# Assigning Cost to Branches for Speculation Control in Superscalar Processors

by

Farzad Khosrow-khavar
B.A.Sc., University of Victoria, 2003

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Supervisor: Dr. A. Baniasadi


## ABSTRACT

Branch prediction is accepted to be the best technique for speculating the direction of the branches in modern superscalar processors. Several algorithms have been proposed to increase the predication accuracy of the branch prediction unit to reduce the number of mis-predictions. However, mis-prediction is associated with all speculation techniques proposed. When it occurs, the processor has to recover to the state prior to misprediction. All the associated functional units and buffers have to be flushed and the instructions have to be fetched from the correct path. Consequently, mis-prediction degrades the overall performance and increases total power dissipation. It is the aim of this research to apply power optimization techniques for mis-predicted branches, while keeping the overall performance unchanged. In order to do so, we introduce a new metric, which we define as the cost of a branch. The cost associated with a branch is the number of instructions that have to be flushed when mis-prediction occurs. Ultimately, we categorize a mis-predicted branch into "low-cost" and "high-cost" by comparing the number of flushed instructions to a given threshold. We show that the cost associated with a branch is highly predictable based on past history of that particular branch. We also observed that high cost branches are responsible for most of wasted power due to mis-prediction while they only contain a small portion of the mis-predicted branches. In order to reduce the power associated with "high-cost" branches during the speculation phase, we propose a cost predictor in combination with other speculation techniques to distinguish them with other branches. The combined branch predictor is used in many commercial processor because of its high accuracy. We show that the cost predictor can be used to reduce the power associated with the combined branch predictor by 20% with no performance loss in most of our benchmarks. Pipeline gating is a technique that uses confidence estimation to reduce fetching those instructions that are more likely to be mispredicted (gating). However, this technique is not useful in terms of reducing the total power dissipation, when it is applied frequently. We illustrate that the combination of our cost predictor with this technique reduces frequency of gratings by 45%, while achieving the same performance degradation and power waste due to mis-prediction.



Supervisor: Dr. A. Baniasadi, (Department of Electrical and Computer Engineering)

# Table of Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **BHR** | Branch History Register |
| **BTB** | Branch Target Buffer |
| **GCHR** | Global Cost History Register |
| **GHR** | Global History Register |
| **ILP** | Instruction Level Parallelism |
| **LCHC** | Low Confidence High Cost |
| **MDC** | Miss Distance Counter |
| **PC** | Program Counter |
| **PG** | Pipeline Gating |
| **ROB** | Re-order Buffer |
| **RS** | Reservation Station |
| **SPEC** | Standard Performance Evaluation Corporation |
| **VLIW** | Very Long Instruction Word |
| **WB** | Write-back |

# Acknowledgment

First and foremost, I would like to thank my supervisor, Professor Amirali Baniasadi, for his invaluable guidance, assistance and time. His wide knowledge and his logical way of thinking have been of great value for me. His understanding, encouraging and personal guidance have provided a good basis for the present thesis.

I also would like to thank the committee member who read carefully this thesis and provided great feedback. Especially, my deepest gratitude to Dr. M. Serra for her great patience and guidance. I also want to thank her for the direct study course that gave me a deeper understanding about embedded systems design which I will always carry with me in the rest of my careerer. I also want to thank Dr. M. Sima for his help during my graduate studies in University of Victoria. His continuous help and support is something that I will never forget.

I also want to thank all my instructors during the last two years: Dr. J. Muzio and Dr. N. Dimopolous.

I want to thank all my friends in the department of Computer and Electrical Engineering: Azarin Jazayeri, Maryam Mizani, Katayoun Farrahi and Ehsan Atoofian. I also want to thank two of my greatest friends who always supported me and were there for me: Garry Vinje and Boobacar Diallo.

I want to thank my great sister and brother, Farnaz and Farzin, for all their support and compassion. They are truly the best siblings one can have. My thanks to my fantastic parents, Faramarz and Narges, whom are my inspiration and idol of my life and their continuous support throughout my life has made this work possible.

# Chapter 1 - Introduction

In late 1980's, two new micro-architectural techniques were proposed to increase the efficiency of the processors: superscalar and VLIW (Very Long Instruction Word). These processors have multiple execution units and have the ability to execute multiple operations simultaneously. However, the techniques used to achieve high performance are different. VLIW processors use compilers for scheduling the instructions while superscalar processor use hardware scheduling at run-time [29]. In this research, we concentrate on superscalar processors.

Out-of-order execution and speculation control are two of the main characteristics of the modern superscalar processors [1]. In such processors, the classic view of in-order fetch, decode and execute model of Van Neumann has been altered so that processors fetch, decode and dispatch instructions in-order while the issuing instructions to functional units is done in an out-of-order manner. Furthermore, modern processors utilize speculation control (branch prediction) in order to improve instruction level parallelism, or ILP in short.

Aggressive speculation techniques can exploit wide issue superscalar processors.
As the issue width of the superscalar processors increases, designers are facing more difficulties to enable high clock frequency and to master silicon area and power consumption [30]. There has been much prior research concentrating on saving power both from architectural and circuit level for such processors. While circuit level implementation has a big impact on power saving, the aim of this research is to concentrate on architectural and algorithmic ways of diminishing power consumption while keeping the performance unchanged, which is the ultimate goal of "power-aware" architectural design.

There has been numerous attempts to utilize micro-architectural techniques to reduce the power in a processor. Srilatha Manne et al introduced the concept of pipeline gating which is used to improve the power based on the concept of confidence estimation of branches [7]. In [12], a just-in-time instruction delivery mechanism is used to decrease the number of in-flight instructions in the processor. In [13], extra hardware is used to estimate power among the main sources of the power consumption in the processors. On a periodic basis, the processor picks the best configuration for the best power-performance configuration. While these methods use some smart mechanism to reduce power, they don't target the prediction of those mis-predicted instructions that are more responsible for waste of power during mis-prediction. When the speculation is incorrect, the processor has to flush all the buffers and functional units associated with the mis-predicted branch. Our goal in this research is to reduce the wasted power due to mis-prediction. This is done by assigning a cost to each branch. We define cost as the number of instructions that are flushed when mis-prediction occurs.

There are two classes of algorithms that are used in architectural design: probabilistic (predictive) and deterministic. Deterministic algorithms need only a few cycle in advance to select the optimization technique [14]. In contrast, the decision making of probabilistic methods are based on the history or pattern of occurrence of a repeated event. Utilizing a probabilistic approach is not a new concept in computer architecture design. Branch prediction, branch target buffer and confidence estimation are examples of such methods. In this research, we propose a new probabilistic approach based on cost prediction of branches. In particular, our contributions are as follows.

- ➔ We differentiate between branches by categorizing them according to their contribution in wasted power during mis-prediction. Ultimately, we use two categories: "low-cost" and "high-cost".

- ➔ We show that the cost associated with a mis-predicted branch is highly predictive. Moreover, high-cost branches are responsible for most of the power waste during mis-

prediction, while they are only a small portion of the mis-predicted branches. As a result, if we can predict and distinguish "high-cost" branches, power optimization techniques can be applied to reduce the wasted power due to mis-prediction. Such techniques don't affect the overall performance because we are targeting only a small portion of branches.

- We introduce three cost predictors and show that the global cost pattern predictor is the best one for a typical set of benchmarks. Such predictor uses the pattern of the last few branches and have a very simple table structure of only 16 entries of 2 bit counters. We expect that the additional circuitry power consumption is negligible due to its simplicity and small table size.

- Pipeline gating is a mechanism that uses confidence estimation to stop fetch those branches that have "low-confidence" or highly probable to be mis-predicted. However, this technique is not useful in terms of reducing the total power dissipation, when it is applied frequently [14]. We show that the global cost pattern predictor can be used in pipeline gating mechanism to reduce the frequency of gating by 45%, while achieving the same performance degradation and power reduction compared to the original pipeline gating mechanism [7].

- We also used the cost associated to a branch to design a new confidence estimator, which reduced the frequency of gating by 25%.

- We show that the global cost pattern predictor can be used to reduce the power of the combined branch predictor by 20% with a negligible performance loss.

The thesis is organized as follows. We begin in Chapter 2 with introduction of some background material on superscalar processors. Furthermore, branch prediction and confidence estimation which are used for speculation control are explained. Chapter 3 explains the simulation tools used for the purpose of this research. In chapter 4, the

concept of cost for speculation control is introduced. Two studies that concentrate on predictability of cost for mis-predicted branches and use of cost for power optimization illustrates the use of cost in superscalar processors. Chapter 5 introduces how different cost predictors are used to improve pipeline (clock) gating in a processor. Furthermore, a confidence estimator that uses cost is also implemented. Chapter 6 uses the best predictor found in chapter 4 to improve the power in a combined branch predictor. Finally, chapter 7 presents the conclusion and some suggestions for future work.

# Chapter 2 - Background

The idea of increasing the throughput (number of instructions / second) of a processor has been the challenge of processor designers for many decades. In the late 1960's, designers realized that they could use the parallelism, in order to achieve this goal. In such systems, multiple instructions exist in different stages of the execution, which is called "pipeline stages" and the process itself is called "pipelining". The analogy of such system is an automobile assembly manufacturing facility, in which every single stage processes different parts of an automobile. In processor architecture terminology, the potential to execute multiple instructions in parallel is referred to "instruction level parallelism" or "ILP" in short. The following diagram explains how pipelining is used to improve ILP and ultimately improve the efficiency of the whole process.

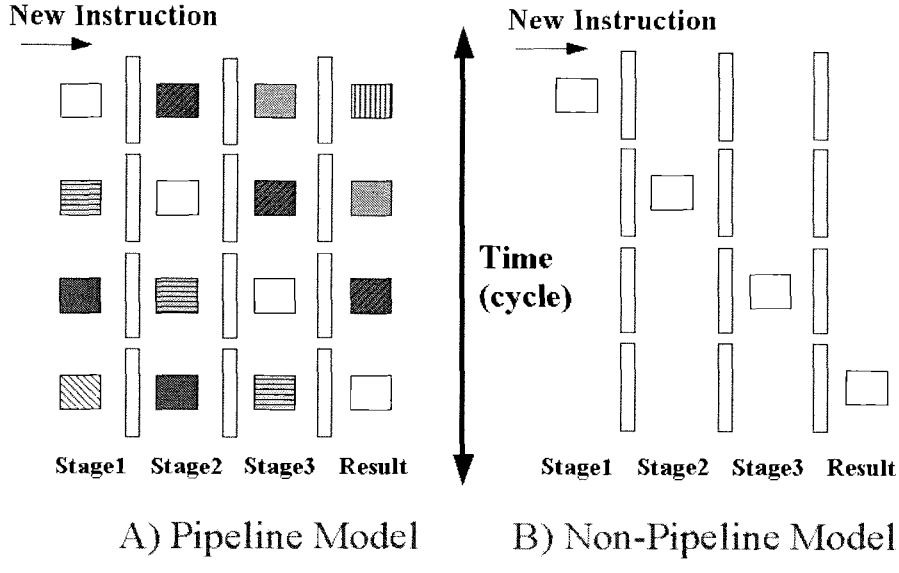



**Figure 2.1 - Pipeline Vs Non-pipeline Model of Execution**

In the non-pipeline model, an instruction goes through different stages of execution one

## 2.1 Architecture of Modern Superscalar Processors

Figure 2.2, which is inspired from the AMD K5 processor [1], is used to present main features of the architecture of superscalar processors.

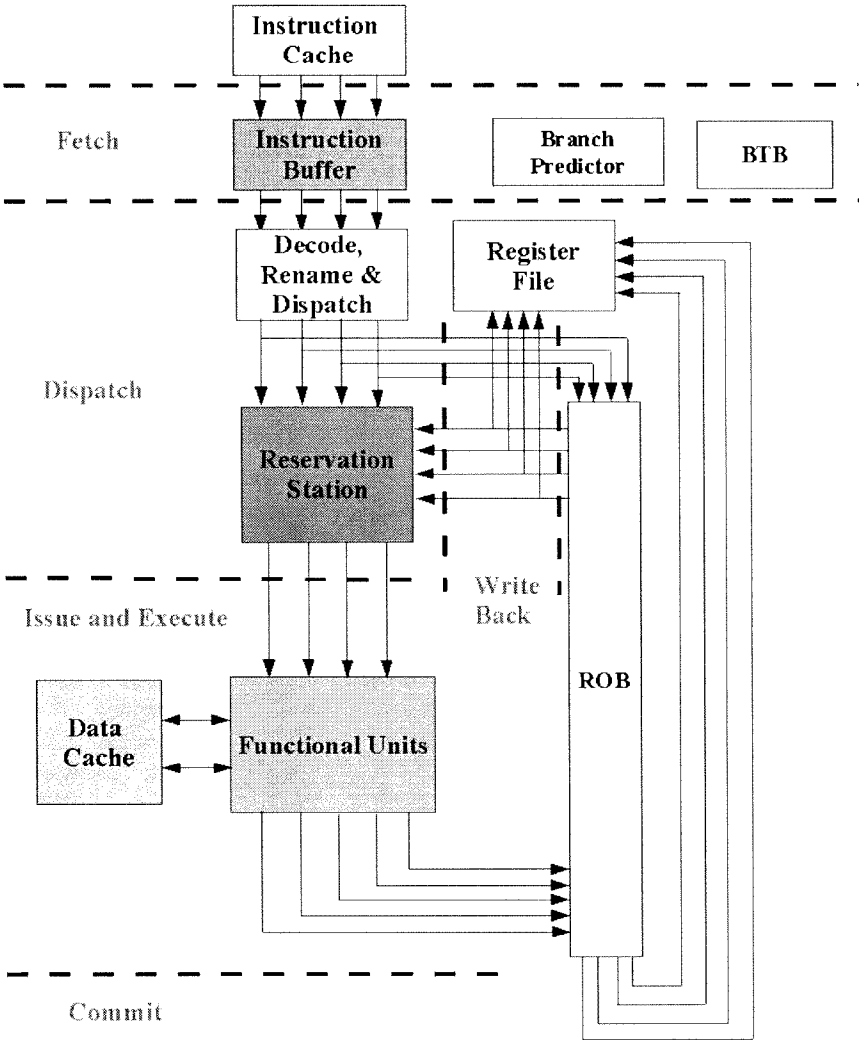



**Figure 2.2 - General Architecture of Superscalar Processor**

Figure 2.2 illustrates five distinct stages for executing an instruction: fetch, dispatch, issue and execute, write-back and commit. In the following sections, all these stages are explained in details.

cycle at time. A new instruction enters the pipeline, when the execution of the previous instruction is finished by the last stage. In the pipeline model, when execution of an instruction is finished by the $n^{th}$ pipeline stage, it will be sent to the next stage, while the instruction from the previous stage is passed to the current stage. Figure 2.1 illustrates the difference between the pipeline and non-pipeline model of execution for an architecture with three distinct stages. For the non-pipeline model, an instruction starts entering the pipeline when the third stage finishes executing the previous instruction. As a result, the efficiency of this model is one instruction every four cycles. In the pipeline model, in every single cycle an instruction is executed. Clearly the pipeline model of execution is by far more efficient and takes advantages of resources in the best manner.

In the early 1990's, instruction level parallelism through pipelining was a method used almost by all processors [1]. However, more efficiency was needed and a whole new set of micro-architectural techniques were started to evolve which revolutionized the architecture of processors. Such processors could execute multiple instructions per cycle and were referred as "superscalar" processors which is explained in details in the following section.

### 2.1.1 Fetch Stage

As it is illustrated in figure 2.2, the fetch unit consists of the following units:

**1) Instruction buffer**

An application begins as a high-level language program, it is then compiled into static binary code. As a static program executes with a specific set of input data, the sequence of instructions form a dynamic instruction stream. During the fetch stage, multiple dynamic instructions are brought from the instruction cache to the instruction buffer. This buffer is used as a "stockpile" so in the case of a cache miss (data is not in the primary cache and has to be brought from lower memory hierarchy), there are still instructions in the buffer that could feed the pipeline. Consequently, the flow of instruction is not interrupted in the case of a cache miss delay [1].

**2) Branch prediction unit**

Often, when a branch is fetched, the data for the branch is not yet available since it is dependent on the previous instructions. The branch predictor is used to determine the direction of such branches based on the history and pattern of that particular branch (speculation control). The architecture of branch predictors are explained in more details in section 2.2.

**3) Branch Target Buffer (BTB)**

In most processors, relative addressing is used to access the memory, which is found by adding some fixed address to an offset value. This is a relatively expensive operation in terms of using hardware resources. As a result, in order to prevent an address calculation for the same branch, the target address could be used by accessing the BTB which holds the target address of the same branch where it was previously executed. This way, the

address should only be calculated one time and it is used in the next consecutive cycles.

### 2.1.2 Dispatch Stage (Decode, Rename and Dispatch Stage)

After the fetch stage, instructions are decoded so that the type of instruction is determined (decode, rename and dispatch in figure 2.2). Some processors use pre-decoded bits which are set by the cache, to make the decoding process faster and simpler. After instructions are decoded, the process of trying to execute them in an out-of-order fashion starts. This is the formal definition of out-of-order execution:

**A superscalar processor executes instructions in terms of their dependency rather than their dynamic order, which improves the overall efficiency by executing instructions in an out-of-order manner [1][16].**

There are two types of dependencies (hazards):

1) True dependencies;
2) Artificial dependencies.

As an illustration, consider an instruction with the following format [3]:
S: R1 + R2 $\rightarrow$ R3
Add register R1 and R2 and put the result into register R3.

The set of input registers R1 and R2 is defined as the domain of S labeled as D(S), the output register R3 is defined as the range of the instruction S, labeled as R(S). The mapping from D(S) to R(S) is depicted by the "$\rightarrow$" sign.
If instruction j is going to be executed after instruction i, the following dependencies could occur :

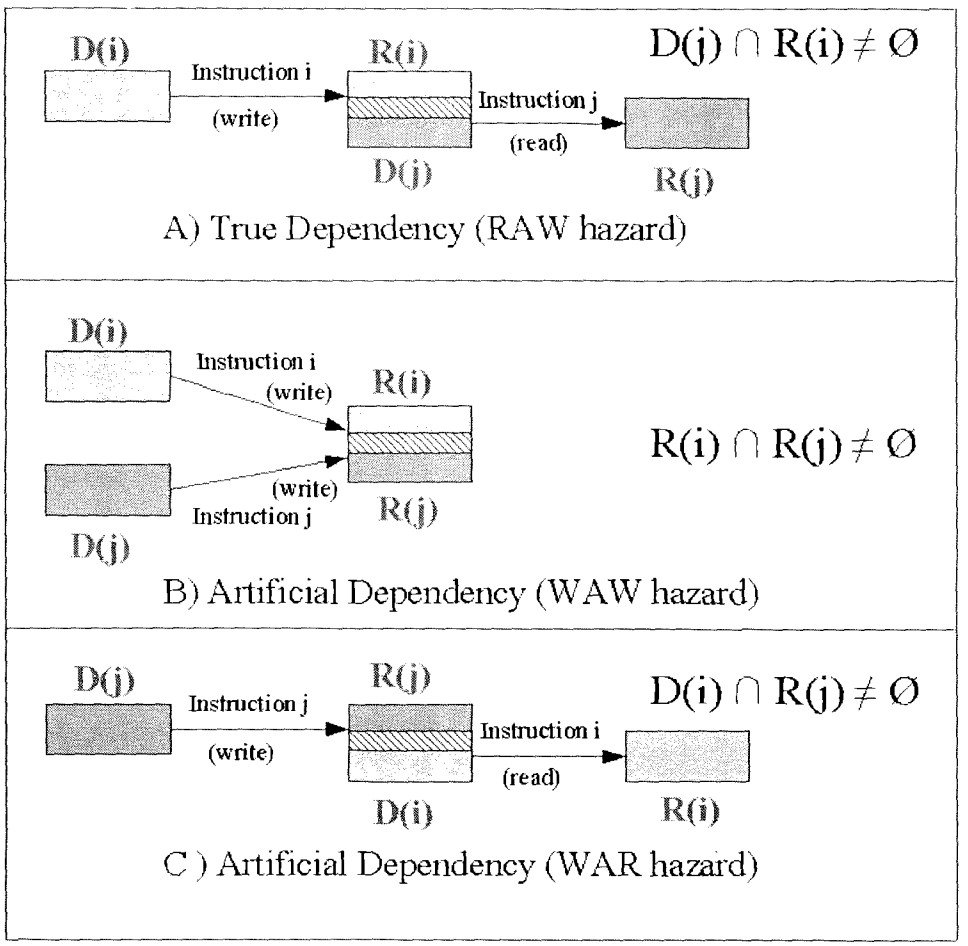



**Figure 2.3 - Dependencies Between Consecutive Instructions**

### 1) RAW Hazard (Read After Write)

It occurs because of the inherent characteristics of dynamic instructions entering the pipeline, and hence, this type of dependencies are referred to as "true dependencies". It occurs when the result of the destination register in the first instruction is not ready for the subsequent instructions and the second instruction wants to read from the same register.

Example :

i: R1 + R2 → R3
j: R3 + R4 → R5

Instruction j can't proceed until the result of R3 for instruction i is determined. In part (A) of figure 2.3, the mapping for instruction i and j results in a common register. This hazard could be prevented if the $D(i) \cap R(j) = \emptyset$.

The second and third kind of dependencies are called "artificial dependencies", since they don't occur because of the inherent characteristic of dynamic instructions and are caused due to limitations and characteristics of hardware resources.

**2) WAW hazards (Write after Write)**

It occurs when two subsequent dynamic instructions have the same output registers.

Example:

i: R1 + R2 → R3
j: R4 + R5 → R3

Both instructions are trying to write into the R3 register at the same time. If the second instruction writes into it first, the outcome could be incorrect. In part (B) of figure 2.3, both instruction i and j want to write into the same registers. This hazard could be prevented if $R(i) \cap R(j) = \emptyset$.

**3) WAR Hazard (Write After Read)**

It occurs when an instruction finishes its execution sooner than the previous dynamic instruction that has a common input register with the current output register.

Example:

i: R3 / R4 → R5
j: R6 + R7 → R3

Division is a much slower operation than addition and as a result the result of instruction j could cause a wrong value to be copied to R3. In part C of figure 2.3, mapping of instruction i and j resulted in a common register. This hazard could be prevented if $D(j) \cap R(i) = \emptyset$.

The rename unit (decode, rename and dispatch in figure 2.2) of the superscalar processor uses internal registers to map the physical register to logical (architectural) registers. Physical registers are visible to the programmer, whereas logical registers are internal to the processor.

**Register renaming unit of a superscalar processor removes the artificial dependencies by mapping the physical destination registers of every instruction into a different architectural register [1][2][16].**

Once the instructions are renamed, the rest of the processor uses the architectural registers. At this stage, instructions are "dispatched" in an in-order manner to reservation stations and the re-order buffer (ROB), which is illustrated in figure 2.2.

Reservation stations are buffers that associated with functional units and keeps the address of the source and destination register (physical) of each instruction. There are different ways to implement reservation stations which is illustrated in the figure 2.4 [2]:

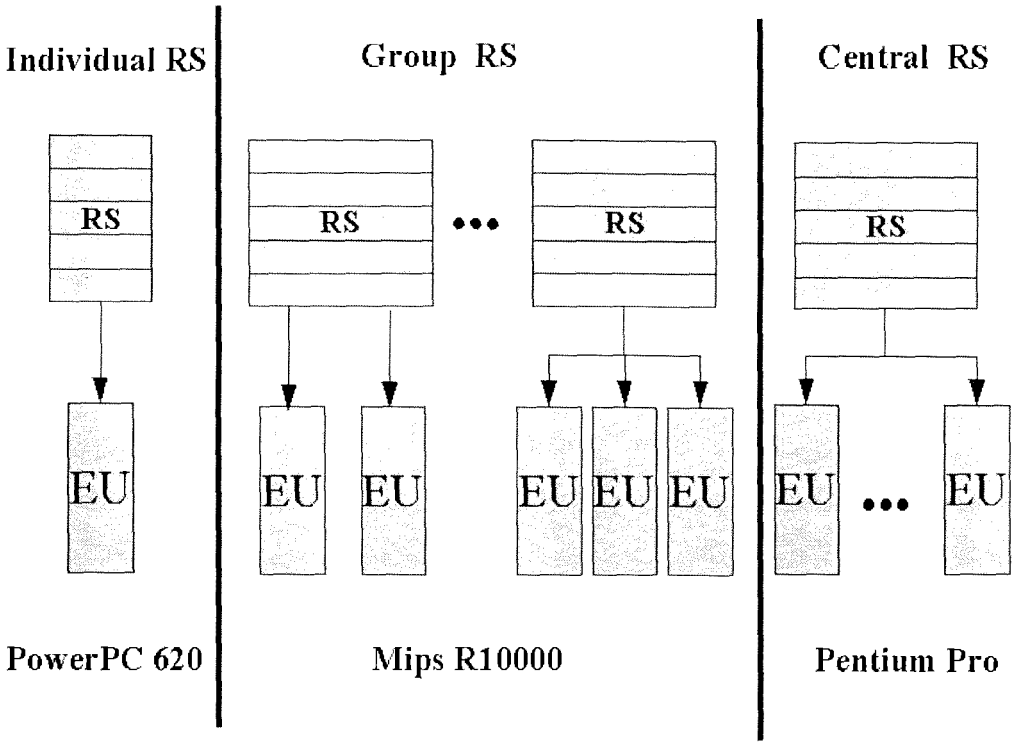



**Figure 2.4 - Different Implementation of the Reservation Station**

In the individual RS (Reservation Station), for every execution (functional) unit, a unique reservation stations is assigned. PowerPC is the example of such implementation. In group reservation stations, a single reservation station is assigned to multiple functional units. For instance, there is a reservation station for the integer unit and another one for the floating unit and so on. Mips R10000 is an example of such processor. In the other extreme, one reservation station is implemented for all the functional units, which is implemented in Pentium Pro processor.

### 2.1.3 "Issue and Execute" Stage

As shown in figure 2.2, once the instructions are dispatched to the reservation stations, the issue logic checks to see if instructions don't have true dependencies to their previous instructions. As soon as there is not dependency and the functional unit is ready, the instructions are "issued" for execution. At this stage, the data cache is also accessed for the load instructions.

### 2.1.3 Write-back Stage

Once the process of execution of an instruction is finished by the functional unit, its source operand (which is associated with an architectural register) is forwarded back to all the reservation stations that depend on it, during the write-back stage (figure 2.2).

It should be noted that there is an additional sets of architectural registers that are used for the instructions that are speculatively executed. After the execution of the branches, the actual direction (taken or not taken) of a branch is determined. At this stage both the branch target buffer (BTB) and branch prediction unit is updated. If the speculation is not correct (mis-prediction), all the buffers should be flushed and the stage of processor before the mis-prediction should be recovered. That is why, all the processors keep a mapping table that saved the information about the state of a processor (architectural registers and son on) for the purpose of speculation control.

Figure 2.5 summarized the dispatch, issue and execute, and write-back stage.

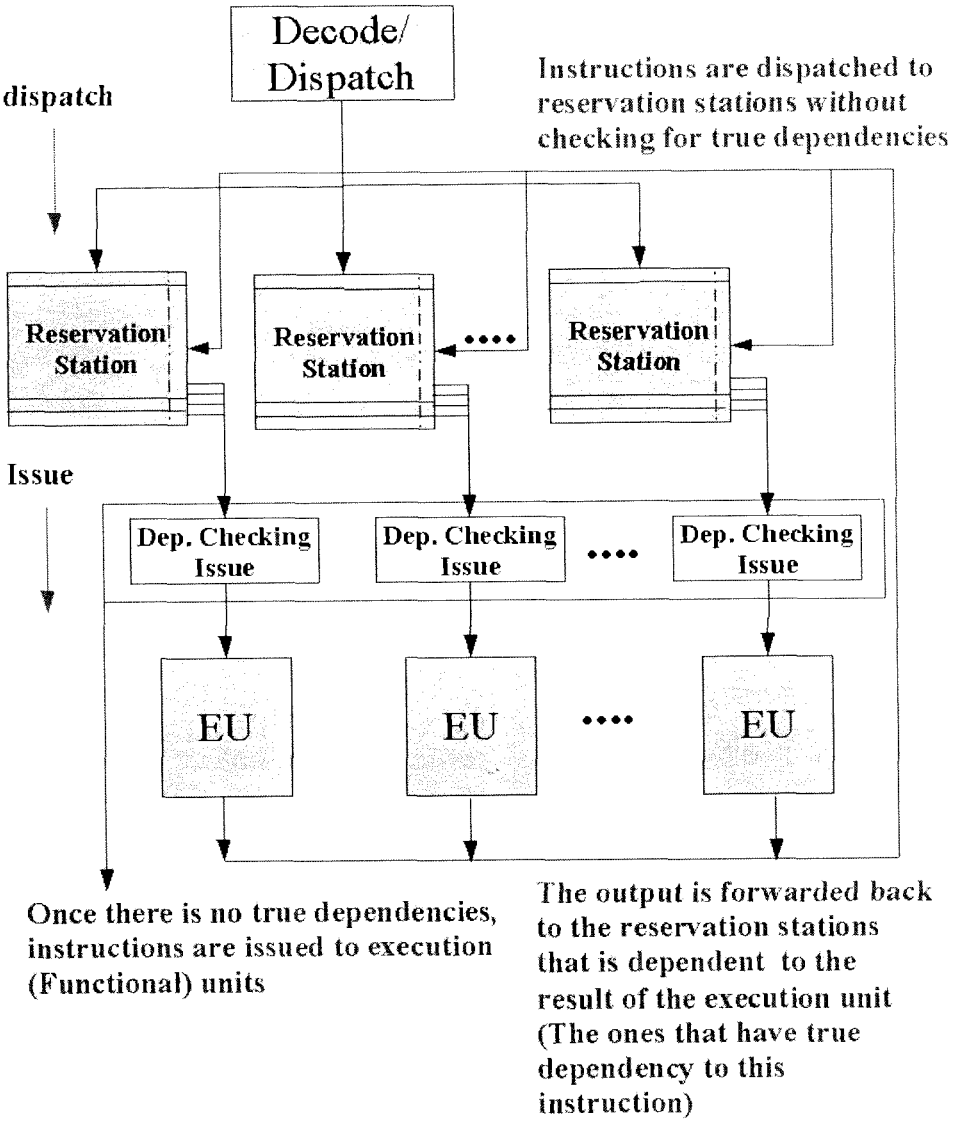



**Figure 2.5 - Dispatch and Issue in Details**

### 2.1.4 Commit Stage (Retire Stage)

Re-order buffer (ROB) is used to assure the sequential consistency of execution in the case of multiple execution unit is executing in parallel (out-of-order execution). Basically, the ROB is a circular buffer with head and tail pointers. The head pointer indicates the location of the next free entry. Instruction are written into the ROB in strict

program order (dynamic order). As instructions are dispatched, a new entry is allocated to each in sequence. Each entry indicates the status of the corresponding instruction: whether the instruction is dispatched, in execution or already finished. The tail pointer marks the instructions which will commit, that is, leave the ROB, next. An instruction is allowed to commit only if it has finished and all previous instructions are committed. This mechanism ensures that instructions commit strictly in-order. Sequential consistency is preserved in that only committed instructions are permitted to commit, that is, to update the program state by writing their result into the referenced architectural (physical) register or memory [2] which can be seen in figure 2.2.

In summary a superscalar processor has the following characteristics:

1) The ability to fetch multiple instructions.
2) The ability to predict direction of branches (speculation control).
3) Techniques that removes the dependencies between instructions and forwarding the result internally for other instructions that need them.
4) Methods of initiating or issuing multiple instructions in an out-of-order fashion.
5) Capability of executing multiple instructions in parallel by utilizing multiple pipelined functional units.
6) Methods for committing the process in an orderly fashion such that the sequential. integrity of the code is maintained.

**Speculation Control and out-of-order execution are two of the most important characteristics of all modern superscalar processors [1][2][16].**

In the following sections, branch prediction and confidence estimation which are the tools used for speculation are described in details.

## 2.2 Saturating Counters

Both branch prediction unit and confidence estimation mechanism are speculation techniques with table structures of saturating counters. Such counters increment/decrement to their maximum/minimum but they don't roll back to their minimum/maximum. For instance, a two bit up/down saturating counter counts upward and downward like this:

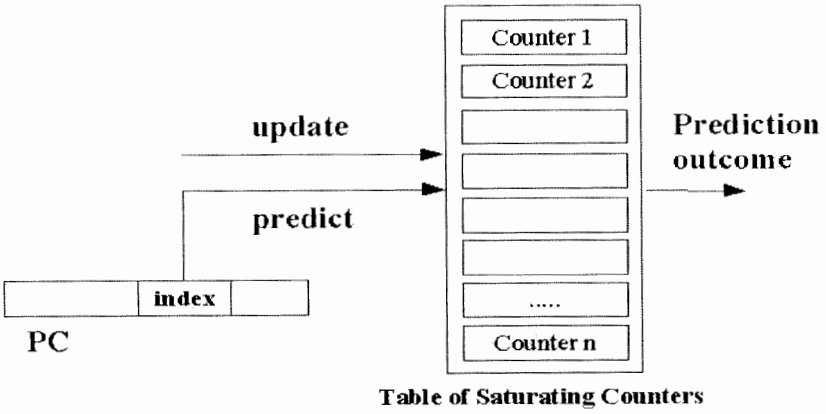
Increment: 00, 01, 10, 11, 11, 11 .... → It doesn't roll back to 00.
Decrement: 11, 10, 01, 00, 00, 00 ... → It doesn't roll back to 11.

The following example shows the example of a 2 bit up/reset saturating counter:

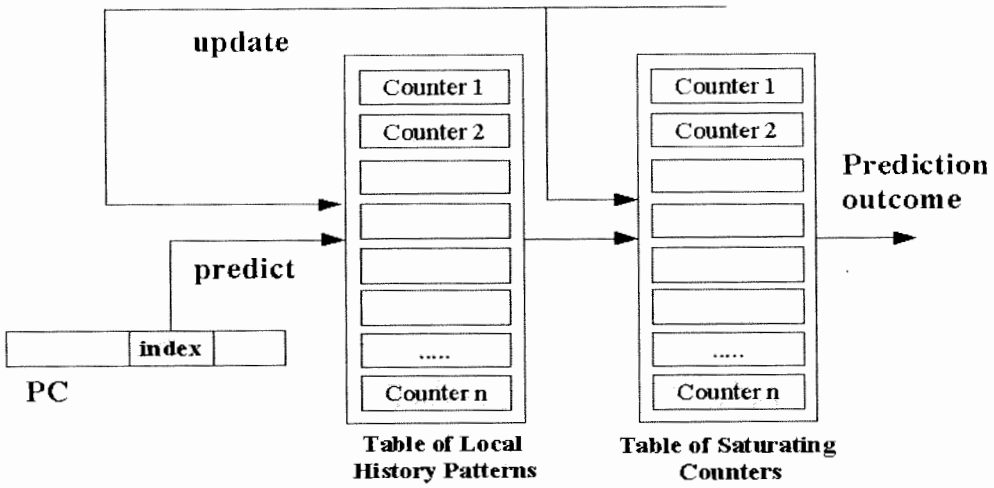
Increment: 00, 01, 10, 11, 11, 11 ... → It doesn't roll back to 00.
Reset: 11, 00 → It is reset to zero.

## 2.3 Branch Prediction

Branch prediction uses a statistical method to find the direction of a branch. As it was mentioned in the last section, the branch predictor is accessed to predict the direction of branches that their result is not completed by the functional units (unresolved branches). In other words, such branches have true dependencies to previous dynamic instructions. Once a branch finishes its execution by the functional unit, its direction is known and the branch prediction table is updated. In general, there are two types of branch predictors: local and global. The local branch predictors keeps track of behavior of branches without considering their global behavior. Figure 2.6 shows the two architectures proposed for the local branch predictors: "bimodal"[4][25][26][27][28] and "two-level local predictors"[4] [23][24]:




Bimodal Branch Predictor




Two-level Local History Branch predictor

**Figure 2.6 - Local Branch Predictors**

As illustrated in figure 2.6, the bimodal branch predictor uses a table of saturating counters, which are accessed by some portion of the program counter (PC), labeled as index. Each counter is incremented if a branch is taken and decremented when the branch is not-taken. This type of the branch predictor works well with usually-taken and usually-not-taken branches. In order to resolve this problem, a "two-level local history branch predictor" is proposed, which consists of two tables. The first table is accessed the same

way as the bimodal branch predictor but the outcome is the history pattern of a given branch, which is used to access the second table. Every entry in the history pattern table is a shift register. The second table has exactly the same structure as the bimodal branch predictor. As a result, the combination of the two tables provides information for a branch that has different patterns in different circumstances. This mechanism works much better for the branches that are not strongly biased toward the taken and not-taken direction. However, the disadvantage associated with this predictor is that the history pattern of the branch should be saved during prediction stage so that the same entry is updated in table of saturating counters when the branch is resolved (the outcome is determined by the functional unit) [4][5][23][24].

The global predictors take advantage of a "global history register" (shift register) that keeps track of the prediction of the last *n* branches. The global predictors work very well for the nested branches and complex programs with regular patterns. The disadvantage of the global predictor is that it has relatively long learning period as well aliasing for the branches that have similar history patterns [4]. Aliasing occurs when two branches with the same PC accesses the same entry in the predictor's table. The simplest type of global branch prediction is depicted in the figure 2.7.

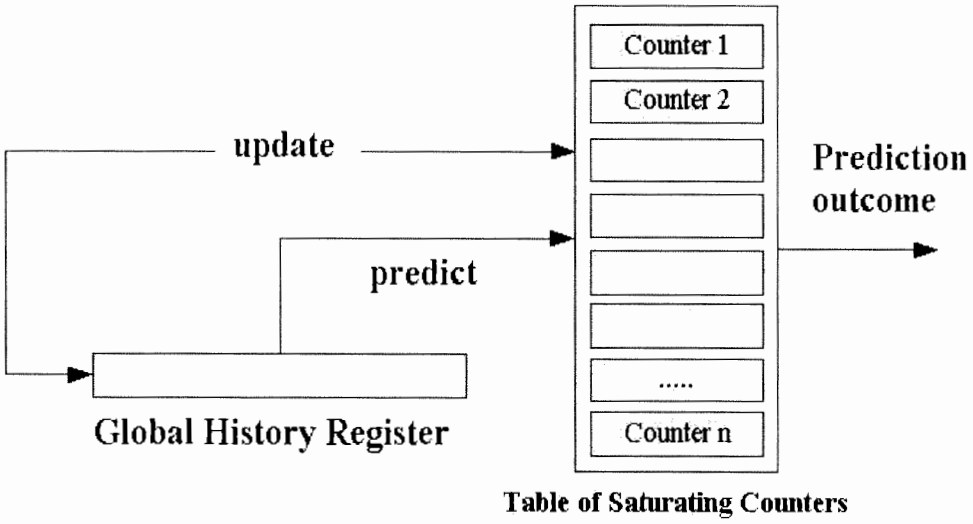



**Figure 2.7 - Simple Global Branch Predictor**

The global history register is used as an index for accessing the table. Once the result of a branch is resolved the global history register is updated by shifting the result into the shift register.

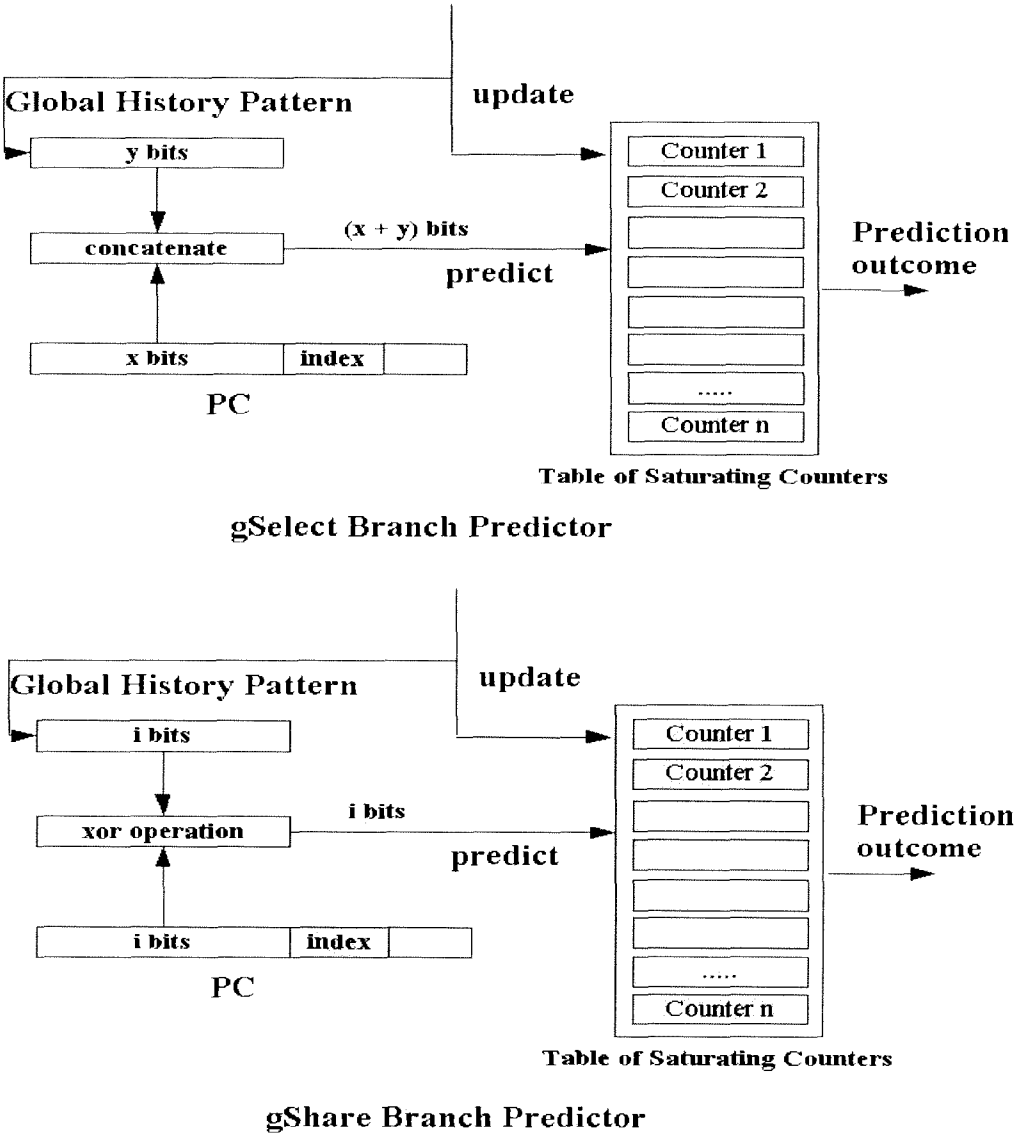



**Figure 2.8 - GShare and Gselect Global Branch Predictors**

More advanced global branch predictors is constructed with combination of the PC and the global history register. Figure 2.6 shows the structure of two of such predictors: gshare and gSelect. The only difference between gShare and gSelect is how the address for accessing the table is calculated. GShare uses the logical exclusive-oring of the

portion of PC with the global history register, whereas gSelect concatenates them. The idea is to keep track of the prediction of the same branches that have different history patterns.

Since global and local branch predictors are effective in different circumstances, a combined (tournament) branch predictor is proposed [4][5][17] . The structure of this kind of predictors is shown in figure 2.9:

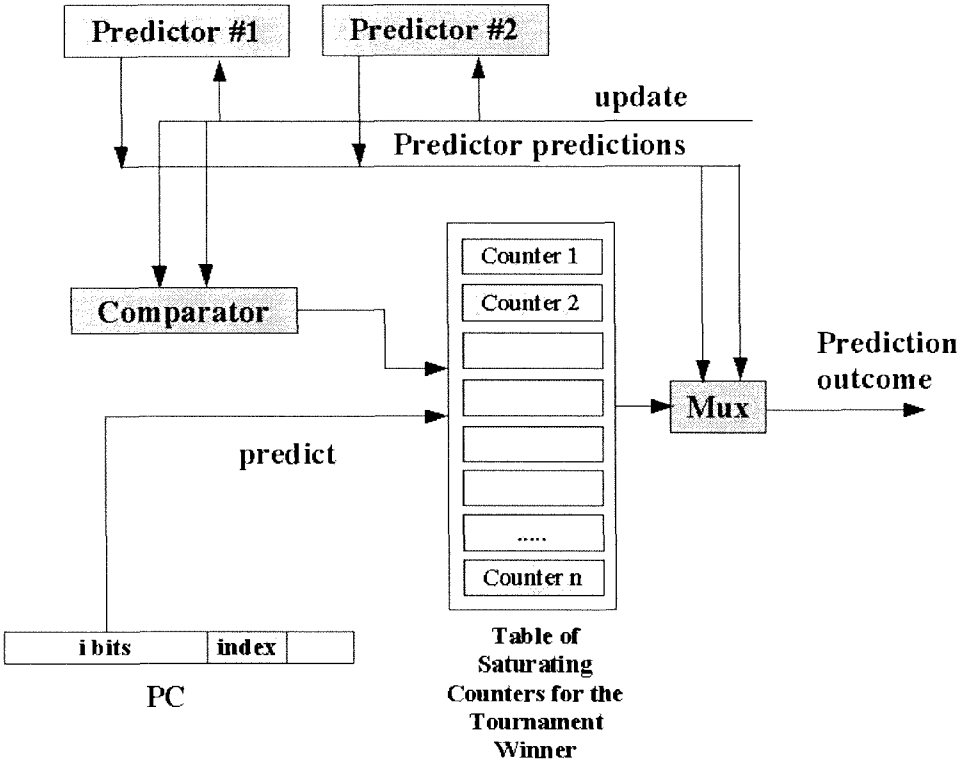



**Figure 2.9 - Combined Branch Predictor**

For every prediction, a tournament table with two bit up/down saturating counters is accessed. If the value of the counter is equal to zero or one, the first predictor (p1) and otherwise the second predictor (p2) is the winner of the tournament. Once the branch is

resolved (the outcome of the branch is known), the following rules are applied to update the tournament table:

- ➔ If p2's prediction is correct, the tournament table counter is incremented.
- ➔ If p1's prediction is correct , the associated counter is decremented.
- ➔ If both predictions are correct or incorrect at the same time, the value of the saturating counter is unchanged.

Combined branch predictors are very effective and have been used in many commercial processors such as Alpha-21264 [4], which is illustrated in figure 2.10.

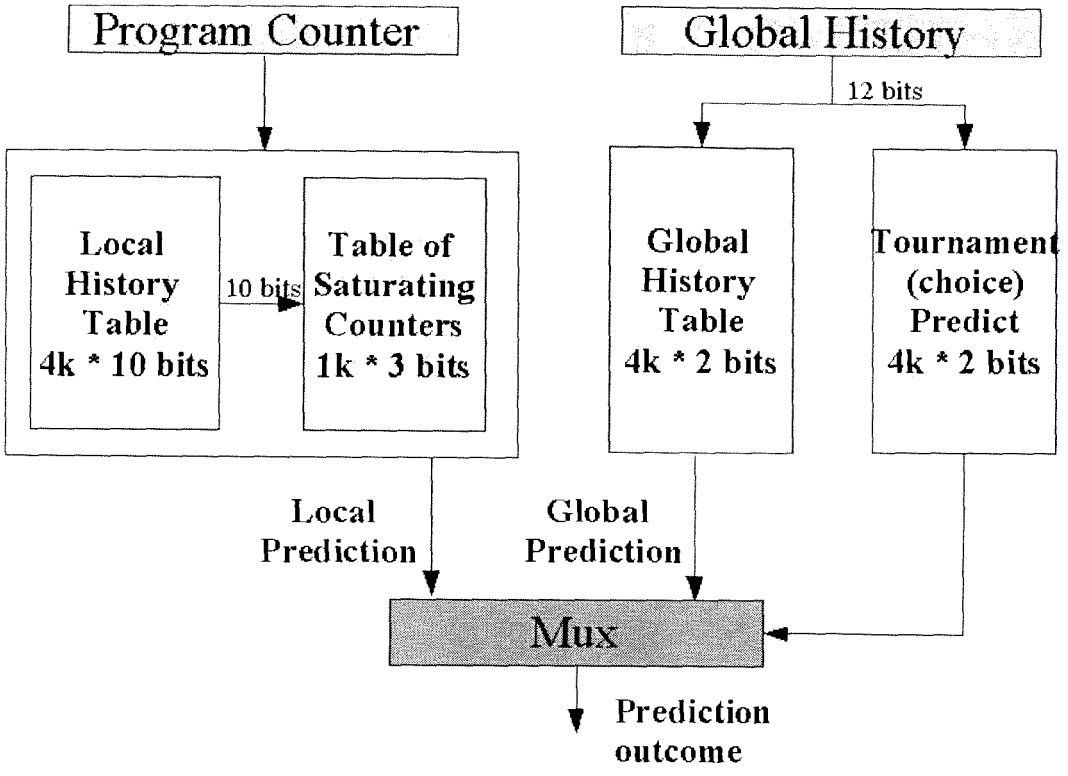



**Figure 2.10 - Combined Branch Predictor for Alpha-21264**

The only difference between figure 2.9 and 2.10 is the way the tournament predictor is updated. The Alpha processor uses the global history instead of the the program counter

to update the tournament predict unit.

## 2.4 Confidence Estimation for Speculation Control

In section 2.2, it was mentioned that almost all modern pipeline superscalar processors use speculative techniques to increase instruction level parallelism. An instruction will be committed if the original prediction in the "branch prediction unit" was correct. In the case of incorrect prediction of a branch, the subsequent instructions that have been entered in the pipeline after the mis-predicted branch are flushed and the previous state of the processor before mis-prediction is recovered.

By 1995, researchers realized that as the complexity of the processors are increasing due to ability of fetching and executing multiple instructions, the penalty of incorrect speculation may be high enough that it may be better not to speculate in those instances where the "probability of mis-prediction" is relatively high. That is, it may be desirable to vary behavior depending on the probability of mis-prediction [6]. This is when the concept of "confidence estimation" were proposed to find the quality of the branch prediction.

Confidence estimation is a statistical tool that has been used in many other scientific and engineering applications such as image, audio and video processing, testing in medical applications, neural networks, and so on [15]. In general, whenever there is an algorithm or procedure that has to do some prediction , a certain "quality" or "confidence" could be associated with that prediction. Prediction of a certain action is usually based on the behavior of a unit in past (history) or the behavior of similar units (pattern) associated with that action. In most of these applications, confidence estimation is implemented in software that assigns a quantitative value for the confidence of the prediction. However, using such sophisticated and in-depth analysis of confidence estimation is not feasible in hardware, since the design doesn't allow sacrificing many cycles to just find the confidence of a branch. As a result, the confidence estimation mechanism should be

simple enough that could easily be implemented in hardware and accurate enough that it could assign correct confidence estimation to each branch.

In [6], James Smith et al., proposed a probabilistic algorithm such that a "confidence-estimator unit" assigned "high" or "low" confidence to all branches during the decode stage. The confidence estimator keeps track of the prediction outcome of the last *n* branches executed. If the number of branch predictor's correct predictions for a particular branch was higher than a given threshold, that particular branch was tagged as a "high-confidence", which means that with high certainty the prediction for that particular branch is correct. Otherwise, a branch is considered as "low-confidence". Assigning low or high confidence to branches only needs one bit for representation. The proposed architecture could be seen in the following figure [6]:

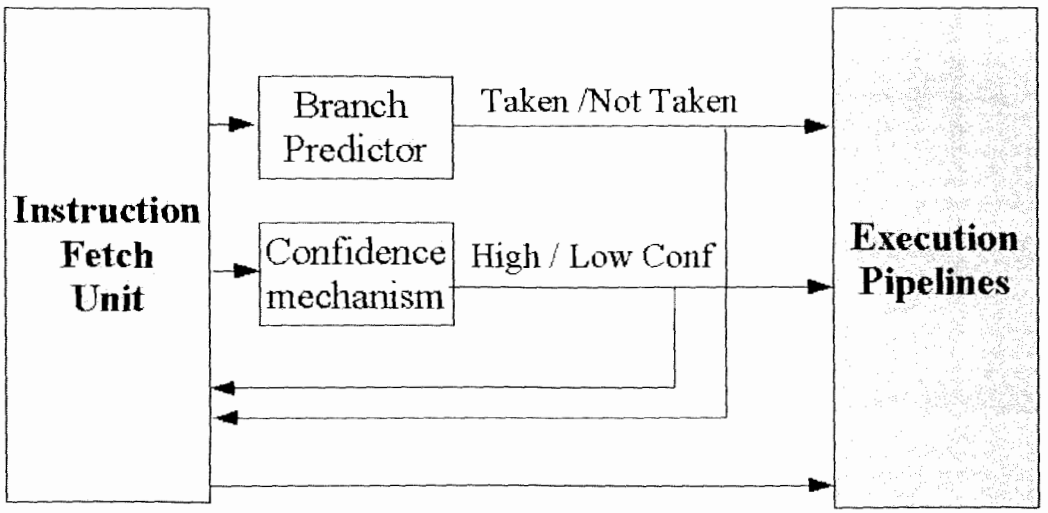



**Figure 2.11 - Architecture Based on Confidence Estimation**

Confidence mechanism could be used to do optimization for branches both for the fetch and other pipeline unit, as it was proposed in [6]. Since then, various confidence estimation based approaches have been used for the purpose of optimization. Pipeline gating[7], branch inversion reversal[8] and dual path execution[9] are examples of such optimization techniques.

In the following sub-sections, three different implementation of the confidence estimators are explained in details [15].

### 2.4.1 JRS estimator (Jacobsen, Rosenberg, and Smith)

This confidence estimator uses a miss distance counter table (MDC), in addition to branch prediction unit. Each entry in the table, is an up/reset saturating counter that is incremented based on the correctness of the branch prediction unit. This estimator essentially has the same structure as the Gshare branch predictor. The entry in the table is determined by "exclusive-oring" some portion of program counter (PC) with global branch history register (BHR), which keeps the confidence of the last *n* global branches. Figure 2.12 shows the architecture of the JRS confidence estimator in details.

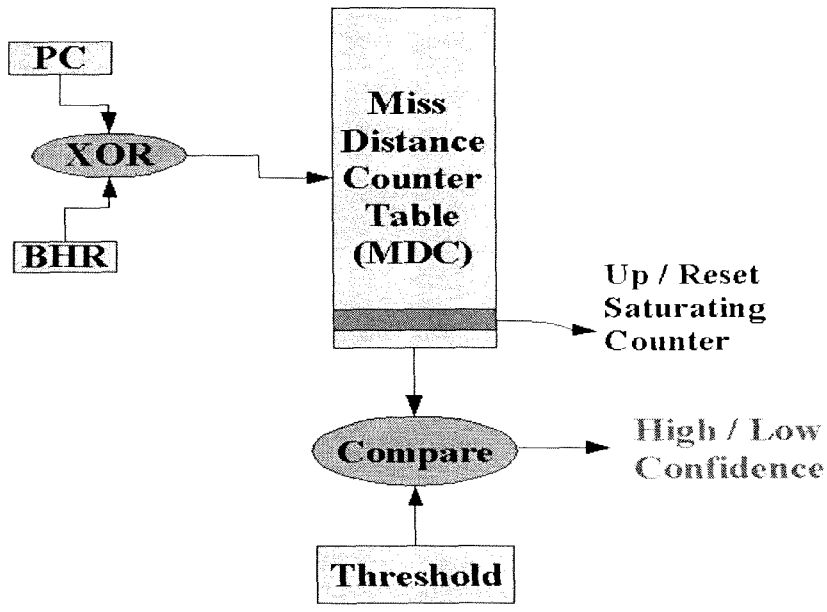



**Figure 2.12 - Architecture of JRS Confidence Estimator**

### 2.4.2 Pattern History Estimator

This predictor uses the specific pattern of the last *n* branches for determining the cost associated with a particular branch.

Example:

As an illustration, consider a predictor that keeps track of the pattern of the last four branches. The combination of all the patterns are:

| | |
|---|---|
| 1111 | Always Taken (Four Taken) |
| 1110/1101/1011/0111 | Almost Taken (Three Taken, One Not Taken) |
| 0000 | Always Not Taken (Four Not Taken) |
| 0001,0010,0100,1000<br>0011,0101,0110,0110<br>1001,1010,1001,1100 | Almost Not Taken (One and Two Not Taken ) |

Two different confidence estimators could use different patterns for categorizing branches to "low" and "high" confidence:

Confidence Estimator 1: "Always Taken" → High Confidence
The rest → Low Confidence

Confidence Estimator 2: "Always Taken" + "Almost Taken" → High Confidence
The rest → Low Confidence

### 2.4.3 Up/Down Saturating Counters Estimator

Up/ Down Saturating counters estimator has the same structure as JRS with the difference that it uses up/down saturating counters for every entry in the table.

In the following section, the application of confidence estimation is illustrated in pipeline

gating.

## 2.5 Pipeline Gating

Almost all modern superscalar processors use branch prediction to speculate the direction of a branch. However, there is always a trade-off between speculation and power consumption. With high branch prediction accuracy, most issued instructions will be committed. However, many programs have a high branch mis-prediction and the issued instructions will never commit.

Study shows that pipeline activity is the dominant source of power consumption in superscalar processors [7]. As a result, if the pipeline resource could be utilized in a more efficient way, considerable amount of power is going be saved.

Pipeline gating uses confidence estimator, in order to track number of "unresolved low-confidence" branches in the pipeline. If the number is higher than a given threshold, the pipeline doesn't allow more instructions to enter the pipeline (gating), until number of unresolved branches is reduced to lower than that threshold.

Figure 2.13 explains the concepts of pipeline gating pictorially:

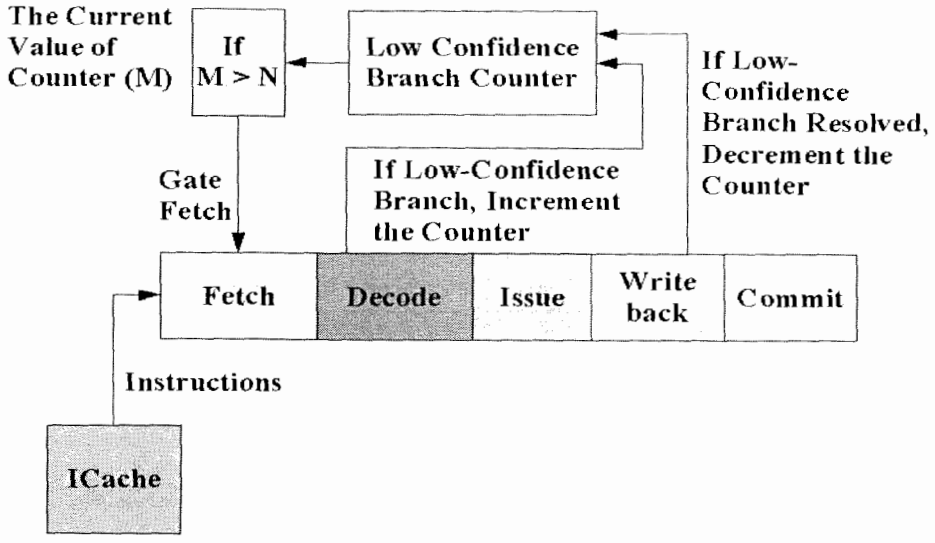



**Figure 2.13 - Pipeline Gating Architecture**

The "Low Confidence Branch Counter" keeps tracks of number of unresolved low-confidence branches. During the decode stage, if a branch is low-confidence, this counter incremented. If the value of this counter is higher than certain threshold, the gating is applied. During the write-back stage, the counter for total number of "low-confidence" branches is decremented for the "low-confidence" branches.

# Chapter 3 - Simulation Tools

In this chapter, simulation tools that are used for the purpose of this research are explained in details.

## 3.1 Simplescalar Tool Set

Simplescalar is a open source simulation tool that is written in the C programming language that simulates a generic superscalar processor. For every stage in figure 2.2, an associated function is implemented. The program accepts a set of benchmarks at its input as well as parameters for different units of processor (such as cache size, re-order buffer size and so on). The benchmarks are in the form of binaries and uses simplescalar instruction set. The output is a text file that gives information about that particular benchmark.

Here is an example of what the output looks likes:

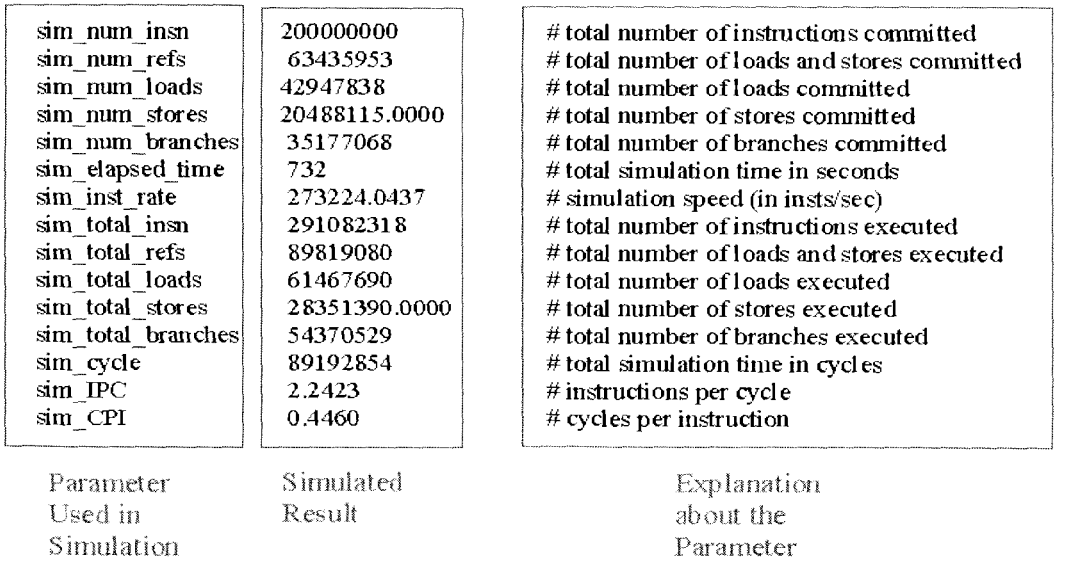



**Figure 3.1 - Part of Simplescalar Output File**

The example in figure 3.1 shows the simulated results based on 200 million committed instructions. The user could change the functionality of the simplescalar and add more of his desired parameter for both input and output.

## 3.2 WATTCH

WATTCH is a tool that uses simplescalar as its backbone to estimate the power based on parameterizable power model for different hardware structures and on per-cycle resources usage counts generated through cycle-level simulation [10]. It is a very fast tool compared to other power simulators and it is a great tool for comparing the effect of two different algorithms for power.

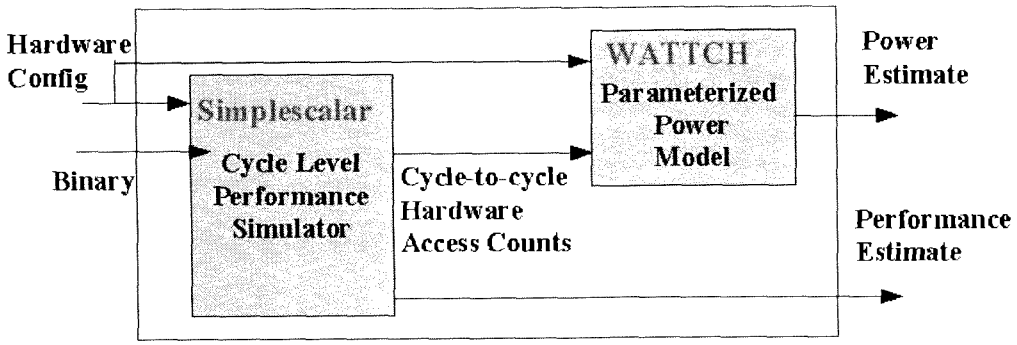



**Figure 3.2 - Overall Structure of Power Simulator [10]**

In figure 3.2, every cycle, simplescalar finds the access to different hardware structures and sends the result to WATTCH, which calculates the power estimate according to the input access parameters.

The following table shows the type of structure as well the units that are associated with these structures:

| *Type of Structure* | *Associated Hardware Units* |
|---|---|
| Array Structure | Data and instruction cache, cache tag arrays, all register files, register alias table, branch predictors and large portions of the instruction window and load/store queue. |
| Fully Associative Content-Addressable Memory | Instruction window/re-order buffer wakeup logic, and load/store order checks. |
| Combination Logics and Wires | Functional unit, instruction window selection logic, dependency logic, and result buses. |
| Clocking | Clock buffers, clock wires, and capacitive loads |

**Table 3.1 - Structures and Associated Units Implemented in WATTCH**

One could add a hardware unit that is associated with any of the structures above. If the structure needed is not in the list, a new structure could also be implemented.

## 3.3 SPEC 2000 Benchmarks

As it was mentioned in the last section, both simplescalar and WATTCH accept a set of benchmarks as their inputs. These benchmarks are based on SPEC 2000 standards. The Standard Performance Evaluation Corporation (SPEC) is a nonprofit consortium whose members include hardware vendors, software vendors, universities, customers, and consultants. SPEC's mission is to develop technically credible and objective component and system-level benchmarks for multiple operating systems and environments, including high-performance numeric computing, web servers and graphical subsystems. Members agree on benchmarks suites that are derived from real world applications so that both computer designers and computer purchasers can make decisions on the basis of the realistic workloads. By license agreement, members agree to run and report results as specified by each benchmarks suite [11].

The following table shows the name of different benchmarks and their descriptions [11]. The gray rows are used for the purpose of this research which are the benchmarks that are commonly used in the field of computer architecture.

| Integer Benchmarks (SPECint2000) | | |
|---|---|---|
| **Benchmark** | **Language** | **Description** |
| **164.gzip** | C | Compression |
| **175.vpr** | C | FPGA circuit placement and routing |
| **176.gcc** | C | C programming language compiler |
| **181.mcf** | C | Combinatorial optimization |
| **186.crafty** | C | Game playing: Chess |
| **197.parser** | C | Word Processing |
| **252.eon** | C++ | Computer Visualization |
| **253.perlbmk** | C | Perl programming language |
| **254.gap** | C | Group theory, interpreter |
| **255.vortex** | C | Object-oriented database |
| **256.bzip2** | C | Compression |
| **300.twolf** | C | Place and route simulator |

**Table 3.2 - SPEC 2000 Integer Benchmarks**

| Floating Benchmarks (SPECfp2000) | | |
|---|---|---|
| **Benchmark** | **Language** | **Description** |
| **168.wupwise** | F77 | Physics: Quantum chromodynamics |
| **171.swim** | F77 | Shallow water modeling |
| **172.mgrid** | F77 | Multigrid solver: 3D potential fields |
| **173.applu** | F77 | Partial differential equations |
| **177.mesa** | C | 3D graphics library |
| **178.galgel** | F90 | Computation fluid dynamics |
| **179.art** | C | Image recognition / neural networks |
| **183.equake** | C | Seismic wave propagation simulation |
| **187.facerec** | F90 | Image processing: image recognition |
| **188.ammp** | C | Computational chemistry |
| **189.lucas** | F90 | Number theory / primality testing |
| **191.fma3d** | F90 | Finite-element crash simulation |
| **200.sixtrack** | F77 | Nuclear physics accelerator design |
| **301.apsi** | F77 | Meteorology: Pollutant distribution |

**Table 3.3 - SPEC 2000 Floating Benchmarks**

## 3.4 Simulation Parameters

We used the following parameters for simplescalar and WATTCH , which is similar to the current technology for superscalar processor:

| | |
|---|---|
| **Branch Predictor** | 8k Combined Branch Predictor |
| **Fetch Unit** | Up to 8 Instructions / cycle. |
| **Instruction fetch queue size** | 16 |
| **Re-order Buffer Size** | 128 entry |
| **Load / Store Queue** | 64 entry |
| **Out-of-order Core** | 8 instructions /cycle |
| **Confidence Estimator ( if any)** | 2 Bit Pattern History with Both Strong Configuration |
| **L1 Instruction Cache** | 512K, 4-way Set Associative, 32-byte blocks, 3 cycle hit latency |
| **L2 Instruction Cache** | 1024 KB, 4-way Set Associative, 32-byte blocks, 16 cycle hit latency |
| **L1 Data Cache** | 256, 4-way Set Associative, 32-byte blocks, 3 cycle hit latency |
| **L2 Data Cache** | 1024 KB, 4-way Set Associative, 32-byte blocks, 16 cycle hit latency |
| **Integer ALU** | 8 |
| **Integer Multiplier / Divider** | 2 |
| **Floating ALU** | 8 |
| **Floating Multiplier / Divider** | 2 |

**Table 3.4 - Parameters Used for Simplescalar and WATTCH**

# Chapter 4 – Cost Analysis for Speculation Control

There are many algorithms that are proposed to improve branch prediction accuracy to reduce total number of mis-predictions [17,18,19,20,21,22,23,24]. Furthermore, "confidence estimation" is used as an complementary mechanism to branch prediction unit to improve speculation control [6][7][8][15]. Most of these techniques have indirect impact on reducing power due to mis-prediction. As it was explained in previous chapters, when a branch is mis-predicted, the associated entries in various buffers and functional units have to be flushed and the previous state of the processor before mis-prediction has to be recovered, which is a very costly performance. The biggest source of power waste during mis-prediction is the flushing of associated instructions in the re-order buffer, which is the queue that allows instruction to be executed in an out-or-order but committed in an in-order manner such that the sequential integrity of the program is maintained [31]. Since even the best branch predictor makes mis-prediction from time to time, we propose a technique that directly targets to filter the set of branches that are more responsible for the power waste due to mis-prediction.

We define the cost of a branch as the number of instructions that has to be flushed in the re-order buffer, if mis-prediction occurs.

If a mis-predicted branch flushes more than a certain threshold number of instructions, it is considered as "high-cost", otherwise it is "low-cost". Such architecture can be used to do more accurate optimization techniques and have more control for speculations. Figure 4.1 illustrates the new architecture that we propose for speculation control:

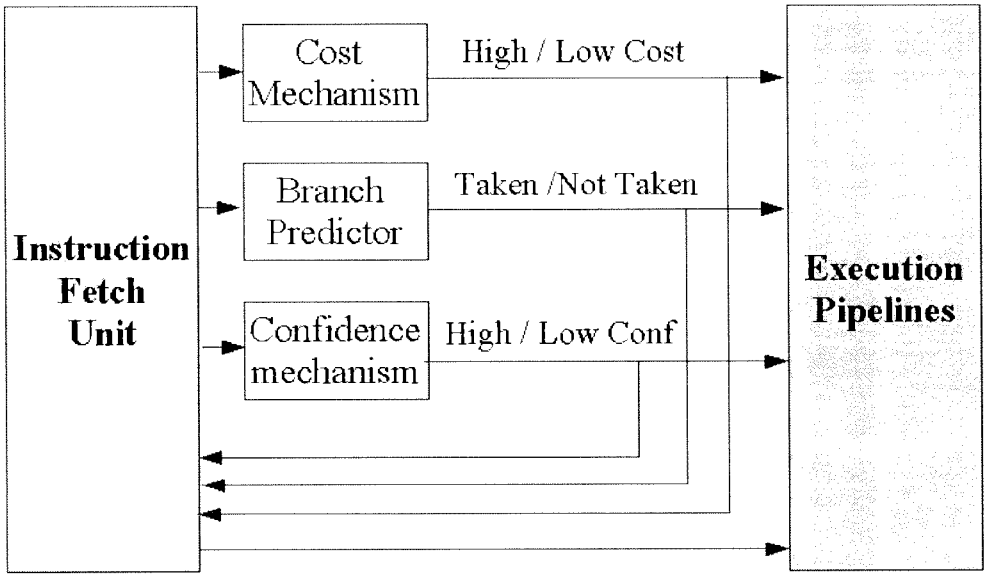



**Figure 4.1 - Architecture based on cost / branch and confidence predictor**

We categorize branches into two sets: "low-cost" and "high-cost" which can be represented by one bit, as illustrated in figure 4.1. The outputs of the branch prediction, confidence estimation, and cost mechanism can be used for optimization in the fetch stage (pipeline gating) and the rest of the pipeline.

Like branch prediction unit, in order to be successful in the implementation of a cost mechanism (predictor), we have to make sure that the cost associated with a mis-predicted branches is indeed predictable, which is the topic of the next section.

## 4.1 Study I: Predictability of Cost for Mis-predicted Branches

Our goal is to investigate if we can predict the cost associated with a mis-predicted branch, when a branch is decoded. In other words, if a mis-predicted branch flushes *n* number of instructions and it is considered as "high" or "low" cost, we want to examine if we can use this knowledge for the same branch next time it enters the pipeline by designing a cost predictor. Figure 4.2 illustrates the setup for cost analysis. We use a PC-indexed table (structure is very similar to a branch predictor). Each entry consists of two

fields: branch PC (Program Counter), and a saturating counter that determines the cost associated with that particular branch. Saturating counters are used since the counters shouldn't roll to the min/max when they are reach to their max/min.

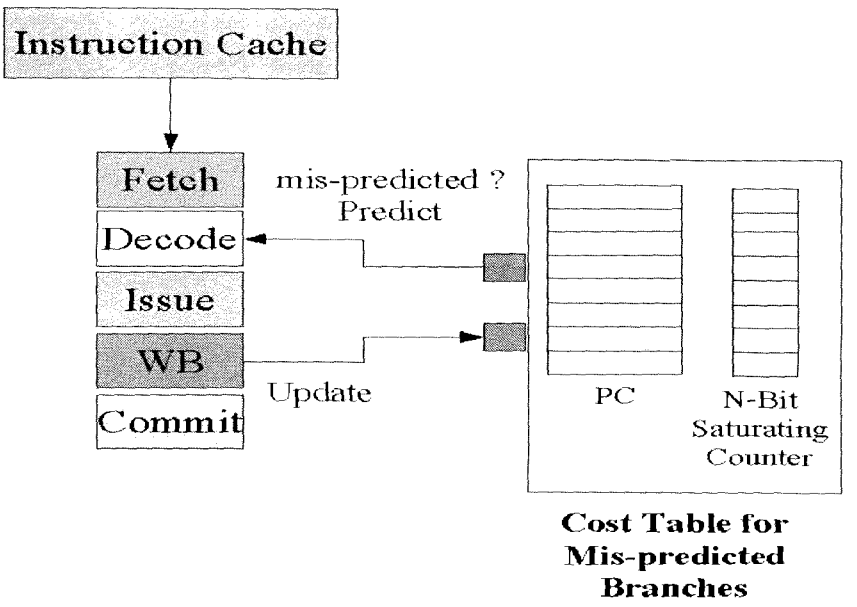



**Figure 4.2 - Cost Analysis**

As presented in the figure 4.2, the cost associated with a mis-predicted branch is determined by accessing the counter associated with a particular branch in the cost table at the decode stage. The value of counter determines the cost associated with a particular branch. For instance, if we use two bit representation for the counters value of 0 and 1 represents "low-cost" branches while 3 and 4 represents "high-cost" ones. It should be noted that our simulator (simplescalar) has the knowledge about a mis-predicted branch during the decode stage (before it is executed). In the write-back stage (once the branch is executed) in a real superscalar processor, it can be determined whether or not a branch is correctly predicted. It is at this stage that we determine the category of cost associated with the same mis-predicted branch. Figure 4.3 illustrates how we calculate the cost associated with a particular branch.

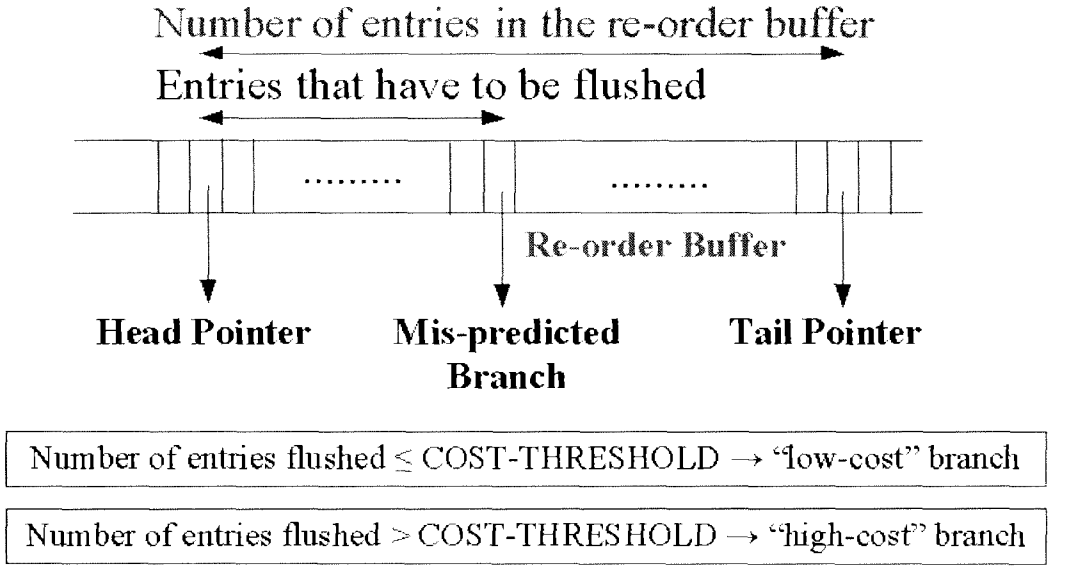



**Figure 4.3 - Cost Calculation during the Write-back Stage**

The re-order buffer is implemented as a circular queue [2]. There are two pointers that are associated with the start and end of the queue that are labeled as the "head" and "tail" pointer in figure 4.3. Instructions are dispatched into the queue from the the location of the head pointer and committed from the location of the tail pointer. When a branch is mis-predicted, we are in the wrong-path of execution. As a result, all the entries between the head pointer and mis-predicted branch has to be flushed. If the number of entries (instructions) that are flushed is smaller than or equal to a given COST-THRESHOLD, that branch is considered as "low-cost", otherwise, it is a "high-cost" branch. After this process, the "head pointer" points to the position of the mis-predicted branch.

For our simulation model, we used the configuration that was outlined in section 3.3. The threshold for categorizing branches to high/low cost (COST-THRESHOLD) is set to 64, half of the re-order buffer size. we used a 2k-entry table with two bit saturating counters (most branch predictors also use two bits) for the cost table. We varied the size of the cost

table until aliasing is minimized. Essentially, we are mapping the whole instruction memory address to a much smaller table and aliasing can occur when two branches with different PC's map to the same entry in the table.

The following table explains how the cost table is accessed both in prediction and update stage when different types of saturating counters are used. This information can be used to see what kind of table and counter type can be used to predict the cost associated with a particular branch.

| Saturating Counter Type | Definition |
|---|---|
| **Up / Down – Low Cost** | The table keeps track of the low cost branches. During the write-back stage, if the prediction is low-cost, increment the counter, otherwise decrement the counter.<br>During the decode stage, if the value of the counter is smaller than 2, then consider that particular branch as high-cost, otherwise it is low-cost. |
| **Up / Down – High Cost** | The table keeps track of the high cost branches. During the write-back stage, if the prediction is high-cost, increment the counter, otherwise decrement the counter.<br>During the decode stage, if the value of the counter is smaller than 2 then consider that particular branch as low-cost, otherwise it is high-cost. |
| **Up / Reset – Low Cost** | The table keeps track of the low cost branches. During the write-back stage, if the prediction is low-cost, increment the counter, otherwise reset the counter to zero.<br>During the decode stage, if the value of the counter is smaller than 2, then consider that particular branch as high-cost, otherwise it is low-cost. |
| **Up / Reset – High Cost** | The table keeps track of the high cost branches. During the write-back stage, if the prediction is high-cost, increment the counter, otherwise reset the counter to zero.<br>During the decode stage, if the value of the counter is smaller than 2, then consider that particular branch as low-cost, otherwise it is high-cost |

**Table 4.1 - Cost Table with Different Types of Saturating Counters**

The following parameter are used for predicting the cost category of each branch.

| Parameters | Definition |
|---|---|
| **Prediction Effectiveness** | The percentage of high-cost branches are identified |
| **Prediction Rate (Accuracy)** | The percentage of branches predicted as high-cost which are indeed high-cost. |

**Table 4.2 - Parameters Used for Finding The Cost Prediction**

The difference between this two parameters is just the fact that prediction accuracy also considers the aliasing rate. Since the values obtained for these two parameters are very close in our simulated benchmarks, the term prediction accuracy is used throughout the rest of this chapter.

Figure 4.4 illustrates the prediction accuracy for integer and floating benchmarks. As explained in section 3.4, these are the common benchmarks that are used in the field of computer architecture for representing general purpose computing. For the 175.VPR benchmark, we can see that the prediction accuracy is almost the same for different types of counters (around 80%) except for the "up/reset – high cost" (around 68%). Moreover, the average prediction accuracy is higher for floating benchmarks compared to integer benchmarks.

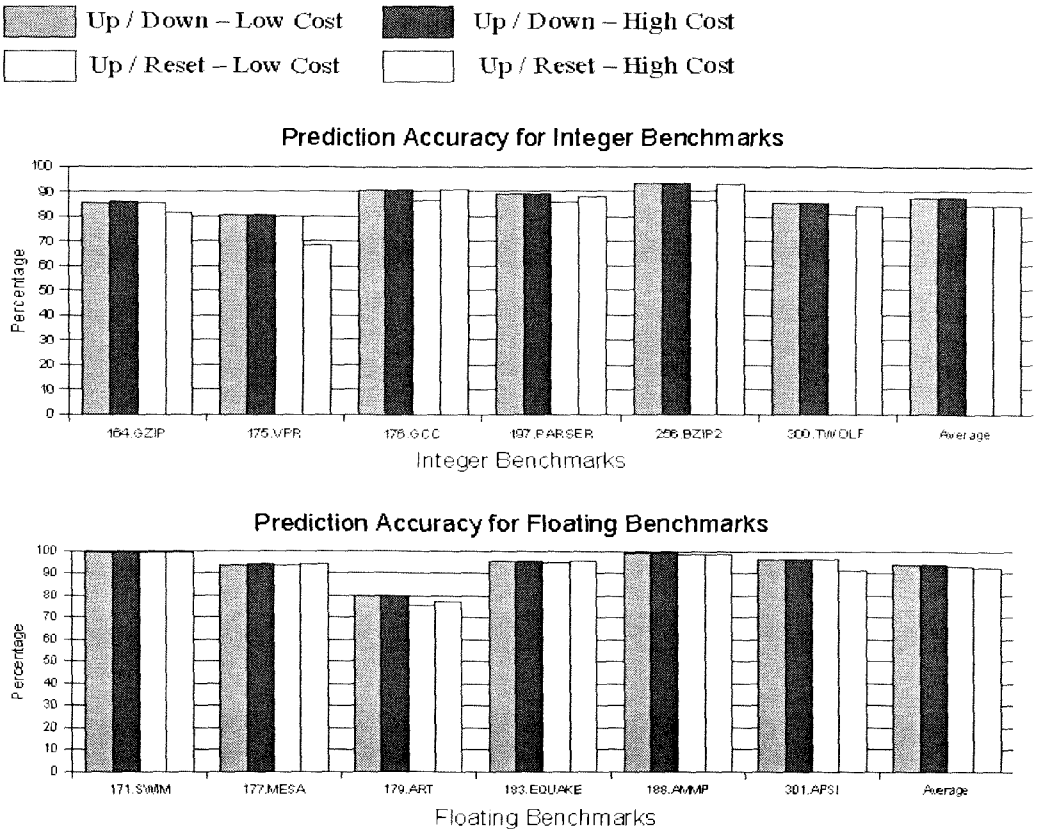



**Figure 4.4 - Prediction Accuracy**

Table 4.3 summarizes the average of the prediction accuracy for the integer and floating benchmarks obtained from the diagrams in figure 4.4.

| | **Integer Benchmarks** | **Floating Benchmarks** |
|---|---|---|
| **Counter Type** | **Average** | **Average** |
| **Up / Down – Low Cost** | 87.27% | 93.93% |
| **Up / Down – High Cost** | 87.29% | 94.01% |
| **Up / Reset – Low Cost** | 84.07% | 93.11% |
| **Up / Reset – High Cost** | 84.12% | 92.76% |

**Table 4.3 - The Average Prediction Accuracy for Integer and Floating Benchmarks**

As it can be seen in the table 4.3, we obtain a high degree of prediction accuracy for both

integer and floating benchmarks.

In the second part of our study, we also want to determine whether cost prediction is more accurate for "low-cost" or "high-cost" branches. The prediction accuracy for both "low-cost" and "high-cost" branches are shown in figure 4.5. For every benchmark, there are two sets of prediction accuracy. The set in the left-hand side corresponds to "low-cost" and the one in the right-hand side corresponds to "high-cost" prediction accuracy. For instance, the prediction accuracy for "low-cost" branches in the 175.VPR benchmark is around 63% for the up/down counters whereas it is around 90% for the "high-cost" branches. We used a cost table with four different counter types for finding the prediction accuracy, the same way as it was done before. The average prediction accuracy for each cost category is summarized in table 4.4. We can draw a set of conclusion from this experiment as follows.

1) Prediction accuracy strongly depends on the particular benchmark. However, the prediction accuracy for the low-cost branches is more accurate than that of high-cost branches. This can be due to the fact that the cost threshold is chosen to be symmetric (64) and the distribution of the low/high cost branches is not symmetric with respect to the center. In other words, number of "low-cost" branches is higher than than of "high-cost" branches for this cost threshold. As a result, the counters in the cost table bias toward the "low cost" branches and makes it more predictable.

2) Prediction accuracy for floating benchmarks is higher than that of integer benchmarks particularly for "high-cost" branches. This can be due to fact that the floating benchmarks have a much higher prediction accuracy [5], and therefore categorizing few branches is more accurate.

3) Considering both integer and floating benchmarks, prediction accuracy for the "low-cost" branches is higher than of "high-cost" branches except for the "up / reset – low cost" counter.

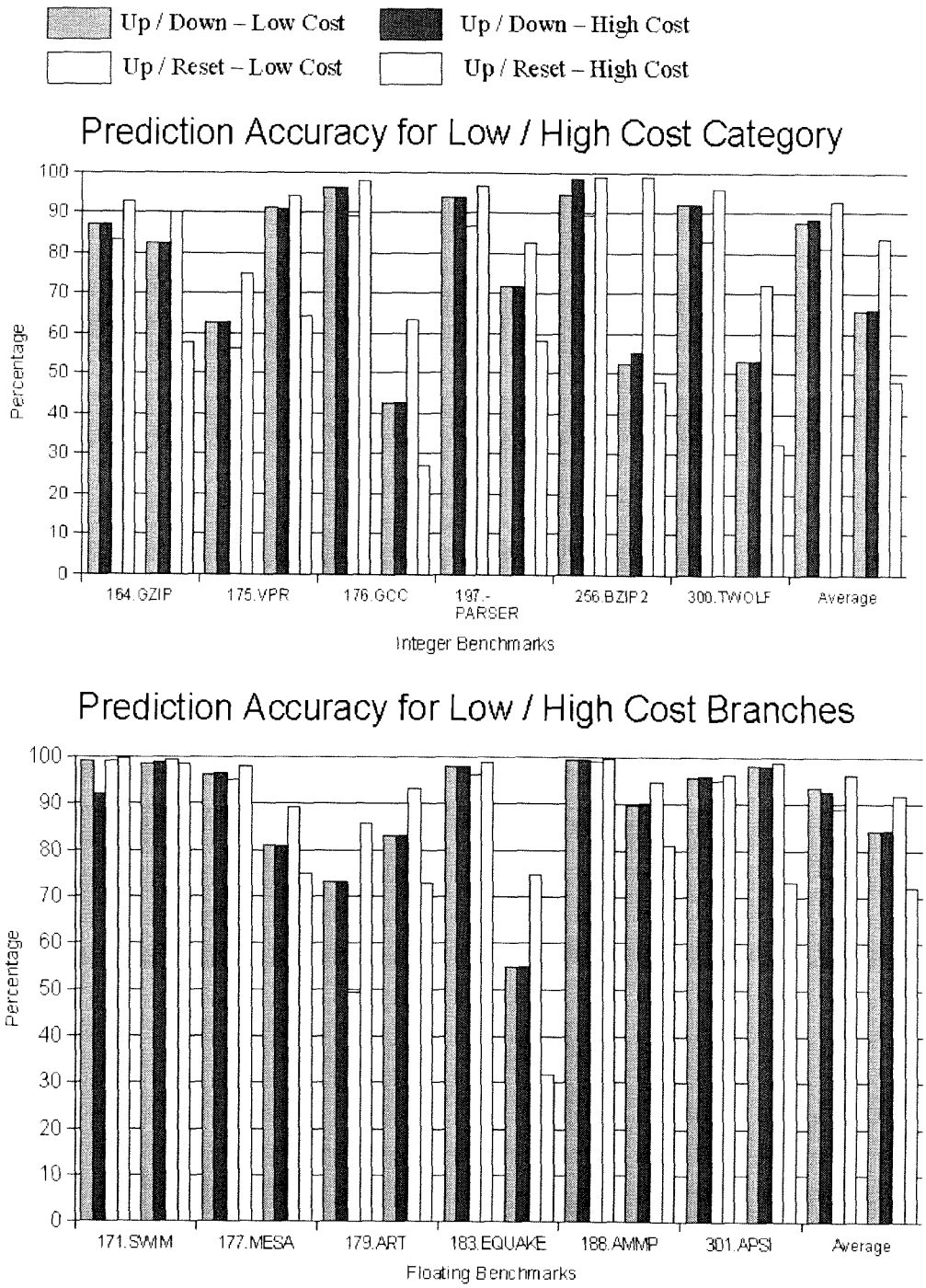



**Figure 4.5 - Prediction Accuracy for Low / High Cost Branches**

| | Integer Benchmarks | | Floating Benchmarks | |
|---|---|---|---|---|
| **Counter Type** | **Low Cost** | **High Cost** | **Low Cost** | ***High Cost*** |
| **Up / Down – Low Cost** | 85.73% | 61.01% | 92.55% | 81.03% |
| **Up / Down – High Cost** | 86.28% | 61.57% | 91.57% | 80.97% |
| **Up / Reset – Low Cost** | 79.08% | 81.51% | 83.39% | 90.87% |
| **Up / Reset – High Cost** | 91.90% | 43.27% | 96.10% | 63.40% |

**Table 4.4 - Average Prediction Accuracy for High and Low Cost Branches**

## 4.2 Study II: Cost Prediction for Optimization in Total Wasted Power

In this section, our goal is to investigate if the cost-predictor can be used to reduce power waste due to mis-prediction in a superscalar processor. Accordingly, we measure four different parameters, presented in figure 4.6 and 4.7.

1) Percentage of total wasted power (due to mis-prediction) to total power.
2) Percentage of number of times a branch is considered as "low-cost" or "high-cost" to total number of times flushed.
3) Percentage of total flushed instructions to total number of fetched instructions for "low-cost" and "high-cost" branches.
4) Percentage of total wasted power for low and high cost branches to total wasted power in a processor.

For the sake of illustration, in the top diagram of figure 4.6, 23% of total power is wasted due to mis-prediction for the 197.PARSER benchmark. In the second diagram from the top, we can see that the number of mis-predicted branches that are considered as "low-cost" are almost three times more than that of the "high-cost" branches for the 197.PARSER. For the same benchmark in the second diagram from bottom, we can observe that "high-cost" branches flush more instructions that "low-cost" branches. Finally the last diagram shows that almost both "low-cost" and "high-cost branches are responsible for the same power waste due to mis-prediction.
The average values obtained in figure 4.5 and 4.6, are summarized in table 4.5.

| | Integer Benchmarks | | Floating Benchmarks | |
|---|---|---|---|---|
| **Parameters** | **Low Cost** | **High Cost** | **Low Cost** | **High Cost** |
| **Percentage number of times a branch is considered as low or high cost to total number of times flushed** | 73.8% | 26.2% | 75.0% | 25.0% |
| **Percentage of total flushed Instructions to total number of fetched instructions** | 14.0% | 19.5% | 4.5% | 4.2% |
| **Percentage of wasted power for low and high cost branches to total wasted power** | 50.1% | 49.9% | 53.1% | 46.9% |
| **Parameter** | **Integer Benchmarks** | | **Floating Benchmarks** | |
| **Total Wasted Power to Total Power** | 21.6% | | 2.42% | |

**Table 4.5 - Average Value for Parameters for Investigating about Use of Cost for Power Optimization**

In this table, we can observe that the number of "low-cost" branches is higher than "high-cost" branches. However, they flush less instructions than "high-cost" branches. Furthermore, "high-cost" branches are responsible for almost the same amount of wasted power due to mis-prediction. We can observe that the wasted power for integer benchmarks is higher than that of floating benchmarks due to higher number of mis-predictions.

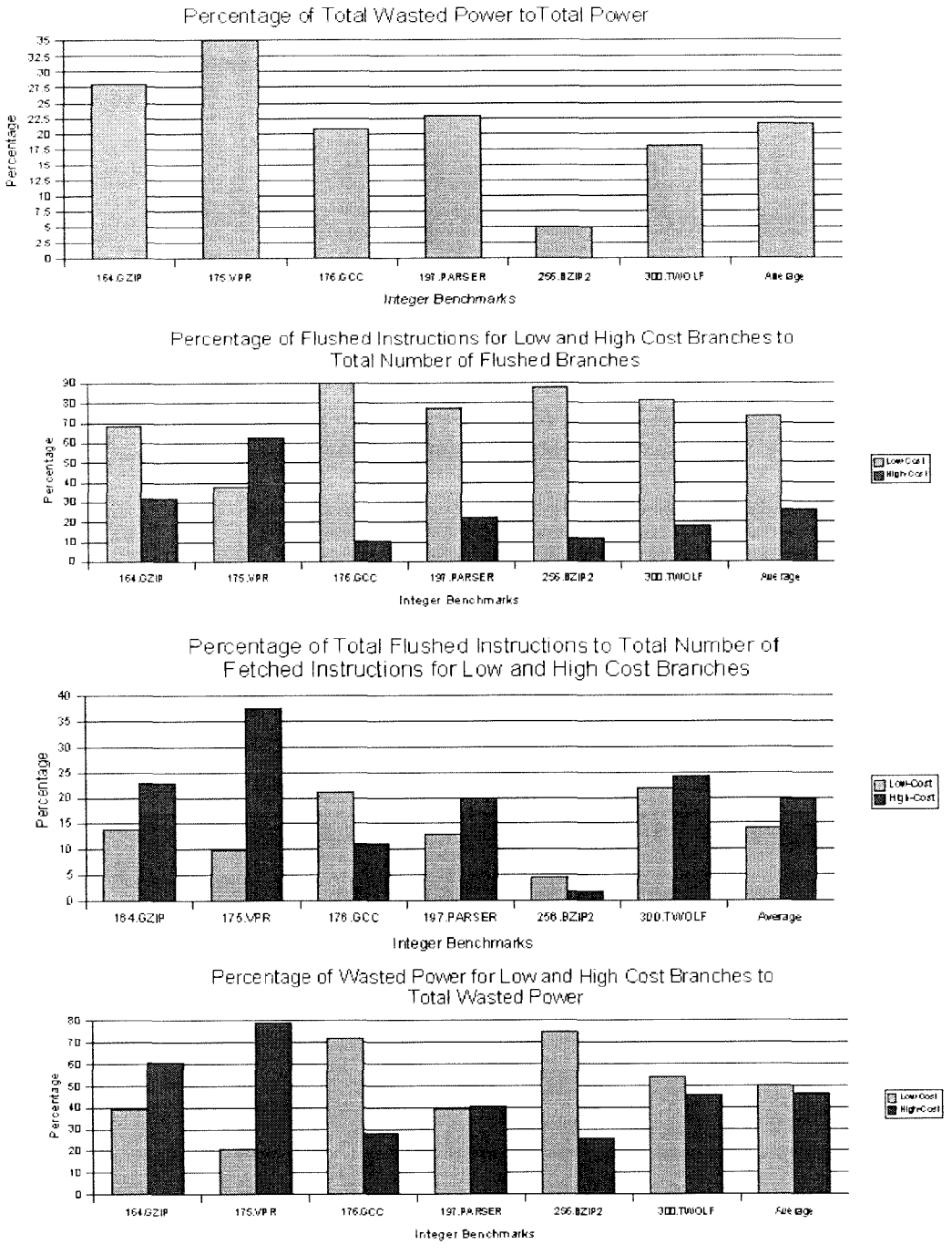



**Figure 4.6 - Relationship between Cost and Wasted Power for Integer Benchmarks**

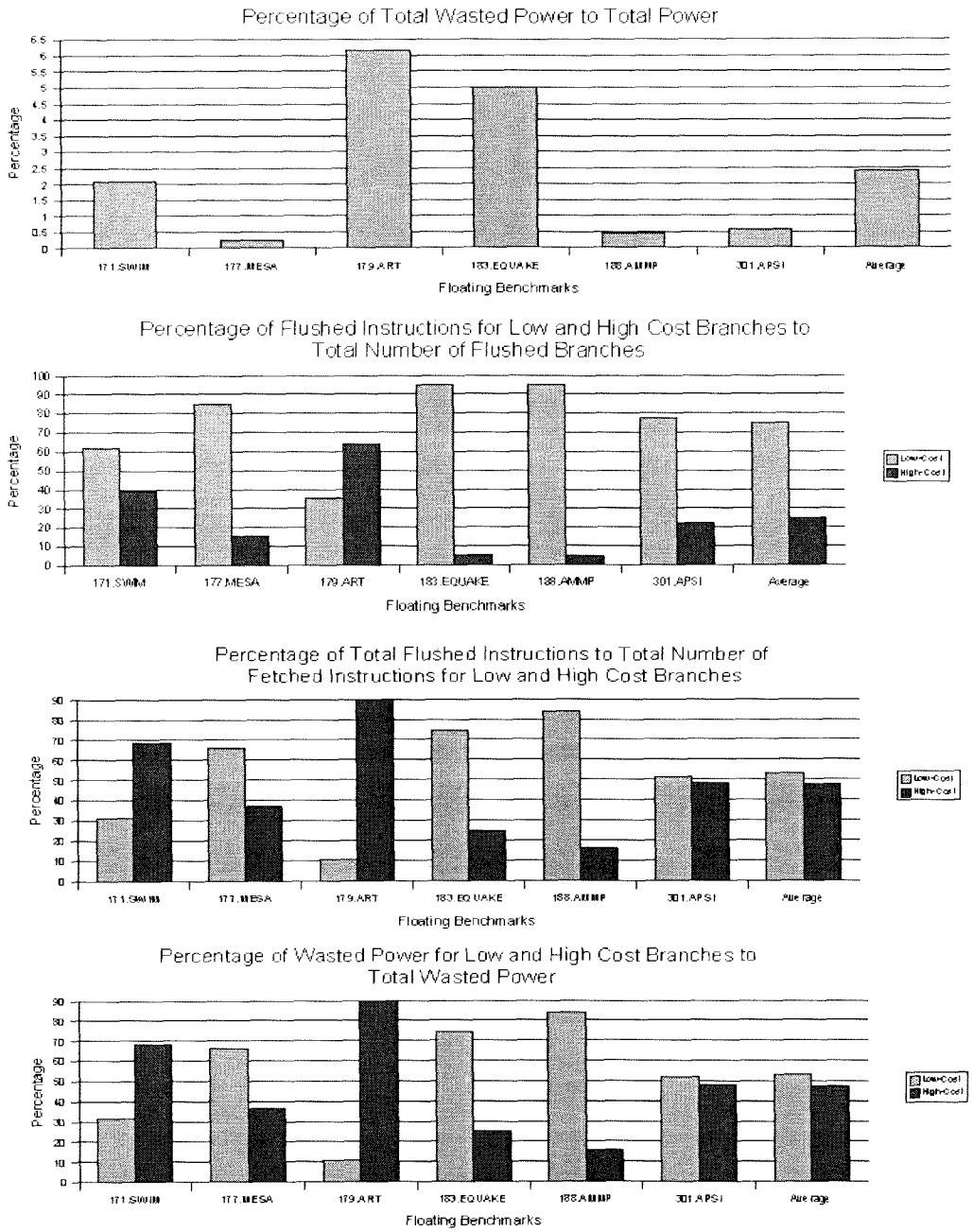



**Figure 4.7 - Relationship between Cost and Wasted Power for Floating Benchmarks**

In summary, we can conclude the following observations:

1) While number of "high-cost" branches is less than "low-cost" branches, they are responsible for almost half the total wasted power due to mis-predictions.

2) If "cost-estimator" is to be used for optimization, threshold of half of the re-order buffer size should not be used, since the values of the counters will bias toward the low-cost branches. As a result, the threshold should be set to a value such that number of "low-cost" and "high-cost" are almost equivalent. In the next chapter, we illustrate this point by presenting the application of cost predictor for pipeline gating.

# Chapter 5 – Cost Prediction for Pipeline Gating

As it was introduced in section 2.5, the aim of the original pipeline gating is to prevent wrong path instructions to enter the pipeline by distinguishing between branches that are more likely to be mis-predicted. Essentially, pipeline gating (clock gating) is achieved by turning-off the fetch circuitry by "Anding" the global clock with a signal that is set by the control circuitry. If this process results in frequent enable/disable of the fetch circuitry, the control circuitry keeps switching between the gated and non-gated mode, resulting in an increased overhead due to the power consumed by the control circuitry [14]. One way to reduce the associated overhead is to reduce the number of times gating is applied which we refer to as gating frequency.

We implemented the original pipeline gating mechanism (section 2.5) to show the effect of "gating threshold" for both IPC (Instruction Per Cycle) degradation and power reduction. The results are depicted in figure 5.1 and 5.2 for integer and floating benchmarks respectively. In this experiment, *"threshold n"* indicates that as soon as *(n+1)* low-confidence branches are in the pipeline, gating is applied and hence no more instructions are fetched .

As the gating threshold decreases, the number of times speculation is applied for the "low-confidence" branches decreases as well, which leads to deduction of wasted power due to mis-prediction. However, as the flow of instructions is interrupted more often due to increase of the gating threshold, the IPC degrades. For instance, "threshold 0" indicates that as soon as a single "low-confidence" branch is detected, no more instructions are fetched until the outcome of that branch is resolved which indicated that we don't apply speculation to that particular branch. However, once that branch is resolved, it takes many cycles until the instructions re-enter the pipeline which leads to degradation of IPC. In the middle diagram of figure 5.1, the IPC is degraded by around 11.5% for "threshold 0" but is only degraded by less than 1% for "threshold 3" for the 176.GCC benchmark. For the same benchmark, power reduction is around 8.5% for "threshold 0" but is degraded to

around 1.5% for "threshold 3" in bottom diagram of figure 5.1. The authors in [7] show that the gating threshold of two is an optimum way of balancing between IPC degradation and total power reduction, which is verified by our experiment. On average, the IPC degradation is around 3% while the total power reduction is around 4% for pipeline gating with threshold of two.

Our goal in this chapter is to take into account cost estimation to perform pipeline gating more efficiently. The ultimate challenge is to achieve the same performance degradation and total power reduction achieved by the conventional pipeline gating method, while reducing gating frequency. In the next section, we introduce three different cost predictors in order to improve the gating frequency.

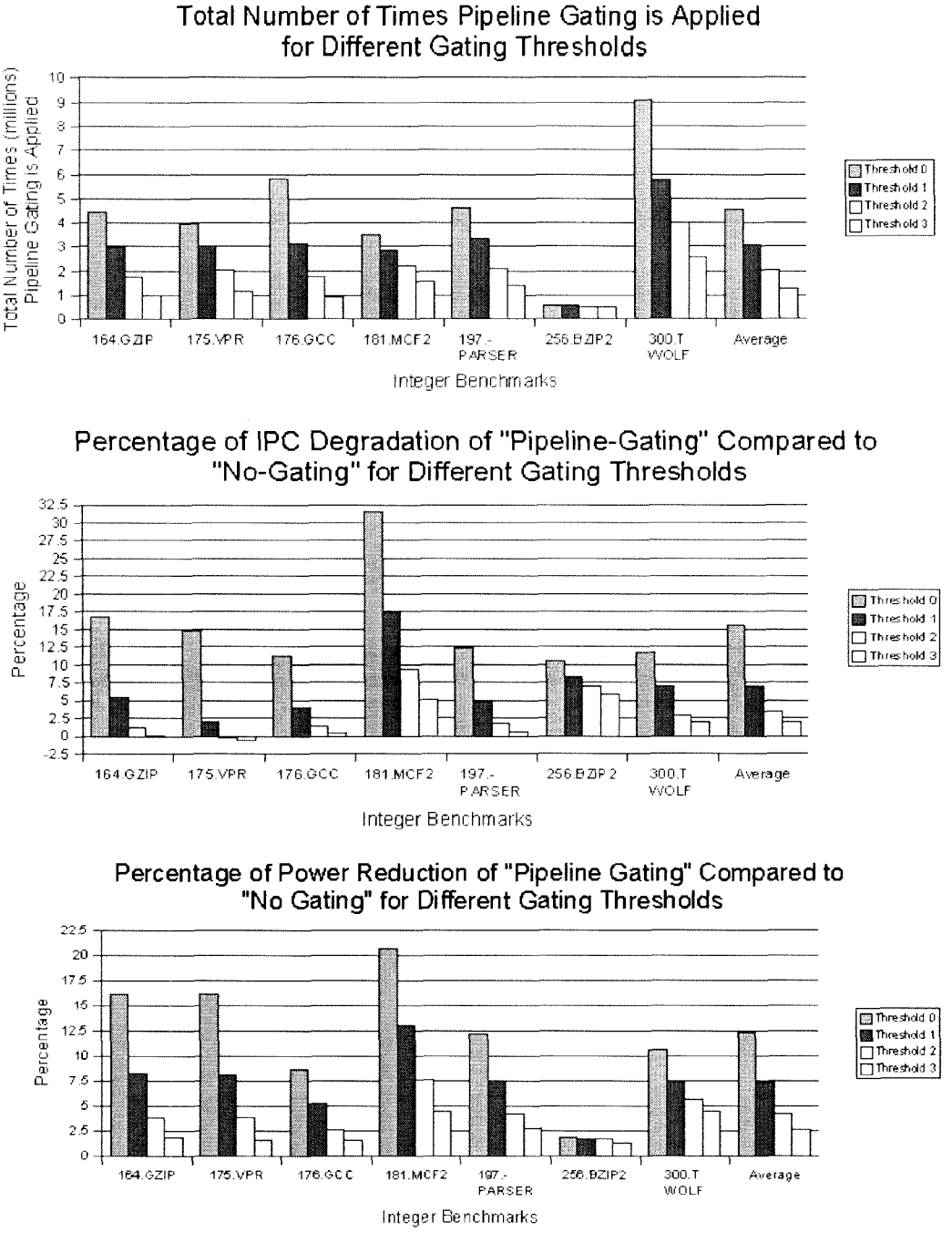



**Figure 5.1 - The effect of "Gating Threshold" on Pipeline Gating for Integer Benchmarks**

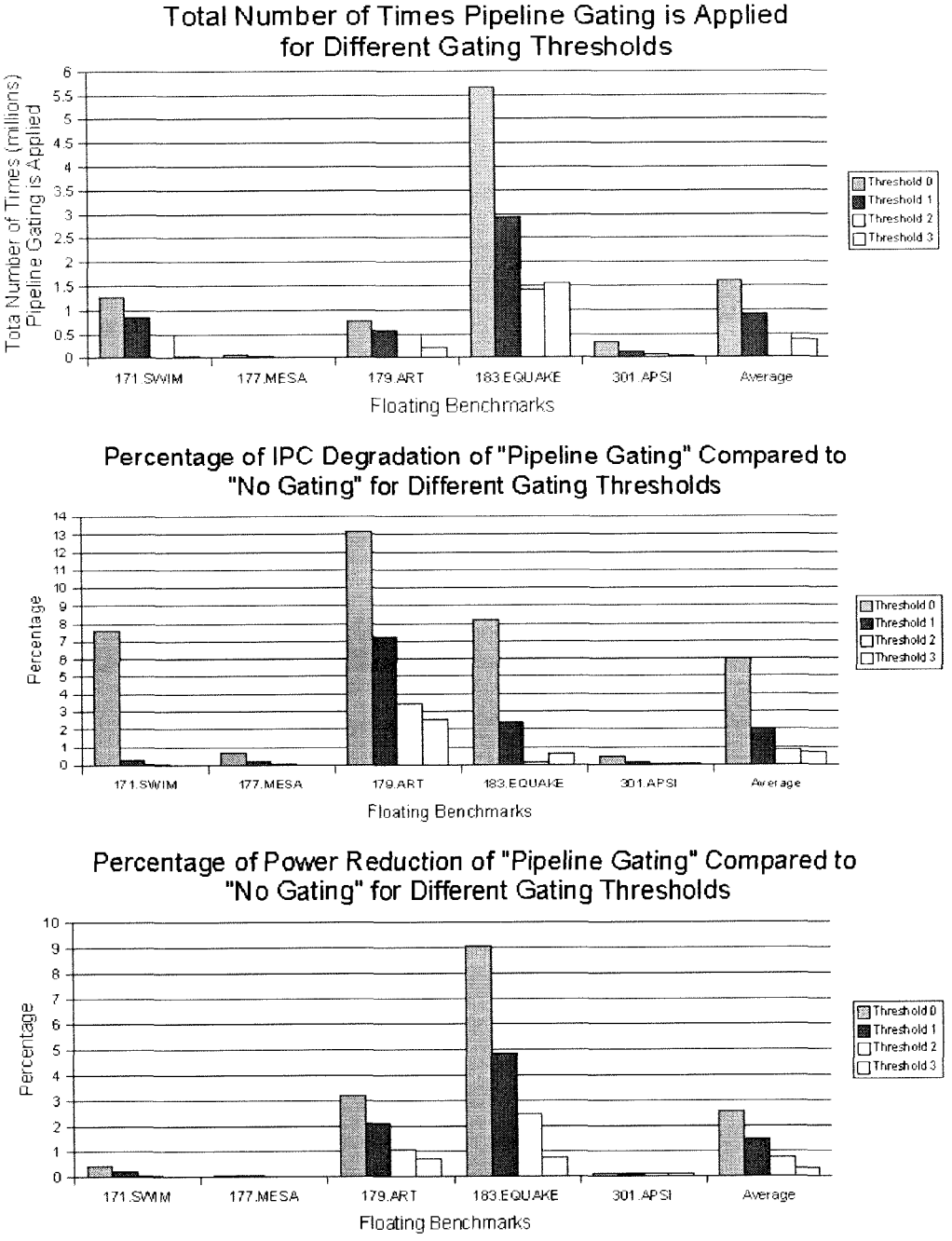



**Figure 5.2 - The effect of "Gating Threshold" on Pipeline Gating for Floating Benchmarks**

## 5.1 Mechanism #1 – Using Different Cost Predictors

As explained in the previous chapter, the possibility to predict the cost associated with a branch can be used to enhance pipeline gating, which is a power optimization technique. Accordingly as presented in figure 5.3, we aim at identifying "low-confidence and high-cost" branch instructions. In the original pipeline gating mechanism, the algorithm goal is to find a set of branches that has a high probability to be mis-predicted, which is shown as the middle rectangle in the left-hand side of figure 5.3. Our goal is to also identify a set of "high-cost" branches (has more contribution in wasted power due to mis-prediction) among those "low-confidence" branches, which is shown in the right-hand side of figure 5.3 As a result, and since fewer branches satisfy both the "confidence" and "cost" constraint, we reduce gating frequency. However, since we gate the pipeline only when mis-prediction cost is estimated to be high, we still achieve high power savings.

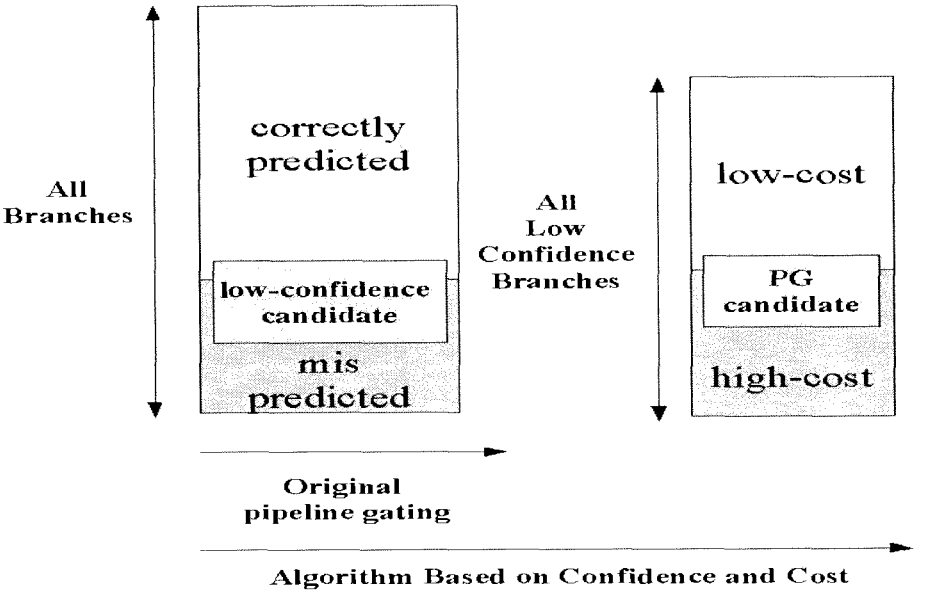



**Figure 5.3 - The filtering Process of The Cost Predictor**

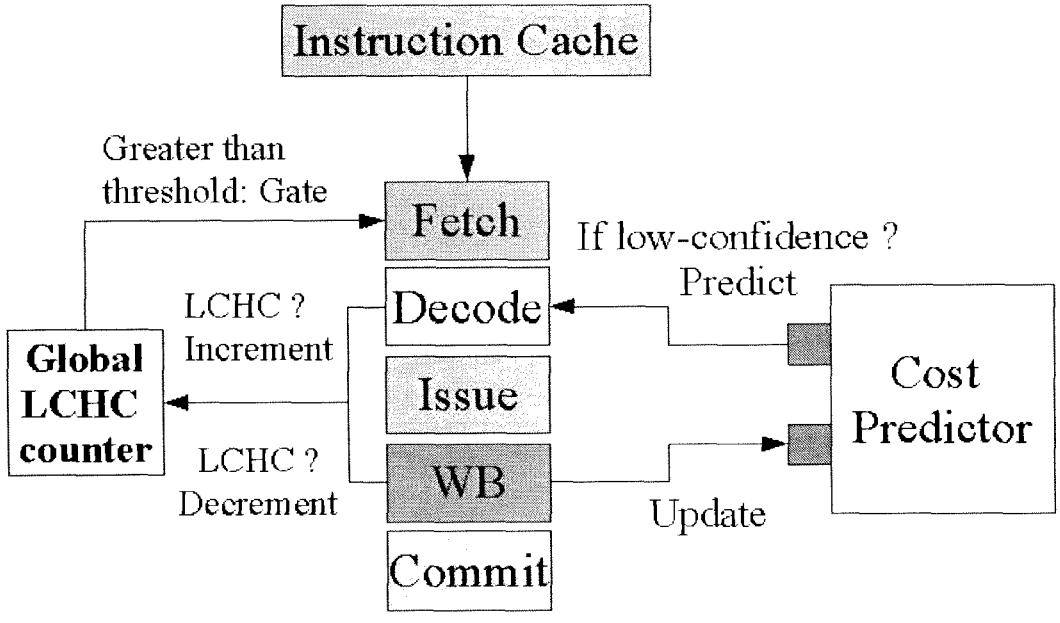



**Figure 5.4 - Pipeline Gating Mechanism Based on Cost Predictor**

Figure 5.4, illustrates how the whole procedure works. During the write-back stage (WB), the cost associated with all branches are determined, which means that we also estimate the possible mis-prediction cost of those branches that are not mis-predicted. We use a cost table, referred to as the cost predictor that keeps track of the cost of the branches. This table has a very similar structure as the branch predictors in section 2.3 and is explained in more details in the next section. We update this table to store the cost associated with a particular branch. Later, at the decode stage, if a particular branch is found to be low-confidence (by accessing the confidence estimator table), the cost associated with that branch is predicted by accessing the cost predictor.

In figure 5.4, a global counter keeps track of the number of "low-confidence and high-cost" ( LCHC) branches that are in the pipeline. If the number of "LCHC" branches is higher than a given threshold value, the gating mechanism is applied. This counter is incremented, when a "LCHC" branch is identified during the decode stage. Conversely,

this counter is decremented if that particular branch is successfully committed or has been flushed by some other branch.

In the following sections, we introduce three different cost predictors and study how they impact performance, power and gating frequency. For the simulation, the cost threshold equal to 32, which is quarter of the size of the re-order buffer, is found to work almost always the best. In other words, if we find that a branch flushes less than or equal to 32 instructions, we consider that branch as a "low-cost" one. As it was discussed in the last chapter, the cost threshold should not be half of the re-order buffer size, since the counters of the cost predictor will have the tendency to bias toward the "low-cost" branches. The cost threshold of quarter of the re-order buffer size, re-emphasizes this point. The cost predictor keeps track of high-cost branches. In other words, if a branch is found to be "high-cost" during the write-back stage, the counter associated with that particular branch is incremented (Table 4.1). In the following section, three different cost predictors are implemented and we show their effects on IPC degradation, power reduction and gating frequency.

### 5.1.1 Global Cost Predictor (PC-indexed Cost Predictor)

The global cost predictor consists of a PC-indexed table of *n* saturating counters. Figure 5.5 shows the structure of such predictor.

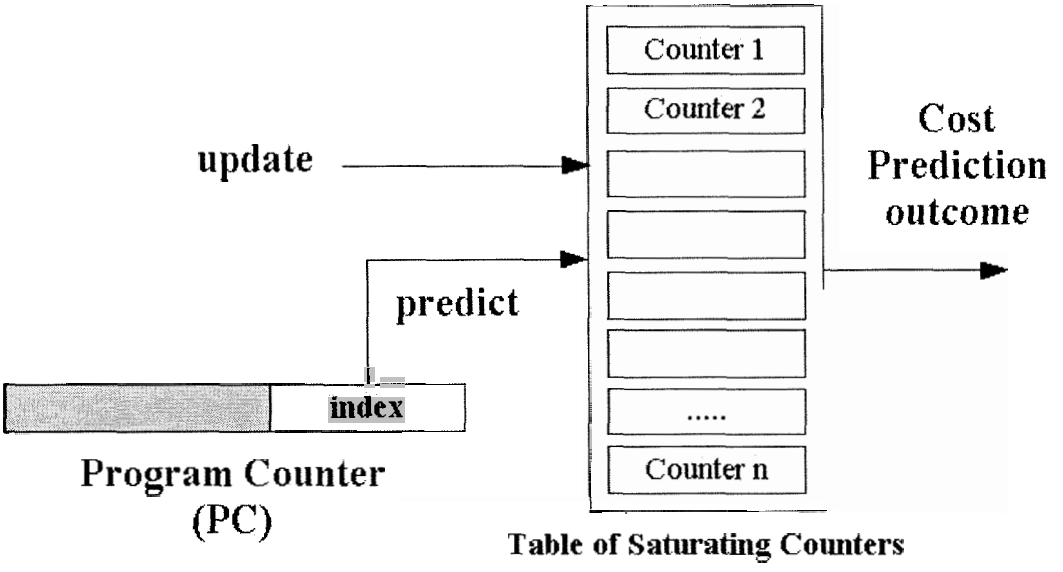



**Figure 5.5 - Architecture of Global cost predictor**

A portion of the program counter (PC), which is labeled as "index" in the figure 5.5, is used to access a table of saturating counters. During the predict stage (decode), if the value of the counter is higher than a given threshold, the associated branch is considered as a "high-cost" one. During the update stage (write-back stage), if a branch is determined to be "high-cost", the associated counter is incremented. Otherwise, it will be decremented or reset to zero depending on the type of the counter (saturating up/down or up/reset).

We achieved the best result with a 16 entry table of 2 bit saturating up/reset counter and threshold of gating of 1. In other words, if two "LCHC" branches are detected, gating is applied. Figures 5.6 and 5.7 report the results achieved by this predictor. In the top two diagrams of figure 5.6, we compare the IPC degradation and power reduction of both

original pipeline gating and the global cost predictor to the case when no gating is applied (no optimization). As illustrated in the top diagram of figure 5.6, on average, the global cost predictor performs better in terms of IPC degradation. The 175.VPR benchmark is an exceptional case, where the IPC is improved when the original pipeline gating mechanism is applied. In the middle diagram of figure 5.6, the pipeline gating performs much better in terms of power reduction by almost 30%. 181.MCF2 and 300.TWOLF show a dramatic reduction of total power compared to the global cost predictor. The bottom diagram of figure 5.6 shows the reduction of the frequency of pipeline gating of the global cost predictor compared to the original pipeline gating mechanism. On average, the frequency of gating is reduced by almost 55%. For floating point benchmarks (figure 5.7), the conventional pipeline gating mechanism provides better performance but achieves similar power reduction.

In summary, for both integer and floating benchmarks, we did not achieve similar performance and power result for the global cost predictor compared to the original pipeline gating mechanism. Nonetheless, for both integer and floating benchmarks, we reduced gating frequency dramatically.

For the floating benchmarks, we can see that the average reflects 179. ART and 183.EQUAKE more than any other benchmarks, since the mis-prediction rate of this benchmark is considerably higher than all other simulated floating benchmarks. In the next two sections, we introduce cost predictors that use the pattern of the last $n$ branches with the goal of achieving better results in terms of balancing among performance degradation, power reduction and gating frequency.

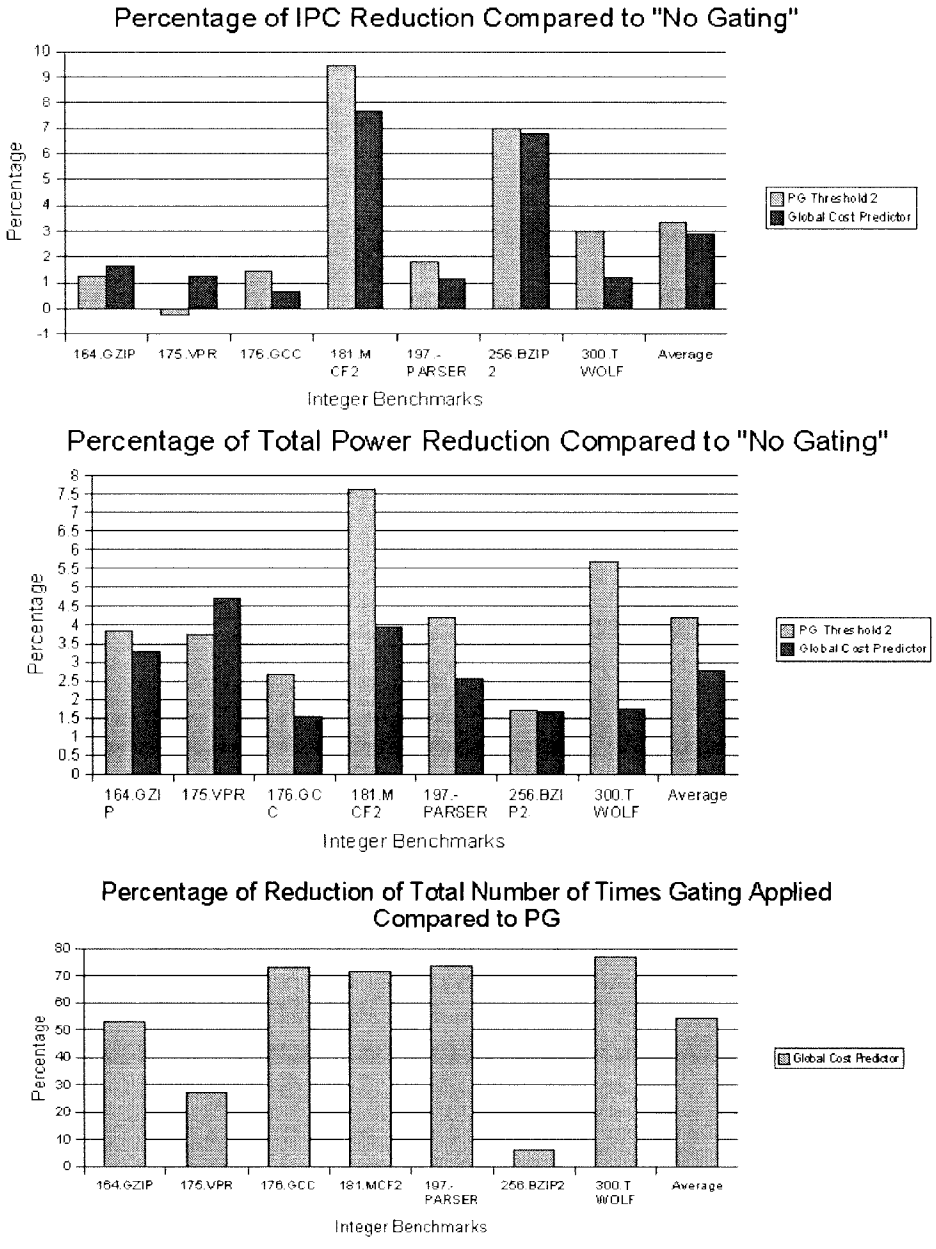



**Figure 5.6 - Comparison between Global Cost Predictor and Pipeline Gating for Integer Benchmarks**

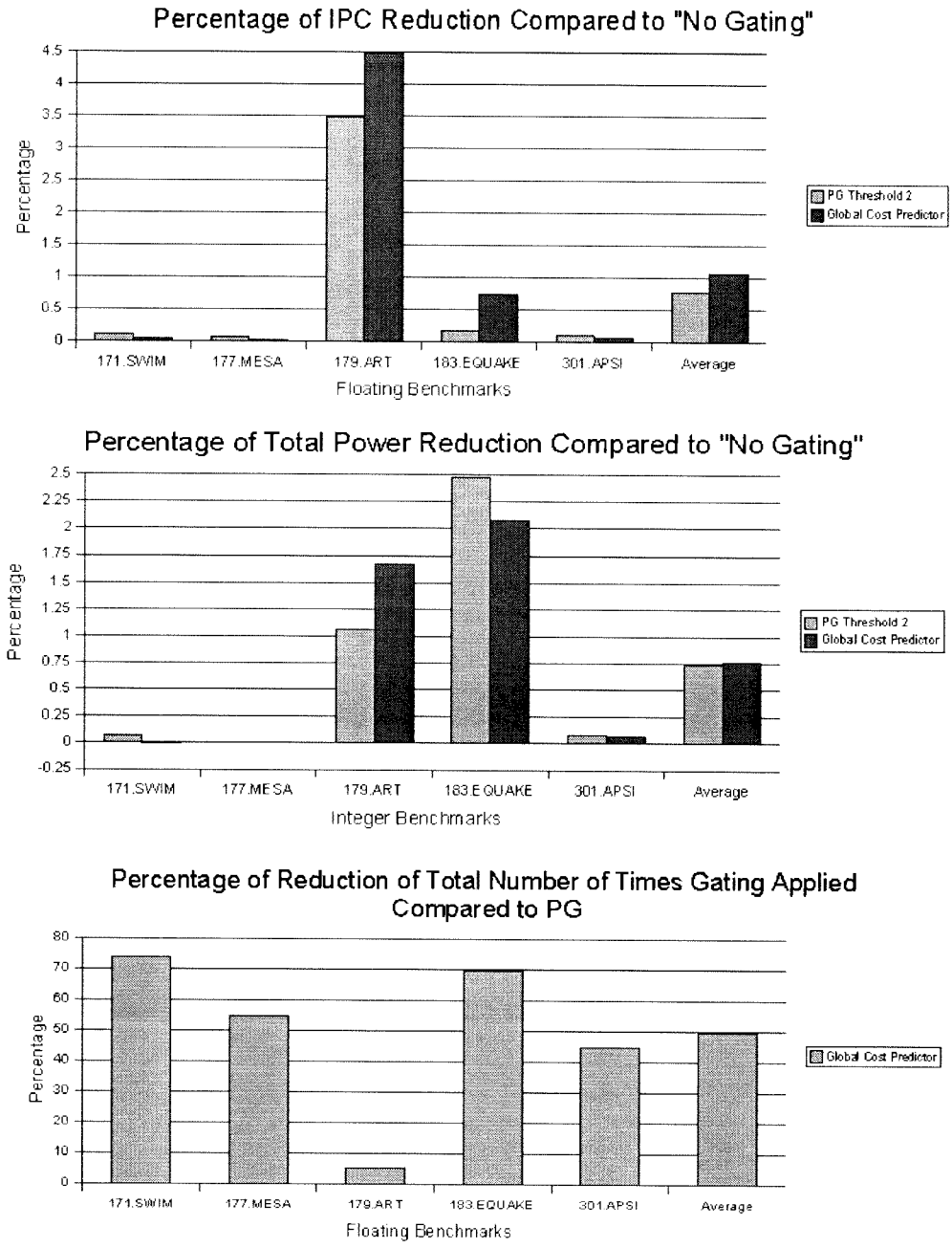



**Figure 5.7 - Comparison between Global Cost Predictor and Pipeline Gating for Floating Benchmarks**

◆ .

### 5.1.2 Global Cost Pattern Predictor

This predictor uses a global $n$-bit shift register, which is used to keep track of the pattern of the last $n$ branches. For every different pattern, there is an $m$ bit saturating counter associated. Figure 5.8 illustrates the architecture of such predictor.

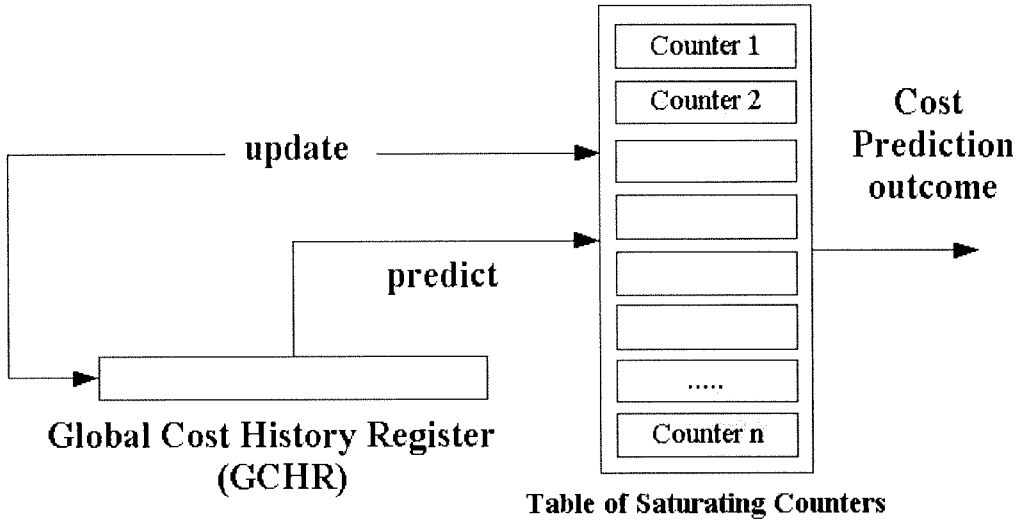



**Figure 5.8 - Local Cost Pattern Predictor**

During the decode stage, the recent global pattern is used to access the table of saturating counters. This is achieved by accessing the GCHR (Global Cost History Register). If the value of the associated counter is higher than a given threshold, that particular branch is considered as "high-cost". The pattern of the GCHR should be saved so that the same entry in the "table of saturating counters" is updated, when the outcome of the branch is determined during the write-back stage. The associated counter in the "table of saturating counters" is updated the same way as the previous predictor. Furthermore, GCHR is updated to reflect the new pattern. Figure 5.9 illustrates how the GCHR is updated.

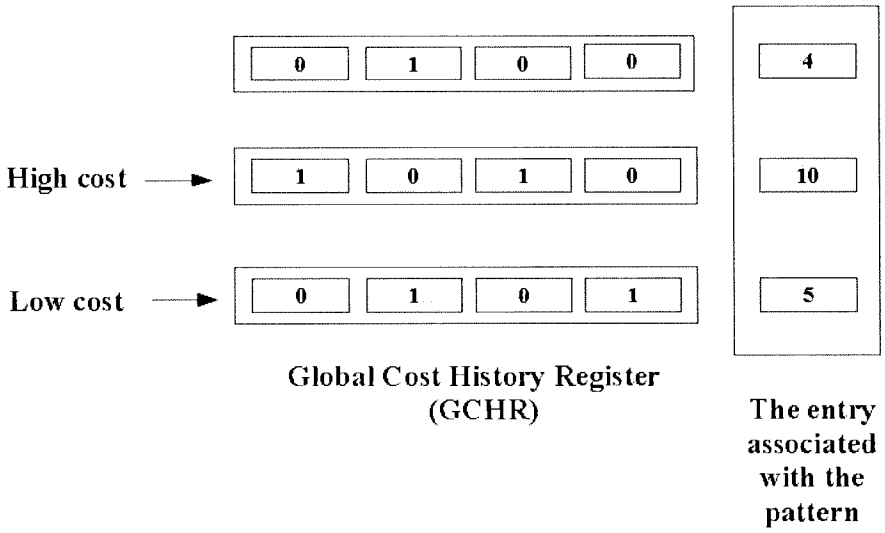



**Figure 5.9 - Example of GCHR Update**

If a branch is determined to be "high-cost", 1 (associated with high-cost) is shifted into the shift register. The value of the shift register determines the entry in the "table of saturating counters". As a result, the size of the GCHR determines the number of entries inside the table. For instance, in figure 5.9, the first pattern is equal to 0100, which is mapped to the $4^{th}$ entry in the "table of saturating counters". During the write-back stage, when the branch is determined to be "high-cost", a value of 1 is shifted into the GCHR. The new pattern is 1010, which is associated with the $10^{th}$ entry in the "table of saturating counters". Similarly, when a branch is determined to be "low-cost", 0 is shifted into the GCHR. The new patter is 0101, which is associated with the $5^{th}$ entry of "table of saturating counters". The maximum size of this table is 16, which is associated with 1111 in the GCHR.

The following diagrams shows the best simulation results for a 4-bit GCHR with a 16 entry table (2 to the power 4) of 2 bit up/reset saturating counters. The same 8-way superscalar processor with 128 entry re-order buffer, cost threshold of 32 and gating threshold of 1 is used.

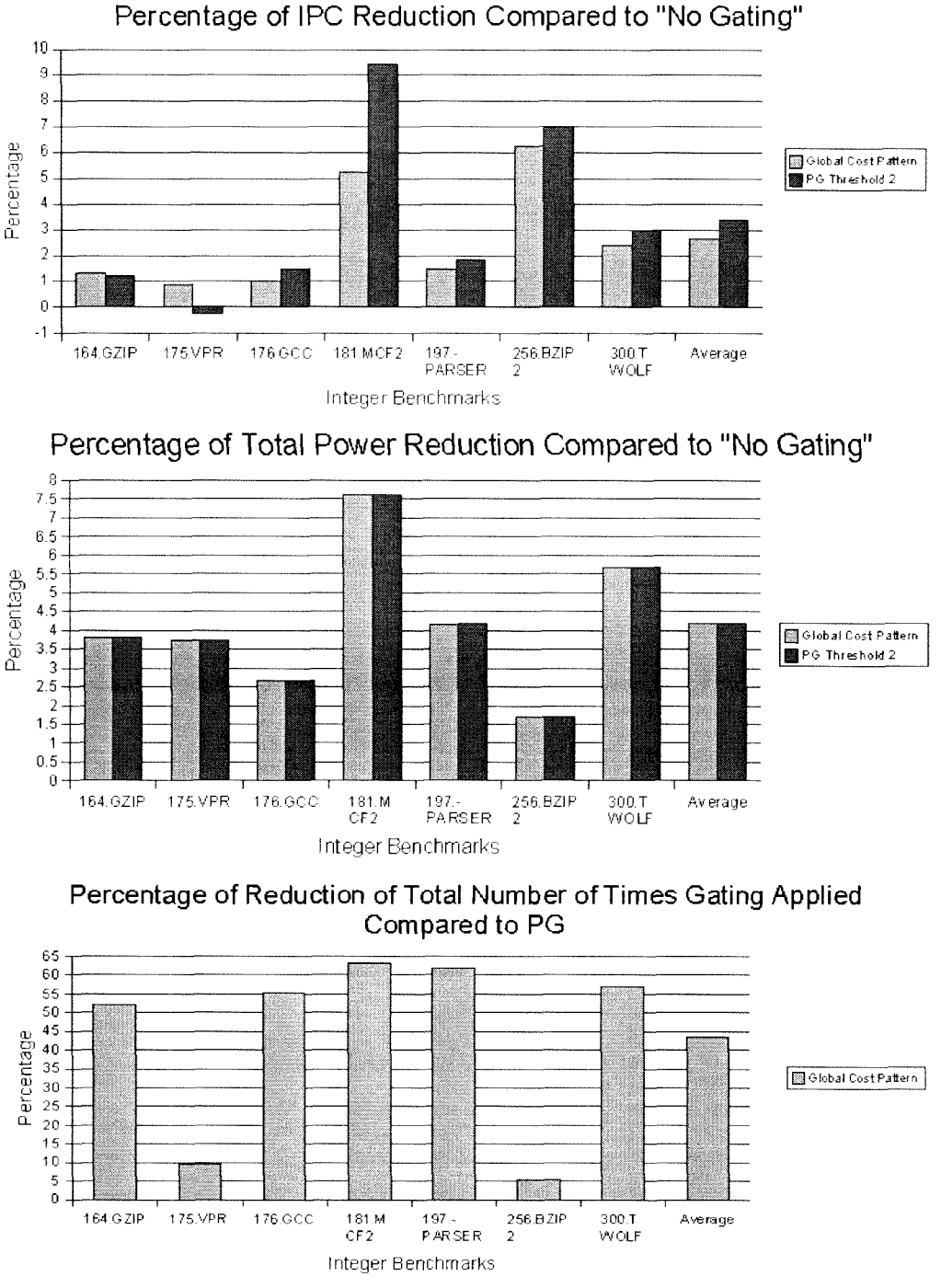



**Figure 5.10 - Comparison between Global Cost Pattern Predictor and Pipeline Gating for Integer Benchmarks**

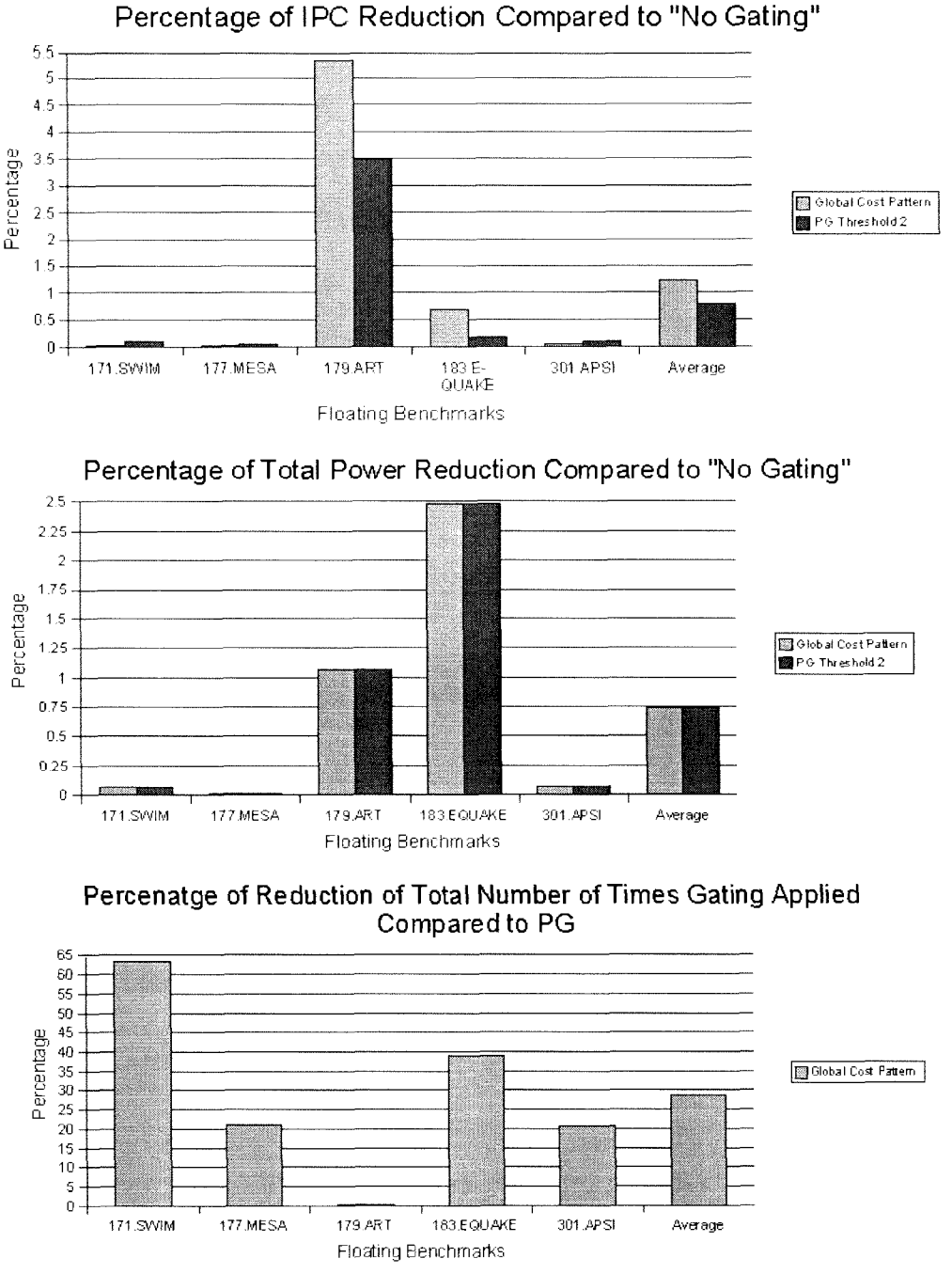



**Figure 5.11 - Comparison between Global Cost Pattern Predictor and Pipeline Gating for Floating Benchmarks**

The first two diagrams of figure 5.10 illustrate that the global cost pattern predictor performs better in terms of IPC degradation, while achieving the same power compared to the original pipeline gating mechanism. In the bottom diagram of figure 5.10, we can observe that the frequency of the gating is reduced by 44%. For floating benchmarks, number of gating has been reduced by almost 30% while the performance and power is almost the same.

This proves that the global pattern is a good measure for predicting the cost associated with a branch. After many simulation, we realized that increasing the size of GCHR to more than 4 bits doesn't have a significant impact on improving the results and proves the fact that cost prediction accuracy mostly depends on the last few branches. In the next section, we are going to introduce a cost predictor that uses the pattern of each branch.

### 5.1.3 Local Cost History Predictor

The local cost history predictor uses two tables. The first table consists of shift registers which keep the patterns that are associated with different PC's. The second table is associated with different patterns in the first table. The local cost history predictor is presented in figure 5.12.

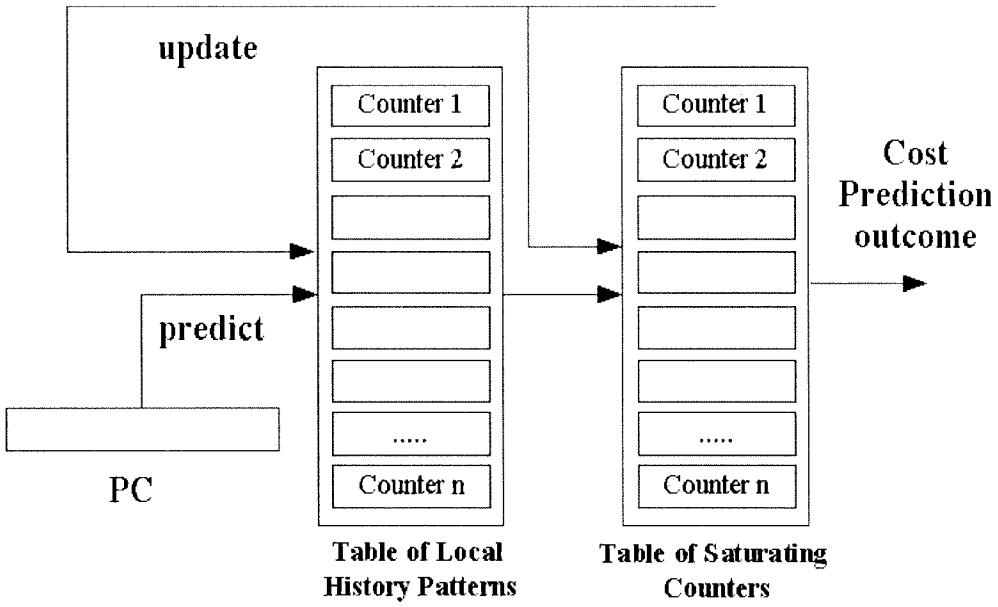



**Figure 5.12 - Local Cost History Predictors**

The only difference between this predictor and the global pattern predictor is that it uses the pattern of last *n* branches of each branch, rather than considering the global pattern of the last *n* branches to access the "table of saturating counters". During the decode stage, a porting of PC is used to access the table of local history patterns. The pattern obtained from the first table is used to access the "table of saturating counters". The prediction and updating of the whole predictor is exactly the same as the global pattern predictor. If the value obtained from the entry in the "table of saturating counters" is higher than a given value, that particular branch is considered as "high-cost". During the write-back stage, both the "table of saturating counters" and "table of local history patterns" are updated.

The problem associated with this predictor is that the architecture of this predictor is more complex. We used the same simulation model used in the previous sections to evaluate this predictor. The best result was achieved by two tables of size 16 and two bit up/down for the "table of saturating counters". Figure 5.13 and 5.14 show power degradation, power reduction, and gating frequency obtained for this predictor for integer and floating benchmarks.

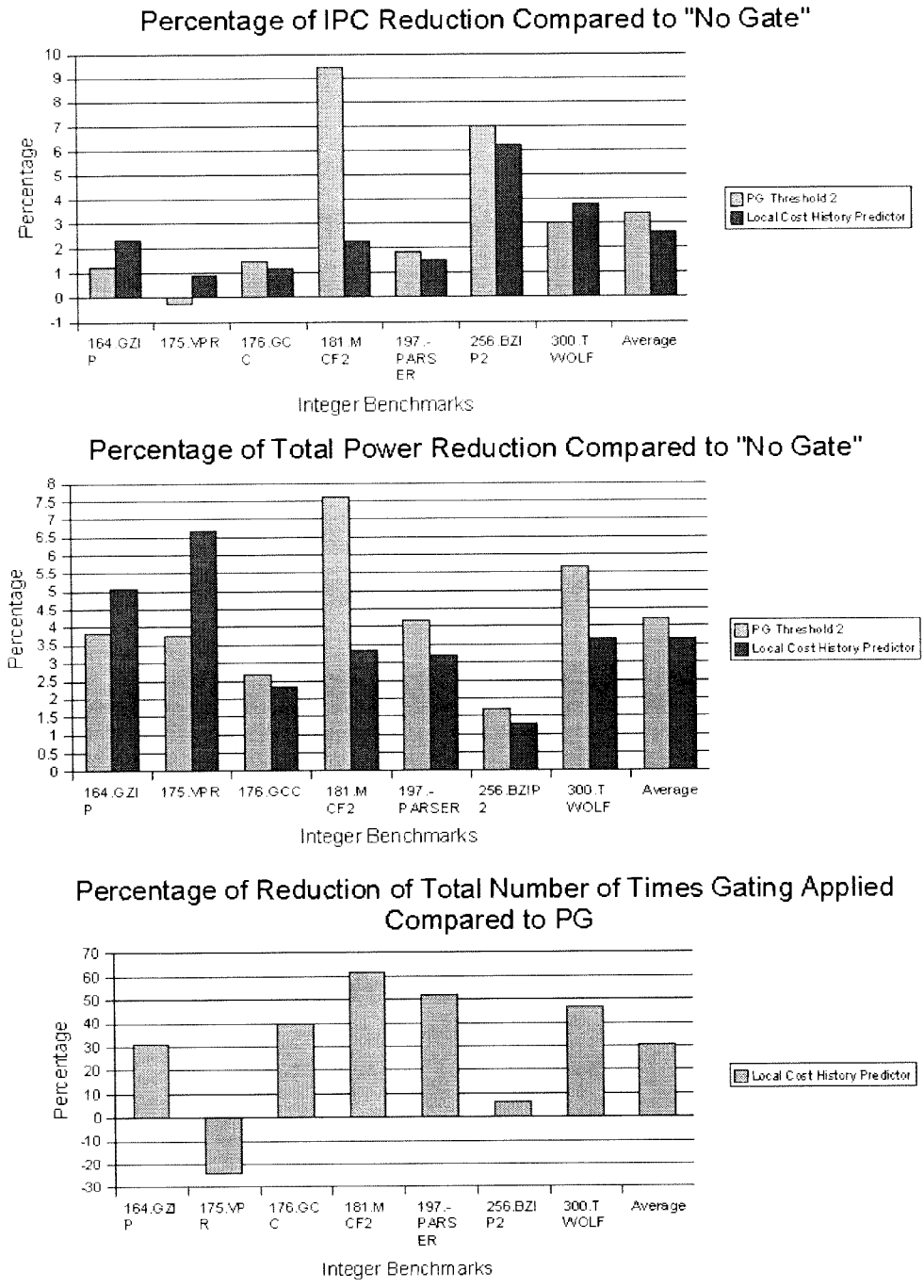



**Figure 5.13 - Comparison between Local Cost History Predictor and Pipeline Gating for Integer Benchmarks**

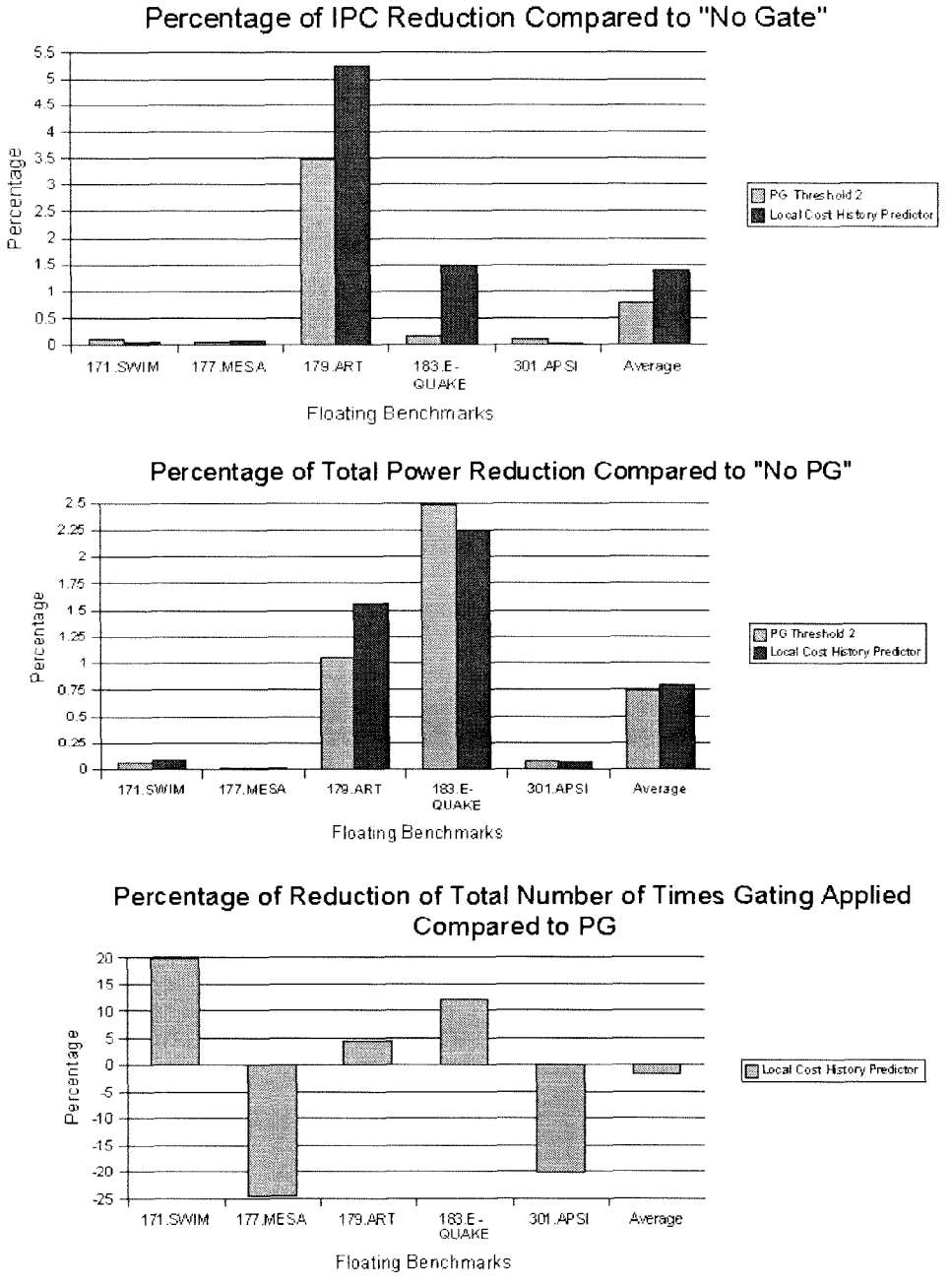



**Figure 5.14 - Comparison between Local Cost History Predictor and Pipeline Gating for Floating Benchmarks**

Figure 5.13 illustrates that we can achieve almost the same IPC degradation and power

reduction for the local cost history predictor compared to the pipeline gating mechanism. However, the frequency of gating is very dependent to the type of the benchmark. For instance, the frequency of gating is increased by almost 23% for the 175.VPR benchmark. This can also be seen for the floating benchmarks. Both 177.MESA and 301.APSI increase the gating frequency.

In summary, we can conclude the following two observations about the local cost history predictor:

(1)The local cost history predictor doesn't do a good job in terms of balancing among performance degradation, power reduction and gating frequency.
(2)The result of the cost history predictor is heavily dependent to the particular benchmark.

Moreover, we can conclude the following observation from the last three cost predictors introduced so far.

Global cost pattern predictor is the best cost predictor in terms of balancing among IPC degradation, power reduction and gating frequency. It uses the global pattern of last few branches to determine the cost associated with a branch.

In the next section, we show that how cost can be used to make a better confidence estimator.

## 5.2 Mechanism #2 – High Cost Low Confidence (HCLC) Confidence Estimator

As explained before, high-cost branches are more responsible for both performance degradation and waste of power due to mis-prediction. As a result, we used the cost associated with high-cost branches to design a new confidence estimator. Figure 5.15 illustrated the architecture of such mechanism.

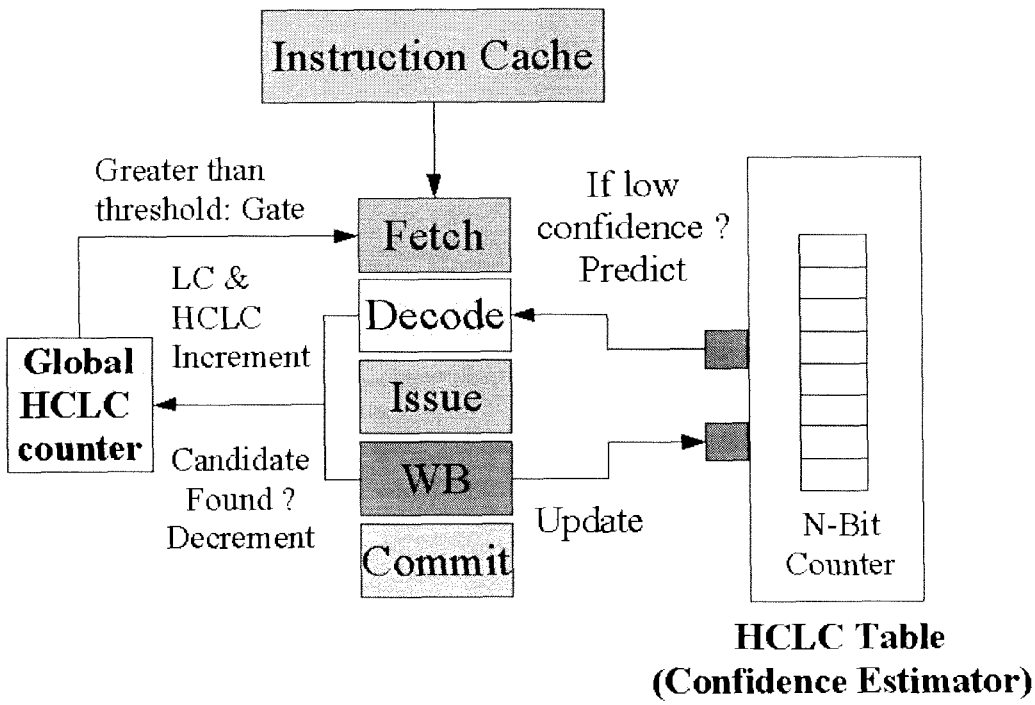



**Figure 5.15 - "High cost / Low Confidence " Confidence Estimator**

During the write-back stage, the cost associated with a particular branch is determined. If a branch is high-cost, the HCLC table is accessed. If a branch is low-confidence, the corresponding counter is incremented, otherwise it will be decremented or reset according to the counter type (up/down or up/reset saturating counter).

In essence, the HCLC table keeps the history of confidence of the previous *n* branches for

"high-cost branches". During the decode stage, if a branch is determined to be "low-confidence" and the corresponding entry in the HCLC is "high-cost" and "low confidence", the global LCHC counter is incremented. In other words, if a branch is "low-confidence" and previously has been "high-cost and low-confidence", there is a high possibility that this branch is a "high-cost / low-confidence branch". If the number of "high cost / low-confidence" branches is higher than a given threshold, we apply pipeline gating.

To evaluate this technique we used the same simulation model used earlier. The size of the table is 16 with a 1 bit counter. For integer benchmarks both the IPC and power remains constant, while gating frequency is reduced by almost 25%. For floating point benchmarks, performance is maintained while the total power and gating frequency is reduced by almost 25%.

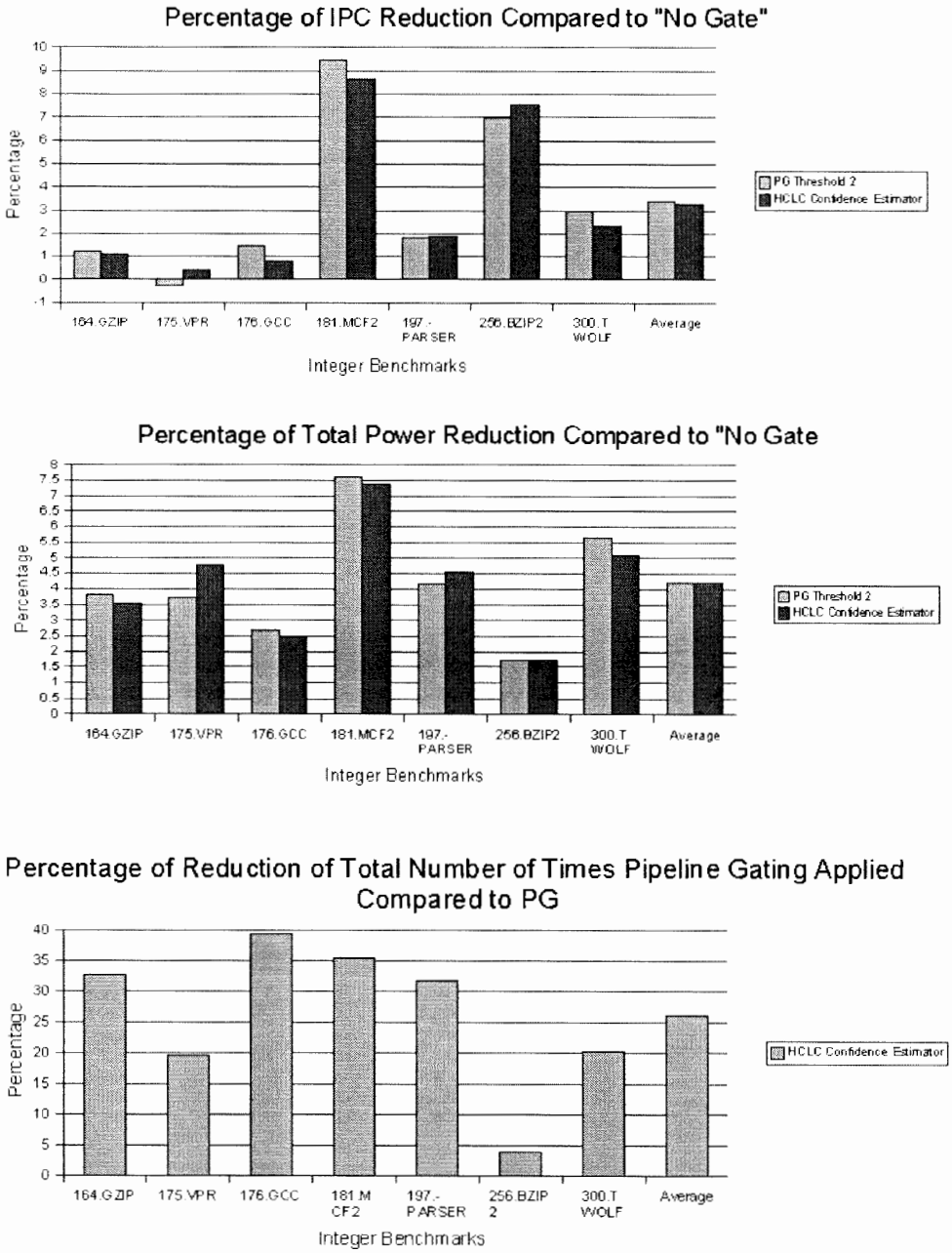



**Figure 5.16 - Comparison between HCLC Confidence Estimator and Pipeline Gating for Integer Benchmarks**

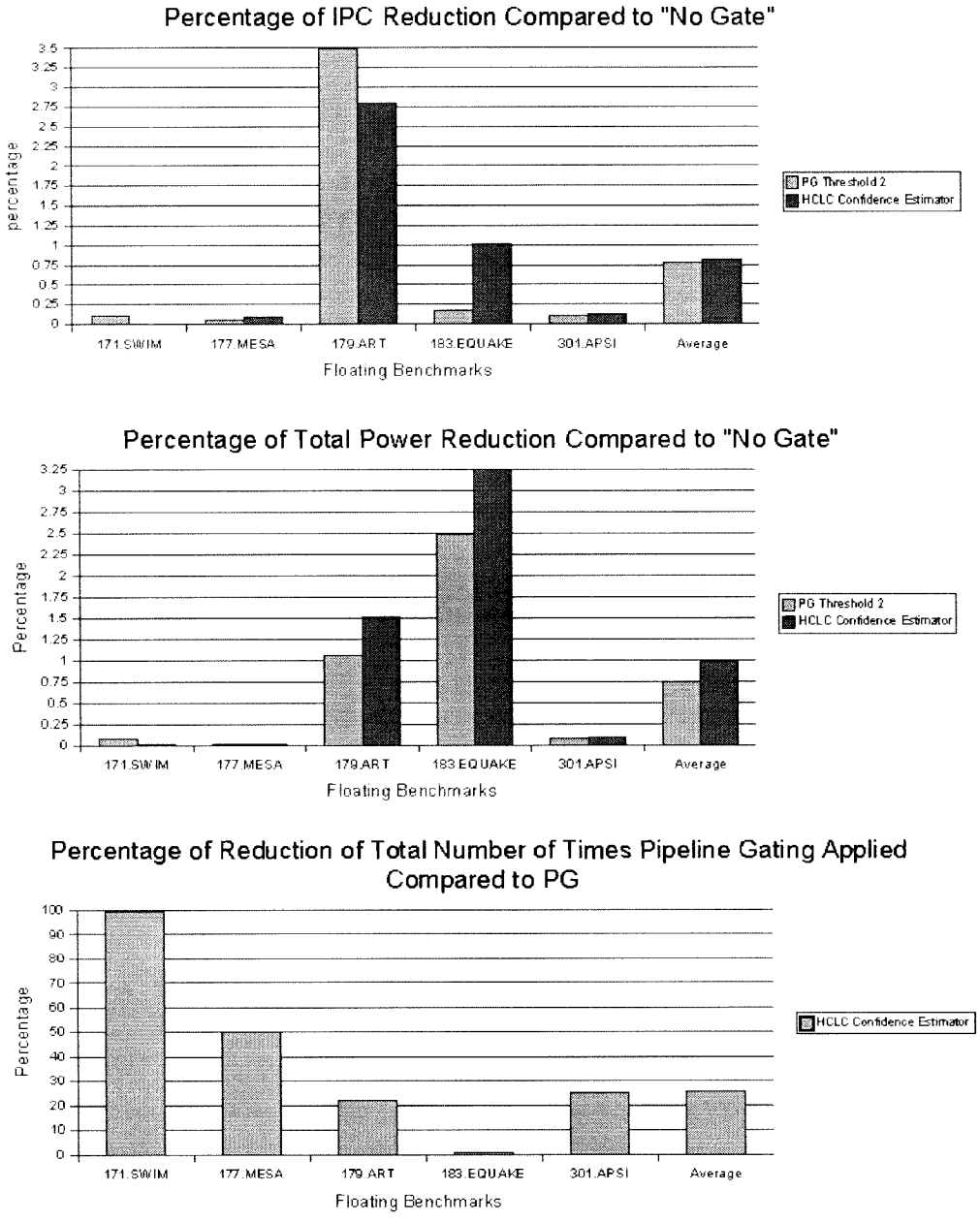



**Figure 5.17 - Comparison between HCLC Confidence Estimator and Pipeline Gating for Floating Benchmarks**

## 5.3 Combination of Methods

In this section, we want to see if we can use the combination of the HCLC confidence estimator and the global cost pattern predictor together. As presented earlier, the global cost pattern predictor is the best predictor in terms of balancing among IPC, power and gating, and frequency of gating. Figures 5.18 and 5.19 shows the result of these two mechanisms combined. Reportedly, such methods do a very good job in terms of IPC and power balancing but not so much in terms of reducing gating frequency. For instance, the gating frequency is increased by almost 58% for the 175.VPR benchmark in figure 5.18. The same applies for the 179.ART in figure 5.19.

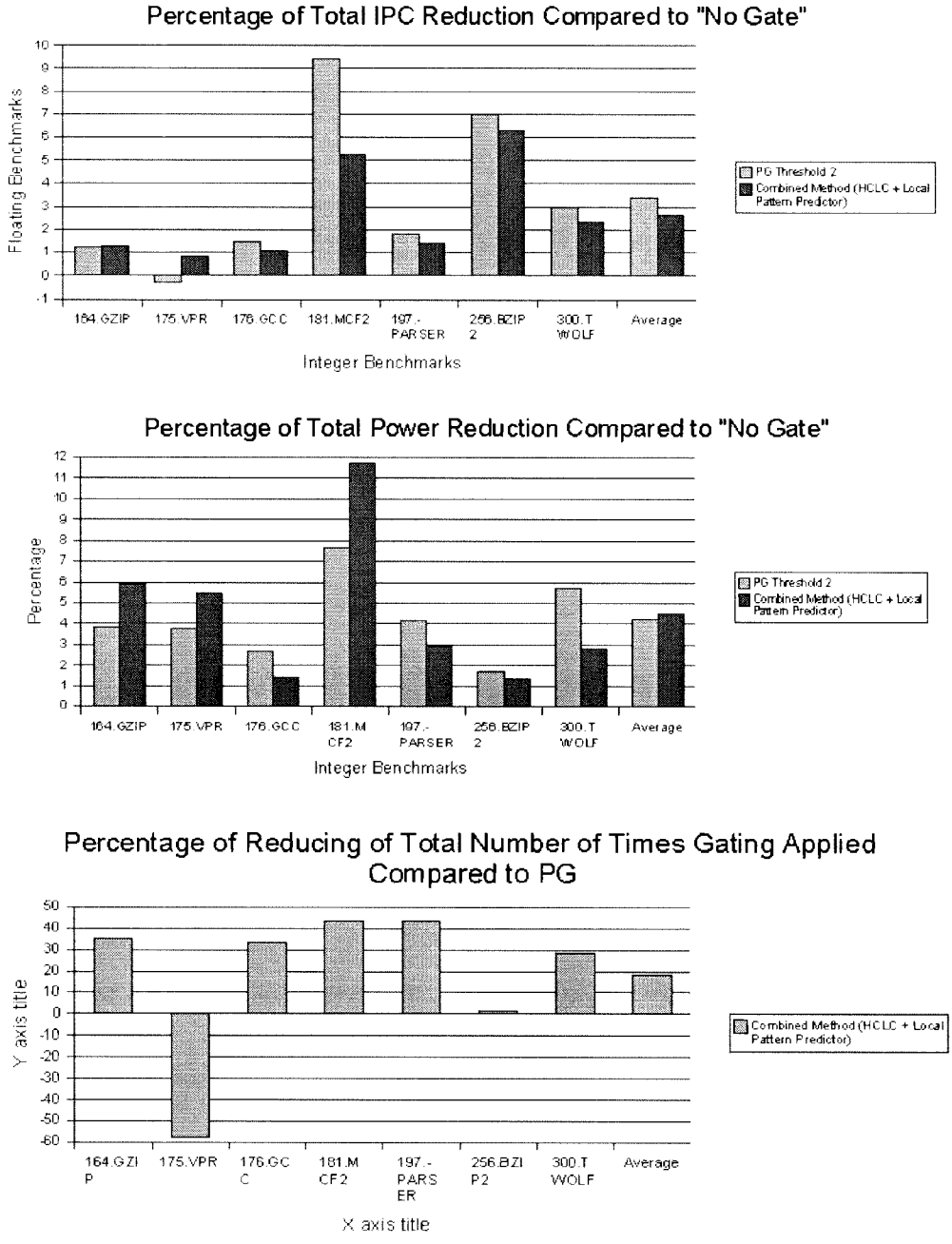



**Figure 5.18 - Comparison between Combined Method and Pipeline Gating for Integer Benchmarks**

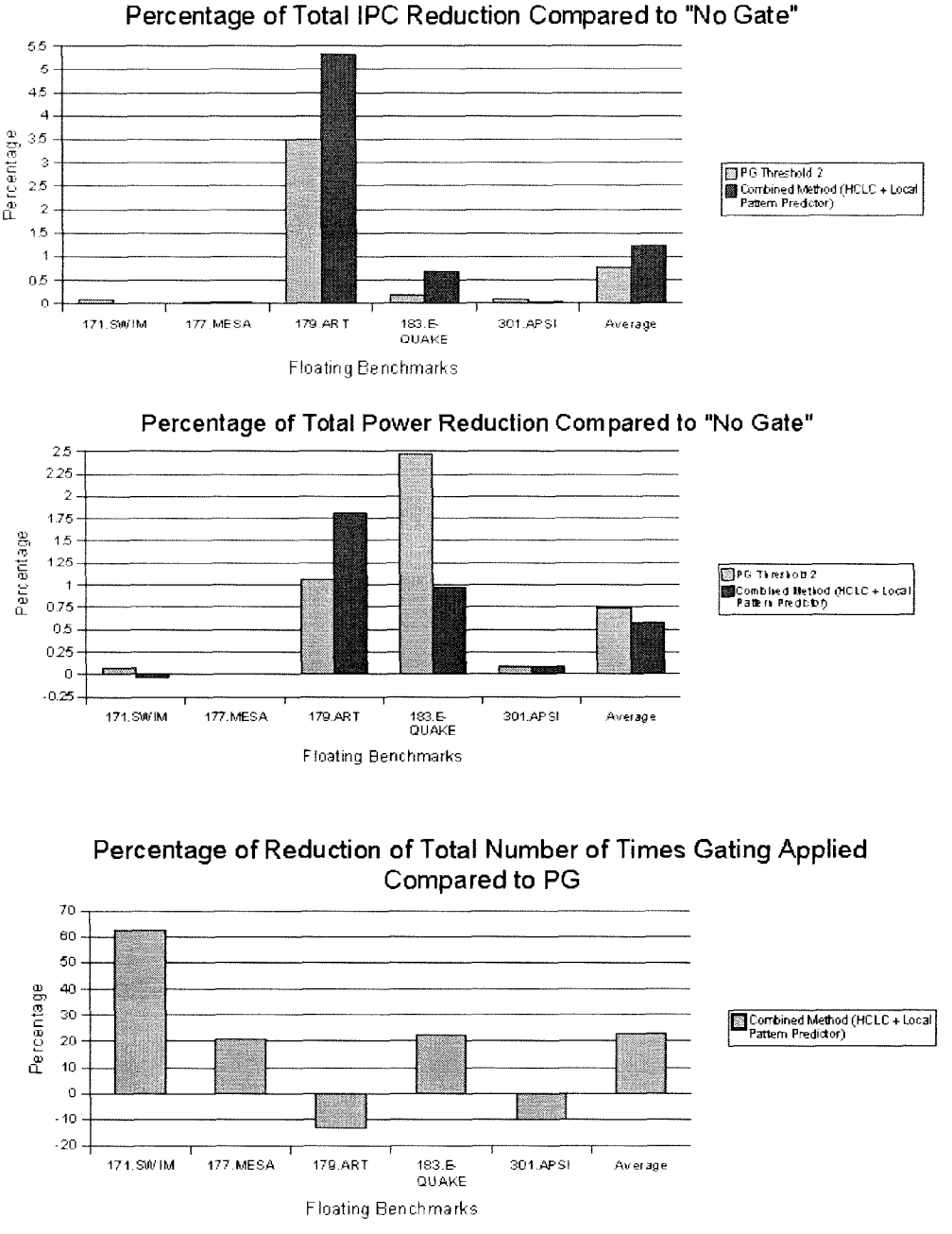



**Figure 5.19 - Comparison between Combined Method and Pipeline Gating for Floating Benchmarks**

## 5.4 Dynamic Threshold of Cost

In the previous simulations, a fixed cost threshold of 32 was chosen. However, we also propose an algorithm based on changing the cost dynamically during the run-time.

We created four different cost regions and associate a counter with each of these regions in the following manner:

Region 1: Cost associated with branches that flush between 0 and 31 instructions.
Region 2: Cost associated with branches that flush between 32 and 63 instructions.
Region 3: Cost associated with branches that flush between 64 and 95 instructions.
Region 4: Cost associated with branches that flush more than 96 instructions.

Every time a branch was flushed the cost associated with that branch is determined and the associated counter is updated. Every 8k cycles, we determine which category of cost is more responsible for flushing branches. We follow the following to decide the threshold dynamically:

If region 1 is responsible: set the threshold of cost to 16.
If region 2 is responsible: set the threshold of cost to 32.
If region 3 is responsible: set the threshold of cost to 64.
If region 4 is responsible: set the threshold of cost to 96.

The same results can be achieved compared to the static threshold of cost illustrated in previous sections.

## 5.5 Dynamic Threshold of Pipeline Gating

In all of the previous simulations, the pipeline gating threshold was set to one. In other words, if two branches satisfied the constraint, the gating was applied. We also introduce an algorithm that can change this threshold dynamically.

A typical threshold selection is presented below.

If the number of entries in the re-order buffer is
between 00 - 31: set the threshold of PG to 3.
between 32 - 63: set the threshold of PG to 2
between 64 - 95: set the threshold of PG to 1.
greater than 96 : set the threshold of PG to 0.

The idea is that if the re-order buffer is only a quarter full, applying the pipeline gating may stop the flow of instruction into the pipeline, which causes the loss of performance. Therefore, the PG threshold is set to 3 so that more instructions enter the pipeline. If the re-order buffer is almost full, entering a new mis-predicted branch could flush the whole buffer which is very power wasteful in terms of mis-prediction. Consequently, additional suspicious instructions are not allowed to enter the pipeline by applying the gating mechanism. The same results can be achieved compared to the static threshold of pipeline gating illustrated in previous sections.

## 5.6 Effect of different parameters for the Cost Table

In this section, we show how variations in the predictor configurations impacts IPC, power and gating frequency. In particular we study the following:

1) The threshold for pipeline gating (THR-PG): As soon as number of branches that follow some constraint in the decode stage is greater than THR-PG, the gating is applied.
2) The size/type of the counters of the LCHC table: In our simulation models, we used both saturating up/down and up/reset counters. Saturating counters don't roll back when they reach to maximum and minimum. Once a branch is high cost, the value of the counter increases up to the max and if it is low cost it decreases up to its minimum for the up/down counters. The same applied to up/reset counters except that if a branch is low-cost the value of the counter is reset to zero.
3) The threshold for defining the cost (THR-cost): This is the threshold that determines if a branch is low or high cost. In most of our simulations, THR-cost of 32 almost always worked the best.

The first two parameters are very important in balancing among the IPC, power and gating frequency. Figure 5.20 shows the results for the 164.gzip benchmarks for the global pattern predictor. A similar behavior is observed for other cost predictors.

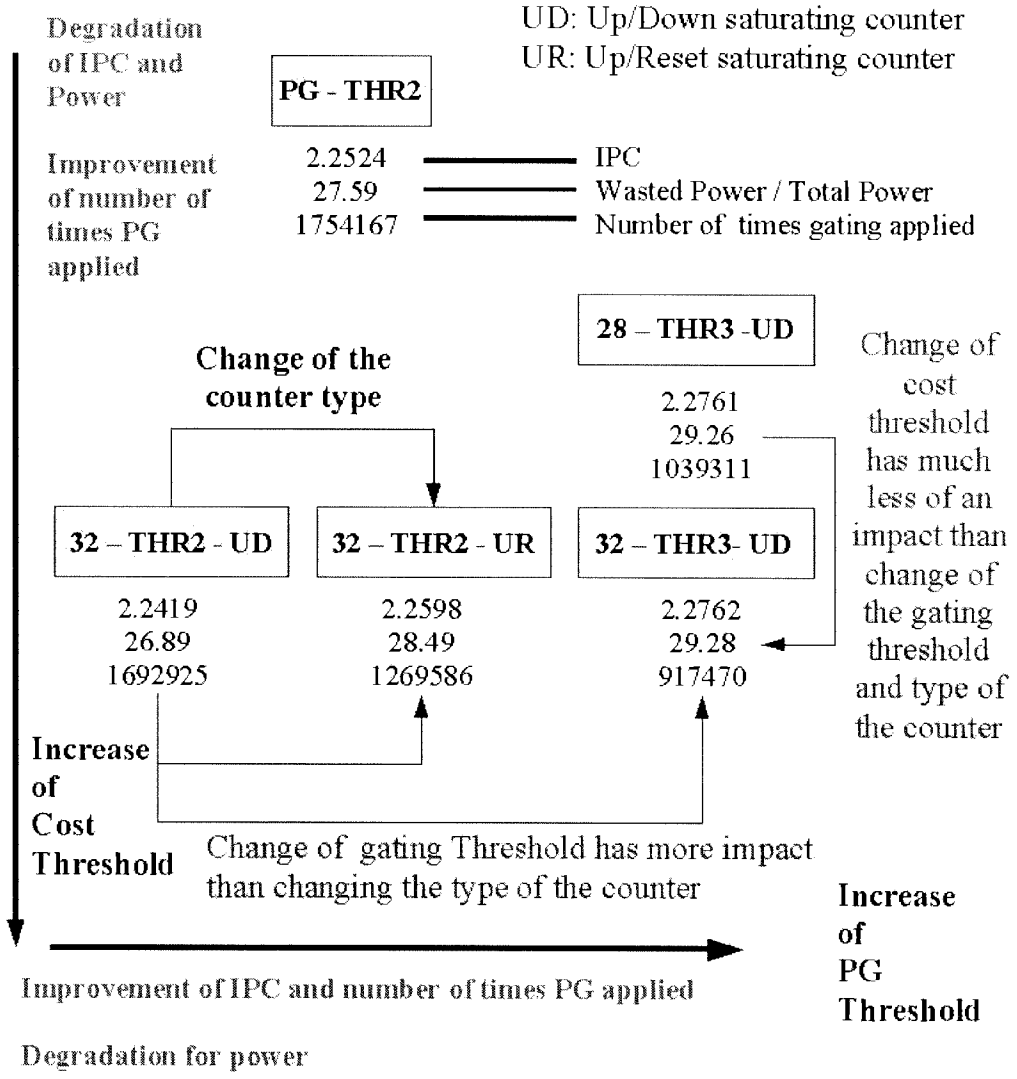



**Figure 5.20 - Effects of different parameters for the cost predictor:**

On the top of the figure 5.20, conventional pipeline gating (PG) parameters are shown. Threshold of 2 is chosen, which means that if the number of low-confidence branches

exceeds two, PG is applied. The rest of the results are for local cost predictors with different parameters. In the figure, the rectangles are titled with this format:
Number-THR(Number) – Type of the counter.
The first number is the cost threshold used to distinguish between high/low cost branches (THR-cost). The number after the THR, is the threshold applied for the pipeline gating mechanism (THR-PG). For instance, consider 32-THR2-UD. This means that branches that flush less than or equal to 32 instructions are considered as "low-cost" and once there are more than 2 instructions in the pipeline that are "low-confidence and high cost", pipeline gating is applied. The type of the counter is up/down saturating counter.

By moving horizontally from right to left, the threshold for defining low/high cost branches (THR-cost) remains constant, while the threshold for pipeline gating is reduced (THR-PG). By doing so, we are increasing the gating frequency. Once gating is applied, the last LCHC branch has to be resolved, in order to resume the flow of instructions again. This will lead to the reduction of total the number of mis-predicted branches (due to lower number of speculations), which is the cause of the waste of power in the pipeline. This explains why power reduction due to mis-prediction is improved. However, since we are also stopping the branches that are not going to be mis-predicted, the IPC will also decrease.

By moving vertically from top to bottom, THR-PG remains constant, while the THR-cost is increased. As reported, THR-cost has a lower impact on IPC and power compared to THR-PG. By increasing the THR-cost, number of branches that will be identified as high-cost will decrease, which leads to a decrease in gating frequency.
It could also be observed that changing the type of the counter from an up/down has more impact (IPC, power and gating frequency) than changing THR-cost and less impact than changing the THR-gate.

# Chapter 6 – Using Cost for Reducing Power in a Combined Branch predictor

As explained in section 2.3, the combined branch predictor provides high accuracy in predictions and it is used in many commercial processors such as Alpha 21264. However, the power for the combined branch predictor is much higher than both local and global predictors, since there are three tables that have to be accessed for both prediction and update of each branch. We propose a new optimized structure based on cost estimation to reduce the power of the branch predictor, while maintaining accuracy.

For the simulation model, we used a combined branch predictor that uses gShare as its global predictor and bimodal as its local predictor. In this configuration, if the cost associated with a branch is low, only the gShare predictor is used. We assume that if the cost associated with a branch is high, a better prediction accuracy is required and hence the combined branch predictor is accessed. Figure 6.1 illustrates the structure of such optimized predictor.

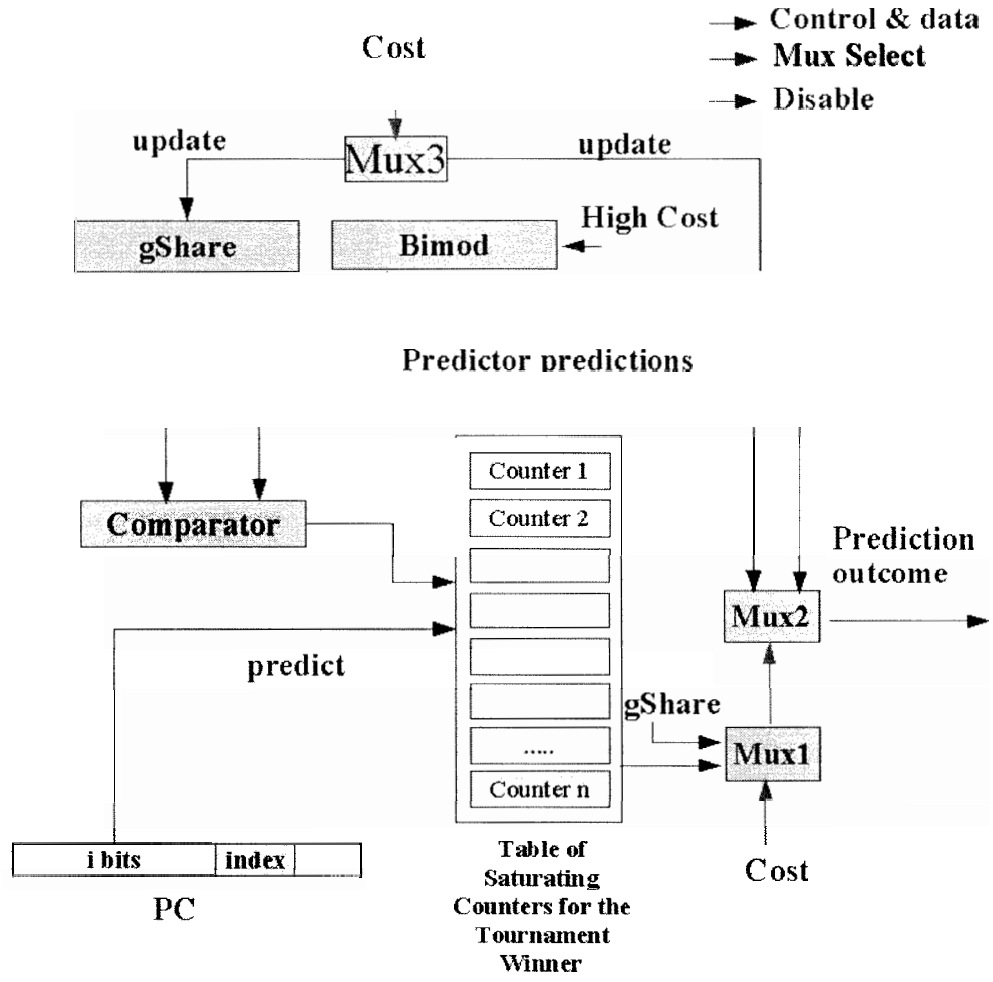



**Figure 6.1 - Optimized Combined Branch Predictor Based on Cost**

As illustrated in figure 6.1, when a high-cost branch is encountered, the bimodal branch predictor is disabled during both prediction and update stage. In the case of a "high-cost" branch, Mux1 selects the combined branch predictor; otherwise, the gShare branch predictor is selected during the prediction. During the update stage, the gShare predictor is only updated for low-cost branches and both predictors are updated for high-cost branches. We used the global pattern predictor for our cost predictor with the cost threshold of 16 and 32. Unlike the cost predictor in the previous chapter, it keeps tack of "low-cost" branches. When a branch is "low-cost", the associated counter in the cost

estimator is incremented (Table 4.1).

Figure 6.2 and 6.3 show the result of the simulation for IPC (Instruction Per Cycle) degradation, total number of accesses to gShare and combined predictor, and branch prediction power reduction. In the top diagram of figure 6.2, we can see that the average percentage of IPC degradation for the cost threshold of 16 is much less than cost threshold of 32. When the cost threshold is 16, only those branches that are highly "low-cost" access the gShare predictor, which contain lower portion of all branches. The two middle diagrams of the figure 6.2 illustrate that the percentage of access to the combined branch predictor (has higher accuracy) for the cost threshold of 32 is higher for cost threshold of 16. However, the branch prediction power reduction for cost threshold of 32 is higher than cost threshold of 16, since more branches access the gShare for the cost-threshold of 32.

For integer benchmarks, an average of 0.3% and 0.78% performance loss is detected for cost threshold of 16 and 32, while 20.4% and 41.8% branch prediction power is reduced respectively. For floating benchmarks, performance is maintained, while the power of the branch prediction unit is reduced by 24.3% and 48% for cost thresholds of 16 and 32 respectively. As a result, we conclude that cost threshold of 16 can be used for power optimization in the combined branch predictor with negligible performance loss.

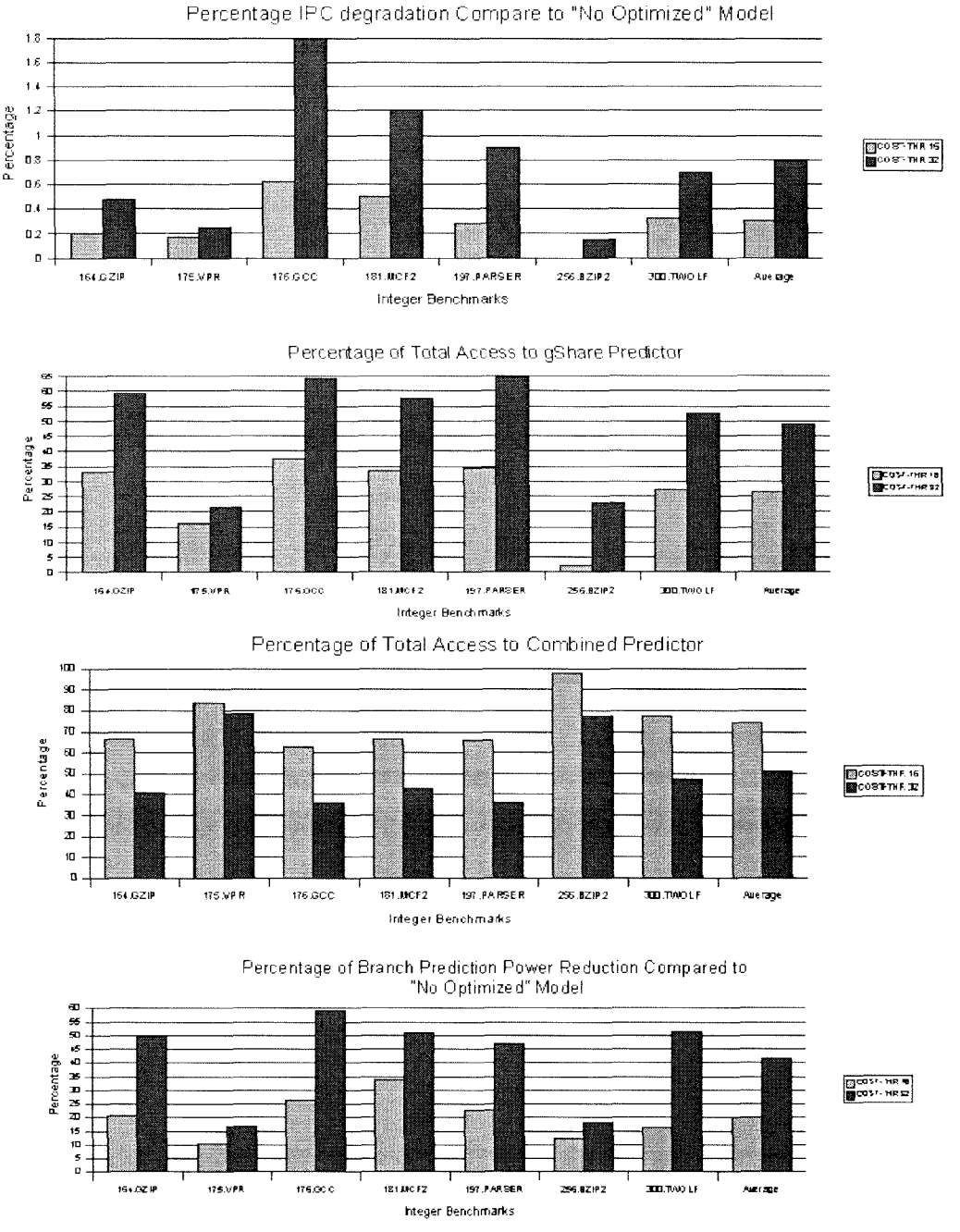



**Figure 6.2 - Simulation Result for Cost Optimization of Combined Predictor for Integer Benchmarks**

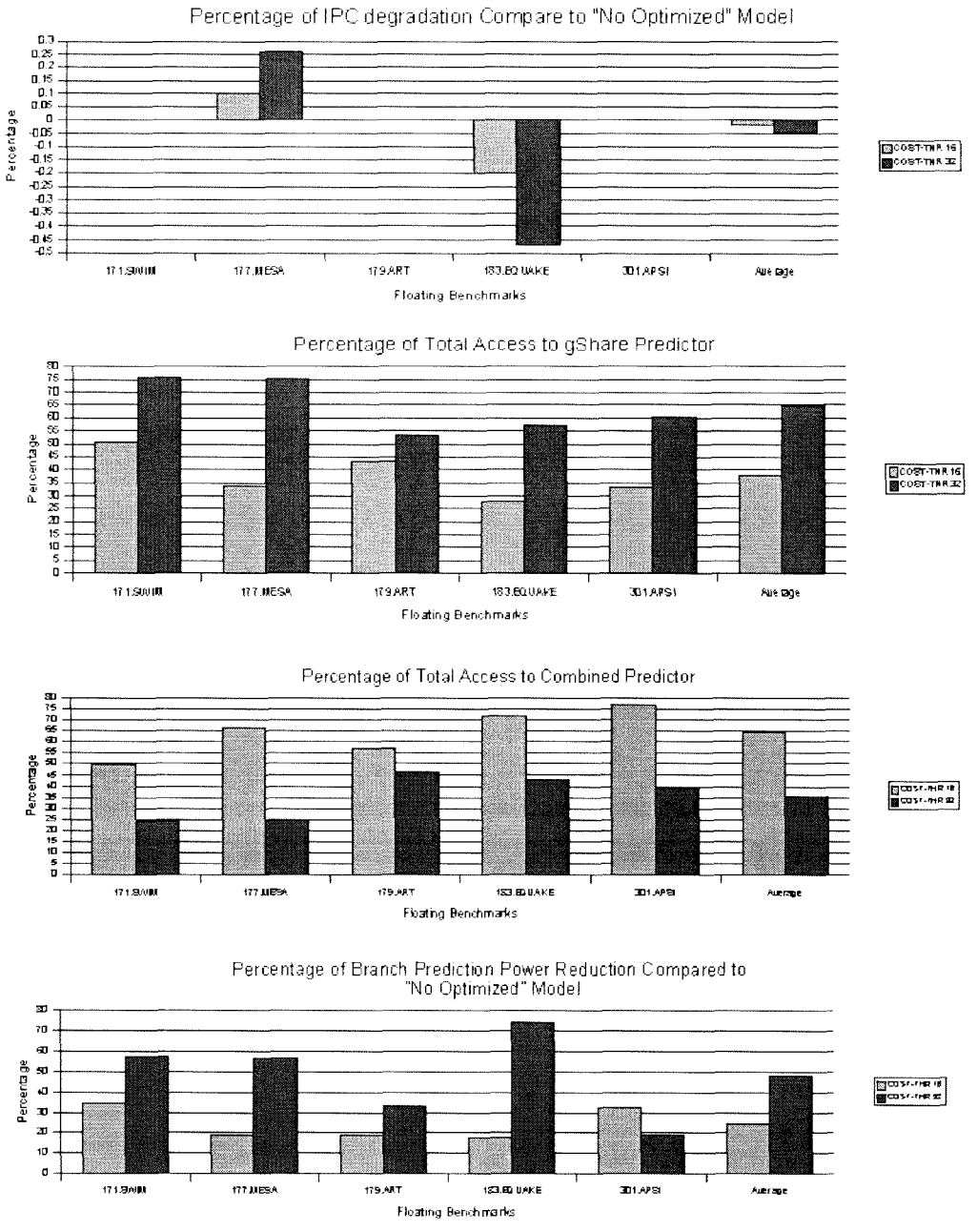



**Figure 6.3 - Simulation Result for Cost Optimization of Combined Predictor for Floating Benchmarks**

# Chapter 7 – Conclusion

The aim of this research is to reduce the wasted power due to mis-prediction for those branches that have more contribution to it. The following objectives are met.

- ➔ We introduced a new method of assigning cost to branches so that we can filter those branches that have higher impact on power waste due to mis-prediction. We defined cost as the number of instructions that are flushed during the mis-prediction. Furthermore, we introduced a one bit representation of cost: "low-cost" and "high-cost" branches.

- ➔ We showed that the cost associated with a mis-predicted branch branch is highly predictable. Moreover, we showed that while the frequency of "high-cost" branches is by far lower than the frequency "low-cost" branches, they are responsible for most of the power waste during mis-prediction in a superscalar processor.

- ➔ We implemented three different cost predictors and we show how they can be applied to reduce the gating frequency for clock gating (pipeline gating) mechanism. In particular, the global cost pattern predictor is the best one and we can achieve a reduction of the gating frequency by 45% while achieving the same performance and wasted power due to mis-prediction.

- ➔ The global cost pattern predictor is used to reduce power in the combined branch predictor by 20% while achieving no performance loss in almost all the benchmarks.

- ➔ The branch predictor consumes around 5% of a total power of a processor with a table of 4k [32]. As a result, with the table size of 16 entries for the global cost pattern predictor, we expect that power overhead associated with such predictor is negligible.

## 7.1 Future Work

In this research, we use a simple methodology to define the concept of cost for mis-predicted branches to have more control for speculation. It is possible to introduce a more accurate definition of cost that can precisely estimate the wasted power during mis-prediction. For instance, two branches could be predicted as "high-cost" according to our definition, while one can have more contribution in the total wasted power in terms of using hardware resources. Furthermore, we can use more categories for the cost associated with branches. For instance, we can have "low-cost", "middle-cost" and "high-cost" branches and apply different power optimization techniques accordingly. It is important to realize that as the complexity of processors increases, the penalty associated with a mis-prediction will have higher impact both on performance degradation and total power. It is the aim of this research to start new techniques to have more control for speculation using the cost associated with branches in order to minimize the power waste during mis-prediction.

# Bibliography

[1] James E. Smith and Gurindar S. Sohi, "The microarchitecture of superscalar processors", *Proceedings of the IEEE*, vol. 83, pp. 1609 - 1624, Dec. 1995.

[2] Dezso Sima, Terence Fountain, and Peter Kacsuk, *Advanced Computer Architectures, a Design Space Approach*, Addison Wesley Longman, 1997.

[3] Kai Hwang, *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., 1984.

[4] Scott Gifford, Chien-Wen Huang, Zimin Yang, and Cong Yu, "A Comprehensive front-end architecture for the verisimple alpha pipeline", http://citeseer.ist.psu.edu/gifford03comprehensive.html.

[5] Scott McFarling, "Combining branch predictors," Digital Equipment Corporation WRL Technical Note TN-36, June 1993.

[6] E. Jacobsen, E. Rotenberg, and J.E. Smith. "Assigning Confidence to Conditional Branch Predictions", *Internation Symposium on Microarchitecture*, Pages 142-153, December 1996.

[7] S. Manne, A. Klauser, and D. Grunwald, "PipelineGating: Speculation control for energy reduction," *International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[8] S. Manne, A. Klauser, and D. Grunwald, "Branch prediction using selective branch inversion", *Proceedings of the International Conference on Parallel Architectures and Compilation Techniquesi,*, 1999.

[9] Juan L. Aragón , José González , Antonio González , and James E. Smith, "Dual path instruction processing", *Proceedings of the 16th international conference on Supercomputing*, June 22-26, 2002, New York, New York, USA.

[10] Margaret Martonosi, Vivek Tiwari, and David Brooks, "Wattch: A framework for architectural-level power analysis and optimizations", *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, June 2000.

[11] John Henning, "SPEC CPU 2000: Measruring CPU performance in the new millennium", *IEEE Computer Society Press*, Pages 28-35, Vol 33, Issue 7, 2000.

[12] Tejas Karkhanis, James E. Smith, and Pradip Bose, "Saving energy with just in time instruction delivery", *Proceedings of the international symposium on Low power*

*electronics and design*, Pages 178-183, 2003.

[13] A. Lyer and D. Marculescu, "Power aware microarchitecture resource scaling", *Proceedings of the conference on Design, automation and test in Europe*, Pages: 190 – 196, 2001.

[14] Hai Li, Swarup Bhunia, Yiran Chen, T. N. Vijaykumar, and Kaushik Roy, " Deterministic clock gating for microprocessor power reduction", *HPCA*, 2003.

[15] Dirk Grunwald, Arthur Klauser, Srilatha Manne, and Andrew Pleszkun, "Confidence estimation for speculation control". *Proceeding 25th annual International Symposium on Computer Architecture. SIGARCH Newsletter*, Barcelona, Spain, June 1998. ACM.

[16] John Hennessy, and David Patterson, *Computer Architecture a Quantitative Approach*, Morgan Kaufmann publishing, Inc, 1996.

[17] P.Y. Change, E.Hao and Y.N.Patt, "Implementation of hybrid branch predictors". *ISCA-25*, 1995.

[18] T. Juan, S. Sanjeevan and J. Navarro, "Dynamic history length fitting: a third level of adaptivity for branch prediction". *ASPLOS-8*, 1998.

[19] J. Stark, M.Evers and Y.N.Patt, "Variable length path branch prediction". *ASPLOS-8*, 1998.

[20] E.Strangle, R. Chappell, M. Alsup and Y.N.Patt. "The agree predictor: a mechanism for reducing negative branch history interference". *ISCA-24*. 1997.

[21 S.Dutta and M.Franklin, "Control flow prediction with tree-like subgraphs for superscalar processors". *MICRO-28*, 1995.

[22] S. Wallace and N.Bagherzadeh, "Multiple branch and block prediction", *ISCA-3*, 1997.

[23] S. pan, k.So and J. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation". *APSLSOS-5*, 1992.

[24] T. Yeh and Y.N.Patt, "Alternative implementation of two-level adaptive branch prediction". *ISCA-19,* 1992.

[25] J.E. Smith, "A study of branch prediction strategies". *ISCA-8,* 1981.

[26] B. Calder and D. Grunwald, "Fast and accurate instruction fetch and branch prediction". *ISCA-21*, 1994.

[27] S.McFarling and J.Hennessy, "Reducing the cost of branches". *ISCA-13*, 1986.

[28] J. Lee and A. Smith, "Branch prediction strategies and branch target buffer design".*IEEE Computer 21(7)*. 1984.

[29] "An introduction to Very-Long Instruction Word Computer Architecture". *Philips semiconductors.*

[30] Andre Seznec, "A Path to complexity-effective wide-issue superscalar processors". *institut national de recherche en informatique et en automatique,* September 2001.

[31] Gurhan Kucuk, Oguz Ergin, Dmitry Ponomarev, and Kanad Ghose, "Distributed Reorder Buffer Schemes for Low Power". *Proceedings of the International Conference on Computer Design*, Page 364, 2003.

[32] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, amd M.R. Stan. "Power Issues Related to Branch Prediction". *Proceedings of. International Symposium on High-Performance Computer Architecture*, February 2002.