

**High Performance Computer Architecture**  
**Prof. Ajit Pal**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 17**  
**Branch Prediction (Contd.)**

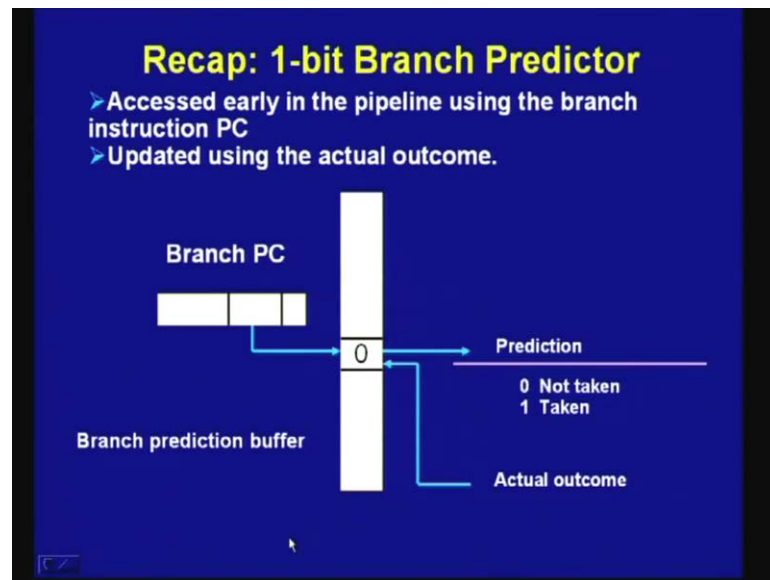
Hello and welcome to today's lecture. We shall continue our discussion on branch prediction.

(Refer Slide Time: 01:03)



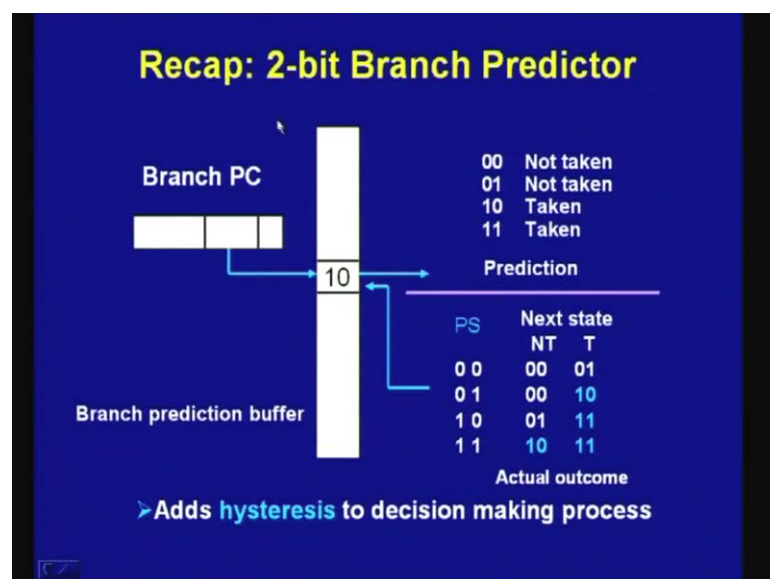
In the last lecture, I have introduced to you the basic concepts of branch prediction and three different schemes have been introduced. That is 1 big branch predictor buffer, two bit branch predictor buffer, then co-relating branch predictor buffer. And, today I shall introduce to you the other techniques related to branch prediction; like tournament branch predictor, then branch target buffer, integrated instruction fetch units and return address predictors. So, these are the topics I shall cover in today's lecture.

(Refer Slide Time: 01:42)



And, let us have a very quick recap of the different schemes that I discussed in the last lecture. This is the simple 1 bit branch predictor and this is accessed early in the pipeline using the branch instruction PC. That means, in the instructions fetch cycle then the that branch PC used to index pillar; particularly the lower order bit is used to index a buffer where 1 bit is stored and that is a prediction it can be taken or not taken. However, if the prediction is right or wrong depending on that it is modified. So, if it is 0, then it is not taken; if it is 1, then it is taken. So, if the prediction turns out to be wrong, then actual outcome is return into this buffer. So, this is how 1 bit predictors works.

(Refer Slide Time: 02:38)



And, we have seen 1 bit predictor does not give you very good performance. So, a two bit branch predictor was introduced, where instead of 1 bit 2 bits are stored and the 2 bits represent 0 0 corresponds not taken, 0 1 not taken. That means, whenever the value is half or more, then it is taken; that means, 1 0 and 1 1 corresponds to taken. And, if this is the prediction and depending on the outcome, you know the modification is done in the branch target buffer. And, for example, if the present state is, it can be described as a finessed machine and so it can be represented with the help of a next state table.


So, here you can see this is the previous state that corresponds to the prediction 0 0, 0 1, 1 0 and 1 1. And, if the next, if the outcome is not taken, then this is how it is modified and it is return into the branch target buffer. So, if it is not taken, if it was 0 0, it remains 0 0. If it is 0 1, it remains 0 0 and if it is 1 0, then it becomes 0 1 and if it is 1 1 and if the not taken it is 1 0. So, that means, if it is not taken as you can see this is decremented. On the other hand, if it is taken then it is incremented. If it was 0, it is incremented to 0 1; it was 0 1, it is incremented to 1 0. And, whenever it was 1 0 it becomes 1 1 and if it is 1 1 it remains 1 1; because it is a we are realizing a saturating counter.

So, the actual outcome is again stored in this two bit branch predictor. And, essentially what is done by this two bit branch predictor, it adds hysteresis to decision making process. That means, unless two predictions are right or wrong, the prediction bit is not modified. So, this is how this two bit branch predictor works.

(Refer Slide Time: 04:44)

### Recap: Correlating Branch Predictor

- How can we capture the behavior of last n branches and adjust the behavior of the current branch accordingly?
- Answer:
  - Use an n bit shift register, and shift the behavior of each branch to this register as they become known.

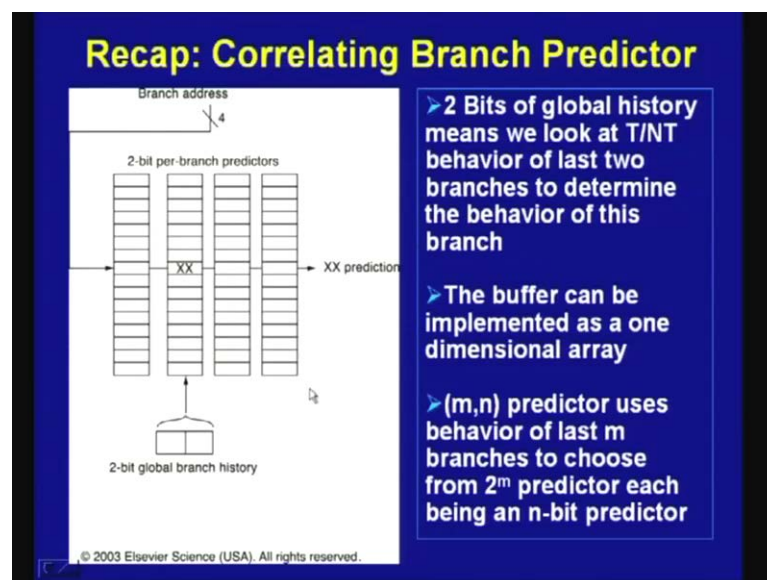


- How many possible values will our shift register have?
- Imagine that many tables and select the table you want to use based on the value of the shift register.

Then, coming to the third type; that is your co-relating branch predictor. Basic idea is that how can we capture the behavior of the last  $n$  branches and adjust the behavior of the current branch accordingly. So, in the 1 bit branch predictor and two bit branch predictor, what is being done; it tries to identify what happened for the current branch. It does not concern about the other branches; that is present in the program. But in co-relating branch predictor, it is a kind of it takes into account the global scenario. So, what happens to the other branches that is taken care of in these co-relating branch predictor? As I have introduced in the last lecture and how it is being done? The answer is use an  $n$  bit shift register and shift the behavior of each branch to this register as they become known. So, the previous  $n$  branches may not be the branch under consideration; it may be other branches, their outcome is stored in a shift register and this is being used.

And, question naturally arises how many possible values will our shift register have? So, the number of bits will again decide the number of possible outcomes you would like to consider. For example, if it is three bits than there will be 2 to the power of 3 possible outcomes, you have to choose 1 out of 8 tables. That means, imagine that many tables and there are many tables and you have to select 1 of the table on the basis of this shift register value and decide it.

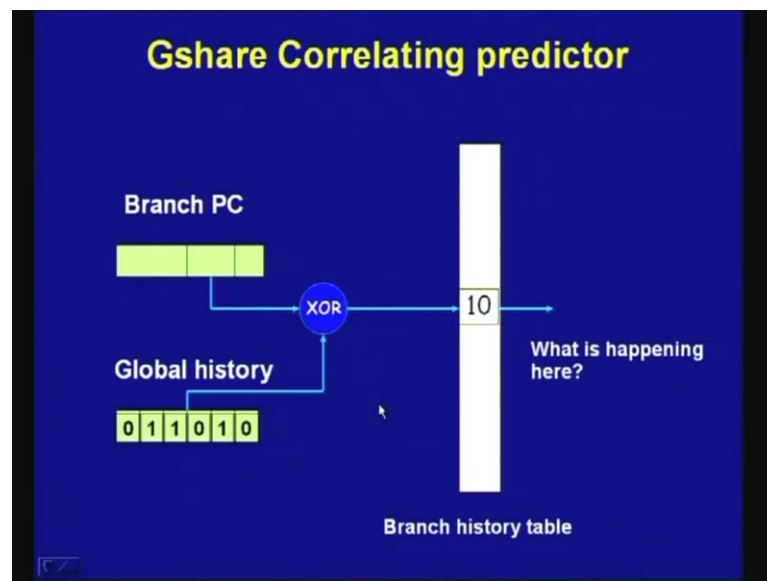
(Refer Slide Time: 06:30)



And, as you have seen I have discussed it with the help of this diagram; how these co-relating branch predictor works. Here, two bits of global history, that is being used and

that selects 1 of the 4 possible tables and where the prediction values to be stored and that can be used for the purpose of prediction. So, this is the co-relating branch predictor. And, in general we can have (m, n) predictor use predictor which were, which uses the behavior of the last m branches; and obviously, total numbers can be 2 to the power n. and, then with the help of the low lower order bit address, branch address it selects 10 bit predictor and that is being used for the purpose of prediction. So, this is the recap of co-relating branch predictor.

(Refer Slide Time: 07:26)



And, 1 popular correlating predictor is known as Gshare co-relating predictor. In the Gshare co-relating predictor as we can see how the global history is being incorporated for the purpose of prediction. So, here the branch PC lowered bits of the of the branch PC is XOR with the global history. And, then that is used for the purpose of indexing in the branch history table and that is being used for the purpose of prediction. So, this is how Gshare co-relating predictor works.

(Refer Slide Time: 08:03)



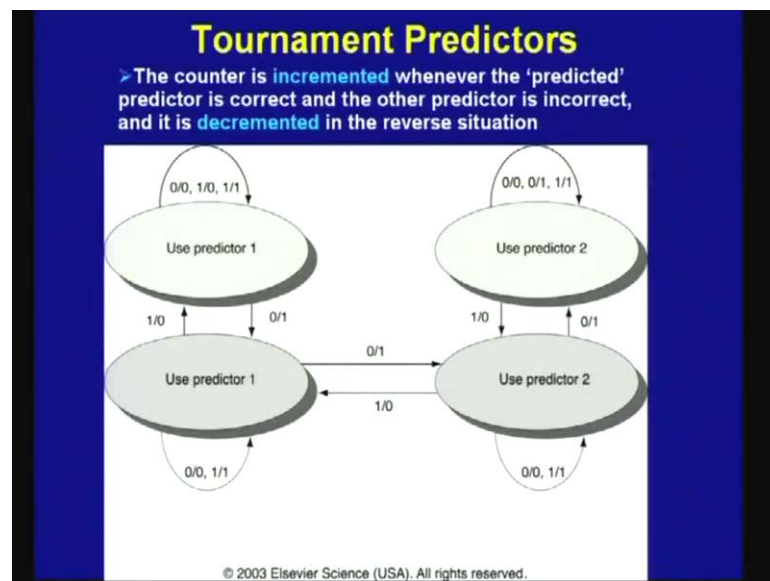
Now, we shall focus on tournament predictors. This is based on the observation that performance is improved by adding global information. I mean based on the observation that performance is improved by adding global information; tournament predictor takes a step further. We have seen in co-relating predictor, we are considering we are combining local and global predictions and trying to come up with the prediction. So, and it we have seen that the, this correlating predictors performed very well and can we really goes a step further. Here is the, that is the basic idea behind tournament predictor.

What is being done it uses multiple predictors; one-based on global information and the other based on local information. So, for simplicity you can have only two predictor; one based on local information. Local information in this particular branch whether it was taken or not taken earlier. And, global information means other loops, other branches whether they were taken or not taken; that information is used. So, this is the difference between local and global. Now, you can have two predictors 1 local and 1 global; you have to select 1 of the two predictors for the purpose of prediction; that is the basic idea of the tournament predictor.

So, it adaptively combines local and global predictors with the help of selector. And, obviously, it does dynamically with the help of hardware. Because at runtime for a particular branch, at a particular instant it can be taken and in, at some other instant it can be not taken. So, it dynamically changes. So, adaptively combines and selects one of the

predictor at runtime. So, this is the most popular among the multilevel branch predictors. Later on we shall have a look at the contemporary processor and we shall see how these tournament predictor is being used in different recent processor. So, this tournament predictors ability to select the right predictor for the right branch. That means, you may have many branches; so, it identifies which predictors will give you good result. So, it identifies right predictor for the right branch; that is the basic approach that is being used in tournament predictor.

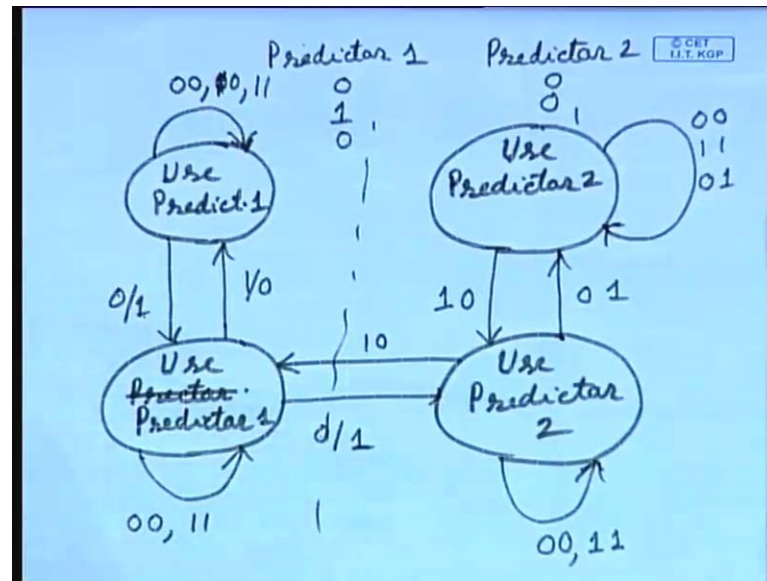
(Refer Slide Time: 10:47)



So, let me explain this tournament predictor.



(Refer Slide Time: 10:52)



Let us assume we have got two predictors; predictor 1 and predictor 2. And, there are 4 states; say here it is used predictor 1 and another is, here also use predict 1 predictor 1 and here it is use predictor 2 and you have got 4 states; predictor 2. Now, how its features from 1 predictor to another. So, what is being done? Let us assume it is in this state, in the state use predictor 1. Now, you have got 2 predictors; predictor 1 and predictor 2. Now, if both of them turns out to be 0; that means, both turns out to be wrong, than its remains in the same it state. So, if it is the outcome is 0 0; that means, both the predictors turns out to be wrong than it remains in this particular state. And, say predictor 1 turns out to be right and then the predictor two is wrong. Then, it since predictor 1 has remain true, so, it remains in this state again for 0 1 0. And, if it is 1 1, I mean if it is 0 1; then, it will switch from these to these. If it is 0 1, then it will go from this state to another state. And, this kind of I mean this predictor 1 if it fells twice; that means, if it is 0 1 and again 0 1, and predictor two becomes right in this case. That means, if it is 0 1 again then its goes to predictor 2.

That means, that predictor 1 has to fell twice and predictor 2 has to becomes true twice; only than it will go from predictor 1 to predictor 2. So, this is the boundary between predictor 1 and predictor 2; so it will go from here to here. So, for all these state 0 0, 1 0 and 1 1; it will remain in this state and only when it is 0 1. That means, predictor 1 has turn out to be wrong and predictor 2 has turn out to be right; it will go from this state to a this state. And, it will remain in this state again as long as both of them are false; both of



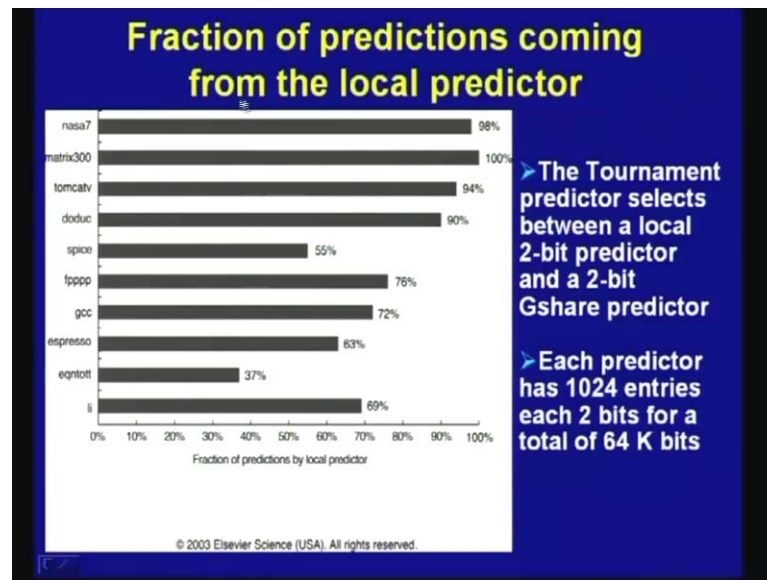
them, both the prediction are false or both the predictions are right. And, only if when it is the other way happens; that means, if it is 1 0, then again it will go back this state. That means, predictor 1 is right and predictor 2 is wrong it moves in that direction; so, it will go from here to here. And, whenever it is 0 1, it will go from predictor 1 to predictor 2.

Similarly, here it will remain in this state or let us consider this one first. So, whenever you are using predictor 2 and I mean let us assume that you are in this state; predictor 2 state, predictor 2 is being used. And, if it is 0 0, again it will remain in this state. If it is 1 1, it will remain in this state as we have seen, that is true for all the cases. 0 0; that means, both the predictors are wrong or both the predictors are right. So, in both the cases it will remain in the same state; state does not change. Now, a it will go, it will move in the direction of predictor 2 and it will remain in this state as long as this predictor 2 is correct and predictor 1 is wrong.

So, it will remain in this particular condition as long as this is true. And, it will come from this state to this state, whenever it is 1 0. That means, predictor 1 is right and predictor 2 is wrong. So, it will come here and again if the same condition holds; that means, predictor 1 is right and predictor two is wrong, if it happens twice it will go to predictor 1. That means, predictor 2 if it fails twice, then it will go to predictor 1. And, the from other hand it will go from this to this is the 0 1. So, from this state to this state it will go; that means, you are moving in the direction of predictor 2. If predictor 1 has fell twice and predictor 2 has turn out to be correct.

And, on the other hand it will go to predictor 1; if it is turns out to be true twice and predictor 2 fill twice. So, this is the state transition diagram for this tournament predictor. And, so, basic idea is the counter is incremented whenever the predicted predictor is correct and the other predictor is incorrect and it is decremented if the reverse situation. So, based on this the tournament predictor, prediction works and you can see here only two predictors has been shown; it is not necessary that you can have only two predictors. So, for the sake of simplicity to illustrate the operation of tournament predictor we have consider two predictors, but there can be more than two predictors in real processors; as we shall see later. Particularly, later on we shall see that (Refer Time: 17:08) processor; where this tournament predictor has been used and of course with much more complexity that we shall discuss later.

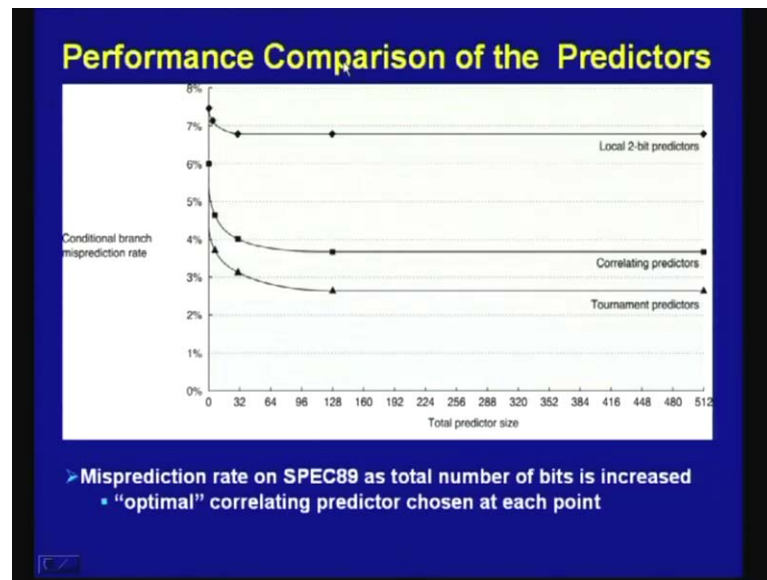
(Refer Slide Time: 17:19)



Now, this particular diagram shows fraction of the predictions coming from local predictor. So, we have seen that earlier in either the predictor 1 can be used or predictor 2 can be used. And, here this particular diagram shows for different branch marks, whether predictor 1 has been used; that is the local predictor has been used or global predictor has been used. So, as you can see the tournament predictor selects between a local predictor, two bit predictor and a two bit Gshare predictor. That Gshare predictor, which I have already discussed earlier, which is a co-relating predictor which used as global information for the purpose of prediction. And, each predictor has 1024 entries, each of two bits for the, for a total of 64 kilo bits of information.

Now, here you can see the local predictor is used different percentage of times for different applications. So, it is very much application dependent. For example, it can vary from 100 percent to 33 percent; that means, local predictor has been used 100 percent for some application and in some cases only 37 percent of the times the local predictor has been used. And, for the remaining part of the time in the global predictor has been used. So, this simulation shows how the tournament predictor works and the variation of the any use of different predictors. So, this is the simple case of two predictors, but in real life as I told they can be more predictors. And, these are the different application programs which I have already mentioned about, mentioned earlier. So, fraction of prediction by local predictor is shown in this diagram and the remaining percentage corresponds to global predictor.

(Refer Slide Time: 19:37)



Now, this is the performance comparison of different types of predictors. We have broadly categorized predictors into 3 types; local, based on purely local information and we have seen two bit predictor gives you better result. So, local two bit predictor; another is which uses both local and global information, that is known as co-relating predictor and third categories turn tournament predictor. So, here as you can see for different predictor size; how the performance changes for different predictors.

So, this is based on misprediction rate on SPEC89 as the total number of bits is increased. So, as the total number of bits is increased how the number of misprediction changes; obviously, as you increase the number of bits the performance should improve. That means, misprediction weight should be reduced but for 1 bit predictor as you can see the reduction is not really much. So, it is a from little more than 7 to little less than 7 that is the decrease and there is no improvement; if and if you increase the size of the branch prediction buffer. So, even by increasing the size of the branch prediction buffer, there is no improvement in performance for local two bit prediction as you can see here.

Then, for co-relating predictor which use as both local and global information and you can see starting from 5 percent. It decreases to little less than 4 percent for SPEC89 programs and you can get a predict quite lately related a good performance misprediction where it is only is less than 4 percent. That is for the co-relating prediction and actually that has used that Gshare predictor for the, for this purpose. And, third is and here what

has been done optimal co-relating predictor has been chosen at each point. So, optimal co-relating predictor has been used for plotting this diagram and the last curve where the misprediction rate is less than 3 percent for the tournament predictor, which I have just discuss in detail. So, for tournament predictor you get very good performance and you can see, you can get a performance; I mean less than 2 percent. That means, for more than 98 percent of the cases prediction is correct. So, it will definitely give you good improvement in processor performance.

(Refer Slide Time: 22:30)



**Branch Target Buffers: The Need**

- In the classic 5-stage pipeline, an instruction is identified as a branch only in the ID stage
  - Branch prediction buffer can help decide whether to fetch from target address (fast pipeline), or fall-through
  - However, IF still ends up fetching a (possibly) useless instruction
- So, even with perfect branch prediction, cannot achieve 0-cycle branch latency
- Solution: Branch Target Buffer (BTB)
  - Accessed during the IF stage
  - A cache that stores predicted address for the next instruction after a branch
- Variants
  - Store only predicted taken branches
  - Works for 1-bit local predictors: store entry when changing prediction to T
  - Use separate target and prediction buffers

Now, coming to another very important need; that is your important the requirement for good performance of the processor; that is branch target buffers. So, far we have focused on branch prediction buffers. So, outcome of the predictions are stored in the buffer and which is used dynamically in different predictors. Now, let us see why do we need branch target buffer on top of branch prediction buffer. So, in the classic 5 stage pipeline and instruction is identified as a branch only in the ID stage. And, branch prediction buffer can help, decide whether to fetch from target address; as we have already seen or from fall through. So, however, instruction fetch is still ends up fetching a possible useless instruction.

That means, whenever you are using only the, I mean branch prediction buffer. So, then what happens? That is been shown given here; so, even when prediction perfect branch prediction it cannot achieve 0 cycle branch latency. The reason for that is you know that

calculation of the address, that will be taken place is done in the either in the execution stage; usually it is done in the execution stage or in a later stage. So, if your prediction is correct taken or untaken, where the branch will taken, take place is not known in the second cycle. So, it is necessary to have branch target buffer. So, solution is to have branch target buffer. And, this branch target buffer which is accessed during the instruction fetch cycle. So, a cache the stores predicted address for the next instruction after a branch.

So, you can use a branch target buffer which will store the information of these of this target address in a cache memory. And, that will be used for the purpose of jumping to a particular address. That means, you know the, whether your branch will be taken or not taken and where it will jump; that is also known. Of course, if your prediction is untaken, then no problem; it is PC plus 4. But when your prediction is taken, then that branches has to be known. And, that branches is calculated if the address is PC relative, we have already seen that in most of our; I mean the example that we have given, that is for the instruction say it of the simple pipeline.

(Refer Slide Time: 25:31)

PC- relative. PC+4

$$EA = (PC) + (\text{Displacement})$$

$$\frac{Ex}{\uparrow}$$

Content

$$\frac{ID}{\uparrow}$$

Addressable Memory.

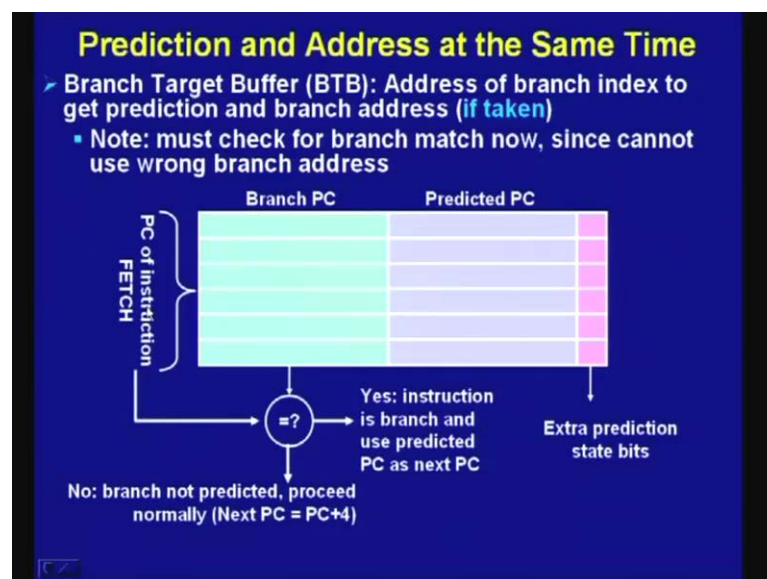
© CET  
U.T. KGP

There you know PC relating address is being used. So, in the PC relative branch case what is being done; that content of the program counter is added with the displacement or offset. That is being added to get to find out the effective address. And, these effective address calculation can be done in the instruction, decode stage or in the instruction

execution stage; depending on whether we are having addition hardware or not. If you do not have any additional hardware or adder, it is done in the execution stage. And, if you use a special adder for the purpose of calculating this effective address, then it is done in the instruction decode stage. So, either in the execution stage or in the instruction decode stage with the calculation is done. That means, you have to wait till the end of either instruction decode stage or the end of the execution stage to get the branch the address. And, obviously, it will lead to; it cannot achieve 0 cycle branch latency.

So, there will be a delay of either 1 cycle or 2 cycles depending on whether when the branch address is calculated. And, now there are some variants, you can store only the predicted taken branches. So, branch may be taken or not taken; it is not really necessary to store the address; address of the, I mean when the branch is not taken because in that case your address is already known; that is your PC plus 4. But only if in branch is taken then you have to calculate this way, if you use pc relative address which is use in the context of branch addresses. And, these works well for 1 bit local predictors and store entry when changing the prediction of t. That means, when the prediction changes; then this that address has to be changed. And, only taken branches information is stored in the branch target buffer. And, you separate target and prediction buffers. So, you can use two separate buffers; one where you will store the branch predictions and another where you will store the branch target buffer. That means, where it will jump and this is done in this way.

(Refer Slide Time: 28:15)



So, branch target buffer, address of the branch index to get the prediction and branch address. You must note that must check for branch match now; since cannot use wrong branch address. So, earlier that branch target buffer, that was used without any tag, but in case of branch target buffer we have to use tag which is conventionally used in cache memories. As we shall see this is very important because for the branch target buffer must check the branch match now; since it cannot use wrong branch address. So, this is how it is being done. So, you can see the cache memory is, in this case is storing not only predicted address; that is the predicted program counter address. But it is also storing the branch PC as part of the cache memory. So, after you get the PC value of the instruction fetch ,after you have fetch this instruction and you know the that branch PC than that is being compared with the branch address. That means, this branch PC.

So, if this done in a, as it is done in conventional manner and it is known as content addressable memory, CAM. So, what is being done; you are essentially comparing with a value and then you are using this address and correspondingly you are getting this value. So, this is the cache memory used in this manner and if there is a no match branch not predicted; so, proceed normally. So, you will go for PC plus 4. So, if there is no match with this value; so, what you are doing? You are in content addressable memory, you know the content is compared with this address; with the address. That is being done here. So, here also you are trying to match a content with the PC of the instruction fetch. And, if there is no match then it is not, the content is not present here and branch is not predicted proceed normally.

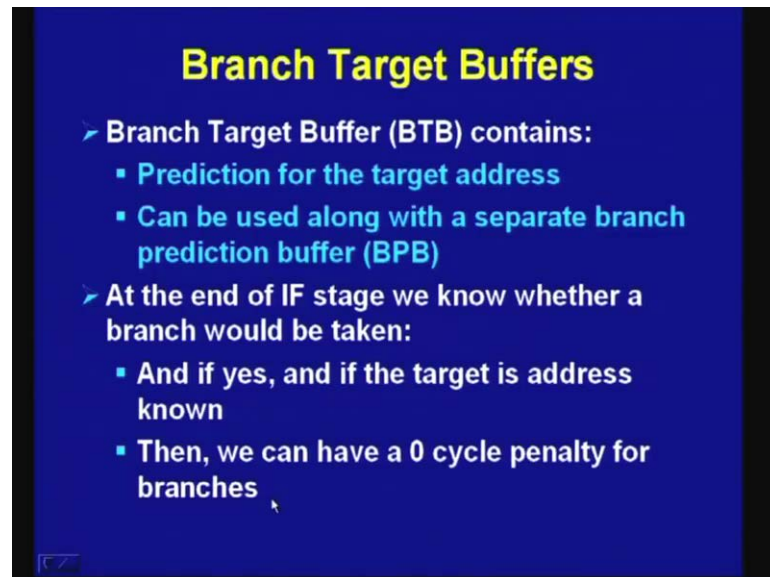
On the other hand, if there is a matching with anyone of the content that is being present here; that means, in your branch target buffer, that target buffer; the PC value which has stored program counter value which has stored, if there is a match with anyone of them than there is a match. And, you say that yes, instruction is branch and use the predicted PC as the next PC. That means, corresponding for the corresponding program counter value, you will get a predicted PC; where the branch a targeted, where the branch should target, brand should take place. So, here you get this predicted PC and that can be used for the purpose of fetching instructions immediately.

So, you can see a using this, you can have 0 delay. The reason for that you are getting the information whether the prediction is taking place or not. You are getting the information where the branch should take place, if it is taken. So, in the next cycle itself; that means,



in the, after the instruction fetch has been done, in the next cycle itself you can fetch the instruction from the target address and its execution can proceed. So, delay 0, 0 delay; it can work with 0 delay. So, prediction and address at the same time; that is the basic idea of using branch target buffer.

(Refer Slide Time: 32:37)

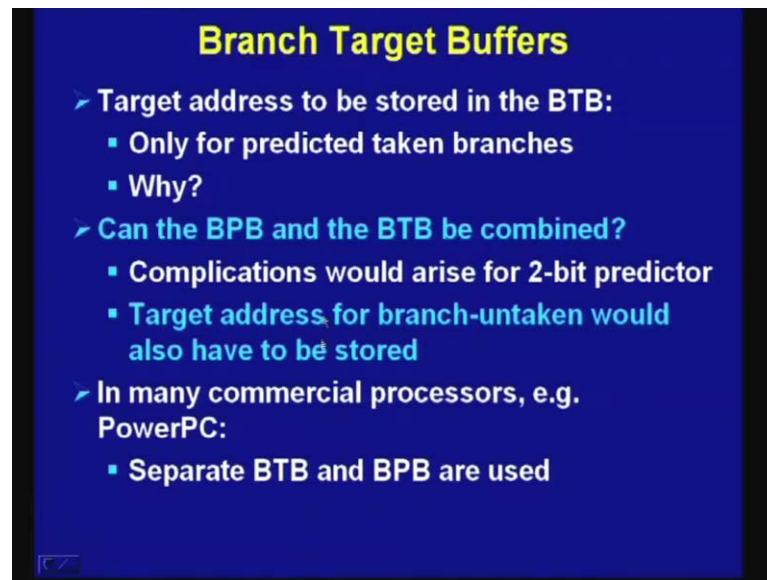


### Branch Target Buffers

- Branch Target Buffer (BTB) contains:
  - Prediction for the target address
  - Can be used along with a separate branch prediction buffer (BPB)
- At the end of IF stage we know whether a branch would be taken:
  - And if yes, and if the target is address known
  - Then, we can have a 0 cycle penalty for branches

So, branch target buffer contains prediction for the target address can be used along with a separate branch prediction buffer. And, at the end of instruction fetch stage we know, whether the branch would be taken and if yes, and if the target is address known; then we can have 0 cycle penalty for branches as I elaborated in detail.

(Refer Slide Time: 33:00)



### Branch Target Buffers

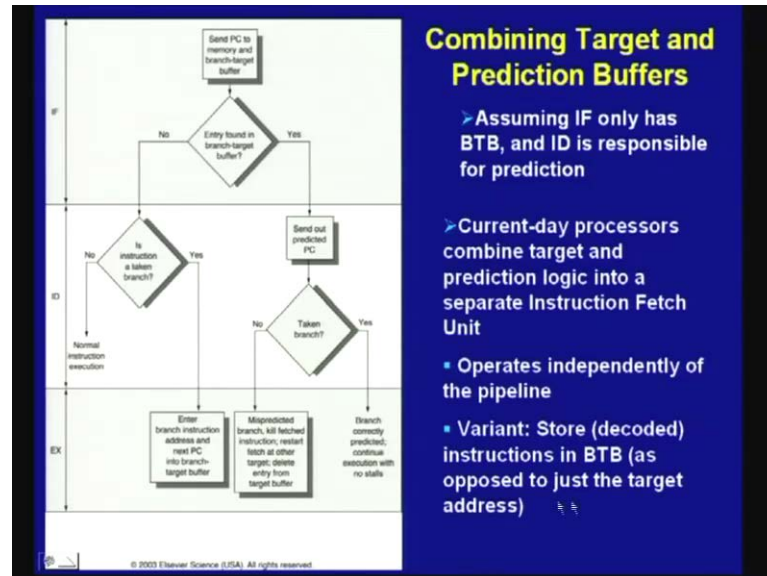
- Target address to be stored in the BTB:
  - Only for predicted taken branches
  - Why?
- Can the BPB and the BTB be combined?
  - Complications would arise for 2-bit predictor
  - Target address for branch-untaken would also have to be stored
- In many commercial processors, e.g. PowerPC:
  - Separate BTB and BPB are used

Now, let us come to some, integrities of this branch target buffers. So, target address to be stored in the BTB. So, will you store only the predicted taken branches? If it is not taken, then what will happen? If the prediction is not taken we know it is PC Plus 4. So, it is not necessary to store the target address, when the prediction is not correct. And, can the branch prediction buffer and branch target buffer be combined? So, this can we use the, here we are trying to tell can we use a single buffer to store the prediction as well as to store the branch target address. This can be used very conveniently if it is a single bit 1 bit prediction. However, whenever you go for two bit predictor, then it leads to come leads to some problem; complication arises.

Why complication arises? We have seen in a two bit predictor 0 0, 0 1, 1 0, 1 1 predict untaken, predict untaken. That means, for that entry and again 0 1 predict untaken, predict taken. So, what you have to do; this will require the target address for branch untaken would also have to be stored. That means, whenever we are using two bit predictor, it is not only necessary to store the information of only the branch taken addresses. But also you have to store the addresses of branch untaken addresses. That is the complications that arises, whenever you go for a combine branch prediction buffer and branch target buffer in the case of two bit predictor. And, in many commercial processors like power PC, separate branch target buffer and branch prediction buffer are being used; to avoid these complications. And, invariably we will see the branch target

buffer is not a single bit; always two bit is used. And, that is there is in why to separate buffers are used one for branch target buffer and one for branch prediction buffer.

(Refer Slide Time: 35:35)



Now, here how we can combine branch target and branch prediction buffer is illustrated with the help of the store chart. And, how the pipeline the behaves whenever you have got branch target and branch prediction buffers. And, we have assumed that instruction fetch only as branch target buffer and ID is responsible for prediction. And, instruction decodes stage is responsible for the purpose of prediction. So, here the first step is send program counter to the memory and branch target buffer. So, the program counter value is send to both in the instruction fetch cycle, it is send the send PC to memory as well as branch target buffer.

So, it is branch target buffer and after sending that whether you have to check whether entry found in the branch target buffer or not. We have seen there is a, if we go back you have to see, check whether that entry is present in the branch target buffer or not. So, that is being done by this, in this step itself; entry found in the branch target buffer. If the entries found in the branch target buffer, than question arises whether there are 2 possibilities; branch may be taken or not taken. If the instruction is taken, instruction is taken branch; if the answer is no, normal instruction execution. Normal instruction execution means, the program counter will be with the address will be fetch; I mean

instruction will be fetched from PC plus 4 for the next pipeline that we have discussed. So, that is the case and you can do it in the ID stage.

Now, if the instruction taken branch, if the answer is yes; that means, your prediction is branch is taken. In such a case what will happen; enter branch prediction address and next be seen to branch target buffer. So, in this case you have to enter your information in the enter branch instruction address and next PC into the branch target buffer; if your prediction is taken. And, that we will do in the execution stage. Now, let us see whenever entry is in the previous case entry was not found in the branch target buffer and that is why you have to go off to the execution stage. And, there will be a delay of you can see, you are losing two cycles; in this case. So, you cannot fetch instruction in this stage, you cannot fetch instruction in this stage. So, there is a loss of two cycle in these particular case.

Similarly, let us see consider the case where the entry is found in the branch target buffer. So, if the entry is found in the branch target buffer, you can send out the predicted PC and if there are 2 possibilities again; branch may be taken or not taken. If the branch is taken, then branch correctly predicted and continue execution with no stalls. So, in the, in this case you can there will be no stalls; you can go ahead with your predicted address. Now, if the, if you are out come turns out to be wrong; so, in this case also, there is no loss of clock cycles. But whenever your prediction turns out to be wrong, in cases then you have to I mean your prediction is wrong; so, misprediction branch. So, what you have to do? You have to kill fetched instruction, restart fetch at the other targeted addresses, delete entry from target buffer.

We have seen only the address of the taken branches are stored in the branch target buffer. So, when it is untaken you have to remove that particular entry and not only that, you have to undo the instructions which have been fetched and you have to fetch from the from the next another address. And, as a consequence here also you will lose 2 cycles. So, in this case you will lose 2 cycles. So, in two cases you are losing two cycles and in two cases we are not losing any cycle. So, current processors combine target and prediction logic into a separate instruction fetch unit and operates independently of the pipeline. And, there is a special variants store decoded; later on I shall discuss about it this is variants.

(Refer Slide Time: 40:13)

Branch Target Buffers			
➤ Branch penalties			
▪ Assuming new target is written into PC only at the end of EX			
Instruction in Buffer?	Prediction	Actual Branch	Penalty Cycles
Yes	Taken	T	0
Yes	Taken	NT	2
No		T	2
No		NT	0

So, I was mentioning about the, when there will be an penalty of two cycles and when there is no penalty; that is being given here assuming a new target is return into PC only at the end of execution cycle. So, instruction in buffer, in whenever it is yes and prediction is taken and actually is taken, there is no penalty. And, your prediction was taken, but actually not taken, there will be loss of two cycle; as I have already explained. And, if the instruction is not in the buffer, than if your prediction is not taken, but actually if it is taken; then again you will lose 2 cycles. And, that means, in this case you will lose two cycles but if your actually if it is prediction is not taken then again there is no loss in your cycles. Because it was not you are it was not taken; so, you will not lose any cycle. So, this is how the branch target buffer works and we can have losses into cases and no loss in two cycles; that means, we can say 0 latency.

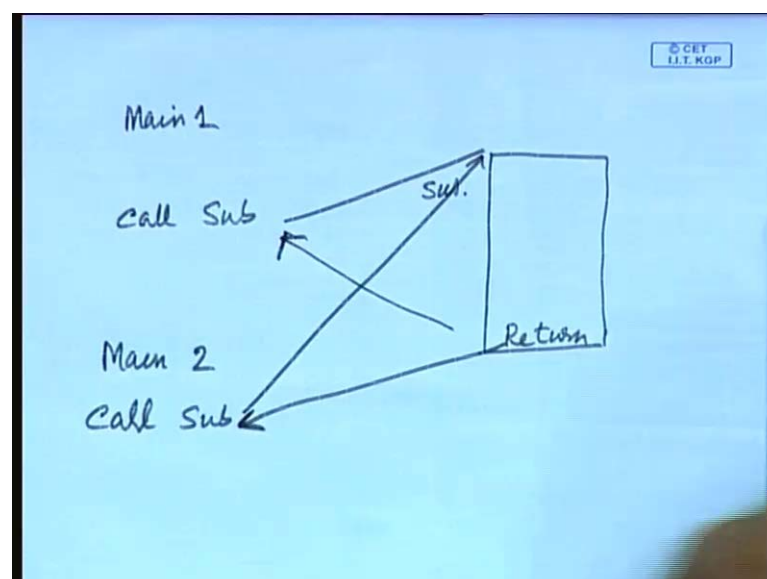
(Refer Slide Time: 41:23)

### Return Address Predictors

- Techniques discussed so far work only with direct branches
  - Target is PC-relative (part of the instruction)
- An important category of indirect jumps: **returns**
  - More than 15% of the branches in SPEC89
  - (more in languages like C++, Java)
- Can we predict the return address?
- **Option 1: Use BTBs**
  - Accuracy tends to be low
  - Return address depends on call site
- **Option 2: Use a Return Address Stack (RAS)**
  - Push an entry to the RAS at a call
  - Pop the entry upon the return
  - Perfect prediction if call depth does not exceed RAS buffer size

Now, there is another requirement, particularly for unconditional cases; return address predictors. You know techniques discussed so far work only with direct branches. That means, you have go direct branches and at this calculation is done with the help of that PC relative. So, and then branch is taken or not taken, but there are many indirect branches whose outcome is known, whether it will be taken or not taken at execution time. So, where you know that target is not PC relative; in such a case an important category of indirect jump is returns. You know you will find you are doing subroutine calls.

(Refer Slide Time: 42:18)



So, here you have got it main program and you may be calling a subroutine. And, so it will jump to a subroutine and here you will have a return. So, you will be returning to this point, in this particular case. But these subroutine, this subroutine may be called not by one main program; this is a main one and another program say main two; another main program is again calling subroutine, say this is same subroutine. So, in this case it will go here, but you have to return not here, but to this point. So, you find that this particular situation, we find that for the same subroutine; that return address has to return to different addresses; if this particular subroutine called by different a main programs for multipoint places multiple places. So, in such this type of returns instruction you will encounter in many situations.

For example, more than 15 percent, the branches in SPEC89 and particularly in object oriented languages likes C plus plus and Java this type of, you know unconditional branches are present in many places. So, in such cases you know that the techniques that we have discussed will not work properly. So, can we predict the return address? So, in this case what you have to do? You have to predict the return address. Just like you are predicting the, with the help of branch target buffer you are predicting where the branch will take place. Here you have to predict where the return will take place.

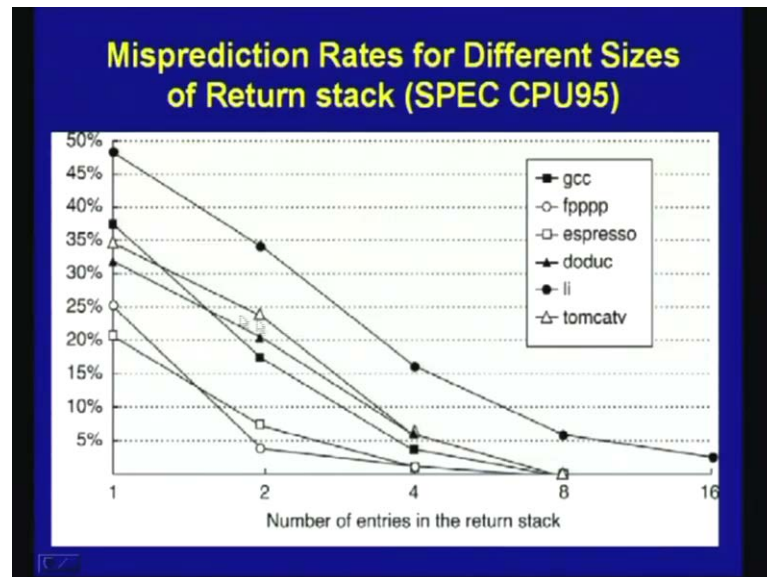
So, there are two possible alternatives, two possible options; 1 is use branch target buffer. So, if we use simple branch target buffer to store the return address; what will be the outcome? Accuracy tends to be low; the reason for that I have already explained. Say the return address cannot be same, cannot be stored for we will not corresponding to a particular branch PC. That means, it can be it can be dependent on which particular program is calling this is subroutine and return has to take place to different points. So, return address depends on the call site; so, call site can be different for different instances of the call single subroutine. So, what is the alternative? Alternative is to use a return address stack. So, this is how the problem can be overcome. So, with the help of return address stack, you can push an entry to the return address stack at a call and talk the entry upon the return. So, this is how you can resolve this problem.

So, in addition to branch prediction branch target buffer you will require a return address stack. So, this, the perfect prediction is call depth, does not exceed the return address stack buffer size. Of course, one thing you have to keep in your mind; stack will have limited size as long as the, there is no stack over flow; your prediction will be that



prediction of the return address will be always correct. Only when there is a stack overflow, you will not get correct branch prediction. So, as long as your, that stack depth is enough; you will get perfect prediction. But if there is stack overflow then of course, it will not give correct result.

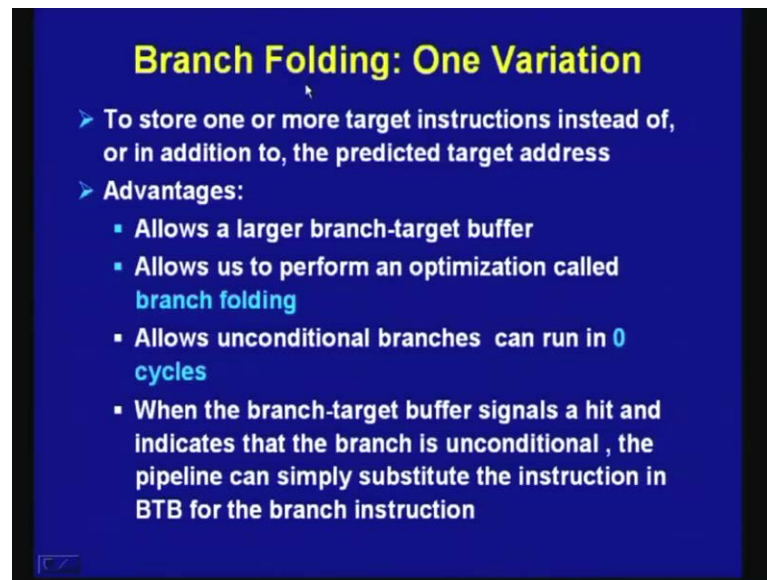
(Refer Slide Time: 46:32)



Let us see, what is the simulation results on misprediction rates for different sizes of returns address carried out on SPEC CPU95 bench marks. So, we find that as the number of entries in the return address stack is increased, the misprediction rate falls significantly. So, we find that for different bench marks program, the misprediction rates falls sharply as the number of entries in the return stack is increase. But you can see you do not really require a very large stack size. Only by using a stack of 16 entries your misprediction rate is significantly lower except for, I mean except for 1 1 particular application program that is your l i. For this particular benchmark it is not 100 percent prediction; I mean prediction is misprediction rate is 0 for all other application except for only 1 application program; that is your l i. Where the misprediction rate is about 2.5 percent even with this a return stack size 16 entry. If you increase the size of the returns stack for this application also it may be becomes 0.

But these particular simulation result clearly demonstrates the usefulness of the returns stack and the size is not be very high; that is also clear from this particular simulation.

(Refer Slide Time: 48:24)

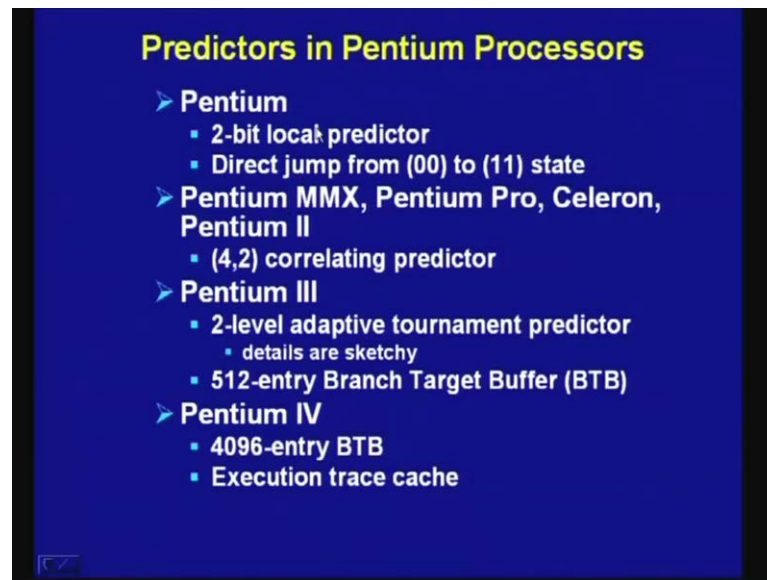


Now, the last topic related to this is known as branch folding. Branch folding can be considered as a variation of the techniques that you have discussed. Here what we have done? So, far we are in a branch target branch prediction buffer, you are storing the predictions 1 bit or two bit. In branch target buffer you are storing the target addresses. And, then from the target address you have to fetch the instruction. Instead of that find why not store the instruction itself; instead of storing be branch target address why not store the instruction itself. And, that is the basic idea of this branch folding to store 1 or more target instructions instead of or in addition to the predicted target address.

The advantage is it allows a larger branch target buffer and it allows us to perform an optimization called branch folding. So, essentially branch folding is this particular technique is known as branch folding. Because it is allows unconditional branches and can run in 0 cycles; that means, normally unconditional branches do not run with 0 cycles. But using this branch folding technique, even this unconditional branches will run with penalty of 0 cycles.

How it is done? When the branch target buffer signals a hit and indicates that branch is unconditional, the pipeline can simply substitute the instruction in the branch target buffer for the branch instruction. So, in this case, this is how your able to achieve 0 cycle; I mean penalty for unconditional branches. So, this is known as branch folding.

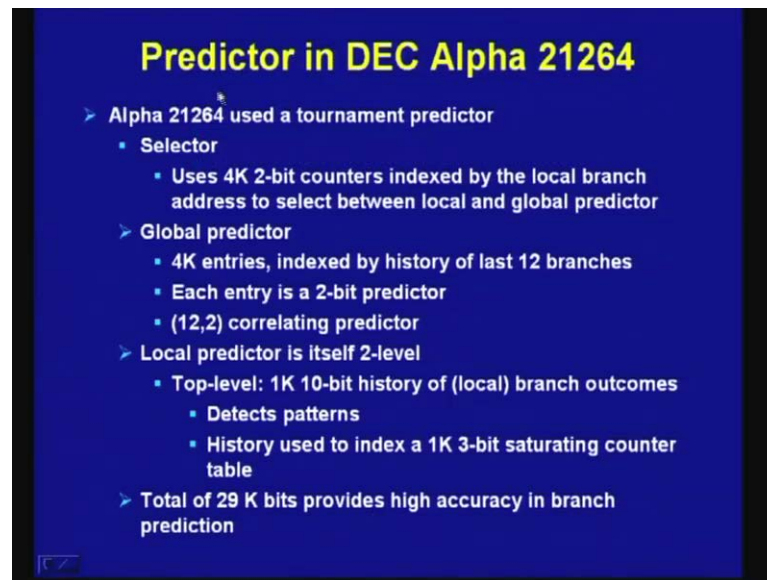
(Refer Slide Time: 50:24)



Now, let us consider how the, how different types of predictor have been used in different processor. First we shall consider the case of Pentium processors; you can see the Pentium 2 bit local predictor direct jump from 0 0 to 1 1 state. So, simple 2 bit predictor is being used, but there is some difference with the predictor that you have discussed; that saturating counter predictor here from 0 0 to 1 1 jump is taking place. So, it is little different from the saturating counter 2 bit predictor that we have discussed. Then, for Pentium MMX Pentium pro and Celeron; Pentium II Celeron and Pentium two. They use (4, 2) correlating predictor. That co-relating predictor where you have got global and local predictors; so, that means, there are 4 global predictor and 2 bit predictor has been used.

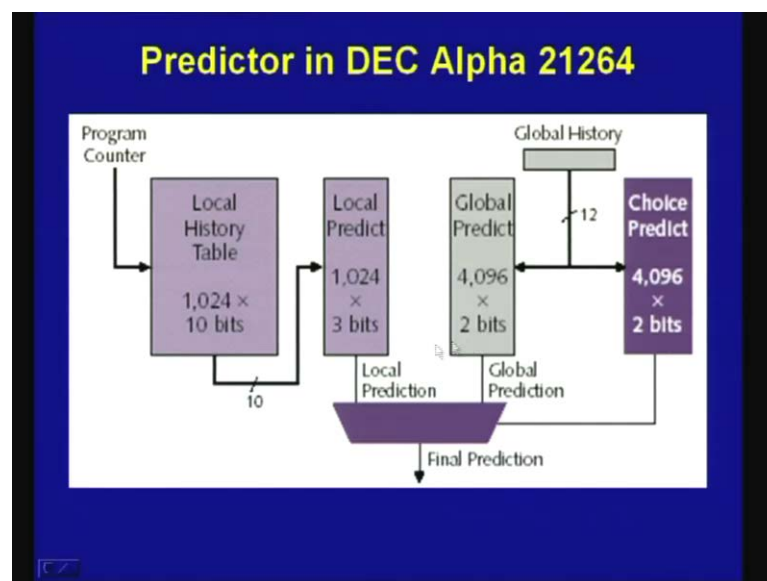
So, you can have 2 to the power 4 different tables you can say; out of 2 to the power of 4, 16 different tables, you have two take 1 of them from the using the lower bits of the address and then the 2 bits is being used for the purpose of prediction. And, Pentium III uses two level adaptive tournament predictor, but unfortunately the details given for Pentium III is very sketchy and so far as the branch target buffer is concerned, it uses 512 entry branch target buffer. On the other hand Pentium four uses for 4096 entry branch target buffer. And, it also uses execution trace cache; later on I shall discuss about it. So, far we have not discussed about this execution trace cache; later on when we shall discuss about cache memories I shall discuss about it.

(Refer Slide Time: 52:26)



Coming to the predictor in DEC Alpha 21264, DEC Alpha sheet where the predictor is the most sophisticated one which has used tournament predictor. As we have seen a tournament predictor requires three components; selector, global predictor and local predictor. So, and the selector will select between a select either local predictor or global predictor. So, three components are there. So, the selector is it uses a 4K 2 bit counters indexed by the local branch address to select between local and global predictor as it is shown in this particular diagram.

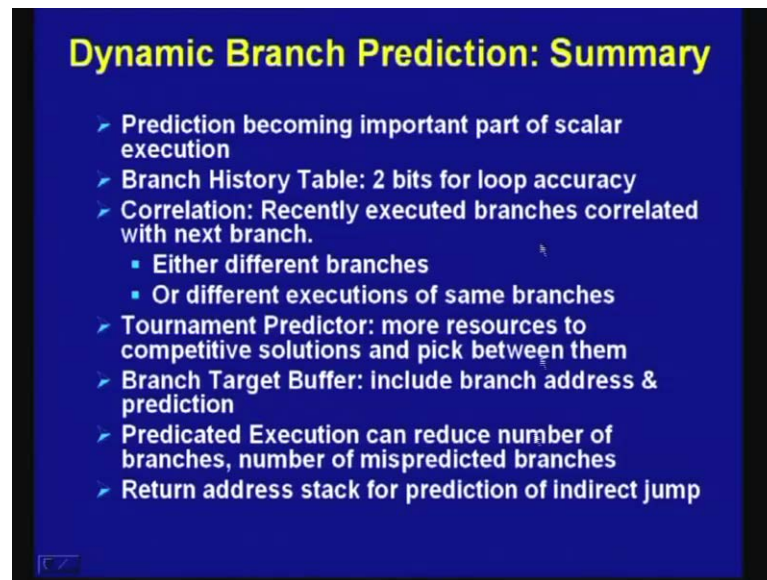
(Refer Slide Time: 53:05)



So, this is the selector, global history which is being used choice predict. So, this is the selector that is being used for the purpose of prediction. Then, you have got global predictor, you have got 4K entries indexed by the history of last 12 branches; each entry is a two bit predictor. So, global predictor uses in 4K entries in the global predictor and each entry is a two bit predictor. And, (12, 2) 4K means you require 12 bit (12, 2) correlating predictor used as a global predictor. So, as it is shown global predictor, where you have got 4K into two bits. This is your global prediction using the global history of 12 bits that is being done. So, this is the global prediction, this is the selection by using the global 12 bits global predictor. And, now let us come to the local predictor. The local predicted is itself 2 level; top level is 1K 10 bit history of the local branch outcomes.

So, it and it detects the patterns. Pattern means you know 10 bit if it all the are taken the bits sequence is 1 1 1 1 1; all 1. If it is alternate bits sequence is 1 0 1 0 1 0 and if it are all are untaken, then it is all 0; that kind of sequences are taken. That is 10 bit history is used and history used to index a 1 k 3 bit saturating counter table and total of 29 k bits provides high accuracy in the branch prediction. And, this is the local history table; as we can see, this is the local history table. Then, the 10 bit is used to for the purpose of local prediction and 1 k into 3 bits. So, this local prediction either local prediction or global prediction is used and that is decided by the selector. So, it will use one of the two for the final prediction. So, this is the predictor used in the DEC Alpha sheet and possibly the most. Sophisticated one used so far in different processor. And, in modern processor some are similar type of predictors are used.

(Refer Slide Time: 55:32)

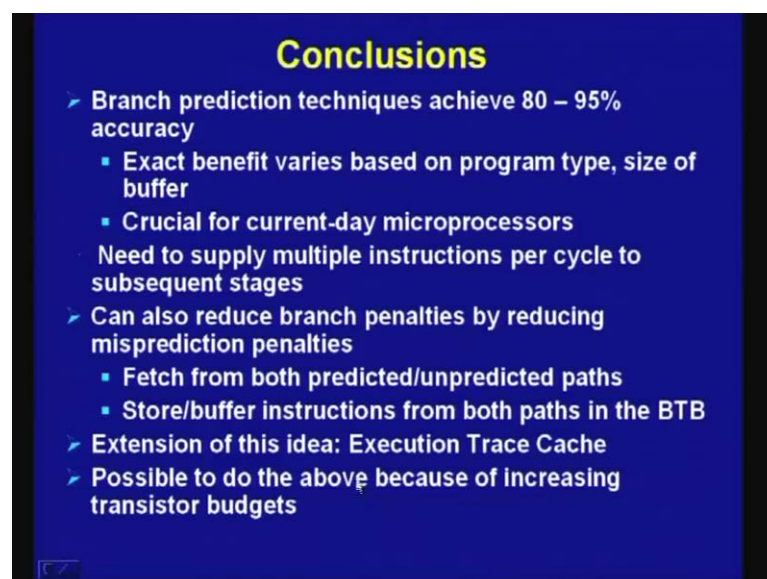


### Dynamic Branch Prediction: Summary

- Prediction becoming important part of scalar execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches
  - Or different executions of same branches
- Tournament Predictor: more resources to competitive solutions and pick between them
- Branch Target Buffer: include branch address & prediction
- Predicated Execution can reduce number of branches, number of mispredicted branches
- Return address stack for prediction of indirect jump

So, you can summarize; now, prediction is becoming important part of scalar execution. So, branch history table 2 bits for loop accuracy, co-relation recently executed branches or correlated with the branch; either different branches or different executions of the same branches. And, tournament predictor is used where which requires more resources to competitive solutions and pick between them. And, branch target buffer includes branch addresses and prediction and predicted execution can reduce number of branches and number of unpredicted mispredicted branches. And, return address stack for prediction of indirect jump that I have already discussed.

(Refer Slide Time: 56:21)



### Conclusions

- Branch prediction techniques achieve 80 – 95% accuracy
  - Exact benefit varies based on program type, size of buffer
  - Crucial for current-day microprocessors
- Need to supply multiple instructions per cycle to subsequent stages
- Can also reduce branch penalties by reducing misprediction penalties
  - Fetch from both predicted/unpredicted paths
  - Store/buffer instructions from both paths in the BTB
- Extension of this idea: Execution Trace Cache
- Possible to do the above because of increasing transistor budgets

And, here we can conclude branch prediction techniques achieve 80 to 95 percent accuracy and exact benefit varies based on the program type size of buffer. And, is crucial for current day processor; because nowadays we are using superscalar architecture, where multiple instructions are issued. And, so, branch prediction is a extremely essential. Because need to supply multiple instructions per cycle in the subsequent stages based on superscalar architecture. And, it can also reduce branch penalties by reducing misprediction penalties. And, fetch from both predicted and unpredicted branches and store or buffer instructions from both paths in the BT.

And, later on we shall see extension of this idea is execution trace cache, you may be asking how all these are possible? The reason for that is we have seen that (Refer Time: 57:27) law as is providing us, large number of transistor this dimensions of these transistor increasing; you can put more and more transistor on chips. And, that is helped us to use sophisticated branch predictor in the processors. So, with this we have come to the end of todays lecture.

Thank you.