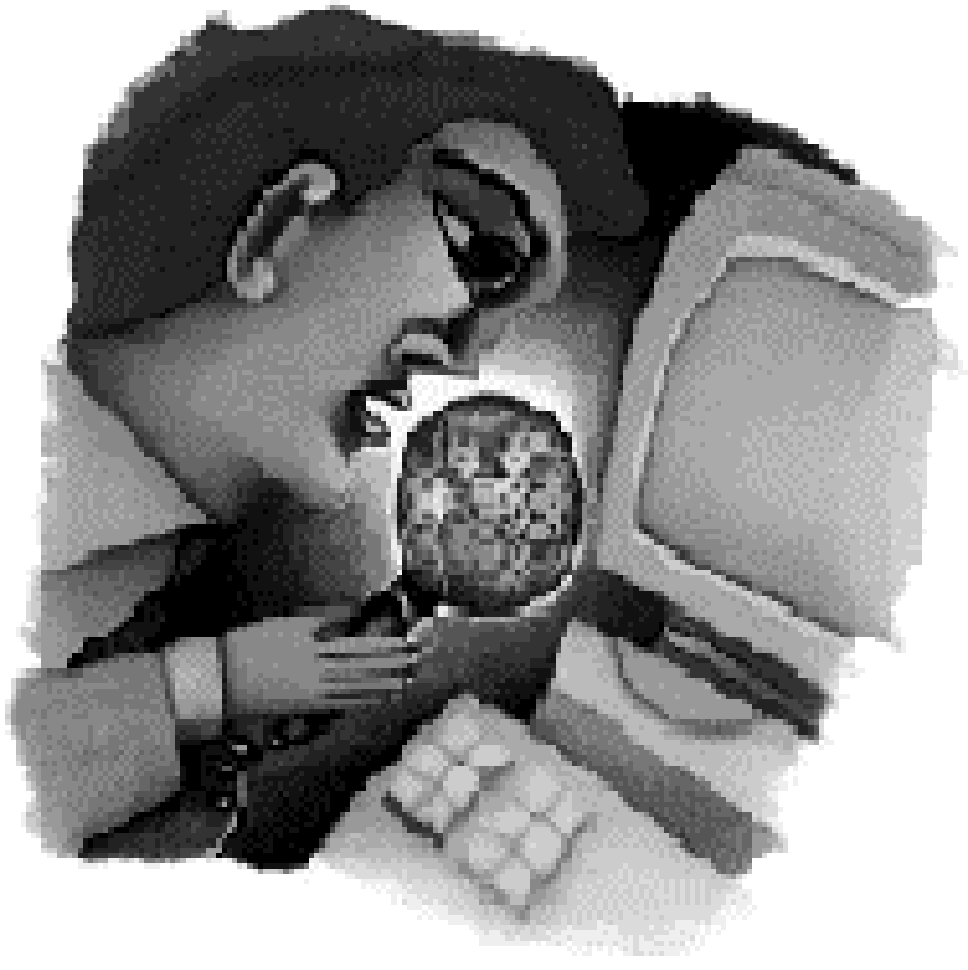


Branch Prediction techniques

Introduction

- Programs are not linear sequences of code but are full of branch instructions.
- The effectiveness of the pipeline, in terms of time and efficiency, is significantly reduced by the branch instructions, called branch instructions, present in the programs.

Jump instructions (I)

- The jump instructions are divided into:
  - Conditional jump instructions: they are a example of an if-then-else statement, i.e. they trigger the jump only if certain conditions occur.
  - Unconditional branch instructions: esse they make the computer finish the sequential acquisition of instructions and make it go to a predetermined address.

Jump Instructions (II)

- Cyclic instructions: they decrement an iteration counter and return to the beginning of the loop if the counter is not yet 0 (i.e. if there are still iterations to do); they can be considered “special” conditional instructions because they know in advance that they will almost always be carried out.

Terminology

- **Branch Target**: it represents the address to which the branch instruction jumps i.e. the jump address.
- **Branch Taken**:it is used to say when the transfer occurs towards the target branch, i.e. when the jump is taken.
- **Branch Not Taken**:when the next statement is executed.
- **Branch Penalty**:represents the cycle loss that the jump caused in the pipeline.

Problems

- If the branch if condition is verified, the pipeline is emptied of the instructions loaded after the appearance of the branch instruction.
- Cycles lost for each branch instruction (high branch penalty).
- Loss of performance.

- Predicting the result of a branch instruction and then pre-fetching data from the expected target branch without waiting for the system to decode the branch instruction and produce its actual result can minimize efficiency issues.

What is Branch prediction?

- It is a method of improving performance by anticipating the results of branch instructions. This method tries to predict the branch, whether Taken or Not Taken, in advance so that the instruction present at the target branch can be loaded without the need to introduce delays (or stalls) in the pipeline

Branch Prediction (I)

- This method has become indispensable in today's era of parallel computers and superscalar architectures in which speed represents the fundamental parameter. And it turns out to be seriously hampered by problems such as branch instructions and cache misses.

Branch Prediction (II)

- The former worsen the speed of the system since the processor must not only find the new address given by the branch instruction but also load the new instruction. They represent a real bottleneck for the pipeline.

Branch Prediction (III)

- The outcome of this prediction depends on the probability that it is correct. In fact it turns out that if the probability that the prediction is correct is high, no stalls will be necessary in the pipeline until the result of the jump is known. If this is low, however, there is an additional cost due to the necessary emptying of the pipeline.

Misprediction Penalty:

- It is the penalty that is incurred in case of bad prediction. It can be minimized through two techniques:

  - Delayed Branches (Branch delay slot): **the instruction or some instructions that follow the jump instruction are always executed**

  - BTB (Branch Target Buffer):**it's a small cache memory used for storing jump addresses.**

Forecasting techniques

- A possible technique could be to load both subsequent instructions in a conditional branch but this is too slow and wasteful.

- Forecasting techniques can be divided into two groups:
  - STATIC: the forecast does not change during the execution of the program, it is fixed a priori.
  - DYNAMICS: they change during program execution.

Static Branch Prediction (I)

- Prediction that all jumps are always Taken:

  All branch instructions are assumed to be addressed to the target branch. The success rate of predictions is 50% and mainly depends on the type of program (if it contains many unconditional jumps it gets worse).

Static Branch Prediction (II)

- Prediction that all jumps are always Not Taken:

The next instruction is always executed to that of jumping. Being the complement of the first technique, its success rate is also 50%. The efficiency of this technique worsens in programs with a high content of loops and forward jumps.

Static Branch Prediction (III)

- Prediction that only branch instructions with certain operation codes are Taken while the others are Not Taken:

  The op codes are examined first and depending on their value some branch instructions are either Taken or not. However, it is a static method since the criterion for choosing the op codes is not changed over time. This method is justified by the fact that some instructions are more intended to be branches than others.

Static Branch Prediction (IV)

- Forecast That everyone the jump backwards branch are Taken and all forward branch jumps are Not Taken:

  It is based on the intuition that most branch instructions (roughly 85%) are loops.

Static Branch Prediction (V)

- Prediction that the first time a certain branch instruction occurs it is always considered Taken:

  It is a semi-static method as there is some dynamism in keeping track of the first branch. However, it is difficult to use in programs containing a large number of instructions because it is difficult to keep track of the branches.

Dynamic Branch Prediction

They use the information present during the execution phase of the program to make predictions about the direction of the jump. The more the past history of jumps is taken into account, the more the accuracy of the forecast improves. Past history is recorded in small associative caches (BTB) or in a BAC (Branch Address Cache).

Dynamic Branch Prediction (I)

- Keep a table of your most recently used Not Taken jump instructions:

  It is implemented using a simple associative cache that stores the last n Not Taken instructions. When a branch is encountered the BAC is checked, if the address matches the branch instruction is predicted as Not Taken, otherwise as Taken.

Dynamic Branch Prediction (I)

If one of the jump instructions present in the table is currently taken (by default or because a certain condition occurs) then the technique for replacing the latter in the table can be LIFO or FIFO. The success rate in this case depends on the size of the table which if large enough exceeds the accuracy of static methods.

Dynamic Branch Prediction (II)

- Keep one bit for each instruction in the cache. For branch instructions the bit goes to 1 if the instruction is Taken:

The prediction here is based on the latest one execution of the branch. In fact, if the Taken instruction was taken the previous time, then it is expected that it will be taken this time too.

Dynamic Branch Prediction (II)

When a branch instruction is cached for the first time the bit is by default set to 1 (based on the static V technique).

The disadvantage of this technique is that some of the time (a very valuable element) is spent finding and keeping track of branch changes.

Dynamic Branch Prediction (III)

- Compress the address of the jump instruction to m bits and access with these m bits a random access memory containing the bits to keep track of the history of the branches. The prediction is made based on these last bits:

  Only m bits are used for the table index. The history bits indicate the result of the most recent branch instruction connected to that address.

Dynamic Branch Prediction (III)

The default prediction is static and the bit is always set to either 0 or 1. The use of random access memory facilitates the replacement of the present instruction in case of prediction
incorrect. An implementation of this technique could be to replace the history bit with a counter that is incremented if the branch is taken and decremented otherwise. This allows many more past executions to be taken into account. The prediction will be Taken if the counter is positive or 0.

Dynamic Branch Prediction (IV)

- Maintain a counter in random access memory for each instruction with 2 complementary counters. The branch prediction will be Taken if the sign of the access counter bit is 0 otherwise it will be predicted as Not Taken:

  This method is a variation of the third method. The history tracking bit is implemented with two counters complementary.

Dynamic Branch Prediction (IV)

The branch instruction is expected as Taken if the sign of the access bit is 0 otherwise as Not Taken. The counter is incremented every time the branch is taken and decremented if necessary otherwise. When more than one address of a branch instruction is compressed to the same address the counter can be used to determine which address to keep in the memory table. It is more efficient to have 2-bit counters so that you can avoid incorrect predictions with loops.

Dynamic Branch Prediction (V)

- Gshare or Global Share:

  It is a prediction method that uses 2 levels to keep track of the past history of hops. The first level keeps the outcome of the last k hops while the second takes into account the directions and paths related to the last k hops when a particular path occurs at the first level. It combines the address of the jumps with their history in order to determine a table indicating the history of the branch paths.

Dynamic Branch Prediction (V)

The Gshare tries to identify the execution of programs through the use of both of the jumping address and of its past history.
It improves the accuracy of forecasts and allows you to find correlations between priority jumps and secondary jumps in programs through the analysis of branch patterns. It is an important method for those instructions with repetitive paths such as an instruction that jumps every second time it is executed.

Dynamic Branch Prediction (VI)

- Combine groups of forecasting techniques into a single forecasting technique

  prediction.Determination of the best prediction technique for each single hop through a selection mechanism:

  this method uses a BTB to record the input jumps. To implement the selection mechanism of N components i.e. of N prediction techniques, N 2-bit counters are added to each input branch.

Dynamic Branch Prediction (VI)

These counters are used to keep track of the most accurate of the N components for that specific branch instruction. For each new incoming branch the counters are initialized to 3. Every time a prediction must occur, the one generated by the technique that in that instant has the value 3 in the prediction section counter for that instruction in the BTB.

Dynamic Branch Prediction (VI)

There are rules for determining priorities in the event of conflicts, i.e. in the case that for that instruction more than one counter in the BTB has a value of 3.
If the prediction is correct, the counter is increased, otherwise it is decreased.

Dynamic Branch Prediction (VI)

This method allows you to differentiate, for example, methods with correct prediction for the last 5 branch instructions from methods with correct prediction only for the last 4.
Among all the methods presented, this method is the one that has the best accuracy of approximately 97.13%.

Conclusions

- From a certain point of view, static forecasts have better performances because they do not need any past history of the branches and the "warm-up" period typical of dynamic forecasts in which the data are loaded, used to keep the history of the branches, is therefore avoided. jumps ,relative to the first branches. If static techniques were implemented with some past history of the branches the accuracy of these types of techniques would increase

  noticeably.

Conclusions

- As regards dynamic forecasts, it would seem that larger tables correspond to greater accuracy. However, it has been demonstrated that beyond a certain size limit the size of the table does not affect the improvement.

Conclusions

- In conclusion, it appears that dynamic prediction techniques have a greater degree of accuracy than static ones, so much so as to justify the higher cost due to the specialization of the hardware and the complexity of the necessary chips.

Static - Dynamic comparison

Comparison between static Predictors and Dynamic Predictors

	Static Predictors				Dynamic Predictors					
Strategies	Always Taken	Selected opcodes always taken	backward branches always taken	First encounter always taken	Table of most recently used branches	Maintaining to historybit	Hashing the branch address	Maintaining to counter	gshare	Combine multiple g predictors into single predictor
Accuracy	76.7	86.7	80.3	90.4	85.5	90.2	90.4	90.3	96.6	97.13

- Accuracy = number of correct predictions out of the total number of predictions

DYNAMIC BRANCH PREDICTION

Advanced Computer Architectures

academic year 2002/2003

Noacco Andrea

Stancich Dario

CLASSIFICATION – Branching Strategies

Branching strategies can be classified as follows:

1. pipeline freezing strategies (when a jump is found at stage s'' the fetching phases are interrupted until the jump is resolved at stage s)

2. prediction strategies (the probability of success associated with the possibility of jumping can be described through a Carnaugh map with predicates
   take branch/ don't take branch and
   from the situation
   I'm in a branch situation/ I'm not in a branch situation

3. instruction reordering strategies

4. Loading strategies of different paths

5. Code condition strategies

branch prediction influences the work of the pipeline and in particular these stages are important:

stage s: is the stage at which the jump condition is known or resolved
s' stage: is the stage where I actually know the address I need to access s'' stage: is the stage where I find the decoded instruction

CLASSIFICATION – Branching Strategies

Another type of classification that can be introduced is a type more closely linked to design. In fact, there are two systems used by hardware designers for architectures that manage branch instructions:

1. Orthogonal instruction pairs: also called *CC condition cache*, provides two instructions, the first of which sets the condition, the second verifies it by jumping the code or making it continue linearly.

2. Non-orthogonal pairs of instructions: a first instruction carries out the test and verifies it the outcome, the second makes the jump; *tb test and branch*

CLASSIFICATION – Branching Strategies

STATIC

A bit present in the branch instruction opcode allows the compiler to have some influence on the prediction. Static branch prediction means (as opposed to dynamic prediction) that the system cannot dynamically alter the branch prediction. This technique includes machine-fixed prediction (for example: "predict ALWAYS TAKEN") and compiler-driver prediction, i.e. driven by the compiler.

SEMI-STATIC

Similar to static but the presence of flag bits near each branch can be set depending on the results of the branch prediction. This method is not adaptive during code execution. It is used by the POWER PC 601 as static branch prediction; in this particular CPU, if a prediction is missing, the instructions executed in the mirror image are abandoned.

DYNAMIC

This technique, unlike static, does not have a hardwired logic inside the CPU which cannot be modified but uses the information collected at run-time to try to predict whether the jump will be made or not. The more history collected, the greater the degree of accuracy.

CLASSIFICATION – Branching Strategies

			DECODE INTERNSHIP	INSTRUCTIONFETCH STAGE	
				STORE TARGET ADDRESS	STORE TARGET INSTRUCTION
CURRENT INSTRUCTION	HISTORY	UPDATE	PREDICT ON HISTORY OF OUTCOMES	PREDICT ON HISTORY OF OUTCOMES	PREDICT ON HISTORY OF OUTCOMES
		LAST	PREDICT ON LAST OUTCOME	PREDICT ON LAST OUTCOME	PREDICT ON LAST OUTCOME
	UNIFORM HISTORY		PREDICT ON UNIFROM HISTORY	PREDICT ON UNIFROM HISTORY	PREDICT ON UNIFROM HISTORY
	NONUNIFORM HISTORY		PREDICT ON NONUNIFORM HISTORY	PREDICT ON NONUNIFORM HISTORY	PREDICT ON NONUNIFORM HISTORY
	CHARACTERISTICS	BRANCH DIRECTION	PREDICT ON BRANCH DIRECTION	PREDICT ON BRANCH DIRECTION	X
		BRANCH MAGNITUDE	PREDICT ON BRANCH MAGNITUDE	PREDICT ON BRANCH MAGNITUDE	X
		OP-CODE	PREDICT ON OP-CODE	X	X

DYNAMIC BRANCH PREDICTION STRATEGIES

STATIC PREDICTION (I)

It is a simple prediction technique, which always uses a prediction direction determined a priori or which allows the compiler to determine this direction. The direction of a jump prediction never changes. Simple mechanisms**hardware-fixed direction** they can be:

1. **Always not taken**:this is the simplest scheme because the intake is a continuous flow of instructions. Unfortunately due to frequent loops in the codes, this technique is not very effective. It should also not be confused with the delayed branch technique. The instruction in the delay slot is always executed, while in the predict-not-taken technique it executes the instructions and destroys the instruction in case of a prediction error.

2.**Always taken**:here the jumps at the end of the loop iterations are correctly predicted. The branch target address must be stored inside the fetch unit to allow zero delay.

3.**Mixed**:backward jumps predict taken, forward jumps predict not taken : the idea is that jumps with branch target address pointing backwards come from loops and should be predicted as taken, while other types of jumps are preferably not taken taken.

Compiler-based techniques:

Sometimes a bit in the opcode of the jump instructions allows the compiler to decide the direction of the prediction either directly (bit set means predict taken, and bit set predict not taken) or by reversing the direction determined by the hardware. The compiler can use several techniques for good static prediction:

can examine program structure for prediction (jumps at the end of loop iterations should be predicted as taken, and situations where there are if-thens predicted as not taken)
can relegate the prediction to the programmer via compilation directives or

can use profile-based prediction, predicting jump directions based on previous program runs, recording jump behavior.

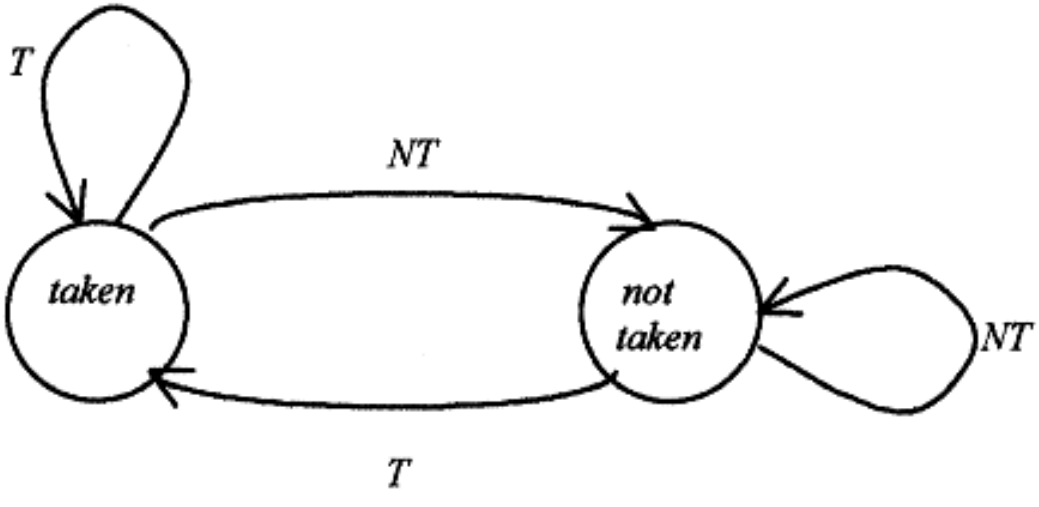
DYNAMIC PREDICTION (I)

They are techniques in which the prediction is decided based on the computational history of the program. After a start-up phase of program execution, in which a static prediction can be efficient, historical information is collected and the dynamic technique becomes highly performing. In general, dynamic techniques give better results than static techniques, but they have to bear the cost of greater hardware complexity.

Techniques:

- One-bit predictor
- 2-bit predictor
- Correlative Predictor (outline)

1 - BIT PREDICTOR



One-bit predictor state machine

Each entry of the BHT (Branch History Table) contains a bit indicating whether the jump was recently taken or less. If the bit is set, the jump is predicted to be taken, otherwise it is predicted to be not taken. If there is an error in the prediction, the state of the bit is flipped and so is the direction of the next one prediction. This car has only two states and only one bit is needed to identify them. Structure considered preserve Therefore memory only of the last one branch executed.

1 - BIT PREDICTOR

In single cycles, everything works correctly for the entire duration of the cycle, until you have to exit.
In nested loops, a one-bit predictor scheme causes two prediction errors for the inner loop: one at the end of the loop, when the iteration exits the loop instead of looping again, and one when executing the first iteration of the loop, when predicts exit instead of continuing within the loop. These types of errors are overcome with the next scheme, the 2-bit one, thus making this system little used.

DYNAMIC PREDICTION (IV)

Techniques compared:

Suppose we have a loop (with 10 iterations) inside another loop:

STATIC(Always taken)

. . . TT NT* TTTTTTTTT NT* T T...

The system fails to predict only the jumps exiting the loop

DYNAMIC (1-bit)

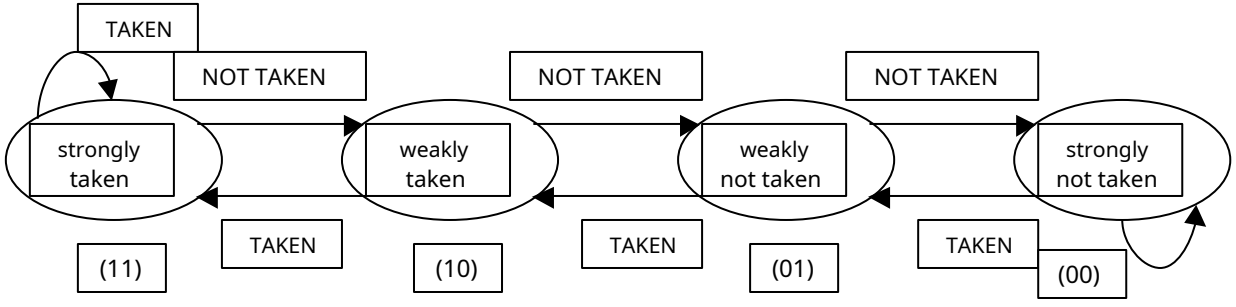
. . . TT NT* T* TTTTTTTT NT* T* T...

The one-bit scheme, however, despite being more sophisticated than the static one, gets both the output branches and the first T of each loop wrong

Using the 2-bit scheme this problem is solved:

. . . TT NT* TTTTTTTTT NT* T T...

2 - BIT PREDICTOR type a



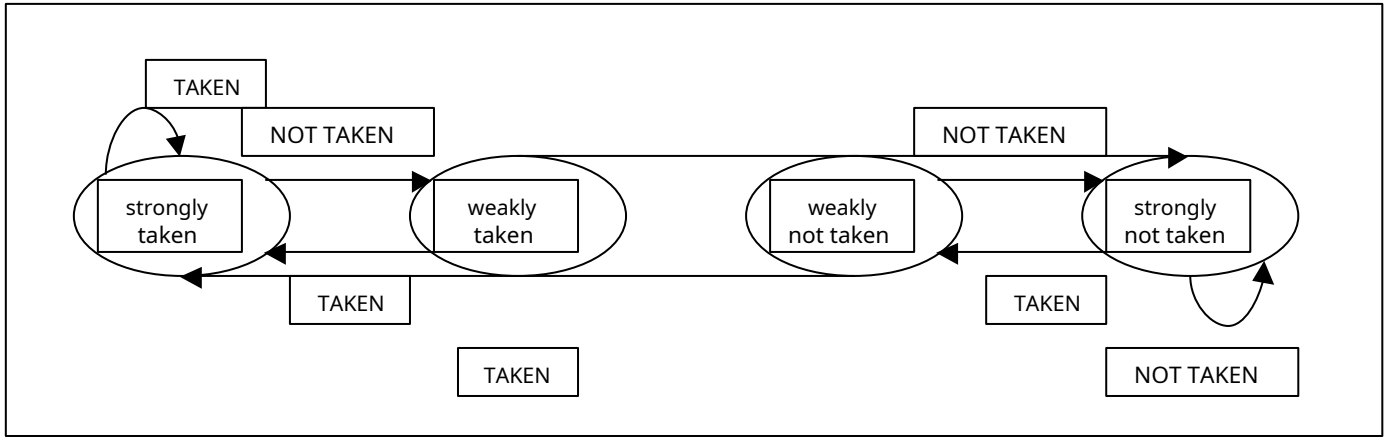
State machine of the 2-bit predictor, the states become 4

In this architecture there are two bits used, and they represent the states strongly-taken, weakly taken, strongly not taken, weakly not taken. In case of prediction errors in the stronglytaken state the prediction direction is not changed but only moved to the weakly state. A prediction must be wrong twice before it is changed.

The fundamental difference between this scheme and the 1-bit one is that here there is a more "prudent" behavior in the sense that before changing state you can see not only what happened in the previous branch but also in the one before it. That is, the results obtained in the two branches preceding the one being examined are kept in memory.

DYNAMIC PREDICTION (VI)

2 - BIT PREDICTOR type b



As before, the counter is incremented at each branch taken event while it is decremented each time the branch is not made. The counter becomes saturated, that is, it cannot be decreased beyond zero and it cannot be increased beyond three. The most significant bit determines the prediction.
The difference compared to the previous one lies in the direct change from the weakly to the strongly state, in the case of a second wrong prediction.

It is used in UltraSparc-I processors. Unpredicted jumps are initialized by this processor in the weakly not taken state.

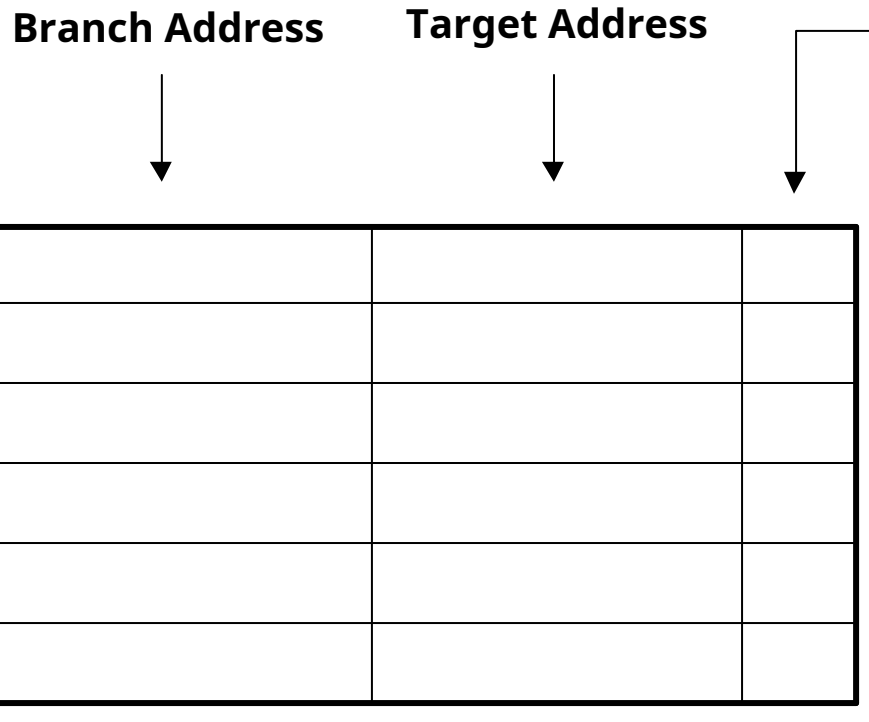
DYNAMIC PREDICTION (VII)

You could have a prediction scheme that adds a bit to each instruction: if this bit is 0 it is assumed that the branch is not taken, if instead it is equal to 1 the branch is assumed to be taken. The negative thing about this technique is that the CPU adds a bit to each instruction but the branch needs all 32 bits so either a bit is removed from the address or a bit must be added to each instruction, but by doing so wastes one bit for non-branch instructions

Solution: put this extra bit in a memory (buffer): this is how the technique was born**branch prediction buffer**.

It is therefore logical that in dynamic techniques structures are needed to store the information with which to process the predictions. An example above all is the HISTORY TABLE. In the simplest cases it can be implemented with the BPB (Branch Prediction Buffer) or for more accurate management with the BTAC (Brach Target Address Cache).

BTAC - Branch Target Address Cache (1)



BTACis a small cache associated with the IF stage of the pipeline and contains:

• The address of the branch instruction that was executed in the past

• The most recent destination addresses for that hop

• The prediction information, i.e. one or more bits depending on the architectural scheme that predict whether the jump will be performed or not

BTAC - Branch Target Address Cache (II)

OPERATION:

1.The IF stage of the pipeline compares the contents of the Program Counter with the addresses of the branch instructions present in the BTAC

2. If a match is found:

Unconditional Jump Case
It reads the most recent destination address related to that hop into the BTAC and then the Program Counter instead of being incremented as usual is updated with the found address. ADVANTAGE: I don't have to wait for advanced stages of the pipeline to know where that instruction will jump but I know it immediately, so I don't have to insert the usual SLOT DELAYS

Conditional Jump Case
It reads the prediction bits and based on their content makes the decision whether the jump will be made or not. If it decides that the jump will be taken, then it will behave like the previous case. If not, continue with the usual execution by increasing the Program Counter.

3. Obviously it can happen that there is a prediction error. It is therefore obvious that when the correct jump address is calculated, the BTAC must be updated with the new prediction and with the respective destination address

DYNAMIC PREDICTION - Performance

The two-bit prediction scheme can be extended to an n-bit scheme. However, studies have shown that a two-bit prediction works as well as an n-bit scheme with n>2. Two-bit predictors work well in floating-point intensive scientific computing programs, which contain many loop-control jumps.

The following example of two hops, one dependent on the other, however, demonstrates that the one-bit and two-bit dynamic prediction technique can still potentially fail every time.

```
if (d==0)    //branch b1
d=1;
if(d==1)     // branch b2
```

If we consider a sequence where d varies between 0 and 2, a sequence of NT-T-NT-T-NT-T will be generated for jumps b1 and b2 as can be seen from the table on the next slide.

initial d	d==0 ?	b1	d before b2	d==1 ?	b2
0	YES	NT	1	YES	NT
2	NO	T	2	NO	T

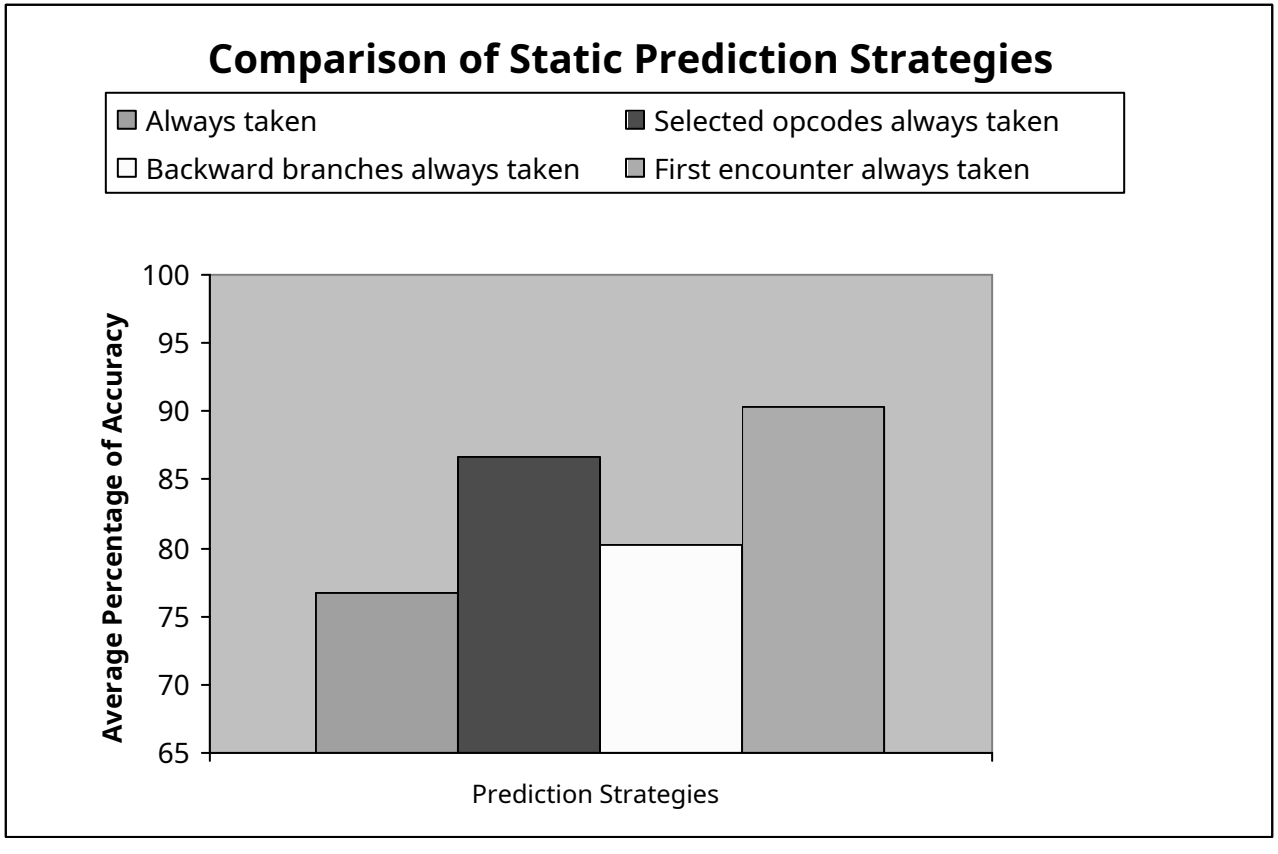
If we apply the one-bit predictor scheme that is initialized to taken for jumps b1 and b2, then every jump prediction will be wrong. The same behavior will occur in the case of the two-bit scheme of figure 2, starting from the predict weakly taken state. In the second type of two-bit predictor, it fails every second execution of jumps b1 and b2. A (1,1)-correlating predictor can take advantage of the correlation between two jumps, making mistakes only in the first iteration, when d=2.

Correlated jump predictors usually achieve higher performance for integer-intensive computation programs than the 2-bit prediction scheme and require only a small increase in hardware cost.

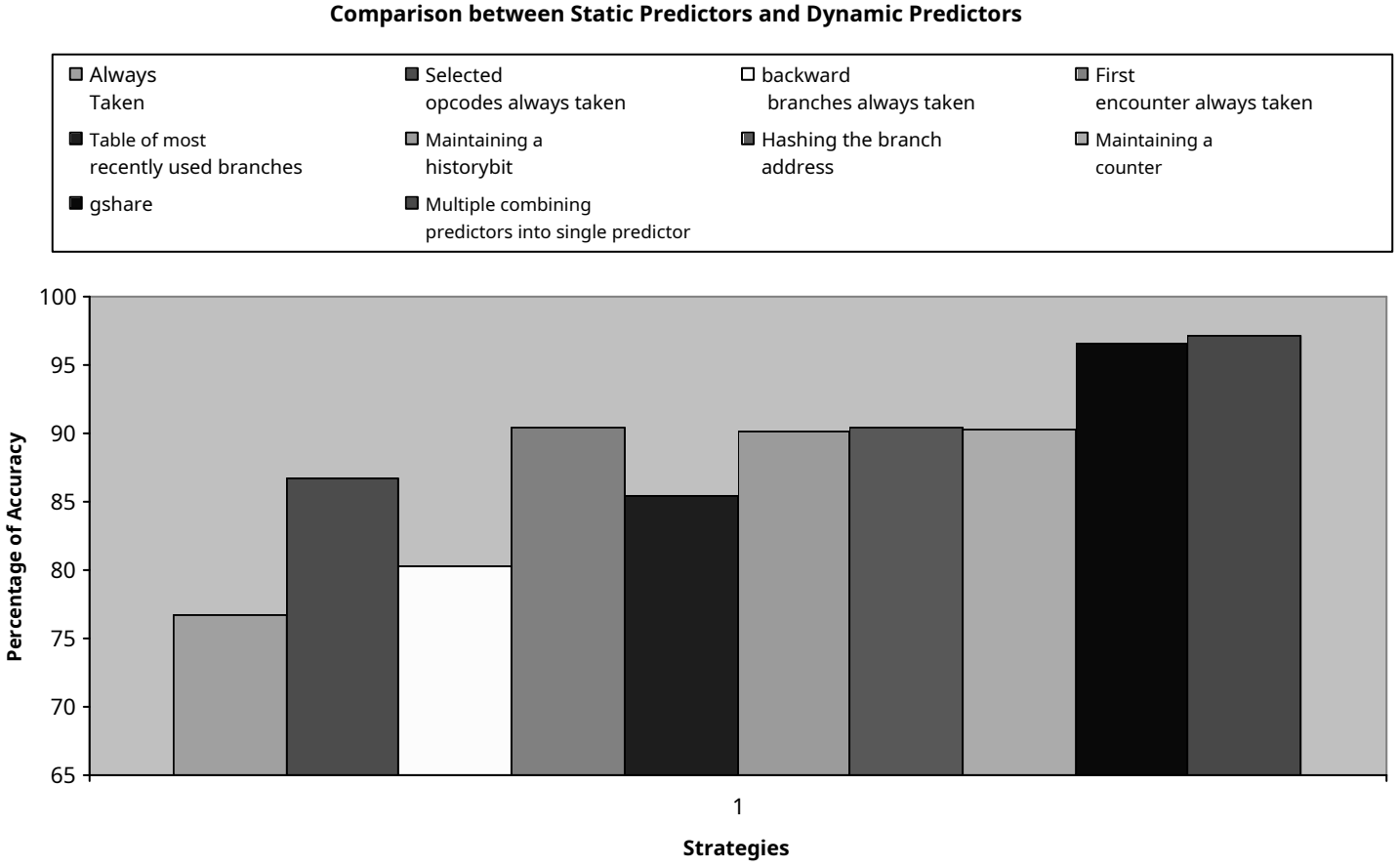
The two-bit prediction technique uses only the recent behavior of a single hop to predict the future of that hop. Correlations between different jump instructions are not taken into account. The techniques so called **correlation-based predictors** or correlating

Predictors are dynamic jump prediction techniques that also use the behavior of other jumps to make a prediction. While 2-bit predictors only use their own history, correlating predictors exploit the history of nearby jump instructions. Many characteristics of integer workloads are characterized by the outcomes of recently executed jumps. In other words the jumps are correlated.

PERFORMANCE – Static Prediction



PERFORMANCE – Dynamic Prediction



DYNAMIC BRANCH PREDICTION Noacco Andrea, Stancich Dario

Branch Prediction Techniques Nicoletta Cossutta