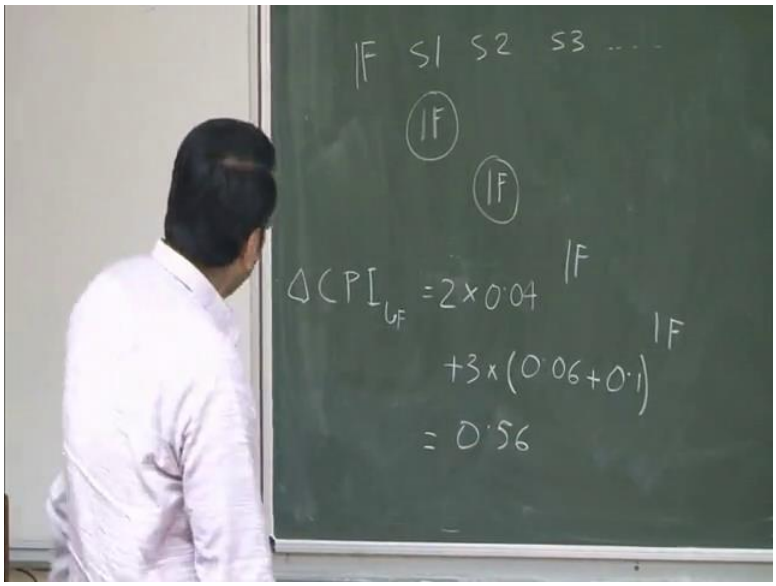
Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

Lecture - 14
Basic pipelining, branch prediction

We talked about couple of static prediction techniques. So, one technique was, that you know it is very simple, you just decide to say the same thing for every branch, that is either always not taken or always taken. So, for loop branches will say that always taken is going to be pretty good if (( )) a loop of an iterations, you may mispredict over the last time, so in the very high the accuracy. Slight improvement upon that is forward not taken and backward taken, and here we say that. So, this particular decision is taken if I have loop branches, which are mostly taken into the backward branches. While as forward not taken comes from the fact that in (( )) else type of construct, most of will execute the e blocks then s block. Now, So, these are all static techniques. You can fix the prediction at compiler, by looking at the source code you can (( )), the backward branch I am going to say taken all the time. These are forward branch I will say always not taken.

So, there is nothing to do at front, because we can make a prediction static; however, that is not enough. So, the problem arises as your pipeline gets bigger; that is we have just talked about 5 stages, but usually today processors have much deeper pipeline. and the reason for having a deeper pipeline is a if frequency, because the simple reason is that, as you cut down pipeline stages into smaller sub stages, your cycle timings becoming by the longest stage then naturally your cycle time goes down. So, you can drop down processor at the higher frequency. So, here is an example just to convey what really happens if we have a deeper pipe than what you have discussed, or although this is not a very deep pipe. So, after instruction fetch MIPS r 4 k takes 2 pipe stages, to compute the target, and one more to evaluate the condition.

(Refer Slide Time: 02:38).

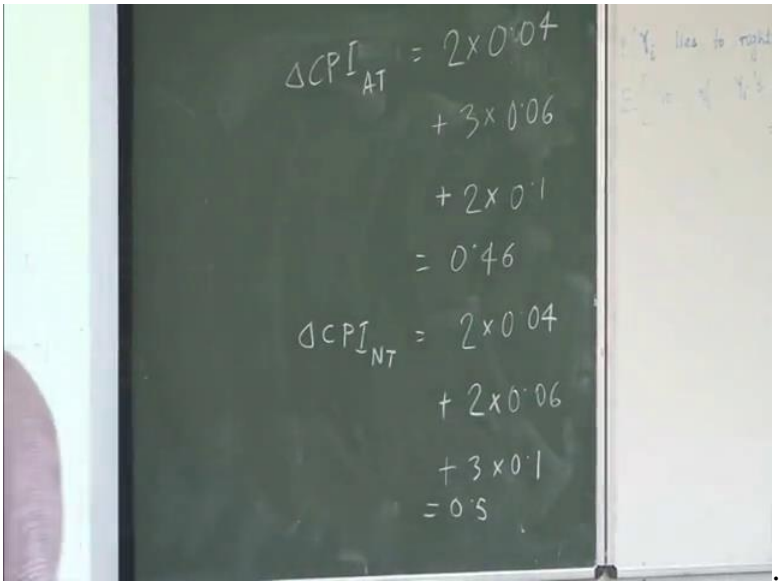


So, essentially what I am saying is that, you have the instruction fetch stage, and then you have three more stages, until you know for sure the target and the condition. So, target is known here, and the condition gets evaluated here. And then of course, you have more pipe stages to access memory etcetera not important for this example. So, let us assume that we have a program which has 4 percent unconditional jumps, 6 percent not taken conditional branches, and 10 percent taken conditional branches , and we will just ignore everything else will assume that everything else that. So, question is evaluate C P I increase for three schemes. So, we have three options, in all branch prediction; one is unconditional flush, that is whenever you come across one of these two types of instruction that is unconditional jump. All are conditional branch, what you do is, you insert singular pipeline, until you know, the target for unconditional jumps, and target and the condition for conditional branches. Second option is predicted always taken. It is this one, whatever maybe it is will be predicted. The third option is predicted always not taken. So, evaluate these three options in terms of the C P I. So, let us do that. So, first you have to figure out, how many bubbles you need to insert in all these cases, these three cases.

So, for unconditional jumps let us first focus on the first one, unconditional flush ; that is whenever you we come across an unconditional jump or a conditional branch instruction, we insert a. So, in unconditional jump how many mobs should I insert 2, because here I know what I need to do, because a target is known here. So, I do not know what to do

here, I do not know what to do here, but I know what to do here, so these are the two bubbles that is equal. So, unconditional jump instructions, under the unconditional flush model, will have a 2 cycle branch penalty. So, in this case C P I will increase by 2. So, unconditional flush. So, C P I, delta the C P I unconditional flush. So, we have two times 0.04 unconditional jump, plus what about not taken conditional branches, how many bubbles, how many bubbles in that unconditional flush model; 3. So, I have to wait until s 3 before I know what to do. So, here only I will be able to fetch the correct instruction. Same for taken conditional branches being, not taken and taken it is a unconditional flush. So, in all in both of these cases I never branch penalty of three cycles. So, 3 times 0.1 6. So, what do I get 0.5 6, so that is my C P I for unconditional flush.

(Refer Slide Time: 06:42)



So, now let us take up the next one predicted always taken; delta C P I always taken. So, what about unconditional jumps, how many cycles do I use. In these prediction model always taken, what is the branch penalty. You must be debating between 0 or 2 I am sure which one is it. Can it be 0. I do not have the (( )) which can tell me a target, I have to wait for two sides to gain correct, so 2. So, what about not taken conditional branches, how many cycles do I use; 3, I use everything, I make a wrong prediction 3 times 0. 0 6. What about taken conditional branches, 0. Are you sure, 2. I still have to wait for two cycles until they go the target I can save the third cycle of course. So, 2 times point 1. So, how much is it come to 0. 4 6. So, you can see that only say it something we say delta C P I 0.1 which is this actually, what about the other one always not taken. what is your

intuition, which one is it, which one is going to be better in this case, looking at this instruction; always delta. Why is that, percentage.

Exactly we have 10 percent taken not branches. So, always taken should be better, bias the side. So, let us see important comes to. So, what about unconditional jumps; 2. What about not taken branches; 2. What about taken branches; 3, how much this count 1 5. So, it depends on your program which one is to be better. So, you can guess what happens if my pipe gets even bigger, even deeper. If I could chop down these pipe stages even into half, to gain in terms of frequency. My branch penalty is going to increase. So, that justifies this particular statement; deeper pipelines increase branch penalty. So, must have better branch predictors for deeper pipes. So, if you are really targeting a very fast clock frequency for your processor, better have a smart department that can design both branch breakers. So, this just one requirement for a deep pipe. We will see many more requirements as you go along. So, just for getting good frequency it will make the pipe deeper, you are running this in other departments. So, keep that in mind, because of course, you will reduce your clock cycle time, but you will gradually increase your C P I, and remember that it is a product, C P I can cycle time will be your performance, you give as constant number of instructions. Any question on these example.

So, I will start again with one compiler technique, before you go on to dynamic branch prediction, and this again comes from MIPS. So, the MIPS your soon figure out that filling the branch delay slot is problematic. Often will find that you do not have instructions to you know you cannot prove that produced code is correct, and one option is of course, that compiler if the compiler could predict which where a branch is going, then of course, for the printed path it could put something in the delay slot, but that is also very difficult it predicting statically. So, if you cannot prove correctness of prediction it has to be conservative; meaning that you will actually filling the delay slot with the essentially you are wasting the delay slot. So, some of as provide nullifying branch or branch likely instruction, what is that, compiler encodes the predicted direction in the instruction and fills the delay slot accordingly. So, when compiling the program makes a prediction, which maybe correct maybe not, and based on that prediction it fills the delay slot accordingly. It naturally pulls up an instruction from a predicted path and put in the delay slot, which may actually the long.

So, now if at run time, the branch turns out to behave otherwise, the delay slot is flushed. So, when the program finally executes you find that the compiler made a wrong prediction, which means the bigger slot should not actually the execute it. So, at the time you cancel the delay slot, you turn into at run time. So, why is it I need better, compare to not having such a instruction. Sometimes it will be correct, so, it keeps the compiler more freedom, now leave it actually, to be more aggressive, which can actually make a prediction, and I have more options to fill up the delay slots, some of which will actually be correct. So, MIPS offers an instruction called cancel if not taken branch. So, that if the compiler thinks that the branch will be taken, it can fill the delay slot from the target. And of course, at run time if you final the branch is actually not taken you can cancel the delay slot. So, why did not they have a cancel if taken branch instruction. They have only one flavor of this, cancel if not taken. So, it has said something here, within parenthesis. Can somebody (( )) that.

Student: (( )) will be taken.

So, what, suppose most of the have not is taken. So, why should not I have an instruction which is cancel if taken.

Student: (( ))

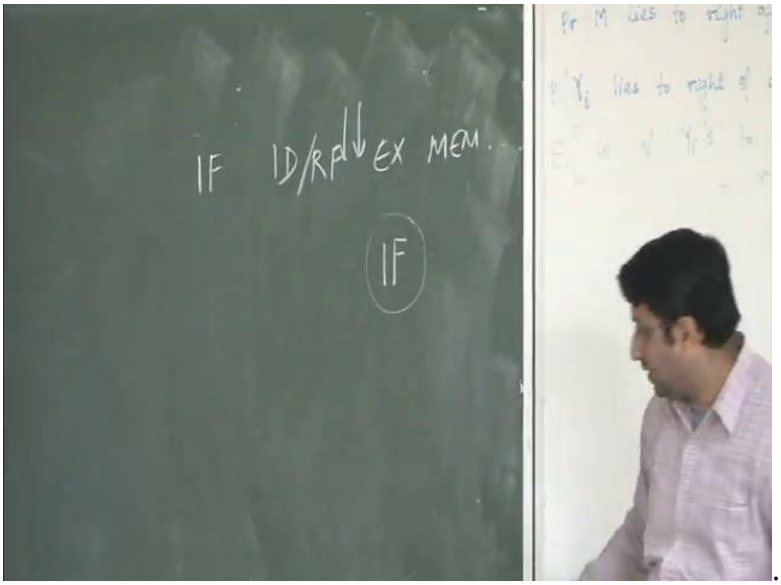
Why is that? So it would be taken; exactly. So, I need this kind of an instruction only, which will be actually useful on the last branch. We mean fall through, the last iteration of the rule. So, this decision actually was taken, to flavor a loop which actually forms a large body of count structures. We usually learn loops a lot. So, they found that if they run, you know it took statics and found that cancel if taken is not that useful; that is why support this instruction. So, MIPS has only1 branch like instruction, which is cancel if not taken. Is this concept clear branch likely, is definitely a big improvement on top of not having one such instruction, and telling the compiler to fill up the delay slot. So, now, will move on to look at dynamic prediction technique, so just to formalize the notion of that. So, this comes from the general phenomena called control dependence, and roughly every fifth instruction is a branch.

So, that that is a going statistics from your program analysis. And you need to be on the control flow path; that is very important, but executing a program, because this is the source of input in the pipeline. So, this determines whether the pipeline is wasting time,

or doing misconduct. If your input is badge, the pipeline is essentially wasting a time. So, we want high quality input meaning that you want to be on the control flow path as often as possible. Static techniques are not enough, because we need highly accurate dynamic predictors, especially when you go for very deep pipelines. And there is a need to speculate past branches, meaning that you predict a branch, start fetching from the predictor path, and while you are fetching on the predictor path, you may count one more branch, which should be able to predict also, and go along the predictor path again. So, you may have multiple predictions going on, until you resolve the first one, you really do not know where you are going, but you should be able to do is highly accurate. So, alpha 21264 no longer there actually is particular company, allows 20 outstanding branches; meaning that you can have twenty unresolved predictions. So, that they put a limit of 20.

So, which means you free it the branch, you start fetching along the predicted path, you encounter another branch you should able to predict that, and depending on this prediction you start fetching again along that path, you encounter another branch, it will predict that, and the limit is 20. So, when you reach 20 they say no more. You have to stall at that point, you cannot fetch in the point. So, on encountering the 21 branch, the front end of the pipeline will stop, waiting for at the first branch to resolve, or at least one of these branches to resolve. MIPS r 20 allows only 4, and this number actually is very important. So, soon we will see actually why.

(Refer Slide Time: 17:35)



.

So, you need to speculate past predicted branches in deeper pipelines, which is actually not a big issue in 5 stage pipe, because if you look at the pipeline that you have talked about, you will never encounter this situation, that you are on a predicted path and encounter another branch, why is that, because if you remember how our pipeline was etcetera. So, we are making a branch prediction here, in this particular stage. So, I will fetch something here, but this branch will resolve at this stage. So, next cycle I know where to go. So, I will fetch exactly one instruction from the predicted path. And there is very you know of course, there is a likelihood, but is most unlikely that there will be two branches back to back, because this instructions has to be a branch also if you want to know, but this will not be predicted, because you will never get to the decode stage if it is mispredicted. So, you will never encounter this situation that, evolve the predicted path and we encounter one more branch which we have to predict, but if you insert a few more stages here you can easily see that, now that is going to happen.

So, prediction accuracy is of course, important and we wanted to be high. So, here is a very quick analysis, just to show why this particular number is very important. So, probability of a correct prediction is p let say. An assuming that this predictions are independent, probability of staying on the correct path after n predictions is p to power of n . So, now, if you plug in n equal to 20; that is what we saying actually they have to be on the correct path after 20 such predictions. So, I want p to the power of 20 to be at least point 5, to be any sense of such a prediction mechanism. So, can anybody imagine what is the value of p will be if I want p to the power of 20 to be bigger than point 5.

Student: it will be close to 1.

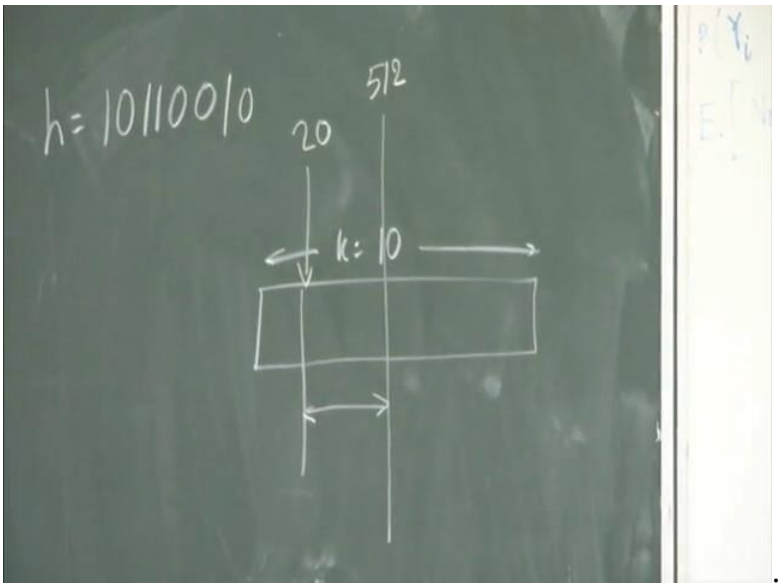
Close to one; yes. So, it turns out to be 0.97. So, I am demanding branch predictor or prediction accuracy 97 percent to make any sense of this number. If you have p to the power of 4, p comes to about 0.85 which is probably achievement. So, this number is very important, it cannot just increase this number arbitrary, then hardly your chance of being on the path. Very soon what will happen is that, you will be on the wrong path for sure, unless you have highly accurate predictor. So, keep this in mind. So, essentially what we are going to do is, we will design predictors, goal is to make p as high as possible. Is this analysis clear to everybody?

So, let us first try to define the problem more formally. So, let us encode each taken branch as 1, and not taken branch as 0. So, the behavior of a conditional branch can be represented as a binary string, it will be 1 0 1 1 1 1 0 0 0 as the branch executes sometimes it be taken sometimes it be not taken. And the and what will a loop branch look like, the branch that is it is the loop. Exactly it will be 1 star 0 yes. So, keep on doing 1 possibly 0 times it will be fall. So, the problem of direction prediction is essentially designed on a estimator, that given an history, tells us the next most likely outcome for a particular static branches. Actually if I give you a branch instruction, I give you its n bit history, and I ask you design an estimator, which will tell me the most likely outcome, on the next execution of this particular branch. Is it clear to everybody, this particular formulation. So, what all branch predictors do is, they compute the probability of seeing a 0 or 1, given the recent patrol history h of some limited length n. So, we have seen the history h of length n, and you ask the priority of getting a 0 or 1 after this. So, suppose the number of times 0 appears after h is c 0, and similarly define c 1. The prediction is 1, if c 1 is bigger than equal to c 0 and 0 otherwise; that is make sense. I have see more ones after h that 0, so I said that well is the most likely prediction . So, instead of actually having two counters, what we maintains c 1 minus c 0. So, it is the difference, is maintained, that is enough. Is it clear?

So, let the difference counter is c h for a certain history h. So, since we have to design the hardware, we have to decide size of these counters. So, let c h be of k bits length. This is independent of the history length remember that. We has nothing to do with how long h is, h maybe other. So, what does it mean; that means, c h can count from 0 to 2 to the power k minus 1. It is a final length, on seeing 0 after h, you decrement c h on seeing 1 after h you increment c h. And it saturates at boundaries, which means it is a saturating counter. So, does not decrement below 0 or increment above 2 to the power of k minus 1. So, by examining c h at any point in time what can I say. We can say which outcome had higher likelihood, in the last 2 to the power of k occurrences of history h , because I can only count in this range, so if the if the counter value is 10 I can say that well 1 appear 10 times more sorry. So, you need to shift the origin to the midpoint first, so 2 to the power of k minus 1 which is very important. So, from the distance from the midpoint, if you are below the midpoint you do that you had seen more zeroes than ones. If there midpoint you know that you had seen more ones than zeros; however, that you can do, that you can say only about the last 2 revolve k occurrences, because it will go up. Suppose you

are seeing ones, it go up gradually saturating we want move very fast. So, cannot really say beyond that point what happened. So, if c h is below the midpoint the prediction is 0, otherwise 1. Is this algorithm clear to everybody?

(Refer Slide Time: 24:25)



Can you say something sorry, one example for this. You want an example sure. So, I have a counter, let suppose k equal to 10, 10 be small. So, what is a midpoint 512. So, that is my threshold 512, and I have a history. So, that corresponding to this particular counter. So, history can be something like 1 0 1 1 0 0 1 etcetera. So, here let suppose I fix 8 bits of history. This is my h. So, what this counter maintains is, after this history h, whenever this history 8 occurs, if a 0 comes this counter value will be decremented, if a 1 comes after this history this counter value will be incremented. Now suppose that some point in time I found that the counter value is 20, what can I say for this value. Well I can probably say that 490. Sorry say it 492 0 more than 1 exactly yes. So, this part is the deficit. This may zeroes I have seen more than ones after h, but that I can say for sure only in this pan, in the last thousand 23 occurrences of h this is what has happened. I cannot see beyond that actually, because of this limited length of the counter.

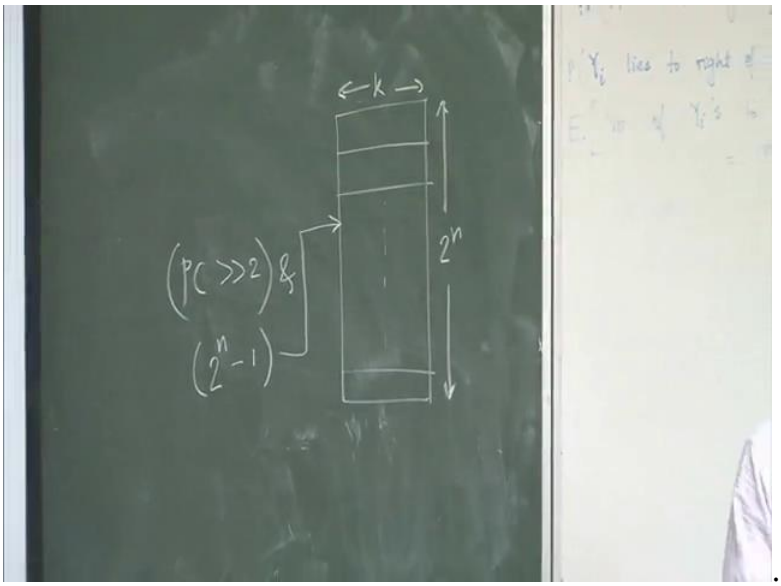
If you get almost it come to here. After that we cannot predict. No you can still predict whenever you see 1 0 you will start moving in this direction. If we want 0 0 that is fine I will predict more; that is what it says, if c h is below midpoint prediction is 0 otherwise 1. You cannot add anything to that, but my prediction is still correct. I will just stay here

at this point and saturated. Any question on this. Are you done? Do you have a branch predictor, well not yet, which still have to determine once more how many different histories do you want. Here we have focused only one counter corresponding to one history, is not it. Now each branch can have millions of different histories, and there maybe thousands of branches in a particular program. So, let us start with a simplest option.

Suppose you do not have k history. We have no history, what does mean. You just have a global counter that counts occurrences of zeroes and ones. So, there is no history, whenever I see a 0 I decrement the counter. Whenever I see a 1 its increment the counter. So, it is not very useful actually, because already its telling you that well how would many braches that I have seen, this many branches are taken, this many branches are not taken, it takes difference of that . Of course, based on that I could make a prediction. So, well this program seems to be bias towards taken branches. I say taken when the value of the counter is allows the threshold otherwise. So, based on the last 1024 branches I have seen, I will make a prediction essentially, based on the population of the history. So, it is not used in any purpose, because it is not very accurate that you can guess.

So, the what is the next option. So, here essentially. So, you can you can improve this a little further you can say that, well I still have no history, but one counter much. So, is somewhat useful, because it helps identify largely bimodal distributions, because what is it giving you. You have one counter per branch, and that counter is telling you the behavior of this branch, and if you have to decade the bimodal nature of the branch. So, this is the branch is largely bias towards not taken, or largely bias towards taken will make mostly correct predictions for those branches, but a branch which fluctuates in between, you know you probably make a large number of mispredictions in those case. So, these called a bimodal predictor, and this actually used in many, so one counter. It is not exactly one counter per branch, we actually use a hash aging for per counters.

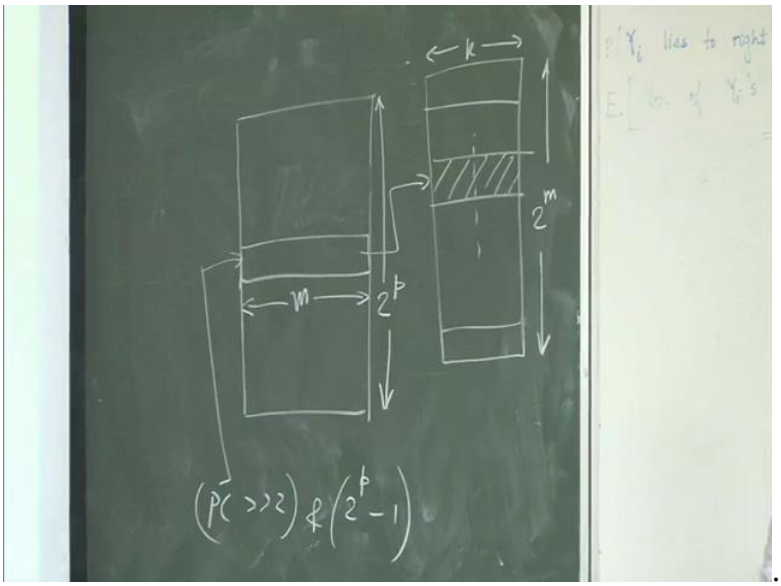
(Refer Slide Time: 29:21)



So, the bimodal branch predictor, essentially it is a table of counter, table of saturating counters. And you take the branch pc shift on the last 2 bits as always. Take a module of number of counters you can afford, and it is called 2 to the n minus 1. See I have 2 to the n counters here, each k bits; that is my bimodal branch predictor. So, this predictor will what extremely well for branches which are primarily not taken with primarily taken , but it would not be able to detect any pattern in branches; that is impossible, actually we does not have any history, it is only counting the number of zeroes and ones in a c that is all. Is it clear to everybody bimodal predictor.

So from no history, let us make one small step. Suppose we give you a 1 global history register; that means, you still do not have one history per branch, but you have all history register, whenever you see a branch you shift in its outcome. So, if you have a history of say n bits, you know the outcome of the last n branches that you have seen. So, it captures cross correlate between branches if there is any. Since history is a sliding pattern h will keep on changing keep in mind. This is not a fixed history that you are maintaining.

(Refer Slide Time: 31:22)



So, it can gradually keep changing, with this capture the sliding window of pattern. So, now, the question is, fine I have this history. Should I have one counter for each history pattern? So usually keep leaves a hash map again. So, what does it look like. We have a global history register, of some length small m let say, which will be used to index into a bank of counter to be the n counters. So, can somebody tell me what am I doing really here. So, whenever I see a new branch what I will do is, I will shift this register by 1 bit position, and shifting the new outcome of this branch. So, it is capture in the outcome of last aim branches, it is a binary string. So, what is it large in nature. Of course the prediction mechanism is still same; whenever I get a branch I use the correct content of the global history register, look up the corresponding counter. And if the counter is above 2 to the k minus 1 I say the branch is taken, if it is below to the k minus 1 I say the branch is not taken. So, what is it doing actually?

No I am not asking you to come up with program constructs where this is going to be good or bad. I am just asking what is it doing can somebody articulate, that in English. Does it make it sense, that is the first question. Actually this is doing anything useful. So, what will the number in a particular counter corresponding at any point any times. Sorry which branch, is it by the group of branches. What is this counter attached to, each counter is attached to something very unique what is that. How I index to it a counter, what do I use to that do. No what is this are you say history to takes it to a particular counter. So, the history attach to a particular counter will remain same throughout , do

you see that, this counter will get indexed only when I see a particular history here . See if I assume that there is no meaning that I have enough counters here, each history gets a new counter in this table. So, now, somebody can tell me, what is this actually measuring this particular counter.

Student: how many time this kind of situation.

Exactly, So, this counter is telling me that the history corresponding to this counter, how many times that had a 0, after that history, and how many times error a 1 after that history, this making the difference if that . So, each counter gets attach to a particular history, it does not have anything to do with a particular branch. So, when a branch shows up, with a particular content of the global history; that means, I am asking well tell me in the past what happened, when you saw this history. Was it a 0 after this or 1 after this, and that is what this counter is going to tell me the most likely outcome. So that is a huge informant over low history. So, this predictor has very high accuracy actually. So, we will actually soon come up with an action of your predictors. So, I am just giving you the glimpses.

I will soon name these predictors, they have names. And then of course, you can extend it, say that well instead of a global history I could have one local history per branch. So, I make this a table, instead of single register. So, this is not a global history register any more, this becomes a history table, and I will make sequence. Can anybody guess. I want a local history per branch, what my next step is branch pc exactly. So, if I have 2 to the p h is here, I essentially do pc shifted by 2, and at with the 2 to the p minus 1 that is my index. So, even though I may not have1 history per branch if my table size is limited. I will at least have you, there may be some collision of course, but in a if in a region of a program as long as I have no more than 2 to the p branches I am happy. Here whether I do not really care when I move on to a new region of the program what happened in that region. So, hash history patterns to counters, but it loses global correlation. Now you will not be able to see there is a relationship between this branch and this branch, these histories, which you actually were having when you had a global history back. So, these are roughly three types of branch predictions that you will see. So, I will soon come up with a naming convention for this. So, any question.

Student: this very slow like we are updating it, we are reading it, and then we compute something.

Updating there is no update going on yet.

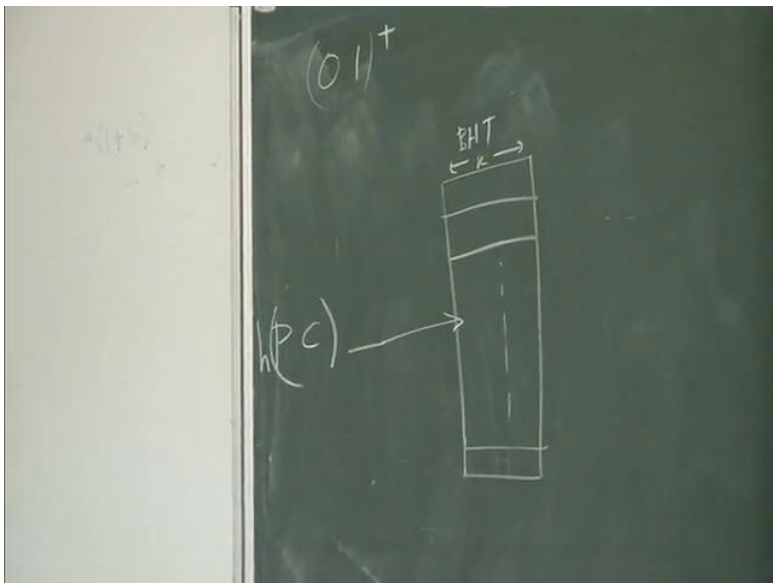
Student: we are updating counter register history.

When the branch finally, executes you will update this and this both, but that is not the time of prediction, that is later. At the time of prediction you look to this up, after you look this up and you make a prediction, and when you when you finally, execute the branch you know the correct outcome, at the time you can update it. You cannot update with the predictor outcome. This has to be correct all the time, whatever you learned.

Student: to get the outcome of.

Yes you can execute exactly. Of course, you can mix as many as you want. Yes you come to know. Yes any other question. So, we have already talked about this 1 bimodal predictor. So, some names here with table how counters are called a branch history table. So, that you will have multiple branches mapping to the same entry, wherein some cases that would that maybe very bad. Two branches may behaving in two different ways they map the same entry; 1 will increment 1 will decrement you will get nothing out of it actually. How wide is each counter, do you really implement the counter. Well how wide is each counter, so that determines how far you can see, the span of your, you know.

(Refer Slide Time: 39:43).



You actually do not really implement a counter here, you just have a machine implementing a saturating counter. So, you know when the state is 0 1, when you have to implement particular counter value what will be the next will be 1 0. Actually do not implement the counter you just implement the final state machine, before two states given the particular. Performance of loops on the bimodal predictor, how will it be good bad, always good. So, talking about this predictor if you have forgotten. We talked about 5 times, sorry 5 minutes ago. So, you have branch history table which is k bits wide this are bank of saturated counters, indexed with some function of the branch pc. The function is basically a module hash.

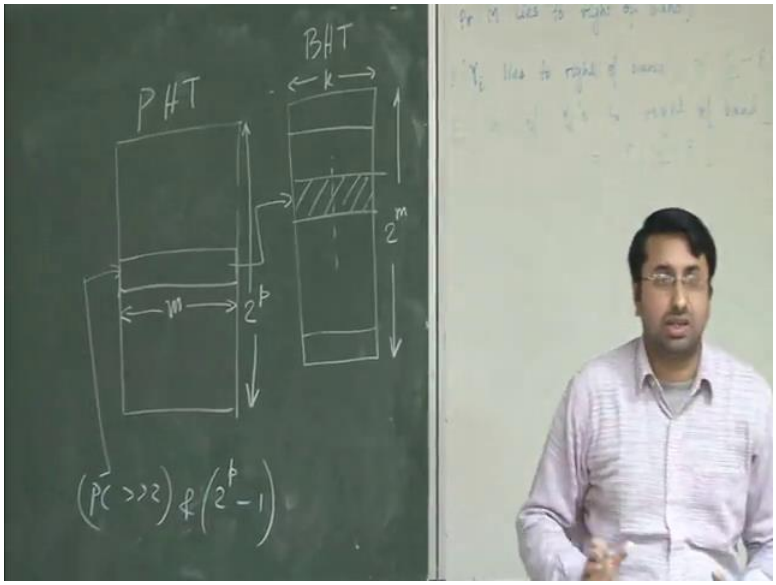
So, we are asking what can you say about a loop branch, the accuracy of. It will be correct, except the last occurs last and first. Does it have any relationship with k. Are you assuming something about k, which is what? I tell you that k is very important here why is that. what happens if k is very large, this should I taught, this should be an easy question to answer .You see a loop keeps on incrementing the counter that is taken gradually, last time it decrements, what happens in it if the value of k is large, it is a very wide counter. It will. I am just trying to figure out if I have a large counter what will happen, can you tell me what will happen. So, if you cannot take values. Suppose k is 10 I have loop of hundred iterations what will happen. Counter will not saturate very good. So, what, or it may saturate, you are assuming something, when do I start from. Did I say anything about that no. So, not make sense, what is the initial value of this counters sorry middle. By starting middle what will happen, if I have hundred iterations, I should be fine, I will make a mis-prediction in the last batch.

Now, the thing is that this counters are actually initialized to 0, when the machine. So, if I assume that it is 0 if I have a counter of 10 bits and a loop of 10 I 100 iterations what will happen. Except the last time I will make mis-prediction all the time. Well it never be because of threshold. So, usually these counters are small, you will often find 2 maybe 3 bits never more than that. So, I really do not want a very large history. I want a small window, I want to know what happened in that small window; that is it. So, I do not need a large window, because the problem is that, often having too much of history may pollute your prediction, because what happened in distant past, may have not changed actually that behavior. You may not see that behavior. Usually these counters are small

to capture, a small window of history. So, what about alternating branches. alternating branches have this particular pattern.

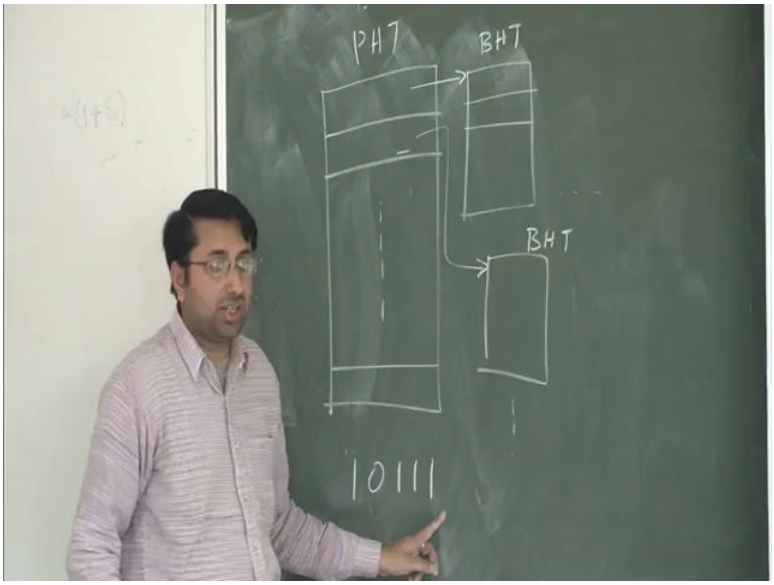
What will happen to them prediction accuracy; 50 percent are we assuming something about the initial value, mid, if I initialize to 0, 50 percent. If I initialize to somewhere if the counter is too large and if I initialize to on the side, still 50 percent. So, it is not very useful that is what it means if you are 50 percent correct, that is more is random I could make a random, actually I could cross a cross an coin and be happy. So, alternating branches we need something better bimodal predictions are not good for this. What about correlating branches, that is branches that have relationship across. I have an example there it is. So, suppose we have this program if f is equal to x then you assign y to f, if g is x then you assign y to g and then you say if f not equal to g you. So, do you see that, the outcome of this branch depends on the outcome of the previously branches . So, these called a globally correlated branch and symbol. So, your branch which depends on the history of few other branches. So, even if I tell you the exact history of this branch, it will very difficult for you to predict what will happen next time in this branch execution, but if I tell you the history of these two branches, you probably be able to come up with a correlator very easily. The pattern will emerge immediately that look if this previous two branches behave like this am getting this, you catch the pattern immediately. Is it clear to everybody?

(Refer Slide Time: 31:22)



So, this requires two levels of table. So, these are the labels of table. The second level table is the branch history table as we just mentioned, and this one is called the pattern history table. And this table may have you when just one entry; that is also fine, that still called a pattern history table. For example, if you had just a global history register that will still be the pattern history table, but having just one entry.

(Refer Slide Time: 46:20)



So, then we can come up with the taxonomy of branch predictors, depending on how you exactly index with the first level table and the second level table. So, the first level table can either be global, can be per set or per branch. So, what does that mean. If it is global; that means, you just have one register, the global history register, when you maintain the history of all branches. So, whenever you encounter a branch, you shift this bit 1 1 1 position to the left, and you shift in the new outcome, and all the branches are mixed up here. So, that is called a global first level table, or the global pattern history table. Then you can have per set where essentially what you do is, you have a table, and a set of branch, is get one entry into the table. So, that will happen when you take the pc, and do a module operation on that. A set of branch will map to one entry and then of course, you can have per branch entry, you can if you know beforehand how many branches are going to see my program, you can size a table, so that every branch gets exactly one entry. Is it clear this three things.

What are the second level table? There also you can do the same thing. You can have a global second level table; that is exactly what that is. There is most specific table, branch history table attached to one particular counter. All counter can update all history can update any counter here, but every counter will get attach to a particular history. For example, suppose this counter gets indexed when you will have the history 1 0 1 1 1 1; that is it. Now this history will appear sometimes in this particular history entry, may appear in this branches history sometimes, may appears here sometimes. So, whenever this history shows up, you will index in to this counter and make an update. So, that is very important to understand that these counters are not attach to any specific branch, a particular counter gets attach to only one history, and that is why the name comes from global. So, global branch.

And then you can have per set branch history table, where you say that well a set of histories, will map to a table. So, I am basically have a table for set of histories here. So, I will have multiple second level tables. And the third one says that well you can have a per branch table, so that for each entry I have a branch history table. So, for that branch it will have all the histories accumulated in that tables counters, and then you have the update method 1is a static method, usually not used. So, here essentially what you say is that you just statically encode this table at book time, you program a table statically , so there is no update as such. The other one is called adaptive update which is more popular, as you go on you learn update to tables. So, combining this people name branch predictors. So, first entry here corresponds to the type of the first level table. Second entry here corresponds to the update method. You invariably find an everywhere you never see an s here, these are all adaptive predictors, and the last corresponds to the type of the second level table. So, for example, pap basically means what. What will a pap predictor looks like.

So, will have a big first level table, each branch gets one entry in this. This will point to one branch history table. This one will point to another branch history table and so on, then how you will update any entry here. So, you take the history here, whatever history you have, that will be used to index into all the counters here, and that counter will be incremented. Similarly the history is used here will be used to update the counters here so and so. Is it good or bad compare to that. It is a gigantic predictor, you can easily see that. You will have many entries in this table and each entry will correspond to a table.

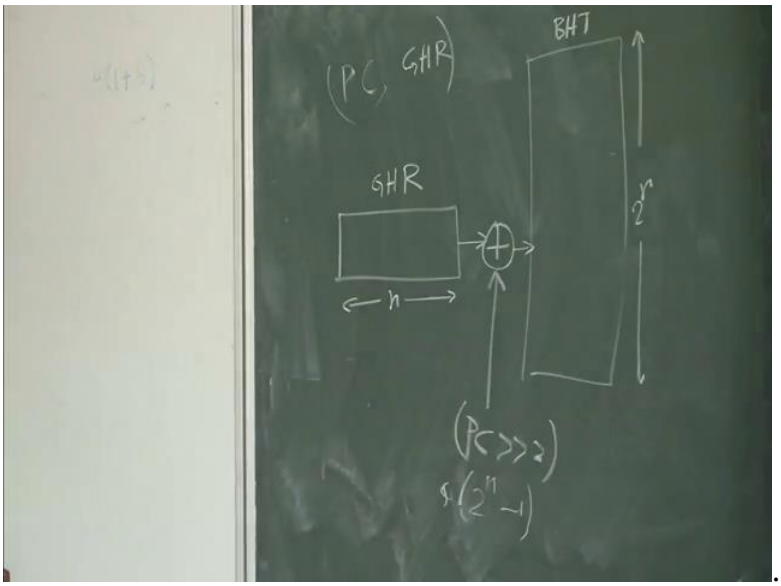
Student: (( )) adding something to (( )) always updating same counter (( )), its nullify the affect.

So, if we have a particular history and it, after this sometimes 0 appears sometimes 1 appears. So, in that case what will happen is that, what were you saying. Exactly, so depending on which branch it is coming from, it may have to different behaviors, but it will update the same counter actually. So, that we have a destructive interference effect. Is that a good effect. So, what he has pointed out is a bad thing, that we should try to avoid, that is exactly what is trying to avoid actually. Two entries in the table updating the same counter when is it good. You learned something in one course, and the same thing, the other course is that good; good or bad, or pretty not better. Is it or will not faster. So, in this case there are two histories, that reinforce they are learning, is going to learn very fast actually, at a much faster rate, to reach the learned pattern quickly. So, this is my pap predictor. Similarly you can come up with sag. Sag is probably the one of the most popular 1; that is a sag predictor. It has a each entry corresponds to a set of branches here, and it is a global table.

Student: Pap (( )) it not be beneficial for correlating branches. So, need the sag.

No sag has some. It will be able to run some correlation provided you are your backing function will be able to map those correlate branches to a same set, but that is an accident. It is impossible to design such a hash function before, and for that purpose you use a gag. So, you have a global entry in the first level and I have a global table in the second level; that will correlating branch predictors. So, what does each one buy. This one allows you to learn each branch separately. This 1 gives you more or less the same accuracy at must cost. This one allows you to learn global correlation.

(Refer Slide Time: 54:14)



There is one special case that is called a g share predictor, which is more or less same as gag except that, it has a slightly different index function. So, we have the global history register as usual. We have your branch history table as usual. So, if this is n bits, how many entries do I have here. Sorry 2 is to power of n. Before you index it, it went with pc. Why, what is it mind. I show it as a x or because that is what it is used most popularly. So, you can count with other smart functions why we have wanted to do this; g share is amazingly good I can tell you, so it is a good thing to do. Mapping the same branch. No there is no such guarantee, there is no such guarantee. You examine it with global history register it will pattern in the (( )). What is it trying to do, why do I bring in the pc. What way lose in a global predictor.

Suppose I do not have g share and a branch history table what is it that I lose? Sorry branch wise prediction I really do not have any learning about the local history of a branch. So, I am just trying to introduce some flavor of that by exhaling with pc, the hope is that a particular branch, when it sees a particular history, it will get a particular different entry here. So, essentially I am trying to map this tuple; pc comma g h r want to this counters. The hope is that each different pair will get a different counter. Of course, I could do other things like I could catenae these 2, but my table will become now gigantic. So, I will have to come up with some function.