國立交通大學

資訊工程學系

碩士論文

低耗電分支目標緩衝器

Low Power Branch Target Buffer

研 究 生： 胡耀中

指導教授： 鍾崇斌 博士

中華民國九十四年五月

低耗電分支目標緩衝器

學生：胡耀中　　　　　　　　　　　指導教授：鍾崇斌 博士

國立交通大學資訊工程學系碩士班

摘要

本研究針對處理器中常見的動態分支預測機制—分支目標緩衝器，進行省電設計。一般的分支目標緩衝器是在每次抓取指令時進行存取，並由其結果決定下次從那個位址抓取指令。但由於分支指令只佔了總指令數的一小部份，因此大部份分支目標緩衝器的存取動作都是不必要的，於是我們可以籍由減少不必要的存取來達到省電的目的。我們藉由程式執行時，記錄下各分支指令的位置來決定何時應該去存取分支目標緩衝器，並且以省電成效及效能影響兩個標準來評估我們的設計。實驗結果顯示這個設計可以在只對效能造成很小影響的情況下有效的節省能源消耗。

Low Power Branch Target Buffer

Student: Yau-Chong Hu Advisor: Dr. Chung-Ping Chung

Department of Computer Science and Information Engineering

National Chiao Tung University

Abstract

This research reduces power consumption of branch target buffer (BTB) — a commonly used dynamic branch prediction component. Conventional BTB is looked up while instruction fetcher is fetching an instruction. The result returned from BTB tells instruction fetcher the address of the next instruction. Since branch instructions occupy a small portion of total executed instructions, most BTB look-up operations are only waste power. We can reduce its power consumption by reducing useless BTB look-up counts. By recording the positions of branch instructions during run time, we can determine what time should instruction fetcher perform BTB look-up operation. This design is evaluated by two metrics: energy consumption and performance loss. The experimental result shows this design effectively saves energy consumption with only a little performance loss.

誌謝

首先我要感謝我的指導老師 鍾崇斌老師．在這兩年的時間內，老師給了我許多寶貴的建議及嚴謹的指導，讓我得以順利的完成這篇論文。除了學業上的指導，也從老師身上學習了許多一生受用的道理。另外，我想感謝本實驗室另一位老師 單智君老師。單老師除了擔任我的口試委員以外，在我兩年碩士生活中，也給了我許多寶貴的指導。還有另一位口試委員 陳添福老師，在口試時給了許多明確的指點，讓我的碩士論文更加的完整。

還要謝謝實驗室的學長及同學們，花了許多時間細心地和我討論，並給了許多建議。實驗室所有成員與我一起渡過的時光以及一起討論課業的情景，我永遠不會忘記。

最後我要感謝我的家人，長久以來毫無保留的給予我支持與鼓勵，讓我能在你們的關心之中順利的完成學業。

謝謝你們！

胡耀中

2005年7月

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Low power becomes an important issue for processor design. On the other hand, branch target buffer (BTB) consumes a significant ratio of total power consumption in processor. It is effective to lower processor power consumption by reducing BTB power consumption. There are many factors influence BTB power consumption. This thesis focuses on reducing BTB look-up counts to reduce BTB power consumption. Three approaches are proposed to reduce BTB power consumption, they are easy to implement and suitable for most processors. The evaluation result shows this mechanism is effectual with only a little performance loss.

1.1 Importance of Low Power Design

Power consumption becomes more and more important factor of IC design. Most consumed power is transferred to heat, and results in unstable and low speed IC. Low power design reduces the heat produced from high performance equipments and increases the life time of battery-powered equipments. For high performance devices, hardware complexity and high working voltage introduce enormous heat, and higher operating temperature will result in many problems, including electron-migration diffusion, enlarged clock skew, ... etc. Low power technique reduces power consumption and lower the heat. Most portable devices are powered by battery, and the life time is vitally restricted by the battery technology. Designing for low power is another solution to increase the life time under current battery technology. While the two problems become more and more serious, low power design hence becomes a very important research topic.

1.2 What is Branch Target Buffer

Almost all processors are highly pipelined today. For a processor with deep pipeline, control hazard becomes a major harm to system performance. And a good dynamic branch handling is vital to the performance of such processors.

Branch target buffer (BTB) is a very commonly used dynamic branch predictor [1]. The major task of branch target buffer is to predict addresses of next instructions. Comparing with static branch prediction mechanism, since static branch prediction can only be performed after instruction is decoded, BTB performs branch prediction at the first pipeline stage, thus further reduces branch penalty. Comparing with other dynamic branch prediction mechanism such as branch history table, branch target buffer reduced the branch penalty to zero if it is correctly predicted. In other words, if all branch instructions are correctly predicted, the pipeline will be kept full, exert the benefits of pipelining.

The major information stored in branch target buffer includes 1) address of the branch instruction (tag), 2) target address of the branch instruction, and 3) direction prediction data, which is used to predict whether this branch instruction will taken or not-taken. Every entry of branch target buffer stores the previous information of a certain branch instruction. Branch target buffer is usually organized as content addressable memory (CAM), every entry is addressed by branch instruction address. If the look-up operation is hit, branch target buffer returns the predicted direction (taken or not) and the target address of this branch

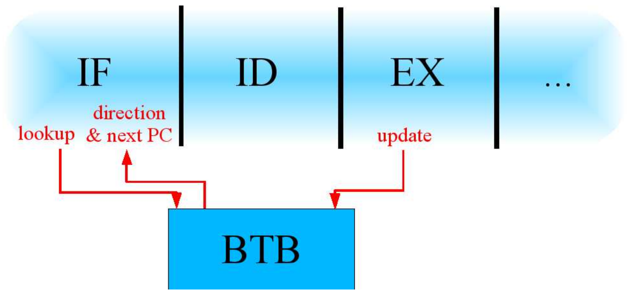


Figure 1.1: Branch target buffer overview

instruction.

Figure 1.1 is the system overview of branch target buffer. To keep pipeline full, branch target buffer is looked up in the first pipeline stage. When instruction fetcher fetches an instruction, it sends the address of that instruction to look-up branch target buffer, and retrieve the information about predicted direction and the target address. If the look-up is not hit or the predicted direction is not-taken, instruction fetcher sets program counter (PC) to current PC plus a word size. If the look-up is hit and the instruction is predicted taken, instruction fetcher sets PC to the target address which is retrieved from branch target buffer. If the address is correctly predicted, no extra penalty cycle is introduced for this branch and the pipeline keeps full.

All information in branch target buffer is gathered during run time. A branch instruction can get into branch target buffer only after it is executed, and may be swept out when a conflict happened. After a branch instruction is executed in EX stage, its target address is computed out and its direction is determined. Then this branch instruction can be stored into branch target buffer, if this branch instruction is already exists in branch target buffer, it still need to update the history of its direction. The history of direction is used to predict its direction next time it is executed.

1.2.1 Branch Target Buffer Power Consumption

BTB is usually organized as content addressable memory (CAM). The organization of content addressable memory is usually composed in two part, the first is tag and the second is data [2]. To access a specific entry of content addressable memory, all tags are compared with the address, and one or none of them match to the address. Then the data entry in SRAM is enabled to be accessible. The major power consumption is caused from tag comparison.

Since BTB contains a large and complex storage, plus this storage is accessed in every instruction cycle, it becomes a significant source of power consumption in processor. In [3] and [4], the power consumed by BTB is about 7%˜10% of whole processor power.

1.3 Motivation and Objective

Branch target buffer look-up operation can gain performance benefit only when it is looked up while instruction fetcher is fetching a branch instruction, because branch target buffer only stores information of branch instructions. If branch target buffer is looked up while instruction fetcher is fetching a non-branch instruction, it always returns predicted not-taken and results in no performance benefit but waste power.

In conventional design, BTB is looked up in every instruction cycle. In other words, instruction fetcher always performs BTB look-up operations when it fetches instructions. According to figure 1.1, branch instructions constitute small portion of total executed instructions, ranging from 8% to 19%, 14.3% in average. The data in [5] also shows similar result.

Since branch instructions do not appear very frequently, most BTB accesses are useless and only waste power. Thus we should only perform BTB look-up operations on branch instruction. To achieve this goal, we have to precisely predict the position of branch instructions and look-up BTB only on branch instructions.

Table 1.1: Ratio of branch instructions to total executed instructions

benchmark	number of branch instruction	number of total executed instruction	ratio (%)
adpcm - rawcaudio	695813	7634937	9.1135
adpcm - rawdaudio	695178	5570597	12.4794
epic - epic	42835301	278394207	15.3866
epic - unepic	2891862	26283864	11.0024
g721 - decode	86691707	473587671	18.3053
g721 - encode	46985665	250676208	18.7436
gsm - toast	9177545	144978438	6.3303
gsm - untoast	6015400	61421333	9.7937
jpeg - cjpeg	1857815	13726918	13.5341
jpeg - djpeg	230531	3467162	6.6490
average			12.1337

Since we want to reduce total power consumption, the mechanism applied should not consume too much power, it must be a very simple and power efficient mechanism. On the other hand, the processor performance should not be affected too much, or the total energy consumption will be increased even the BTB energy consumption is reduced.

1.4 Related Works on Reducing BTB Look-up Counts

The problem of skipping useless BTB look-up can be translated into the problem of identifying whether an instruction is a branch instruction. There are two sets of state of the art approaches to solve such a problem. The first approach is to identify branch instruction by compiler analysis, and the second is by pre-decoding. In compiler analysis approach, compiler analyzes control flow to retrieve branch information, and stores the information in a special hardware or by inserting hint instructions. Then processor skips unnecessary BTB look-up according to the branch information during run time. [6] and [4] is classfied into this approach. In pre-decoding approach, instruction is pre-decoded and is identified if it is a branch instruction during run time. According to the pre-decoded information, processor decide whether to look-up BTB. [3] is classfied into this approach. The detailed information about these three papers are introduced now.

Petrov and Orailoglu [6] analyze the control flow of programs at compilation time, and load the branch instruction information into a special hardware at run time. The BTB look-ups are then performed only on branch instructions, since the control flow is already known. Monchiero et al. [3] insert special instructions in code to inform the processor of the next branch instruction's arrival. If any no-operation instruction exists in a VLIW word, then the special instruction can be insert into this VLIW word to replace the no-operation. Parikh and Skadron [4] use a pre-decoder to identify if an instruction is a branch instruction at run time, the fetcher accesses BTB for all instructions in this cache line if any branch instruction exists in the cache line.

1.4.1 Application-Customizable Branch Target Buffer

In [6], a mixed hardware/software low-power branch identification approach is proposed. The required components include a profiler, a compiler, and a special hardware called ACBTB (Application-Customizable Branch Target Buffer). On the other hand, a conventional BTB is still required because ACBTB is only be activated inside hot spots of program. Branch prediction of the rest code is handled by conventional BTB.

The profiler identifies the hot spots of a program, and only the hot spots are analyzed by compiler because of the amount of information of total program is too large to store in ACBTB. The compiler analyzes the control flow to retrieve the branch instruction information in these hot spots. The information includes the positions of branch instructions and the distance to the next branch instruction under taken and not-taken conditions. The information is loaded into ACBTB at run time. When the control flow enters the hot spots, the conventional BTB is reposed and the ACBTB is activated. ACBTB stores all the information in BTB, with extra information of the distance of the next branch instruction, and the index of the entry that stores the information of the next branch instruction.

The fetcher reads the distance of the next branch instruction from ACBTB and reposes it until the next branch instruction arrives. The fetcher then reads the new information from the next entry of ACBTB.

1.4.2 Hint Instruction

In [3], compiler retrieves the information of branch instructions, and inserts hint instructions to tell the fetcher when to look-up BTB. The fetcher does not look-up BTB unless it is told to. When the hint instruction is executed, it tells the fetcher the distance of the next branch instruction, sometimes with the extra information including the branch target address or so.

A problem of this approach is the hint instruction increases code size and need extra execution time. To reduce the effect, this [3] applied this approach only on VLIW machine. If a no-operation slot is found before a branch instruction and they are in the same basic block, the no-operation slot is replaced by hint instruction, thus the extra execution time can be overlapped.

1.4.3 Prediction Probe Detector

In [4], instructions are pre-decoded to identify any branch instructions. When the instructions are loaded to instruction cache from memory, they are decoded to identify if there any branch instruction in a certain cache line. The results are stored in a hardware called PPD (Probe Prediction Detector). The instruction fetcher look-up PPD while it is fetching instructions from instruction cache. If the PPD indicates that the current cache line contains branch instructions, the instruction fetcher looks up BTB for all instructions in this cache line. If the PPD indicates that the current cache line does not contains branch instructions, the instruction fetcher reposes BTB for all instructions in this cache line.

Because PPD entry maps to instruction cache line one-by-one, PPD is also organized the same as instruction cache. If instruction cache is N-way set associative, PPD is also N-way set associative. This means PPD look-up operations also wastes a lot of power. This approach reduces the number of BTB look-up but PPD is still looked up every instruction cycle, total power consumption is not reduced a lot.

1.4.4 Summary and Discussion

The approaches proposed in [6] and [3] reduce the BTB accesses with the aids of compiler. This restricts the binary compatibility of the program, and the hardware can not gain any power reduction for unmodified programs. In fact, hardware even can not work if the software is not modified.

The approach proposed in [4] is a pure hardware-based approach, but the problem is that since BTB look-up is not performed every instruction cycle, PPD has to be looked up every instruction. To reduce PPD look-up count, all instructions in a cache line is treated equally, this restricts the performance of power reduction.

In contrast, my proposal requires only a little simple control logic and some extra space to record branch instruction information. And every instruction is treated individually to increase power reduction on non-branch instruction. For multimedia/DSP applications in which simple loops dominate, these proposed methods yield good power reduction.

1.5 Organization of this Thesis

The rest of this thesis is organized as follow. Chapter 2 explains my designs. The evaluation methodology and experimental results are discussed in chapter 3. Finally, chapter 4 is conclusion and future works.

Chapter 2

Design

To reduce useless BTB look-up count, a mechanism is proposed to predict what time the next upcoming branch instruction would be encountered. In more detail, the *distance* to the next branch instruction is predicted. BTB look-up is only performed when the predicted distance reached. If BTB misses, it is kept looked-up until it hits. The next distance is predicted when BTB hits.

2.1 Overview

Generally, branch instructions do not appear consecutively. In other words, there are usually several non-branch instructions follow a branch instruction. Figure 2.1 shows the distance between two branch instructions profile: The X-axis is the distance of two branch instructions, and the Y-axis is the accumulated ratio of branch instructions. For example, this figure shows that about 60% of the distances between two branch instructions are more than 4. According to figure 2.1, it is expected that reposing BTB for a few instructions after a branch instruction will not harm performance too much.

When a branch instruction is encountered, I predict the non-branch instruction count

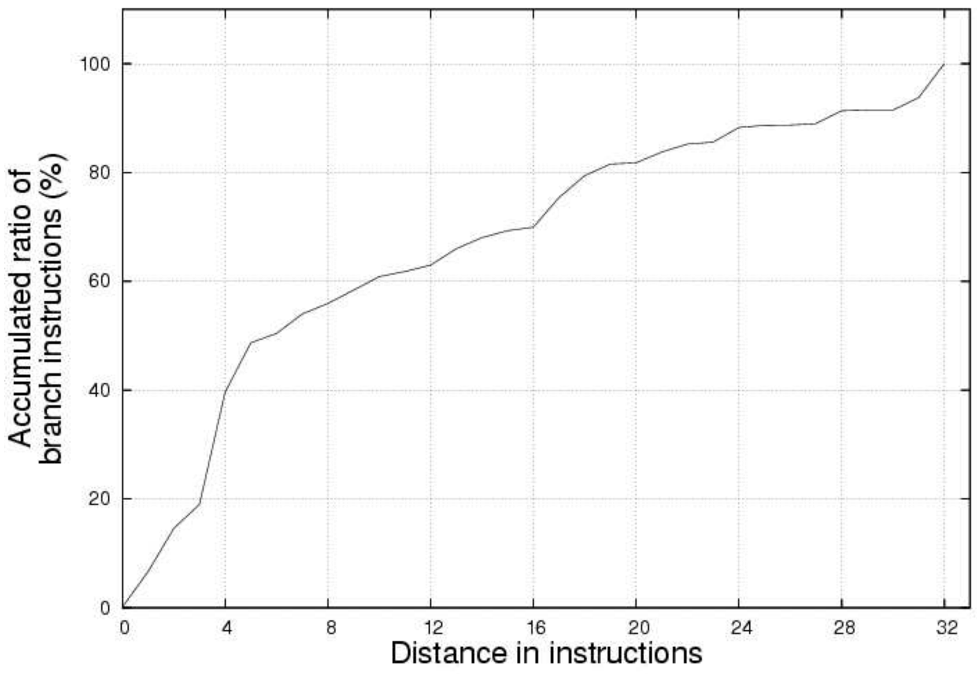


Figure 2.1: Distance between two adjacent branch instructions

(NBIC). Then BTB is reposed for the next NBIC instruction since they are non-branch instructions. Figure 2.2 shows the overview of my design. A *repose counter* is added into instruction fetcher. The value of repose counter is decremented when instruction fetcher fetches an instruction. The counter will be decremented until it is zero. Depending on whether the repose counter is zero or not, instruction fetcher looks up BTB or not. In other words, instruction fetcher only perform BTB look-up operations when the repose counter is zero. If BTB look-up hits, the repose counter has to be updated to a new NBIC value. The new NBIC value to update repose counter can be determined in several methods, and will be introduced later.

Figure 2.3 is an example of my design. The first column is the executed code sequence, branch instructions are highlighted. The second column represents whether instruction fetcher performs BTB look-up operations. The third column is the value of repose counter. At first, repose counter is zero and instruction fetcher performs BTB look-up while fetching

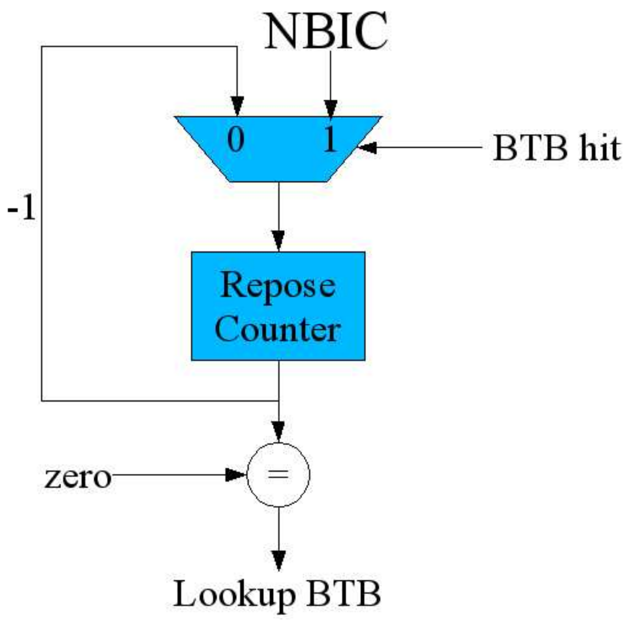


Figure 2.2: Overview of my design

instruction. When a branch instruction (the third instruction) is encountered, we update the value of repose counter. If the repose counter is updated as 4, then the following 4 instructions will be fetched without BTB look-up. When the repose counter reaches zero, it performs BTB look-up and BTB will hit because this is a branch instruction (we assume that this branch instruction is already recorded in BTB). Since BTB is hit, we update the repose counter again. The detail of how we decide the new NBIC will be introduced later.

2.2 Effect of NBIC Precision

The precision of predicted NBIC affect the result enormously. If a non-branch instruction is fetched with BTB look-up, it consume power but do not gain performance benefit. If a *taken branch instruction* is fetched without BTB look-up, it introduces performance loss because branch is not predicted and causes branch prediction miss penalty.

If the predicted NBIC is *exactly match* the number of non-branch instructions after a branch instruction, all unnecessary BTB look-up will be skipped without performance loss.

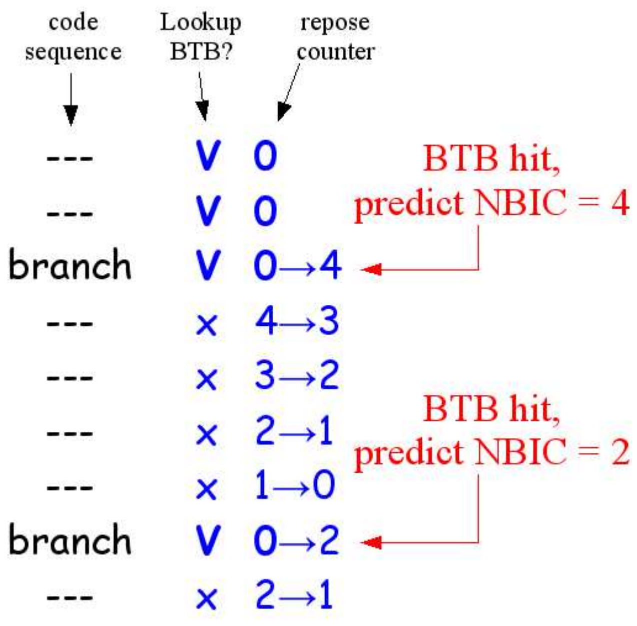


Figure 2.3: Example of my design

Figure 2.4 is an example of a perfect predicted NBIC. BTB look-up operations are performed on all branch instructions, and all non-branch instructions are not fetched with BTB look-up. This results in most power reduction without any performance loss.

If the predicted NBIC is *longer* than the number of non-branch instructions after a branch instruction, the next branch instruction will be fetched without BTB look-up. This will cause performance loss if this branch instruction is a taken branch instruction. Figure 2.5 is an example of a longer predicted NBIC. Instruction fetcher does not perform BTB look-up on the second branch instruction. If this branch instruction is taken, it causes performance loss. On the other hand, the non-branch instructions after the second branch instruction are fetched with BTB look-up, causes unnecessary power consumption.

If the predicted NBIC is *shorter* than the number of non-branch instructions after a branch instruction, BTB look-up operations will be performed on non-branch instructions and waste power. Figure 2.6 is an example of a shorter predicted NBIC.

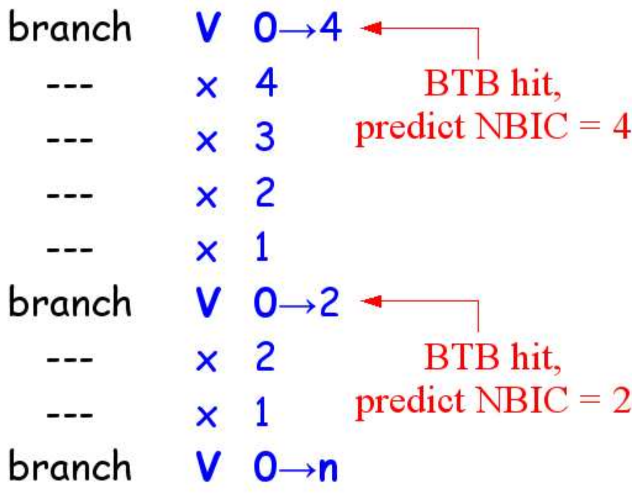


Figure 2.4: Example of exactly matched NBIC

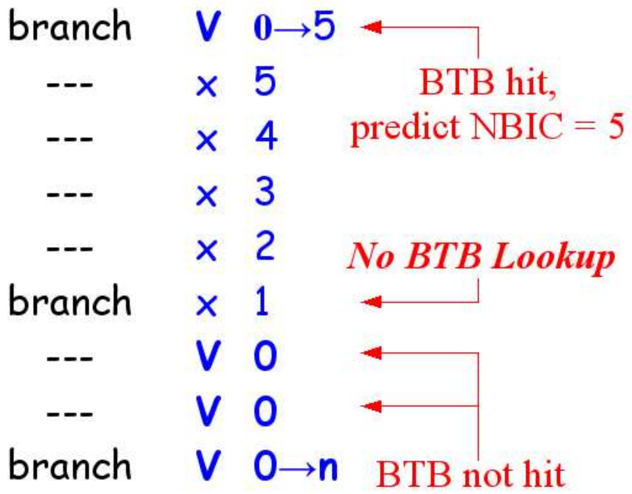


Figure 2.5: Example of longer NBIC

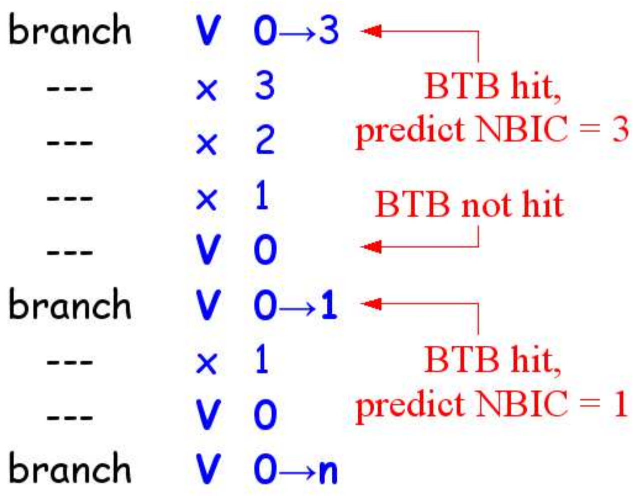


Figure 2.6: Example of shorter NBIC

2.3 NBIC Determination

The new NBIC value to update repose counter can be determined in several methods. I proposed three approaches to determine the NBIC value to update repose counter. The first approach is always predict NBIC as a fixed number. The second is to predict NBIC is the same as the latest NBIC. The third approach records every branch instruction's own NBIC.

2.3.1 Fixed NBIC Approach

The simple method to determine NBIC is always assigning a fixed NBIC. The value of NBIC can be determined arbitrarily or by off line profiling. In most cases, an arbitrarily determined NBIC about 3 is enough. But if we want to improve the precision of NBIC, we can determine the NBIC value by off line profiling.

Figure 2.7 is the design of this approach. The NBIC value is determined arbitrarily or determined by profiling. After the NBIC is determined, it will not change during run time. Whenever a NBIC is needed for updating repose counter, it simple sends the pre-determined value to repose counter.

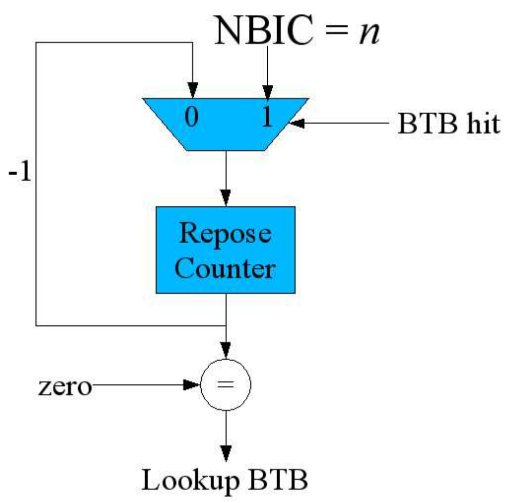


Figure 2.7: Design of the first approach

2.3.2 Last NBIC Approach

In the previous approach, NBIC is a fixed value. Actually, the distance between two branch instructions will varies during run time. The first approach can not react the variation. To improve the precision of NBIC prediction, dynamically adaptive approach is necessary.

A intuitive run time adaptive approach is to predict NBIC is the same as the distance between previous two branch instructions, i.e. last NBIC. For a program with simple loops, which contains only one branch instruction in the loop, the NBIC value in these simple loops are always the same. In such a case, if the predicted NBIC can be dynamically adapted to previous NBIC value, the precision will be enormously increased.

Figure 2.8 is the concept of this approach. When the program is executed to the second branch, the distance between the first branch and the second branch , last NBIC, is known to be 2. And then we predict the distance from the second branch to the next upcoming branch instruction is likely to be the same as last NBIC, 2.

Figure 2.9 shows the implementation of this approach. Note that this figure only shows the last NBIC counting logic, the BTB reposing logic is not shown in this figure. A *NBIC counter* is added to count the number of non-branch instructions between the previous two

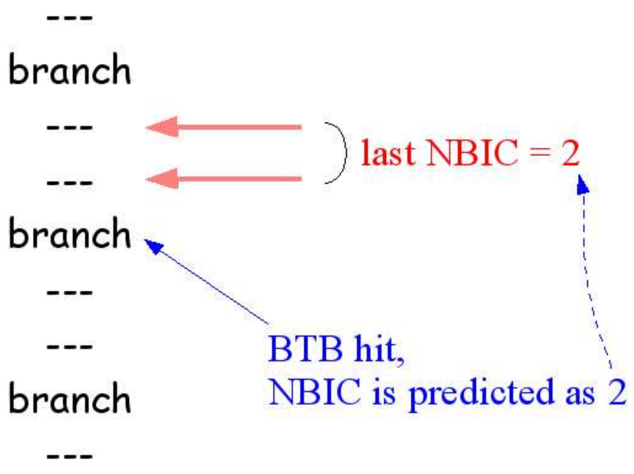


Figure 2.8: Concept of the second approach

branch instructions. A *LD* register stores the last NBIC which is counted by distance counter. Every time a non-branch instruction is encountered, the distance counter increments. When a branch instruction is encountered, the value of distance counter is written into LD register, and the distance counter is reset. When we want to predict NBIC, we predict the NBIC will be the same as the value stored in LD register.

In this approach, we count the distance between two branch instructions. Since an instruction can be identified as branch instruction after arriving instruction decode stage, the distance counter is implemented in instruction decode stage. This introduces a little difference from my original notion. Refer to figure 2.8, the predicted NBIC of the second branch should be the distance between the first branch and the second branch. But if the distance is counted in instruction decode stage, the distance can be used as predicted NBIC of the second branch. Because the NBIC of the second branch is predicted in instruction fetch stage of the second branch, but the distance is counted in instruction decode stage of the second branch, which is one cycle later. So, the distance is used as predicted NBIC of the third branch. Although it is a little different from my original notion, it does not affect the system to much, the mechanism still works.

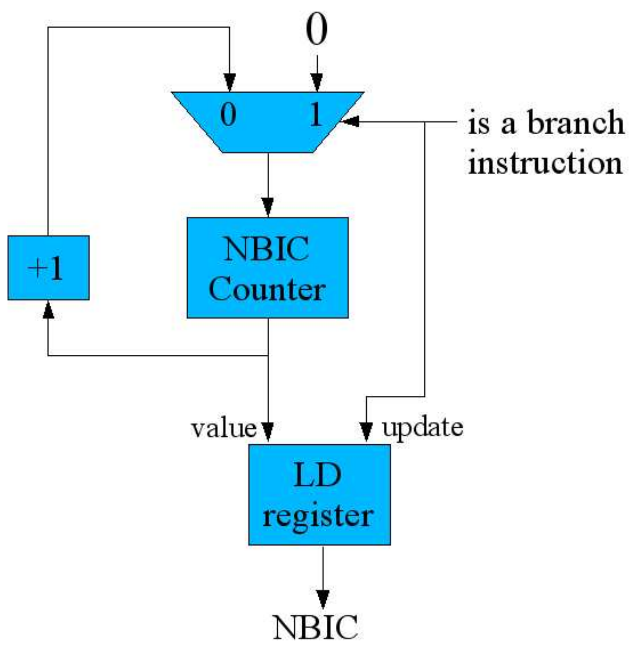
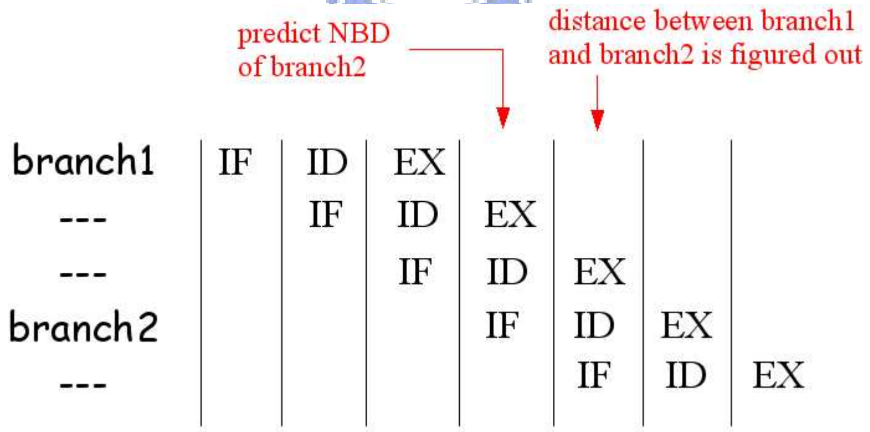


Figure 2.9: Implementation of the second approach



branch1	IF	ID	EX				
---		IF	ID	EX			
---			IF	ID	EX		
branch2				IF	ID	EX	
---					IF	ID	EX

Figure 2.10: Issue of the second approach in pipelining

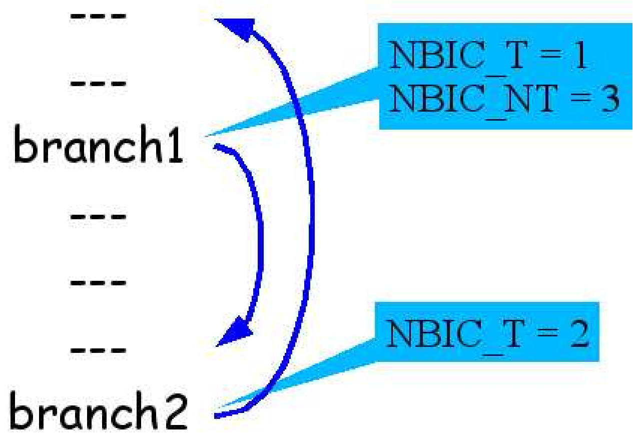


Figure 2.11: Concept of the third approach

2.3.3 NBIC Fields in BTB Approach

The third approach further improve the precision of NBIC prediction by recording every branch instruction's own NBIC value individually. Besides, since the NBIC under taken and not-taken branch are different, the NBIC is separated into NBIC_T and NBIC_NT which represent the NBIC under taken and not-taken conditions. Figure 2.11 is the concept of the third approach. Refer to this figure, after the code is executed, the NBIC information is gathered. Next time when the branch instruction **branch2** is encountered, instruction fetcher looks up BTB, BTB will predict whether this branch will taken or not. According to the information, NBIC is updated from NBIC_T or NBIC_NT of **branch2**.

On the other hand, it is inefficient to record NBIC of all branch instructions because this results in a lot of need of storage. The solution is to record the several least recently executed branch instructions. Take the advantage of temporal locality, the storage needed to store the information is enormously reduced but the prediction precision is still kept pretty good. Since BTB already stores the several least recently executed branch instructions, we can simply add NBIC_T and NBIC_NT fields into BTB. Every branch instruction recorded in BTB has its own NBIC_T and NBIC_NT fields.

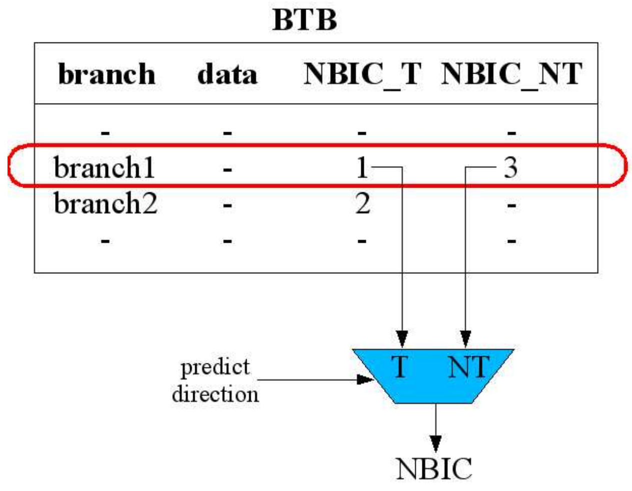


Figure 2.12: Design overview of the third approach

Figure 2.12 shows the design overview of the third approach. The information of NBIC_T and NBIC_NT fields is gathered after the code is executed. Refer to figure 2.2, whenever BTB is hit, the new NBIC value is loaded from NBIC_T or NBIC_NT fields to repose counter. Since BTB also predicts the direction of branch, the new NBIC value is loaded from the two fields depends the predicted direction. If BTB predicts this is a taken branch, new NBIC is loaded from NBIC_T, or it is loaded from NBIC_NT.

The contents of NBIC_T and NBIC_NT fields in BTB is gathered during run time. At first, the initial value of the two fields is set to zero in my design, but actually it can be set to other value such as two or three. Figure 2.13 shows the mechanism of gathering the contents. The first component is the NBIC counter. It counts the number of non-branch instructions between the previous two branch instructions, just as what it does in the second approach. When a branch instruction is encountered, the value of distance counter is stored to the NBIC_T or NBIC_NT fields of the previous branch instruction. If the previous branch is a taken branch instruction, the value of distance counter is stored in NBIC_T field, otherwise it is stored in NBIC_NT field.

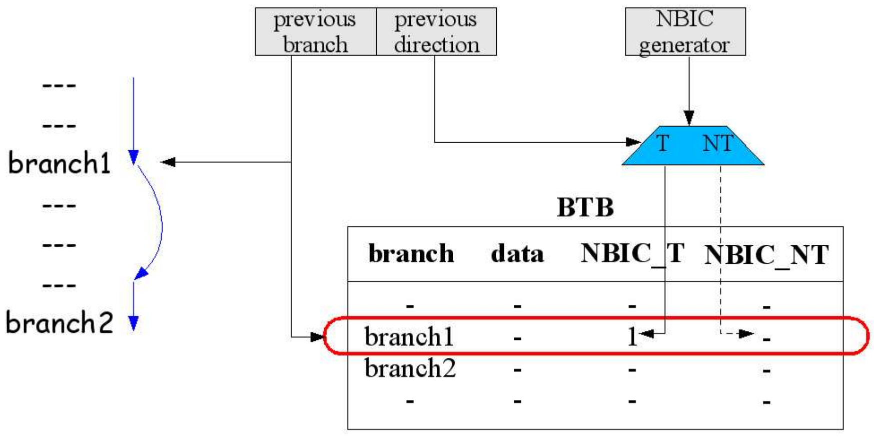


Figure 2.13: Information gathering of the third approach

2.4 Comparing My Design with Related Works

Comparing my design with related works, the advantages of my design include the following three ingredients:

Processable scope covers all executed code

Since all needed information is collected during run time and no pre-processing is required, the scope that my mechanism can apply covers all executed code. Comparing with the two compiler analysis approaches, because the information collected in static time is too much to store, these two approaches are usually applied apart of code, not all. But by my approach, information of all executed code will be collected, when the size of information is too large, old information would be swept out, but the swept information is likely not to be used. Next time it is needed, it would be collected again.

Pure hardware implementation

Any approach that want to reduce useless BTB look-up must modify the hardware to add some mechanism to repose BTB. But some approaches also have to modify compiler, ISA, source code, etc. My design is a pure hardware implementation, we can apply my design on an existed system by only changing the processor core.

No every-cycle-accessed table

In [4], although BTB is not looked up every cycle, but PPD is. Since PPD is correlated to instruction cache, it is organized in CAM, and consumes a lot of power consumption. In my design, only the counter works every cycle. Even NBIC table is only accessed when necessary.

Chapter 3

Evaluation

My design is evaluated by trace-driven simulators. The benchmark suite I selected is a subset of *MediaBench* [7], which is a benchmark suite for multimedia and communication applications. The results are evaluated by five metrics: branch instruction look-up precision, non-branch instruction look-up precision, overall instruction look-up precision, energy saving, and performance loss.

3.1 Method

It is very difficult and time-consuming to re-design a processor and implement my designs into it. Another practical approach is to develop a simulator for experiment. Behavioral simulation is a very commonly-used approach in architecture domain, because it is efficient than re-design a processor, and its result is still accurate.

I evaluate my designs by a trace-driven simulator. My simulator analysis the trace and simulate the result. When a branch is encountered, I simulate the result of branch prediction, if my predicted direction is different from the trace, I plus several cycles of penalty to executed cycle count.

3.2 Environment

I collected the trace of my benchmark programs on ARM emulator. The benchmark programs selected are subset of MediaBench, which is a benchmark suite for multimedia and communication applications. ARM is a popular processor family designed for embedded systems. Although my design is not only for embedded processor, ARM is still a rather good platform for evaluation because of its integrated development environment.

MediaBench is a benchmark suite for multimedia and communication applications [7]. It is designed to captures the essential elements of modern embedded multimedia and communication applications. MediaBench includes 11 applications, but some of them are difficult to run on ARM emulator. Due to this problem, only 6 of them are selected for evaluating my designs. The applications of MediaBench are listed below. The first six applications are selected to evaluate my design:

1. **JPEG**, a still image compression method.
2. **MPEG2**, a moving picture compression method, used in high quality digital video transmission.
3. **GSM**, European GSM 06.10 is a standard of full-rate speech transcoding.
4. **G.721 Voice Compression**, G.711, G.721 and G.731 voice compressions.
5. **EPIC**, an image compression utility which allows fast decoding without floating-point computations.
6. **ADPCM**, a standard of audio coding.
7. **PGP**, a standard of digital signatures.
8. **PEGWIT**, a public key encryption and authentication utility.

9. **Ghostscript**, interprets the PostScript language.

10. **Mesa**, a clone of OpenGL (a 3-D graphics library).

11. **RASTA**, a speech recognition program.

My approaches are designed for any processor with BTB. ARM is selected because its integrated development environment, which is suitable for evaluation. The ARM emulator I used is armsd, a powerful utility for emulating ARM processor, plus tracer funcionalities that generates trace files of these benchmark programs.

3.3 Evaluation Metrics

In this research, I use the following metrics to evaluate my design:

- Branch instruction look-up precision
- Non-branch instruction look-up precision
- Overall instruction look-up precision
- Energy saving
- Performance loss

The previous three look-up precision are raw data to evaluate my design. Energy saving and performance loss are computed from the look-up precision and are more meaningful for end user.

3.3.1 Branch Instruction Look-up Precision

Branch instruction look-up precision is to evaluate how many branch instructions are fetched with BTB look-up operation. The higher precision means the less performance loss. Branch instruction look-up precision is defined as:

$$BLP = \frac{B}{BIC}$$

BLP is branch instruction look-up precision. B is the number of branch instructions fetched with BTB look-up. BIC is total branch instruction count.

3.3.2 Non-branch Instruction Look-up Precision

Non-branch instruction look-up precision is to evaluate how many non-branch instructions are fetched without BTB look-up operation. The higher precision means the less useless BTB look-up operation. Non-branch instruction look-up precision is defined as:

$$NLP = \frac{N}{NIC}$$

NLP is non-branch instruction look-up precision. B is the number of non-branch instructions fetched without BTB look-up. BIC is total non-branch instruction count.

3.3.3 Overall Instruction Look-up Precision

Overall instruction look-up precision is take the previous two into account. It is used to evaluate the precision of overall instruction. Overall instruction look-up precision is defined as:

$$LP = \frac{B+N}{IC}$$

B and N is the same as defined in BLP and NLP. IC is instruction count.

3.3.4 Energy Saving

Precisely speaking, the purpose of my design is to save "*energy consumption*", not only power consumption. To evaluate the energy saving, power consumption and execution time must be taken into consideration. Refer to figure 3.1, Y-axis is execution cycle count, X-axis is power consumption on a certain time point. The area under the power consumption line is the total energy consumption of the application. According to figure 3.1, there are two directions to reduce energy consumption. One is to reduce power consumption, the other is to reduce execution time. In my designs, power consumption will be reduced, but execution time will be increased.

In figure 3.1, P_O is the original power consumption (power consumption before applying my design). After applying my design, P_L represent the power consumption when BTB is looked up. In my third approach, two fields (NBIC_T and NBIC_NT) are added into BTB, and increases BTB look-up power. P_R is the power consumption when BTB is reposed. For whole processor, if BTB consumes $b\%$ of total processor power consumption, P_R will be $(100 - b)\%$. C_O is the original execution cycle count. After applying my design, the execution cycle count may be increased because some taken branch instruction is fetched without BTB look-up and branch prediction is not performed. The execution cycle count after applying my design is represented as C_M. The C_{Li} is the cycle count of performing BTB look-up operation. Thus the total energy consumption before applying my design is:

$$E_O = P_O \times C_O$$

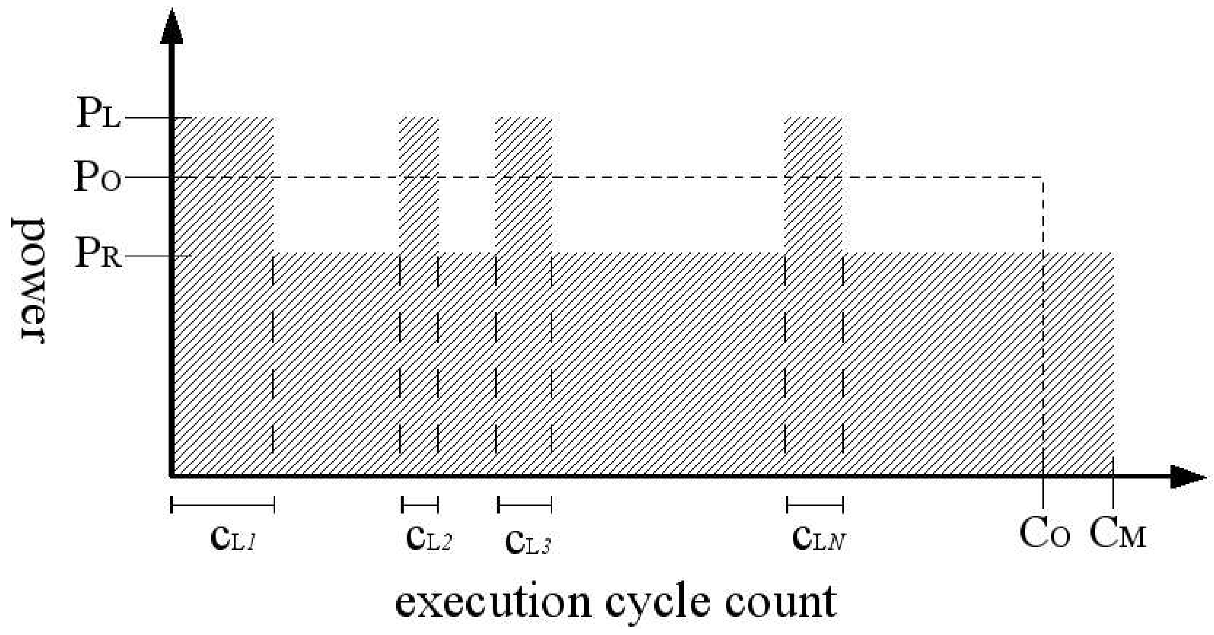


Figure 3.1: Energy consumption of a program

and the total energy consumption after applying my design is:

$$E_M = P_R \times C_M + \sum_{i=0}^{N} C_{Li} \times (P_L - P_R)$$

3.3.5 Performance Loss

Performance loss here mainly means the increment of execution cycle count. In conventional design, BTB is accessed every instruction cycle, no branch instruction will be fetched without BTB look-up. In my design, BTB will be reposed under some situation, results in some branch instructions that are fetched without branch prediction. If a branch instruction is taken but is not fetched with branch prediction, branch penalty would be introduced because the branch instruction is identified as taken in execution pipeline stage, but at the moment, instruction fetcher already fetched several instructions that will be flushed. The number of instructions to flush depends on number of pipeline stages between instruction fetching and branch taken identification.

In my experiments, number of taken branch instructions that are fetched without BTB look-up is counted and is represented as $\mathcal{BN}$ in experimental results. The number of penalty cycle is represented as $\mathcal{PC}$. Thus the total performance loss is:

$$\frac{\mathcal{BN}}{\mathcal{IC}} \times \mathcal{PC}$$

, in which $\mathcal{IC}$ represents instruction count. In my experiments, $\mathcal{BN}$ and $\mathcal{IC}$ are calculated by my simulator, and $\mathcal{PC}$ is assumed to be 2 cycles.

3.4 Experimental Results

Figure 3.2 is the branch instruction look-up precision, non-branch instruction look-up precision, and overall look-up precision. Figure 3.3 is the energy saving and performance loss. The Y-axes in the two figures are the ratio in %. The X-axes are the NBIC size in bit.

3.5 Discussion

According to my experiments, it can be identified that whether my design can save overall processor energy consumption depends on the portion of BTB power consumption to processor power consumption. On the other hand, comparing with static branch prediction, my design also saves energy consumption because it reduces execution time.

3.5.1 Beneficial Threshold

The previous experimental result is produced under the assumption that the original BTB consumes 10% of total processor power. But does my design is practicable when the original BTB consumes less portion of processor power, and what is the threshold value that my

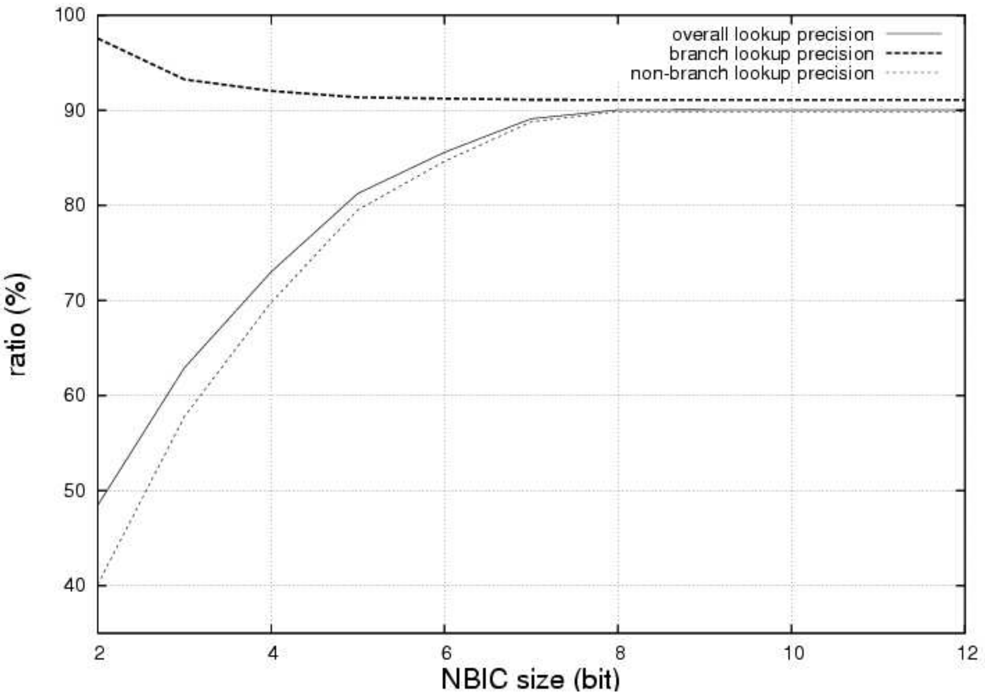


Figure 3.2: Look-up precision

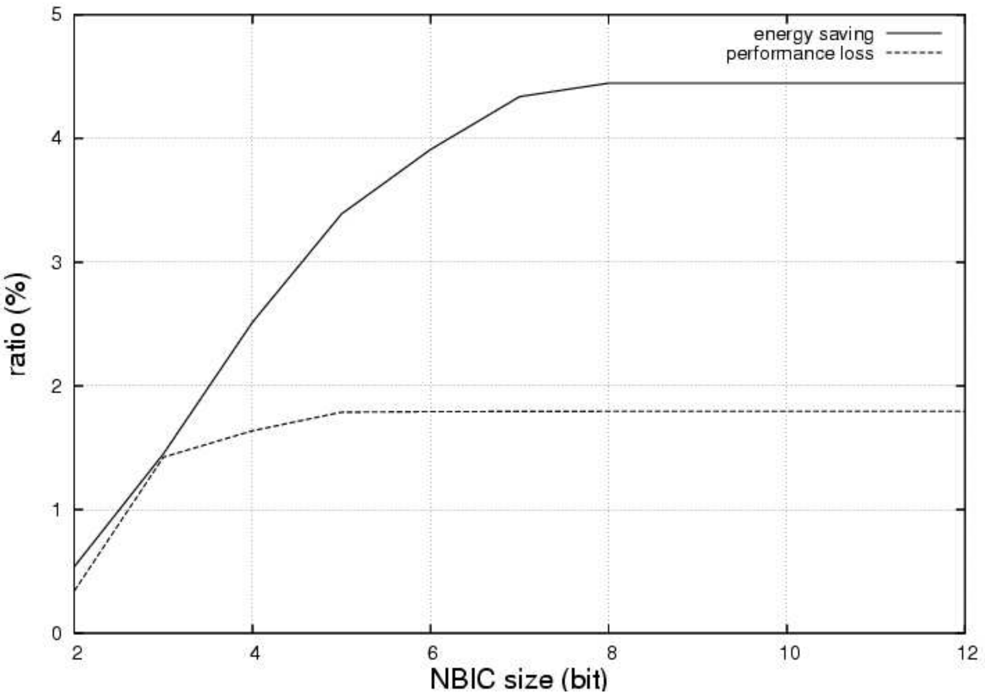


Figure 3.3: Energy saving and performance loss

Table 3.1: Energy saving, performance loss, and beneficial threshold

approach	energy saving	performance loss	beneficial threshold
First (NBIC=16)	-5.067%	10.689%	19.1%
Second (8-bit NBIC)	-1.128%	5.124%	12.9%
Third (8-bit NBIC)	5.535%	1.097%	1.7%

design can gain benefit. Beneficial threshold is defined as the threshold of the ratio of BTB power consumption to whole processor power consumption that can gain benefit after applying my design. For example, beneficial threshold x indicates that if the original BTB consumes more than x% of whole processor power, then my design can be applied to reduce the total energy consumption. On the other hand, if the original BTB consumes less than x% of whole processor power, my design will introduce more energy consumption if it is applied.

Table 3.1 is the energy saving, performance loss, and beneficial threshold of the three design. The experimental results show that the performance loss enormously affects energy saving and beneficial threshold. The first and the second approach even can not save total processor energy consumption because they introduce too much performance loss. But the third approach introduces only a little performance loss and finally saves overall processor energy.

3.5.2 Comparing with Static Branch Prediction

Comparing with a processor only with static branch prediction, applying dynamic branch prediction reduces execution time but increases power consumption. Can my design reduces overall energy consumption on a processor originally without dynamic branch prediction?

Table 3.2 is the energy consumption of a processor with 1) static branch prediction, 2) conventional BTB, and 3) low power BTB I proposed (my third approach). The first column

Table 3.2: Performance and energy consumption of static branch prediction, conventional BTB, and low power BTB

Machine	System Performance	Energy Consumption	Energy Consumption ($x = 0.1$)
Static branch prediction	100.000%	1	100.000%
Conventional BTB	109.468%	$(1 + x) \times \frac{1}{1.09468}$	101.499%
Low power BTB	108.276%	$(1 + 0.268x) \times \frac{1}{1.08276}$	95.108%

is the three machine. The second column is the performance which is normalized to first machine. The third column is the energy consumption which is normalized to first machine. The parameter x indicates the ratio of conventional BTB power to overall processor power. The last column is the energy consumption that computed from the third column, assuming x is 0.1. According to this result, it can be identified that the low power BTB not only increase system performance but also reduces overall energy consumption.

Chapter 4

Conclusion and Future Works

I proposed an effective approach to reduce BTB power consumption of branch target buffer by skipping useless BTB look-up counts. On the other hand, BTB power consumption can be further reduced by other mechanism such as reducing the static power consumption, this is a direction for future works.

4.1 Conclusion

Since most BTB look-up operations are not necessary, they can be skipped. My proposal is to predict the position of the following non-branch instruction count (NBIC) and then reposes BTB for these instructions. If NBIC is precisely predicted, BTB power consumption can be reduced without performance loss. But imprecise predicted NBIC may introduce more energy consumption. Three NBIC prediction mechanism are proposed. The first is always predict NBIC is a constant number during run time. The second predict NBIC is the same as the previous executed NBIC. The third records every executed branch instruction's own NBIC value in BTB. According to my experiments, the third approach is very precise and is effective to reduce overall energy consumption with only a little performance loss.

On the other hand, my proposal is very practicable because it can be easily applied on existing processor, no any modification of ISA, compiler or software is needed.

4.2 Future Works

In this thesis, BTB power consumption is reduced by skipping unnecessary BTB look-up counts, but there are several other directions to further reduce BTB power consumption.

4.2.1 Hierarchical Branch Target Buffer

Although the size of BTB is usually not small, only a part of total BTB entries are used in a period of time. Accessing a smaller number of BTB entry will reduce the access power consumption. A hierarchical BTB save access power consumption in most cases, and is expected to save total power consumption in average.

4.2.2 Saving Static Power Consumption

As the process technology improves, static power consumption occupies more and more portion of total power consumption. In this research, dynamic power consumption is focused and is effectively saved. Since the position of branch instruction can be predicted by my design, the entry that would be used can be known. We can activate the target BTB entry only, but switch other entries into drowsy mode to save static power consumption.

4.2.3 Instruction Cache Line plus Branch Target Buffer Entry

Cache and BTB are both organized as CAM. Most of CAM access power is consumed by tag comparison. This proposal skip the tag comparison of BTB look-up by appending BTB entry to instruction cache line.

Figure 1.1 shows that branch instruction occupies about 12.3% of total instructions. According to the statistics, it is expected that there would be one branch instruction per eight instructions. If a instruction cache line contains eight instructions, one of them would be branch instruction. If one BTB entry is appended into a instruction cache line, it can be used to store the branch prediction information of the branch instruction in this cache line.

Bibliography

[1] C. H. Perleberg and A. J. Smith, "Branch target buffer design and optimization," 1993.

[2] T. Juan, T. Lang, and J. J. Navarro, "Reducing TLB power requirements," 1997.

[3] M. Monchiero, G. Palermo, M. Sami, C. Silvano, V. Zaccaria, and R. Zafalon, "Power-aware branch prediction techniques: A compiler-hints based approach for VLIW processors," 2004.

[4] D. Parikh, K. Skadron, Y. Zhang, and M. Stan, "Power-aware branch prediction: Characterization and design," 2004.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, 2003.

[6] P. Petrov and A. Orailoglu, "Low-power branch target buffer for application-specific embedded processors," 2003.

[7] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *International Symposium on Microarchitecture*, 1997, pp. 330–335.