instruction and just replay that. So, let us define a forward slice.

(Refer Slide Time: 04:51)



A forward slice is defined as follows; we have let us say one instruction I_0 it produces a value the consumers of those values are I_1 and I_2 . So, let us say I_2 whatever I_2 produces nobody consumes it, but let us say I_3 and I_4 produce something I_4 consumes both and so on and so forth.


So, we can create a graph of this type. So, this will be what is called a dag or directed acyclic graph and so, this is the structure if you do not know what a directed acyclic graph is this is the right time to actually look it up in a book on data structures because this concept is used rather heavily in computer architecture. So, you cannot escape this concept. So, it is far better to look it up right now.

So, this structure that we have here note is not a tree. So, this is not a tree. So, let me do this it is a directed acyclic graph and furthermore this is rooted at the instruction whose value we are predicting, it is rooted over here. So, this entire set of instructions is known as the forward slice of instruction I 0, it contains the instructions consumers the its consumers and so on.

So, the entire forward slice including I 0 and all its consumers and so, on the forward slice has to be replayed, but not the rest of the instructions. For example, it is possible that another instruction was issued along with I 0. But it is independent from I 0 and even some of its instructions are in are independent not in the forward slice.

So, this set of instructions can execute and it should also be allowed that these instructions execute, but of course, the commit happens in program order still, but at least these instructions are not re-executed only the forward slice is the candidate for replay or re-execution.

(Refer Slide Time: 07:13)



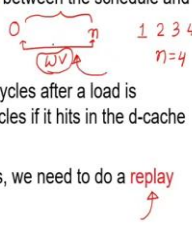
NPTEL


Non-Selective Replay

Trivial Solution: Flush the pipeline between the dispatch and execute stages

Smarter Solution

- It is not necessary to flush all the instructions between the schedule and execute stages
- Try to reduce the set of instructions
- Define a window of vulnerability (WV) for n cycles after a load is selected. A load should complete within n cycles if it hits in the d-cache and does not wait in the LSQ
- However, if the load takes more than n cycles, we need to do a replay





NPTEL

McGraw-Hill | Advanced Computer Architecture

22

So, the trivial solution could be that we flush the entire pipeline or somehow we flush the pipeline between the dispatch and execute stages if it is doable, but this again is wrong and impractical and hard to implement. So, we should go for a smarter solution which is non selective replay. So, we will discuss three replay schemes non selective replay, a delayed replay and a token based replay.

So, this is the first such scheme. So, this scheme is simple, but as we shall see there are some issues, but the issues can be fixed these are not major issues. So, these are all fixable issues. So, the first is that it is not really necessary to flush or remove all the instructions.

So, I would not call it schedule let me call it dispatch. So, between the dispatch and execute stages all the instructions that are there which could potentially be consumers, consumers, consumers and so on we do not have to remove all of them.

See even if they are in the window they can be there we do not have to remove all that is the first insight. So, we can be slightly prudent and judicious about it. What we can do is we can reduce the set of instructions and only focus on a set of instructions that have possibly gotten the wrong value right because of the miss speculation.

So, what we do is, we define what is called a window of vulnerability WV that is its abbreviation which is the window is n cycles after a load is selected. This window of n cycles is known as the window of vulnerability. So, let us say that the load is selected in cycle number 0. So, what we do is that we consider a window of n cycles after it till the n th cycle.

So, this entire period is known as the. So, how do we determine n ? Well n is determined based on the predicted latency. So, for example, let us say we are predicting that the load will hit in the data cache.

So, then what we do is that if we are selecting in the 0th cycle in the first cycle we will read the register file, in the second cycle we will compute the address and after that in the third and fourth cycle assuming a two cycle latency for accessing the data cache at the end of the fourth cycle we will get to know if the value is there or not in the data cache.

So, if we expect that a given load will hit in the data cache which means it will be found, then we can set the window of vulnerability = 4 or we can set $n = 4$ over here and within this period we expect that the value will be available. If it is not available then of course, a replay has to be initiated.

So, this means that the load will take more than n cycles and this is where a replay definitely has to be initiated. So, the reason that the window of vulnerability is defined within is defined in this manner is basically because any instruction that is waking up let me change that to say that any operand that is waking up during this period in the window of vulnerability might have been woken up speculatively.

Because it might get selected it might go down the pipeline and it might not find its value. So, we should particularly be interested in those instructions whose operands have woken up in this period.

(Refer Slide Time: 11:42)

Example

1: ld r1, [r2] (Predict the value)

2: add r4, r1, r3

3: add r5, r6, r7

4: add r8, r9, r10 (squash them)

- Let us say that instructions 2, 3, and 4 had one operand waking up in the WV of instruction 1
- If there is a misspeculation, all three instructions get **squashed**
- Instruction 1 gets **reissued** with the correct value later

NPTEL

McGraw-Hill | Advanced Computer Architecture

23

Let us consider the example of the speculation. So, let us assume that we have a load instruction. The load instruction loads from this address, the addresses stored in register r 2. Furthermore, assume that we predict the value the predicted value is stored in register r 1 which is the destination register of the load.

Now, let us consider these three instructions, instructions 2, 3 and 4 which have had at least one operand that wakes up in the window of vulnerability of instruction 1. So, one

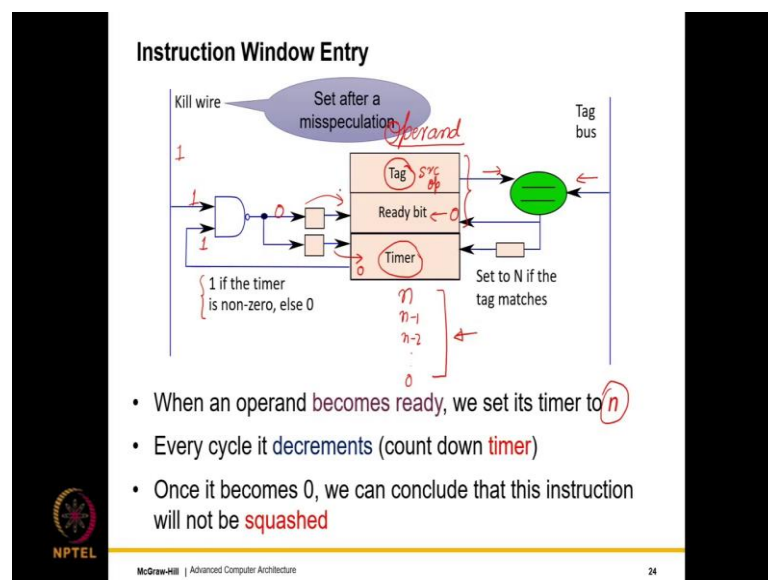
thing that is obvious is that register r1 will clearly wake up because it is in the WV of instruction 1, but let us assume that the other two instructions also wake up because of other broadcasts.

So, we will discuss this issue later, but for the time being let us assume that windows that instructions 2, 3 and 4 are within the window of vulnerability of instruction 1. So, if there is a miss speculation what will happen is that, we will squash what does squashing mean? T we will essentially set them from ready to not ready. So, we will in essence kill their readiness that is called squashing.

So, we will squash these three instructions and then we will wait for instruction 1 to get the correct value which is the correct value of r1 we will reissue it and then we need to reissue these three instructions again once again to the pipeline.

So, it is possible there are two cases one is that the operand woke up, but the instruction was not selected. So, instruction is still in the instruction window that is one case. The other case is it was selected and it has been sent down the pipeline as the other case regardless of the case in both cases we squash them and we re-issue.

(Refer Slide Time: 13:57)



So, how exactly is this done? Well, the way that this is done is that the instruction window entry for every operand is augmented with some additional circuitry. What is that additional circuitry? Well the additional circuitry is that we store the tag over here.

So, the tag is the tag that we are expecting from the tag bus or we can say that this is let us say a source operand can be source operand 1 or 2 does not matter. So, the tag goes here from the tag bus. If there is an equality, we set the ready bit. So, this part we have already seen because it was a part of the instruction window.

So, there is nothing new over here, but what is new is that in addition to this two the tag and ready bit we have a timer. So, the timer is basically a simple piece of logic and state where once the operand wakes up not the instruction once the operand wakes up we set the value of the timer to n . So, this is a capital N small n here. So, kindly ignore that.

So, regardless of whether its lowercase or uppercase we set the value of the timer to n and every cycle it decrements n , $(n - 1)$, $(n - 2)$ and so, on until it reaches 0. So, every cycle we decrement the timer once the value of the timer becomes equal to 0 we can conclude that the window of vulnerability is over. It is not a part of the window of vulnerability and we can conclude that this instruction will not be squashed.

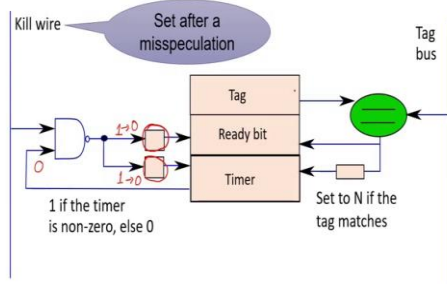
So, what happens? So, what happens is that we augment every instruction window entry particularly the entry for the source operand with a timer. Once the operand wakes up because of a broadcast, we set the timer to n and subsequently in every cycle we decrement the timer and if let us see it receives what is called a kill. So, I will discuss what is a kill in one second when the timer is non zero then we squash the instruction which means we set the ready bit back to 0.

So, how do we actually squash an instruction? So, what we do is that we assert the kill wire. So, the kill wire is meant to go and kill all those instructions whose timer is not equal to 0. So, for the timer we have a little bit of additional circuitry over here. So, it is a logic block I am not showing that this is one if the timer is non zero 1 0 so see in the timer is non zero. So, this is one and if the kill wire is one this is one this is a NAND gate.

So, the output of the NAND gate is 0. So, pretty much pretty much the ready bit gets set to 0 and also the timer gets set to 0 in the sense the timer also gets reset. So, that is the important thing that the both the timer and the ready bit both get reset if we are asserting the kill wire. So, the kill wire essentially says that look go through all the operands in the instruction window, for any other operands in the timer is non-zero then set its ready bit to 0 which means that squash it.

So, what is the key idea of the non selective replay scheme? The key idea of this scheme is that we assert the kill wire when we detect a MIS speculation and the kill wire then goes and does this via this logic which is a NAND gate. So, let us now look at all of those entries for which the timer is 0. So, I have not discussed that case. So, in that case what will happen is that if the timer is 0 regardless of the kill wire this will be 1 because a NAND gate if any one of the inputs is 0 the output is 1.

Instruction Window Entry



The diagram illustrates the logic for an instruction window entry. It features a vertical line on the left representing the 'Kill wire' and a vertical line on the right representing the 'Tag bus'. A callout bubble points to the Kill wire with the text 'Set after a misspeculation'. The logic consists of an AND gate with two inputs: one from the Kill wire and another from a signal labeled '0'. The output of the AND gate is connected to the 'Ready bit' of a three-part register (Tag, Ready bit, Timer). The 'Timer' part of the register has a feedback loop from its output to its input, labeled '1 if the timer is non-zero, else 0'. The 'Tag' part of the register is connected to a green circle, which is also connected to the 'Tag bus'. A signal labeled 'Set to N if the tag matches' is connected to the 'Timer' part of the register.

- When an operand becomes ready, we set its timer to n
- Every cycle it decrements (count down timer)
- Once it becomes 0, we can conclude that this instruction will not be squashed

NPTEL

McGraw-Hill | Advanced Computer Architecture

24


So, recall that for both of these logic blocks at the same inputs if its 1 we do not do anything only if it becomes 0 which is if both the kill wire is a circuit and the timer is non-zero in that case what we do is that this is when this circuit gets activated. So, it sets the ready bit to 0 and it resets the timer.

(Refer Slide Time: 19:45)


More about Non-Selective Replay



- We attach the expected latencyⁿ with each instruction packet as it flows down the pipeline

Wherever there is an additional delay (such as a cache miss)

- Time for a replay
- Set the kill wire 
- Each instruction window entry that has a non-zero timer, resets its ready flag (squash)

We now have a set of instructions that will be replayed

Methods of replaying instructions 

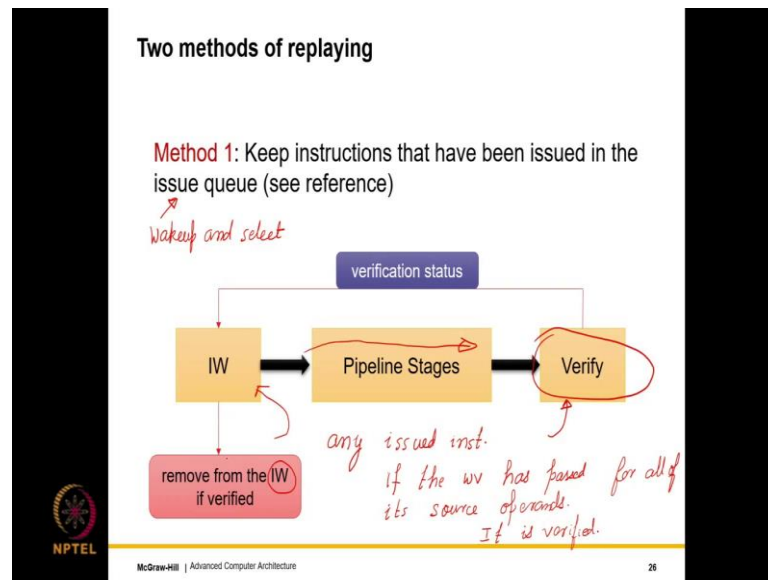


25

So, how do we do this? Well with each instruction if there is speculation we attach the expected latency which is essentially the value of n . Let us say there is an additional delay in sense this threshold of n gets breached maybe because we are doing latency speculation and we predict that it will hit in the d cache, but it actually misses in the data cache in that case it is a time for a replay.

So, what do we do? Well as you can see the monster over here we assert the kill wire. So, each instruction window entry will have a non-zero timer that has a non-zero timer will reset this ready flag which basically means a squash. So, this is happening at the operand level not at the instruction level. So, we now have a set of instructions that need to be replayed how do we replay? Well we will discuss this in the next two slides where we discuss methods of replay.

(Refer Slide Time: 20:47)



So, the simple method of replaying which we call method 1 is keep instructions that are already been issued in the sense that they have left the instruction window because they were woken up and selected we still keep them in the instruction window. So, we do not remove them. So, in a simple version of the pipeline when they got selected. So, what is issue again? Well issue let us define us the entire process of wakeup and select. So, after which it goes to the execution units.

See if it has not been issued which means it is still lying around in instruction window then there is no problem, but if it has been issued we still keep it in the instruction window we let the instruction flow through the pipeline stages and then we verify. How we will verify? Well we will discuss it is different for different schemes, but at least for this specific scheme what we will do is that, we will we can definitely verify for the load instruction if its speculation has been correct or not.

For the instructions in the forward slides that have been issued there is no quick fire way of verifying right now, but that does not matter. For any issued instruction once the timers for all of its operands becomes 0 it can be removed from the instruction window. So, this we can treat as verification.

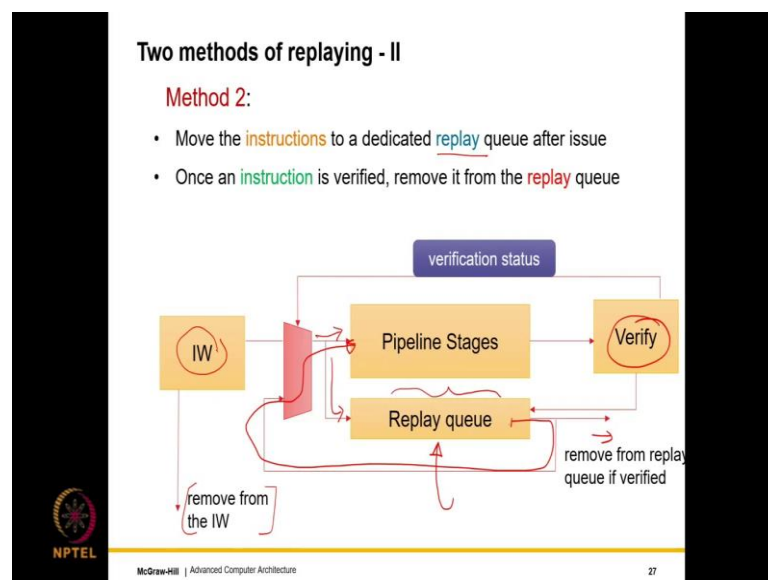
So, what we can do is that for this particular scheme. So, of course, for other schemes it will differ, but for this particular scheme any issued instruction if the window of vulnerability is passed which means the timers are 0 for all of its source operands, it is

verified. So, we can have a small separate circuit to check that or periodically we can check. So, it depends on the implementation.

So, as I said we can either have a periodic passes every 10 cycles we look at all of the instructions that have been issued and they are not within the window of vulnerability none of their operands are within the window of vulnerability they can be removed. So, whatever logic we choose this is the verification logic that an instruction has correctly executed for this particular scheme.

So as I say it will vary depending on the scheme, but regardless of this we have a generic verification conceptual block over here and once an instruction is verified to be correct it can be removed from the instruction window. So, this is one approach.

(Refer Slide Time: 24:10)



The other approach that we are calling method 2 is that, we move the instructions to a dedicated replay queue after issue. So, after we issue we remove the instruction and we move the instruction to a dedicated replay queue. So, as the instruction moves to the pipeline stages after the instruction window we also send it to a dedicated replay queue and the verification logic is the same.

So, assume that it gets verified which means that there is no miss speculation it's not in the foreword slides of any miss speculated instruction, then what we do is that we remove from the replay queue if verified. So, we just remove it throw it out and let us see

if there is a replay then instructions from the replay queue will enter the pipeline via this mass.

So, the replay queue does two things it is a temporary storage area for instructions that have been issued, but not yet verified to be fully correct. Fully correct basically means that they are not in the forward slides of any misspeculated instruction. So, this is a temporary staging area once they are verified they are removed and if we need to do a replay, then from the replay queue we can replay these instructions like this that is why a multiplexer has been provided.

So, this replay question we did not answer for the previous design which is method 1. So, I will start by first erasing the ink on the slide and then telling you that if we have to do a replay, then what will happen is that since the instructions that need to be replayed are already there in the instruction window, they will simply proceed to the rest of the pipeline stages instead of the instructions that are not being replayed.

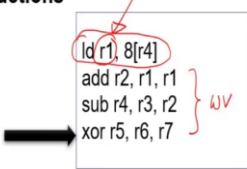
So, we will give higher priority to the replay instructions the instructions that are being replayed and they will move through the pipeline stages and this time the verification will be successful because they are guaranteed to be on the forward slice of at least that misspeculated load. So, in this case the temporary staging area is within the instruction window and they do not leave the instruction window unless it is guaranteed that they will never be a part of a miss speculation and the replay proceeds from here.

Whereas in this case the replay proceeds from a dedicated replay queue. So, in most designs this is kind of preferred because it is simpler it is more scalable it's easy to create it in hardware and also there is no need to increase the complexity of the instruction window given that it is so, complex as it is. So, we made it even more complex with this mechanism.

So, typically using the replay queue was more common, but again I do not want to take sides for the purposes of simplicity the instruction window can also be used.

(Refer Slide Time: 27:28)

Orphan Instructions



- Assume that the *load* instruction misses in the L1 cache
- The *add*, *sub*, and *xor* instructions will need to be **squashed**, and **replayed**
- For the *add* and *sub* instructions, tag will be broadcast

What about the *xor* instruction?
Say that r6's ready bit was forcefully set to 0

NPTEL

McGraw Hill | Advanced Computer Architecture

28

Now, we need to discuss one important corner case with the non selective replay scheme. So, let us look at this scheme in somewhat more detail, let us now look at a very very important corner case.

So, this corner case is present in a lot of replace schemes that consider a superset of the forward slice and clearly one such scheme is this scheme the nonselective replay scheme. So, in this case, this is the first instruction that we are speculating and we are speculating let us say the value of the load which is the value that will be loaded into register r1.

Let us further assume that these three instructions are in the window of vulnerability. So, these three instructions will get squashed. So, let us say you assume that whatever speculation you are doing either the latency or the value it does not matter, but the speculation is that within a certain time period the value will be there with us. If this is not happening let us say for a cache miss, the add sub and xor instructions will need to be squashed and subsequently replayed.

So, one thing that is rather clear is that the add in sub instructions the tag will be broadcast and they will be replayed in the natural course of actions. Why?

(Refer Slide Time: 29:08)

Orphan Instructions

```
ld r1, 8[r4]
add r2, r1, r1
sub r4, r3, r2
xor r5, r6, r7
```

Assume that the *load* instruction misses in the L1 cache

- The *add*, *sub*, and *xor* instructions will need to be **squashed**, and **replayed**
- For the *add* and *sub* instructions, tag will be broadcast

What about the *xor* instruction?

Say that r6's ready bit was forcefully set to 0

NPTEL

McGraw Hill | Advanced Computer Architecture

28

Well we can clearly see a dependence. So, the, so this is the WV instructions and r 1 is basically the value whose tag will be broadcast is the register whose tag will be broadcast. So, the second instruction over here the add instruction both of its source operands are r 1, we can clearly see a read after write dependence over here. Now, if we take a look at the next instruction.

(Refer Slide Time: 29:48)

Orphan Instructions

```
1 ld r1, 8[r4]
2 add r2, r1, r1
3 sub r4, r3, r2
xor r5, r6, r7
```

Assume that the *load* instruction misses in the L1 cache

- The *add*, *sub*, and *xor* instructions will need to be **squashed**, and **replayed**
- For the *add* and *sub* instructions, tag will be broadcast

What about the *xor* instruction?

Say that r6's ready bit was forcefully set to 0

NPTEL

McGraw Hill | Advanced Computer Architecture

28

Then that would be reading r 2 again there is a read after write dependence. So, we can clearly see that instruction 1 will wake up instruction 2 which will wake up instruction 3

which here is the fun part that is xor instruction xor r 5, r 6, r 7 the xor instruction is independent of the first three instructions.

So, there is clearly no read after write dependence as a result in the entire forward slice of the load the xor instruction is not there. So, it is one of those instructions that is a part of the WV, but not a part of the forward slice of instruction 1. So, in a certain sense if for some reason this get squashed why will it get squashed? Well, it is possible that the operand the source operand corresponding to r7 that was woken up in the window of vulnerability.

Now the ready bit of this will be set to 0 and as we can see there is nobody to set the ready bit of r7 to 1 which essentially means that there is nobody to wake up the xor instruction and so, of course, I did not see the example here. So, instead of r7 we consider r6 see it says that let us assume. So, let me go by what is written.

(Refer Slide Time: 31:29)

Orphan Instructions

```
ld r1, 8[r4]
add r2, r1, r1
sub r4, r3, r2
xor r5, r6, r7
```

add r6

I(WV) = forward slice

orphan

- Assume that the **load** instruction misses in the L1 cache
- The **add**, **sub**, and **xor** instructions will need to be **squashed**, and **replayed**
- For the **add** and **sub** instructions, tag will be broadcast

What about the **xor** instruction?

Say that r6's ready bit was forcefully set to 0

NPTEL

McGraw Hill | Advanced Computer Architecture

28

So, let us assume that r6 is ready bit is set to 0, then as we can see no other instruction before it is actually writing to r6. So, it is very well possible that one of the early instructions that let us say was writing to r6 that is committed a left the pipeline after that there is no instruction to broadcast the tag associated with this register.

Hence, this instruction will become an orphan there is nobody to wake it up and unless we do something it will remain in the pipeline forever. So, this needs to be understood

that this instruction over here is an orphan. We require another instruction to wake it up that other instruction there is a possibility that it might not be there if that is happening then this instruction will stay in our pipeline forever.

So, we have to deal with orphan instructions and why do we have orphan instructions? Well we have them because our window of vulnerability. Let us say all the instructions that were squashed in the window of vulnerability are essentially a superset of the forward slice. So, it can be a superset or it can be equal right of the forward size slice and this is causing the problem.

(Refer Slide Time: 32:59)

Orphan Instructions - II

Impractical Method

- Keep track of **squashed** instructions.
- **Re-broadcast** tags of orphan instructions.
- → We need to dynamically **detect** which instructions are orphans.

Better Approach

- Let the orphan instruction **reach** the head of the ROB
- **Execute** and **commit** it.

The slide includes a diagram of an instruction window (IW) and a Reorder Buffer (ROB). The 'Impractical Method' section lists the need to track squashed instructions and re-broadcast their tags, noting the difficulty of dynamically detecting orphans. The 'Better Approach' section suggests letting orphan instructions reach the head of the ROB and then executing and committing them, accompanied by a happy emoji. The NPTEL logo is in the bottom left, and the footer reads 'McGraw Hill | Advanced Computer Architecture' and '29'.

So, there are two ways of solving this. The first is an impractical method, but it nevertheless should be discussed because it is the classic example of a bad idea. See in this case what we can do is, we can devise a mechanism to keep track of the squashed instructions.

One way of keeping track of the squashed instructions can be that we have some auxiliary structure that is some kind of an appendage with the instruction window. So, what we do is that whenever we squash instructions within the instruction window we transfer them we essentially copy their id's to an auxiliary structure.

And then basically we detect which ones are a part of the forward slides and which ones are not and then we can identify the orphan instructions and then rebroadcast their tags to

wake them up or not kill them in the first place. So, even if let us say they have been their operands have been set from ready to non-ready from 1 to 0, we can convert it to 0 to 1. If they have already been issued and remove them from the instruction window at a later point in time.

So, even though having this auxiliary structure that can be used to temporarily store some of the instructions that have been squashed that appears to be a feasible idea, but this is very hard to implement. So, this can be attempted as long as somebody is willing to do it and implement it correctly, but a far better approach which is simpler I am only stressing about simplicity here is that we let the orphan instruction reach the head of the rob.


So, we do not do anything then the orphan instruction will reach the head of the ROB at that point we will find that this instruction is an orphan and it is essentially waiting for a register operand which clearly means that it has been squashed. So, the instruction the head of the ROB cannot wait for a register operand and the only thing that it can wait for is that, it can wait for some value to come back from memory. But it clearly cannot wait for let us say the result of an add, subtract or multiply instruction at that point we can execute and commit it.

So, this is if you think about it this is something that does give us some simplicity of course, we do sacrifice a little bit of LP because of it because we could have executed this we could have woken up its consumers and done a lot we are not doing it. So, there is clearly a trade off and something in the middle is also possible where maybe we lead orphan instruction reached a certain point in ROB and then we initiate a check.

So, it is doable, but as I said that creating orphan instructions as it is either as they are that is a problem. So, we should try to create as few of these as possible. So, let us now look at another method that tries to reduce the number of orphan instructions and is more efficient.

(Refer Slide Time: 36:43)

Delayed Selective Replay

- Let us now propose an idea to **replay** only those instructions that are in the **forward slice** of the **misspeculated load**
- Let us extend the non-selective **replay** scheme
- At the time of asserting the **kill signal**, plant a **poison bit** in the **destination register** of the load. *64 + 1 poison bit RF*
- Propagate the bit along the **bypass** paths and through the **register file**. *consumer insts. (n) dummy 1* *bypass path*
- If an **instruction** reads any operand whose **poison bit** is set, then the instruction's **poison bit** and its **destination register** are also set. 
- When an instruction finishes **execution** → *ad*

NPTEL

McGraw Hill | Advanced Computer Architecture

38

So, this method is known as delayed selective replay. So, here the idea is to replay only those instructions that are in the forward slice of the misspeculated load, they are the only ones that are executed once again and the rest are pretty much allowed to go through we will see how? So, here also orphan instructions are created, but we will have a new method a different method of actually dealing with them.

So, let us extend the basics nonselective replace schemes. So, we will still have the nonselective replace came with all of its elements and the elements will be the kill signal will be there, the window of vulnerability will be there everything will be there.

But what we will do is that we will have something extra and this extra is what gives this scheme its edge its advantage over the previous scheme say in this case whenever we detect a miss speculation what we do is along with asserting the kill signal we plant a poison bit in the destination register of the load.

So, the destination register instead of containing 64 bits in this case contains $(64 + 1)$ bit and this bit is known as the poison bit and this poison bit is set to 1. So, wherever the value is stored this includes the register file wherever the value is transmitted this includes the bypass paths we increase the width of the data path from 64 to 65 bits and we also say that look this is a value which has been poisoned.

So, what we do is that we propagate this bit along the bypass path and the register file. So, then in. So, in this case, what happens is let us say that we predicted that a load will hit in the data cache. So, let us assume we did that and let us assume that after n cycles we find out that the load value will not come from the data cache it will come from somewhere else.

So, what we do is that, we basically create a dummy value which is the dummy value has no meaning. So, this value is created and along with that is set the poison bit to 1, we write this dummy value to the register file and we also broadcast this dummy value on the bypass paths. What is the need for doing it? Well the need for doing it is that we would have speculatively woken up instructions.

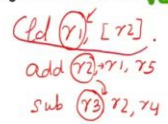
So, we would have speculatively woken up consumer instructions and those consumer instructions maybe would have gotten selected maybe would have proceeded down the pipelines. So, they will anyway read the register file see if they read the register file instead of reading an erroneous value, it is far better that they read a value for the poison bit is explicitly set.

So, then what happens is that if an instruction reads any operand either from the bypass path or from the register file this poison bit is set, then the instructions poison which is set and then it kind of propagates it. So, its destination registers poison bit is set. So, basically what happens is that the poison bit propagates via the instruction the instructions destination register, it is passed to the consumer consumers and so on.

(Refer Slide Time: 40:58)

Delayed Selective Replay

- Let us now propose an idea to **replay** only those instructions that are in the **forward slice** of the **misspeculated load**
- Let us extend the non-selective **replay** scheme
- At the time of asserting the kill signal, plant a **poison** bit in the **destination** register of the load
- Propagate the bit along the **bypass** paths and through the **register file**
- If an **instruction** reads any operand whose poison bit is set, then the instruction's **poison** bit and its destination register are also set.
- When an instruction finishes **execution** →



NPTEL

McGraw Hill | Advanced Computer Architecture

38

So, let me give an example let us say that we have a load instruction and this load has a miss speculation. So, we set the poison bit is this, then if we have an add instruction of this form, then we set the poison bit of r2 because it is reading r1. Then if you have a subtract instruction of this form, we set the poison bit of r3 because it is reading r2 so, on and so forth. So, pretty much along the forwards lies all the values get poisoned.

So, which basically says that all of these values are in a sense invalid, they are potentially wrong and we say that all these values are poisoned. So, this is one way of nicely keeping track of the forward slice it has its issues, but at least you can see that a bit is being propagated along the forward slice of this instruction.

(Refer Slide Time: 41:59)

Delayed Selective Replay - II

When an instruction finishes **execution** →

- Check if its **poison** bit is set.
- If yes, **squash it**
- If no, **remove** the instruction from the IW (it is verified to be correct) *or the replay*

Issues with this scheme

- It is **effective**, but assumes that we know the value: n
- This might not be **possible** all the time
- Instructions in the WV that have not been issued might become orphans

[I(WV) - If operand - slice]

Issued *Not issued*

NPTEL

McGraw Hill | Advanced Computer Architecture

31

So, what do we need to do in terms of changes to the pipeline? So, when an instruction finishes execution, we check if its poison bit set which means after the E x stage. So, after the E x stage recall that we actually set a bit in the ROB indicating the fact that the instruction has completed its execution at this point we check if its poison bit is set if yes we squash it.

So, squashing it would basically mean that we set the ready bits in the instruction window to 0 and we also stop ourselves from setting the ready bit in the ROB the instruction completed bit in the ROB we do not set that. So, we basically squash it which essentially means that this instruction is not completed.

If we go back to the replay queue where we talked about the verify logic, we can add one more verify stage over here where it will verify this poison bit and if it is set well then the instruction remains in the replay queue it does not go anywhere. Otherwise, we just remove it we remove the instruction from the IW or the replay queue depending upon either which scheme which replay scheme we are using.

So, what are the issues with the scheme? Well this scheme is effective in the sense that as long as we know the value of n which we may not know all the time, but as long as we know the value which is true most of the time the scheme is undoubtedly effective. So, now, it is important to understand where do the advantages lie. Well the advantages lie in

the fact that if we consider the two kinds of instructions that are in the WV, but not in the forward slice then we can see that at least one of the kinds of benefits.

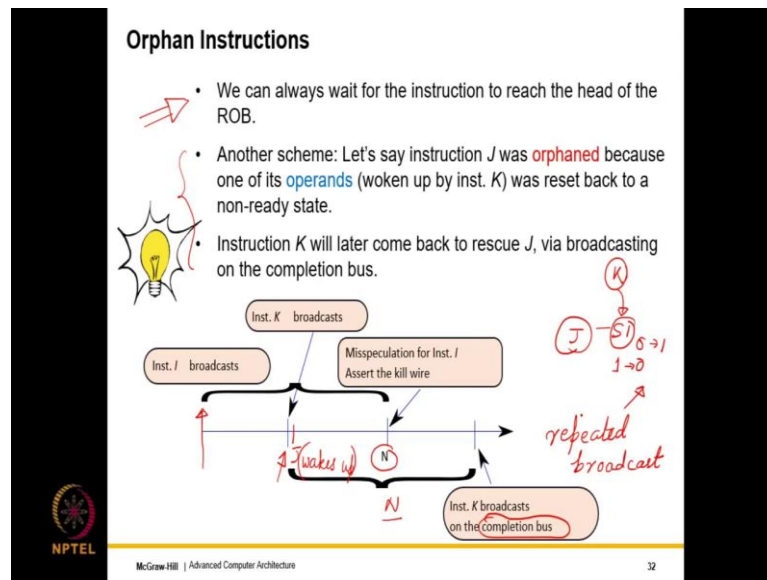
So, let me. So, let me kind of explain this in different manner. So, let me consider all the instructions which were there in the window of vulnerability, but were not there in the forward slice. So, let me consider the difference. So, these instructions ideally should not have been squashed, but since we do not explicitly have a way have a mechanism of maintaining the forward slice where we still need this.

Now, what we do is that we can divide these into two types one type let us say would have issued one type would have not issued. See if I consider all the instructions that would have issued then we would have cleared them. So, why would you why would we have cleared them? Well, we would have cleared them because we would have seen clearly that their poison bits were set equal to 0. So, the instructions were clean they would have gone through.

So, this would have reduced the number of orphan instructions right here because this set would have been treated correctly. What about the other set of instructions which are there in the window of vulnerability, but let us say for some reason they are not issued for a long period of time. So, even after the window of vulnerability these were not issued. It is very much possible that because of the kill wire one of their operands got invalidated.

So, we need to worry about this set not about the other set. See if I were to consider this set of the instructions which are there in the WV the window of vulnerability, but those who have not issued they can become orphans and if they become orphans we will have a similar problem.

(Refer Slide Time: 46:27)



So, what we can do is we can always wait for the instruction to reach the head of the reorder buffer and at that point of time we can say that look this instruction is indeed an orphan. So, we can execute it. But what was done in this paper which originally suggested this scheme is that, they looked at a different idea they introduced a different mechanism which as you will see is effective.

So, all such mechanisms do add their hardware overhead, but they are indeed effective. So, let us say instruction J was orphaned because of one of its operands which was woken up by instruction K was reset back to a non ready state. So, J is the one that is getting orphaned it is getting orphaned because one of its source operands the ready bit of that was set from 1 to 0 and originally instruction K which was clearly before J in the instruction sequence had set the ready bit of the operand S 1 to 1.

So, let us look at this on the timeline ok. So, what will happen is that instruction I will broadcast its tag and it will say that look I have been selected it will do early broadcast of its tag. Then what will happen is that in this case n cycles later we will detect that there is a miss speculation for instruction I.

So, we will assert the kill wire. In the middle what will happen is that instruction K will broadcast and one of Js operands will this source operand is ready bit will get set.

So, then I am assuming it gets broadcasted and then subsequently J is operand wakes up and after J is operand wakes up, J is kind of ready to get selected subject with other operands, but let us assume that it does not get selected it remains within the window.

After the N cycle after the window of vulnerability is over because one of Js operands is vulnerable it gets set from 1 to 0 well no problem. So, what you will see is we can add one more bus where instruction K will come back later to rescue instruction J. So, instruction K will again wait for N cycles.

So, instruction K of course, has broadcast and gone away, but after N cycles a timer will elapse. Instruction K will broadcast the tag once again on the completion bus. So, this is one way for instruction K to actually rescue all of those instructions like J which it had woken up.

So, what does instruction K do? It actually broadcasts twice. It broadcasts once at this point of time which is its regular broadcast and then it broadcasts once again which is n cycles later in this case it is a new broadcast it is a fresh broadcast it is not something that was there in an earlier scheme.

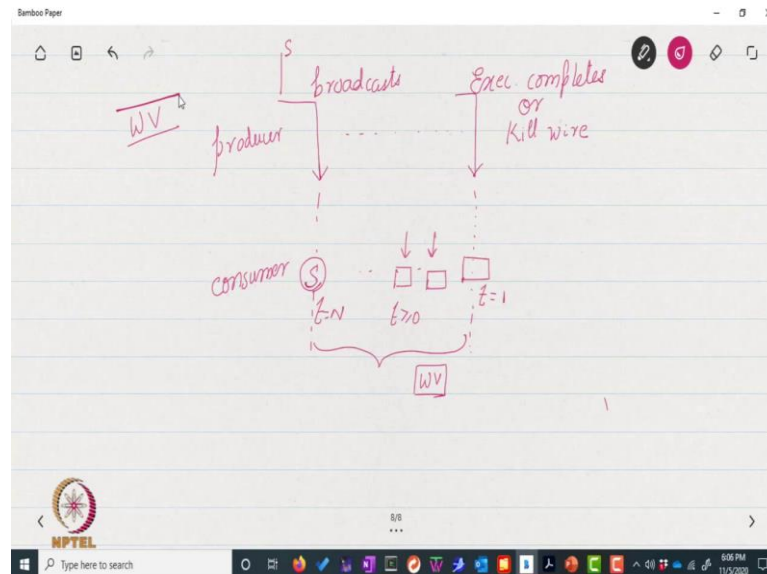
See we are assuming that along with the tag bus we add another bus called the completion bus in which the second broadcast is made and in that again the ready bit of this operand S 1 is set from 0 to 1 and instruction J can proceed subject to the ready bits of other operands.

So, this is one example of a repeated broadcast where you kind of come back and rescue those instructions which for some reason have kind of fallen in the shadows of some window of vulnerability and who has kind of from a ready have gone to a non ready state.

So, this is one way of taking care of operands as you can see in this method the important operative point over here is that in this method we because of the poison bit we were able to kind of split the potentially orphaned instructions which is this set into issued and not issued. Issued instructions no problem their poison bit will not be set not issued instructions we can either have the default scheme which is wait for them to reach the head of the ROB or otherwise what we do is we have this repeated broadcast scheme to rescue them later.

Now, let us have a slightly deeper discussion about the scheme and discuss a few of the subtle points which would kind of tell you about some of the intricacies of having such schemes and implementing a replay scheme in practice.

(Refer Slide Time: 51:49)



Let us now discuss a few subtle problems that will happen while constructing the window of vulnerability. So, what was discussed in the lecture on non selective replay was that, the window of vulnerability starts when we select the instruction that is not wrong turns out we can do better.

So, consider this cycle this instant of time when the producer broadcasts. So, this pretty much tells the rest of the world particularly the consumers about the existence of the producer about the fact that it is broadcasting its tag.

So, it tells the consumers that look this is when you can get yourself selected and you can enter the pipeline you will get the value either from the register file or the bypass path. After that other producers executes other stages like reading from the register file execution and so, on and then at one point of time the execution completes. Now from the point of view of a consumer it is essentially this point of time that is important.

So, we assume that in this cycle either the execution completes or we are aware that the execution will not complete. So, we assert the kill wire. So, either one of these two events happen. Now from the point of view of the consumer when it receives the

broadcast let us say that it gets selected in the same cycle. So, in this cycle if it is getting selected, then it needs to set its $t = N$ let us say and in this cycle when it is it may receive the kill signal it sets its $t = 1$.

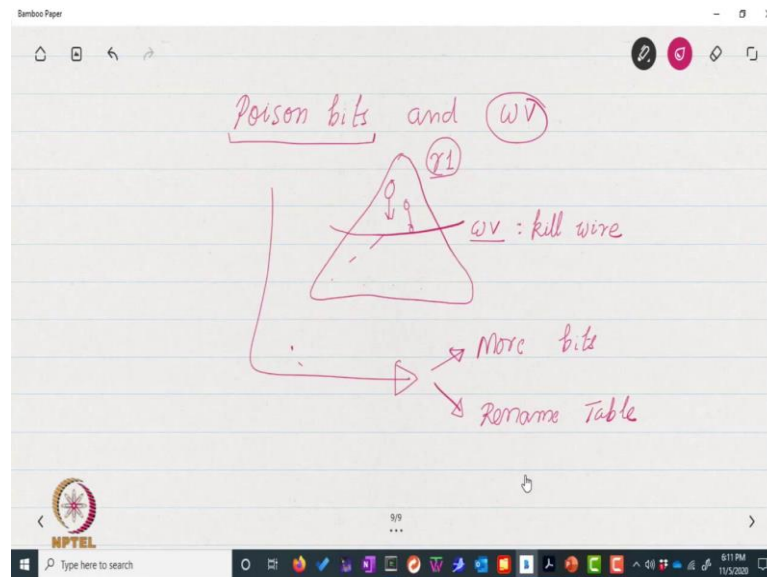
and of course, for in a realistic pipeline sometimes the latency might be variable. So, this might be the latest possible time at which we will get the value of the kill signal it could be the latest possible time at which the value of that kill signal will be gotten.

So, then at if let us say we get the signal kill signal at any earlier point of time then of course, the value over here will be nevertheless ≥ 0 because the value of the timer at the latest possible $t = 1$. So, for us this should be the window of vulnerability which pretty much starts with the producer actually broadcasting.

So, this is a slightly more accurate way of doing it as compared to when it can be at other points as well it can be when the producer is selected for example. So, even that would not be wrong see if I were to consider the select point which is an earlier point and define the window like this, see it is possible that n will be higher it will be greater than what it actually should be which means the WV we will slightly extend to the right.

So, this will just produce more squashes more orphan instructions, it will not be wrong. So, the correctness criteria over here is that we should not allow an instruction to complete if it should be squashed in the sense it should not be getting a wrong value. So, this is one aspect one subtle aspect of the timing.

(Refer Slide Time: 56:11)



Another question that many times students ask is that, why have poison bits in the delayed selective the replay scheme why have poison bits and the WV together at the same time? What is the advantage of doing that? Well, if I just have poison bits just think about it the forward slice can just become huge in the sense the forward slice will never terminate.

So, let us see if I have one load instruction its value will be poisoned then so on and so forth. it will just keep going on and on and on after a certain point of course, the correct value of the register r 1 will come and after the correct value of register r1 comes again we will start re executing them, but this forward slice will actually become very large and it will just keep on growing.

So, the size of the forward slice has to be limited. So, one way of limiting the size of the forward slice is to essentially draw a line which is the line at the end of the window of vulnerability which is drawn by setting the kill wire. So, this is drawn by setting the kill wire. So, in the kill wire what happens is that it abruptly terminates the forward slice. So, it does not allow the forward slice to increase because all the instructions that are getting that are in the forward slice with operands that wake up during the window of vulnerability.

If they have not been issued they pretty much get killed. So, this terminates this limits the size of the forward slice it cannot grow any further and whatever few orphan

instructions are there they can be taken care of. If you just have poison bits and if you do not do something extra which we will see in a token based replay scheme the size of the forward slice will become huge. So, the WV in essence limits that and hence it is required.

So, for us the direction is to have something that exclusively uses poison bits, that we will show that one bit is not enough we need more bits. So, there are two things. So, we will show that look one bit is not enough because it is not distinguishing between the forward slices of different loads because we might be speculating different loads at the same point of time.

And the other is that we need to make changes at the level of the rename table making changes at the level of the instruction window is not good enough. So, with this we will proceed to the next scheme that uses token based replay.

(Refer Slide Time: 59:20)

Token Based Selective Replay

Let us use a **pattern** found in most programs:

- Most of the **misses** in the **data cache** are accounted for by a relatively small **number** of instructions
- 90/10 thumb rule → 90% of the misses are accounted for by 10% of instructions
- Predictor → Given a PC, **predict** if it will lead to a d-cache **miss**
- Use a predictor similar to a **branch predictor** at the fetch stage

Handwritten notes: An arrow points from the word "predict" in the third bullet point to the word "decode" in the handwritten text "decode stage". A bracket is drawn under the words "branch predictor" in the fourth bullet point.

NPTEL
McGraw Hill | Advanced Computer Architecture 33

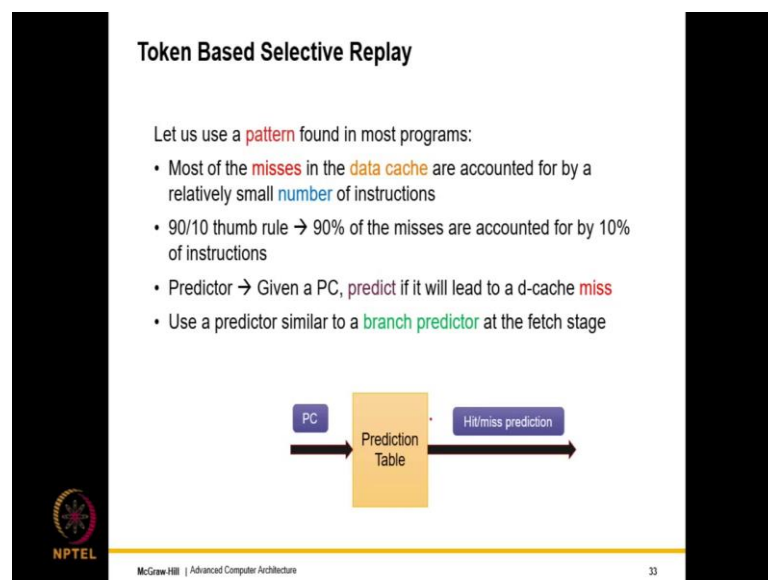
So, let us now look at token based selective replay which extends the idea of poison bits. So, let us first leverage a pattern that is found in most programs. So, what we see is most of the misses in the data cache are accounted for by a very small number of instructions.

So, typically the 90/10 thumb rule is followed where 90% of the misses are accounted for by only 10% of the instructions, again 90/10 is not hard and fast rule it can be 80/20,

but you roughly get the sense that most of the misses are accounted for by a minority of instructions.

So, we can design a predictor where given the PC the program counter we predict if it will lead to a data cache miss or not correct. So, given a PC we predict whether it will lead to a miss or not and we use a predictors similar to a branch predictor, we call this a hit miss predictor at the. So, similar to the branch predictor the fifth stage this predictor can be there at let us say the decode stage. Once we know it is a load of store and let us say for loads we can use this predictor and we can predict if it is predicted to hit or miss.

(Refer Slide Time: 60:55)



So, what do we do? What we do is that we have a hit miss prediction that is similar in principle to a branch predictor and that gives us a good idea of whether we will hit or miss.

(Refer Slide Time: 61:12)


After Predicting a d-cache Miss

Instructions that are predicted to miss, will have a non-deterministic execution time (most likely) and lead to replays (set S1)

Other instructions will not lead to replays (most likely) (set S2)

Let us consider an instruction in set S1 *prediction*

- At decode time, let the instruction collect a free token
- Save the id of the token in the instruction packet
- Example: assume the instruction: `ld r1, 4[r4]` is predicted to miss
- Save the id of the token in the instruction packet of this instruction
- Say that the instruction gets token #5
- This instruction is the token head for token #5
- Let us propagate this information to all the instructions dependent on the load
- If this load fails, all the dependent instructions fail as well



NPTEL

McGraw Hill | Advanced Computer Architecture

34

So, instructions that are predicted to miss will have a non deterministic execution time most likely and will lead to replace let us call this set S1. So, most likely these instructions will lead to replace other instructions will most likely not lead to replace the ones that are predicted to hit. So, let us call this set S2.

So, let us consider the set S1. So, these instructions are where we would want to do some kind of latency speculation or value speculation some kind of prediction basically. See, if we do some kind of prediction and since there is a high likelihood that things can go wrong we will have to provision for replace. So, let the instruction collective free token I will discuss in a second what a token is at decode time and we save the id of the token in the instruction packet.

So, essentially a token is a number. So, let us say if we have 10 tokens the tokens can be numbered 0 1 2. So, we can take any free token let us say token number 2 and this for us is the token. So, we will have a dedicated token allocator similar to a free list that we allocate a token to the instruction which we are predicting to have a problem it might need a replay and instead of a witness predicted if there is a different kind of speculation.

We can also use a confidence predictor where let us see if it is low confidence, then we can either decide not to speculate or if we are speculating we should provision for a replay. So, we give it a token.

(Refer Slide Time: 62:57)


After Predicting a d-cache Miss *confidence*

Instructions that are predicted to miss, will have a non-deterministic execution time (*most likely*) and lead to replays (set S1)

Other instructions will not lead to replays (*most likely*) (set S2)

Let us *consider* an instruction in set S1

- At decode time, let the instruction collect a free *token* #5
- Save the id of the *token* in the *instruction packet*
- Example: assume the instruction: *ld r1, 4[r4]* is predicted to miss
 - Save the id of the *token* in the instruction packet of this instruction
- Say that the instruction gets *token #5*
 - This instruction is the *token head* for token #5
- Let us *propagate* this information to all the instructions dependent on the load
 - If this load fails, all the *dependent* instructions fail as well



NPTEL

McGraw Hill | Advanced Computer Architecture

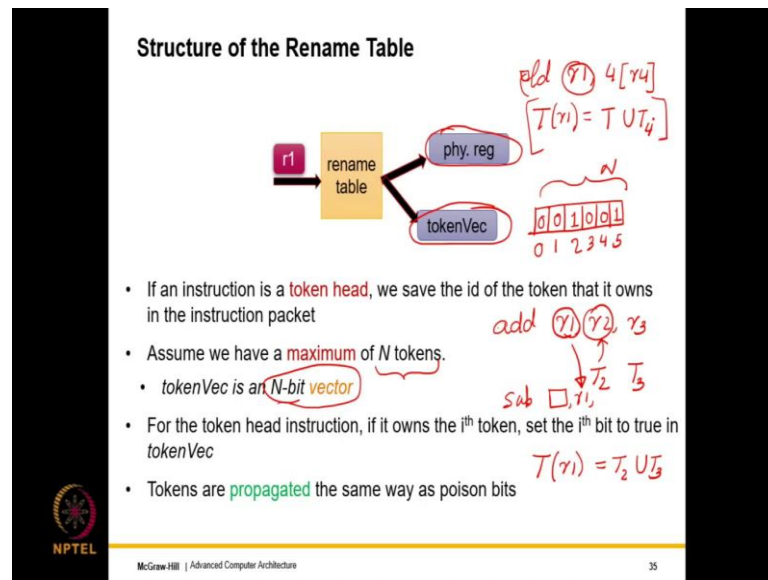
34

So, assume the instruction load r1, 4[r4] is predicted to miss. So, then what we do is we give it a token and we save the id of the token in the instruction packet of this instruction as well as in other places, but let us first we store it in the instruction packet and we say that this instruction is the token head for the token it collected. See if we let us say this collected token number 5, we say that it is the token head for token number 5.

So, that is what we say. So, then what we do is that we propagate this information to all the instructions that are dependent on the load which is similar to the way that we were propagating poison bits. So, the consumer consumer's consumer and so, on such that the entire forward slice will have token number 5. So, the load of course, will have it and the entire forward slice of the load will also have token number 5.

So, this can be propagated. So, now, if the load instruction fails in the sense that let us say we were speculating that we made a guess that it will hit in the d cache and then we did an early wake up for other instructions and as we are predicted there might be a problem and there was a problem. So, it is very easy to identify the forward slice of the load instruction just by this token id we can just squash them we will see how this is just a high level idea.

(Refer Slide Time: 64:58)



So, what we do is to implement this we change the structure of the rename table. So, we had discussed this load instruction right. So, in the rename table given a register as input, the rename table previously was yielding only the physical register and of course, the available bit, but I am not talking about that right now.

Along with that let us also store a token vector which is a bit vector of tokens and let us see if token number 5 a given instruction has received the number the bits 0 1 2 3 4 and 5 and if it contains token 5 there is a 1 here, you can also contain multiple tokens if it contains token 2 there is a one here the rest are 0.

So, if an instruction a token head we will save the id of the token that it owns an instruction packet, we have already seen that. We assume that we have a maximum of n tokens and token vec is an n bit vector which means that this vector is N bits for the instruction that is the token head.

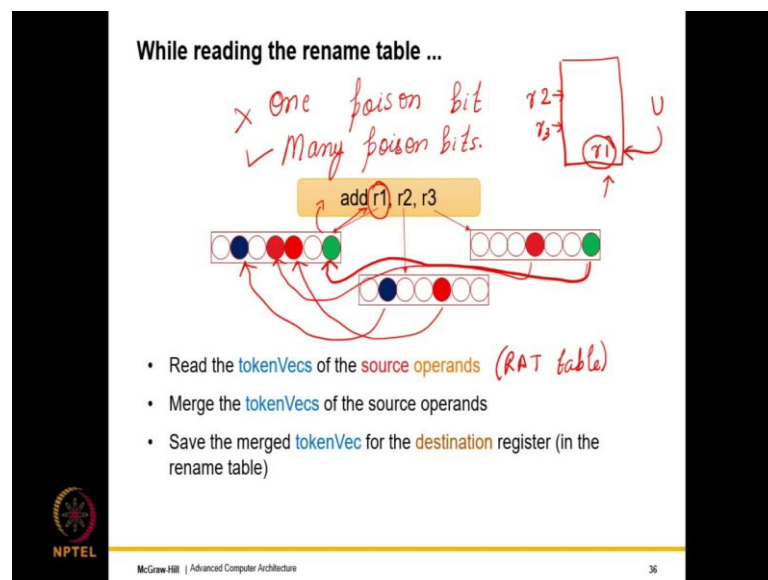
So, what we will do is that we will have a token associated here with each source register and we will have a token vector associated with each source register and we will have a token vector associated with the instruction.

So, consider any instruction of let us say this type. So, for $r2$ and $r3$ we access the rename table. So, let us say the token vector that we get is $T2$, then for $r3$ we get $T3$

right then the tokens that r1 will contain which is also the same token that the instruction will contain will be $T_2 \cup T_3$ which means that it is a part of the forward slice of both the instruction that produces r2 and the instruction that produces r3 and of course, now these are all the tokens that r1 contains.

So, any subsequent instruction that uses r1 that will automatically inherit all the tokens that r1 contains the tokens are propagated in that fashion. Now if we consider a load instruction of this type well then the tokens that r1 will contain will be the token that was assigned to r1 because it's a token head which let that be union the tokens of r4 let us call it T_4 . So, this will essentially be the tokens corresponding to the load instruction as well as the destination register of the load instruction.

(Refer Slide Time: 68:18)



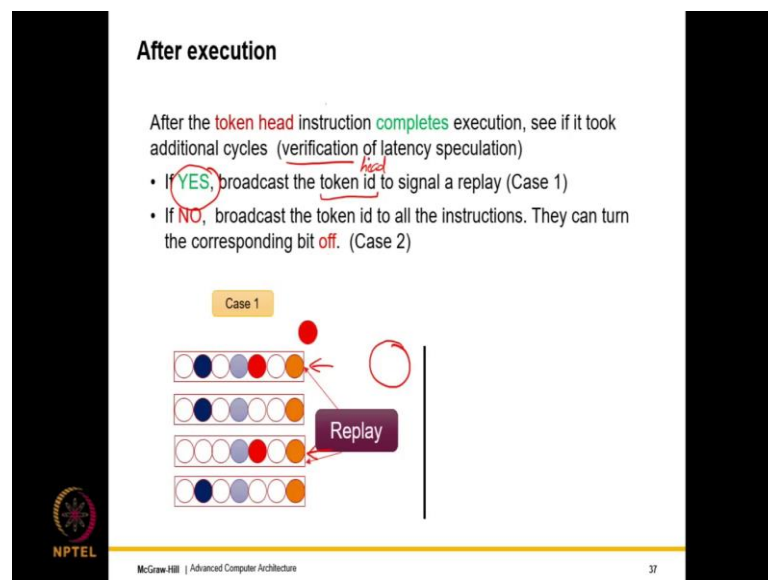
So, this is shown nicely in a diagram over here let us assume that for r2 and r3 these are the tokens shown with different colors. So, when we add r1, r2, r3 the tokens for the instruction as well as the tokens for the register r1 are basically these. So, you can see that all of these tokens show up here it is a clear union operation that we have.

So, then what we do is that in the rename table we read the tokens for r2, we read the tokens for r3 we perform the union operation and then again we write back the tokens for r1 and of course, let us see if this were a load instruction and there was a token head. We would add that also and that would be the tokens of r1. So, what do we again do? We read the tokenVECs of the source operands from the RAT table.

So, the RAT table is again being made more complicated. So, in any architectural scheme some of these structures you know have to be made more complicated we merge the tokenVECs of the source operands as you can see we save the merge tokenVEC for the destination registered in the rename table as you can see. Additionally, this is also stored in the instruction packet such that later on in the pipeline this information can be used.

So, here what is the key insight? The key insight is that instead of having one poison bit we actually have many poison bits. So, previously in the delayed selective replay scheme we had one poison bit we do not have that here, we have many poison bits right not 1, we have many poison bits and each one of them is a token and they are propagated.

(Refer Slide Time: 70:45)

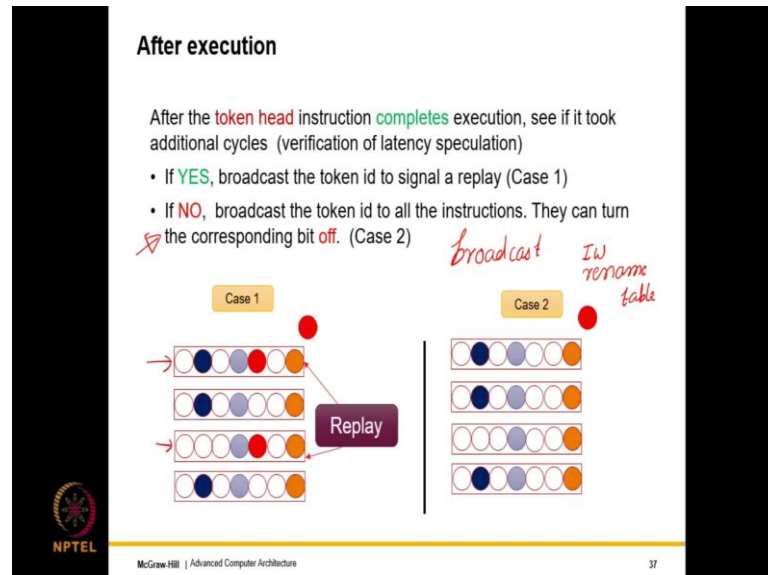


So, after the token head instruction completes execution. So, after the load instruction in this case we see if it took additional cycles. This is essentially the verification phase that we have been talking about if yes well then it means that we have a problem. So, we broadcast the token id.

So, token id is the id of that token head to signal a replay which is case 1 and as you can see let us say that this was the token head of the load instruction it is broadcasted and this signals a replay for these two instructions. Case 2 is that no problem happened then we broadcast the token id to all the instructions, they turn the corresponding bit off. So, I

should have shown this animation. So, I am clearing of the pointer and I am showing this once again that we broadcast the token id and we replay these two instructions.

(Refer Slide Time: 71:44)



But if there is no problem which is the second case then we can turn the corresponding bit off which as you can see over here that we broadcast it to all the instructions in the instruction window. And I said as I said the tokenVEC is a part of the instruction packet and we will also store it within the entry of each instruction window.

So, once you broadcast it to all the entries of the instruction window and the rename table all of them will turn this off. So, as far as we are concerned the token is released. So, this was one problem we are having in earlier case that forward slices forward slices used to grow and grow. So, this will not happen.

As soon as the token head instruction has executed correctly we will just release the token, but for that first we will have to broadcast it to all the entries in the instruction window and rename table such that they set the corresponding bit associated with the token to 0 and then the token can be added to the free list of tokens.

(Refer Slide Time: 73:11)

Instructions in S2

- Assume an **instruction** that was not predicted to **miss**, actually misses
- No **token** is attached to it
- Wait till it reaches the head of the ROB; **flush the pipeline.**

NPTEL

McGraw Hill | Advanced Computer Architecture

38

So, this was for the instructions in which we are predicting a problem we made a provision for a replay. S2 is the set of instructions for which we were not predicting a problem like for example, if we are doing latency speculation, we are predicting that for S2 there is a very high likelihood that they will hit in the cache, but then nevertheless it is a predictor. So, assume an instruction that was not predicted to miss actually misses.

Well no token is attached to it we have not attached a token. So, the only thing that can be done is we adopt the more expensive operation option over here which is to flush the pipeline. So, we have had a fairly deep discussion of replay based mechanisms.

So, we will stop this lecture video over here the next lecture video we will look at a simpler version of an out of order pipeline. And then of course, the last part of this chapter will focus on compiler enhanced techniques for enhancing ILP parallelism reducing the number of instructions etcetera.