國立交通大學

資訊科學與工程研究所

碩 士 論 文

藉由提早產生分支目標減少分支目標緩衝器記憶體使用量及耗電

BTB Storage and Power Reduction via Early Branch Target Generation

研 究 生：石博仁

指導教授：單智君　博士

中 華 民 國　九 十 八 年 二 月

藉由提早產生分支目標減少分支目標緩衝器記憶體使用量及耗電

學生：石博仁　　指導教授：單智君 博士

國立交通大學資訊工程學系（研究所）碩士班

摘要

現今的處理器設計，趨向於同時注重效能的增進以及省電能力。在製程技術的演進之下，積體電路中的靜態漏電(leakage power)問題日顯嚴重。實驗數據指出，在 70 奈米的製程下，處理器中多於 60%的耗電來自於靜態漏電。而造成靜態漏電最主要的元件便是晶片上的記憶體(on-chip memory)，如分支目標緩衝器(Branch Target Buffer, BTB)及快取記憶體系統(cache memory system)等。

在這篇論文所發表的設計中，我們提出以使用算術運算單元(arithmetic unit)以及存取小型緩衝器(buffer)的方式，減少分支目標緩衝器的記憶體使用量和耗電。概念上，此方法是以小量的動態耗電(dynamic power)換取大量的靜態漏電。藉由提早產生分支目標(early target address generation)，原本必須存放於分支目標緩衝器中的資訊量可望大幅度地減少。此方法可完全不造成任何的效能下降，甚至有機會得到可觀的效能增進。實作提早產生分支目標機制，我們發現程式中超過 77%的分支將不再需要存放於分支目標緩衝器中。此結果幫助我們減少了超過 75%的分支目標緩衝器記憶體容量，同時減少了超過 60%的分支目標緩衝器耗電。

BTB Storage and Power Reduction via Early Branch Target Generation

Student: Po-Jen Shih Advisor：Dr. Jyh-Jiun Shann

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Modern processor designs focus on both higher performance and lower power consumption. As the evolution of process proceeds, static power consumption grows dramatically. It has been shown that under 70nm process, more than 60% of the total system power is statically consumed. And large on chip memories like Branch Target Buffer (BTB) and Cache systems are held most responsible for such leakage.

In this thesis, we proposed a method of exercising arithmetic unit and accessing small-size buffers to reduce the storage and power consumption of Branch Target Buffer (BTB). The idea is to reduce a large amount of static power by adding relatively small amount of dynamic power overhead. By performing an early branch target generation, information necessarily stored in BTB can be significantly lessen. The approach proposed suffers strictly no performance degradation, and can even deliver noticeable system performance improvement. With early branch target generation, more than 77% of the branches no longer need to reside in BTB. Leads to an approximately 75% of BTB storage reduction, and more than 60% of BTB power reduction.

致謝或序言

感謝我的指導老師 單智君教授這兩年來對我細心地指導，並給予我親切的關懷、幫助與勉勵，使我能夠克服求學中所遇到的困難，完成研究及碩士學位。也感謝實驗室的另一位老師 鍾崇斌教授的諄諄教誨，讓我瞭解到必須更加努力才能不落人後。另外也謝謝我的口試委員 洪士灝教授及 邱日清教授，由於你們的指導和建議，使得此篇論文更加完整與充實。

感謝喬偉豪學長帶領我進入 low power 領域，並且在研究上給予我許多的建議，使我學習到不少東西以及可以完成此研究。同時也謝謝不吝給予幫助和意見的翁綜禧學長和實驗室其他學長姐、同學與學弟妹，你們讓我的研究生活更為充實。

最後要謝謝我的家人與親友，你們的支持與陪伴，一直是我精神上最堅強的支柱。

石博仁　2008. 2.4

Table of Contents

摘要 i
Abstract ii
致謝或序言 iii
Table of Contents iv
List of Figures vi
List of Tables viii
Chapter 1 Introduction 1
1.1 Branch and Target Generation Method 3
1.2 Research Motivation 4
1.3 Research Objective 5
1.4 Organization of this thesis 6
Chapter 2 Backgrounds and Related Works 7
2.1 Backgrounds 7
2.1.1 Branch Target Buffer (BTB) 7
2.1.2 Instruction Buffer (IB) 9
2.2 Related Works 10
2.2.1 Related Work 1: Partial Resolution of BTB 11
2.2.2 Related Work 2: Cost-Effective BTB 12
2.3 Opportunity and Evaluation 15
Chapter 3 Branch Handling Unit Design 17
3.1 Architecture and BHU 17
3.2 Idea of BHU 19
3.3 Design of BHU 21
3.3.1 Early Branch Target Generator (EBTG) 22
3.3.2 Branch Identifier 26
3.4 Constrains of BHU 28
3.4.1 Instruction Buffer Refilling 28

3.4.2 BHU Latency 29
3.4.3 Indirect Branch Misprediction 36
3.5 BHU in More Complicated Pipeline 37
3.5.1 Multiple Issue Systems 38
3.5.2 High Clock Rate Systems 40
3.5.3 Summary ofBHU Adaption 42
Chapter 4 Experiment 43
4.1 Baselines Determination 49
4.2 Benchmark Evaluation Results 52
4.2.1 Case N = 1 53
4.2.2 Case N = 2 57
4.2.3 Case N = 3 59
4.3 Summary for Simulation Results 62
Chapter 5 Conclusions and Future Works 63
5.1 Conclusions 63
5.2 Future Works 64
References 66

List of Figures

Figure 2-1: A typical 32- bit Branch Prediction Mechanism overview. 9
Figure 2-2: Instruction Buffer overview. 10
Figure 2-3: (a) Conventional BTB (i.e. full tag) and (b) Partial Resolution in BTB. 11
Figure 2-4: Conventional BTB vs. Paired-Entry BTB. 13
Figure 2-5: Conventional BTB vs. Variable Entry Size BTB. 14
Figure 3-1: Branch handling policy comparison. 19
Figure 3-2: BHU and RBTB co-work scenario. 20
Figure 3-3: System overviews. 21
Figure 3-4: BI and EBTG working scheme. 22
Figure 3-5: Overview of the implementation of a BHU in Alpha instruction set. 25
Figure 3-6: Implementation of Partial Decoder. 27
Figure 3-7: Two possible scenarios of Instruction Buffer miss. 29
Figure 3-8: Latencies of every components of the Branch Prediction Mechanism 30
Figure 3-9: Critical path of the BHU 31
Figure 3-10: Difference segment length of PC-relative branches. 32
Figure 3-11: The accumulative result. 32
Figure 3-12: Actual length of adder required in implementation. 33
Figure 3-13: The expectation of Lookahead Pipelining BHU 34
Figure 3-14: An example of a three-cycle target generation (i.e. N=3). 35
Figure 3-15: Instruction format of indirect branches in Alpha instruction set. 37
Figure 3-16: BHU adaption in a 2-issue pipeline. 39
Figure 3-17: BHU adaption after eliminating the redundant parts. 40
Figure 3-18: Components in BHU. 41
Figure 4-1: Average accuracy of different configurations of conventional BTBs. 50
Figure 4-2: Average accuracy of conventional BTBs in sorted order. 51
Figure 4-3: Average IPC of different configurations of conventional BTBs. 51
Figure 4-4: Average IPC of different configurations of conventional BTBs in sorted order 52
Figure 4-5: Average IPC in N =1 case. 53
Figure 4-6: Average accuracy in N=1 case. 54
Figure 4-7: Normalized power in N=1 case. 56
Figure 4-8: Normalized power sorted in increasing order of IPC in N=1 case. 56
Figure 4-9: Average IPC in N=2 case. 57
Figure 4-10: Average accuracy in N=2 case. 58
Figure 4-11: Normalized power in N=2 case. 58
Figure 4-12: Normalized power sorted in increasing order of IPC in N=2 case. 59

Figure 4-13: Average IPC in N=3 case. 60
Figure 4-14: Average accuracy in N=3 case. 60
Figure 4-15: Normalized power in N=3 case. 61
Figure 4-16: Normalized power sorted in increasing order of IPC in N=3 case. 61

List of Tables

Table 2-1: Proposed design vs. existing methods. 16
Table 3-1: Branch instructions and their target address generation methods in Alpha instruction set. 18
Table 3-2: PC-relative branches in Alpha. 26
Table 3-3: Indirect branches in Alpha. 26
Table 4-1: Simulation environment. 43
Table 4-2: Integer benchmarks in SPEC2000. 44
Table 4-3: Floating point benchmarks in SPEC2000. 45

Chapter 1 Introduction

In recent years, with the pursuit of higher computation power, general purpose processor design focus on performance exploitation. A lot of methods and ideas are introduced to bring more powerful computational ability into reality. One of the major factors that affect execution smoothness is branch instruction. As we know, a branch instruction may cause two possible outcomes: proceed to next instruction sequentially or leap to target address. In order to decide the next move while a branch instruction was encountered, Branch Predictor was hence proposed for the very purpose.

A Branch Predictor is necessarily composed of two major parts: Direction Predictor and Target Address Predictor. The Direction Predictor obviously makes the judgment of whether a certain branch instruction would jump to a distant address or stay put by going inline. The Target Address Predictor on the other hand, can be viewed as an entourage of the Direction Predictor, since the target is only required at the point of certainty of a branch jump. In implementation, both predictors are history-base storage units made of SRAM. Base on the record taken down as the program execution, predictor can make accurate predictions of future behavior for branches encountered again. In nowadays processor designs, a Branch Predictor usually has two separated physical components: a Direction Predictor (for direction prediction) and a BTB (for target address prediction).

Aside from computational ability, power consumption of electronics draws more and more attention today. Portable devices like cell phones and laptop computers strongly depend on longer battery lifetime for a better using experience. Even with plugged devices, we expect less power consumption in order to achieve more energy-efficient solutions. The power consumption of electronics can be put into to two parts: dynamic power and leakage power. While dynamic power comes from every exercise or access to a circuit component, the

leakage power is statically consumed when the device is turned on. It has been shown that under 70nm process, more than 60% of the total system power is statically consumed [1]. In storage units in a system, like SRAM-based Branch Target Buffer (BTB), the power leak is particularly serious, and the degree of this power consumption increase linearly with the storage size.

BTB servers many purposes in a system, one of them is Branch Identification. BTB is accessed in a way similar to cache: indexed by program counter (PC) following by performing a tag comparison to determine a hit. A hit in BTB implies that the current instruction is actually an executed branch. In order to identify branch instructions, every cycle the BTB is accessed by the PC. To improve the hit rate, designers tend to increase the number of BTB entries. And to determine a hit, BTB must maintain a tag bit field in its structure, along with a target address field. The nature of BTB as being a large and frequently-accessed structure has destined it to become a power hungry component in the system. Statistic shows that BTB is the second largest on-chip memory: second to cache systems, and consumes approximately 5%~10% of total power, in processor system such as Pentium Pro and Alpha 21264 [2] [3].

Overall speaking, branch predictor is an irreplaceable unit when it comes to efficient program execution. Meanwhile, it comes with a price of not only chip area but also power consumption. The very existence of the key member of branch predictor, BTB, perfectly demonstrated the dilemma older than time in architecture design philosophy: compromise between cost and performance.

1.1 Branch and Target Generation Method

Branch predictor undoubtedly handles branches. First, let's get to understand what we are dealing with here. We can basically put branches into two different categories according to how their target addresses are generated:

- PC-relative Branch — Also known as Direct branch. The PC-relative branches may be conditional or unconditional. Despite of how the condition is set to determine direction, all these branches share the same method of target address generation. That is, target address = [PC + 4 + branch offset]. While the PC is known to system, the branch offset can be extracted from the branch instruction body according to the defined instruction format. Once the PC and instruction is in hand, target address can be generated in just a latency of an Adder. In a conventional five-stage system pipeline, after an instruction is decoded and certainly identified as a branch, its instruction address (PC) and offset extracted are add together during the EXE stage and target address is determined.
- Indirect Branch — Indirect branches usually have their target address kept in register file entries. Often times, the instruction set would define certain fixed register entries for this purpose. However, this category of branch may differ from instruction set to instruction set. For example, Alpha instruction set shows very strong characteristic: all indirect branches in Alpha are *unconditional* (so are also referred to as *indirect jumps*), and their target address are provided with certain predefined register file entries (i.e. $ra, $pv, $AT). As for other instruction set, like ARM instruction set, indirect branches could be a little tricky to identify, since any MOV instruction with destination of $15 (i.e. the PC) should be treated as an indirect jump instruction. Nevertheless, in ARM instruction, there is still a register

file entry defined as Link Register, where an address is meant to be load into as CALL and then later accessed from as RETURN.

According to our experiment results, most branches in common programs are PC-relative branches. The direct branch category counts for approximately 90% of total branches in average, where as the indirect branches count for the rest 10%. Since branches in the same category have similar or even identical target generation method, we suppose it is possible to design a simple mechanism to provide targets for branches with reasonable hardware overhead. And by doing so, branches would have an alternative, light-weighted handler besides the conventional huge, power-hungry BTB.

1.2 Research Motivation

So far we have introduced the purpose of a BTB: branch identification and target providing; how it is one of the largest on chip memory in system: second to cache systems in size; and last but not least, how power-consuming it is: 5%~10% of total power in a general purpose system. In addition, we have discovered that targets stored in BTB are rather easy to obtain. Simply by exercising integer arithmetic unit and access register entries, we can generate the target addresses for all branches.

We've found that BTB uses up an inefficient amount of storage for the information it contains. If we can find a way to generate target addresses without sabotaging the whole branch prediction timing scheme, the information load in the BTB can be greatly lessened, thus the number of BTB entries can also be greatly reduced.

Storage reduction of BTB not only linearly lowers the static power consumption, but also dynamic power due to the fact that the smaller structure consumes less power every access. By both the static and dynamic power reduction, BTB storage reduction contribute even significantly to overall power saving.

1.3 Research Objective

We intend to design a mechanism that logically serves the same purposes as the BTB: **branch identification** and **target address providing**. Here we name our design a “Branch Handling Unit” (BHU), since its original goal is to “handle” branches and ease the information load of BTB. The BHU design is composed of two parts: a **Branch Identifier** and an **Early Branch Target Generator,** to provide the above mentioned functions. By implementing BHU into the system, there are three main goals to accomplish:

1. To reduce BTB storage requirement:

   By handling branches by BHU, less BTB entries would be required. Less storage benefits both to less power consumption and chip size shrinking. However, this doesn’t mean BTB is no longer necessary in the system. On one hand, due to some branch unpredictable nature (e.g. indirect jumps), storing the last target address for certain branches is still an irreplaceable method. On the other hand, BHU design would face some physical constrains, making BTB still required in the system to function as its counterpart. We name the remaining BTB a Reduced BTB (RBTB), since it has far less number of entries than the original one. This will be discussed in later chapter.

2. To reduce BTB power consumption:

   The power reduction part can be evaluated by static and dynamic part.

   a. Leakage power:

      Leakage power of BTB is linearly affected by the size of the storage used. Under advanced process today, where static power counts for more than half total system power, it is effective to focus on static power reduction, in the aspect of overall power saving.

b. Dynamic power:

Dynamic power of BTB is affected by the power consumption of each access and the number of access. A BTB of less number of entries can offer a lower access power; by processing branches by BHU, less BTB updates need to be done. Both facts noticeably reduce dynamic power of BTB.

c. To maintain or improve system performance:

Branch penalty can dominate the performance of a system. When it comes to branch predictor design, performance degradation is undesired, and under most circumstances, unacceptable. Since BTB count for 5%~10% of system power, any performance degradation would be punished 10X~20X as worse; with the same sense, any performance improvement would be evaluated 10X~20X as better. Our goal is to at least maintain the system performance or even to provide better performance with reasonable cost, while achieving the above mentioned storage and power reduction.

1.4 Organization of this thesis

The remaining chapters of this thesis are organized as follows: In chapter 2, we would provide background knowledge for BTB and related works would be introduced and a brief comparison would be made indicating the opportunity we find worth trying for. In chapter 3, we would present two different approaches of our design and propose a plain evaluation of the two methods; also, challenges encountered in implementation would be discussed and provided with practical methods or conceptual solutions. Chapter 4 would demonstrate the simulation technique and results of this work; some environmental assumption would also be listed in this chapter. And finally, chapter 5, a summary would be made and some future work would be proposed.

Chapter 2 Backgrounds and Related Works

In the first part of this chapter, we explain necessary background including conventional BTB structure, access strategy, and replacement policy. Also, provide basic knowledge of another important component in our assumed system, known as Instruction Buffer. In the second part, a brief introduction of BTB power saving techniques would be made. Two previous works done on BTB size reduction would be presented and evaluated in more detail.

2.1 Backgrounds

Two aspects of background knowledge are introduced in this section. The first part is the Branch Target Buffer (BTB), and the second part is the Instruction Buffer (IB).

2.1.1 Branch Target Buffer (BTB)

In modern processor designs, Branch Target Buffer (BTB) is widely used to lower branch penalty. Conventionally, BTB integrates both direction predictor and target address predictor and provides predictions for both. By using up an amount of SRAM, BTB keeps track of all the information needed to make a proper prediction, that is, both direction and destination. However, in recent year, a lot of researches have proven that a standalone Direction Predictor often achieves better prediction accuracy [4]. The reason is mostly because independent structures make number of entries along with the indexing method more flexible and thus optimal. Also, when a replacement takes place, valuable history of direction predictions would not be evicted along with the target address. Therefore, in nowadays processor designs, a Branch Predictor usually has two separated physical components: a Direction Predictor (for direction prediction) and a BTB (for target address prediction). Thus BTB referred to throughout this thesis stands for a BTB with no direction prediction ability. This also explains

the reason why we are able to reduce the number of BTB entries without degrading direction prediction accuracy.

Most BTB has two essential fields: **tag field** and **target address field**. Every taken branch register an entry in BTB after it is executed. The correct target address calculated in EXE stage is stored into the target address field of BTB as the registration. For the case an encountered branch already registered an entry in BTB, a possible target address update maybe performed according to the calculated target address. Note that BTB keeps only the latest, correctly-calculated target addresses. As for replacement policy, most BTBs apply LRU strategy for eviction.

In order to identify a branch before decode stage, BTB is accessed every cycle at the first pipeline stage for each instruction. Conventionally, it is access in a way similar to cache system: First, ignore the two least significant bits in the instruction address (PC), since we address in word address instead of byte address. Secondly, look up BTB by index using a low order portion of the PC. Finally, perform tag comparison to determine a hit or a miss in BTB. A miss in BTB implies currently we have no information of the instruction pointed by PC being a branch or not, so it should be proceeded as a non-branch instruction; while a hit in BTB indicates a branch is being dealt with, and the last target address is expected to be landed on again this time, thus target address is provided from BTB. Figure 2-1 shows the basic overview of a Branch Prediction Mechanism in a 32-bit system.

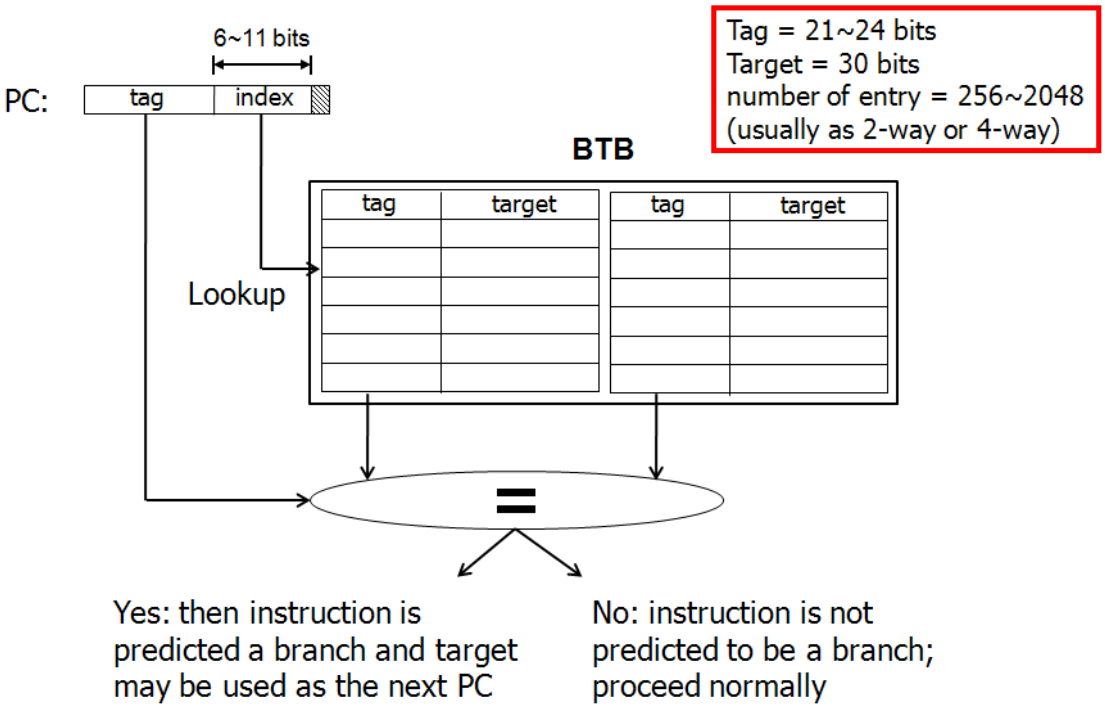


Figure 2-1: A typical 32- bit Branch Prediction Mechanism overview.

Direction prediction and target address prediction are delivered by two independent parts, Direction Predictor and BTB, respectively. In practical implementation, BTB has 64~2048 entries, thus gives us the tag and target address length estimation as shown above.

2.1.2 Instruction Buffer (IB)

Instruction Buffer (IB) is a simple idea proposed to save fetch energy consumption of instruction cache (i-cache). Since sequential access dominates the program stepping, instructions in the same i-cache line are most likely to be accessed in the near future. IB keeps a copy of i-cache line that is most recently accessed, in the hope of providing a fast and low-power instruction fetch in the near future. Every cycle the tag and index portion of current PC are compared to those of the PC last cycle. A match indicates a same-i-cache-line access, and the instruction thus can be provided from the IB immediately. Usually, IB and i-cache are access simultaneously. Assuming it takes a much shorter time to determine if a hit would occur in IB than to actually finish a proper i-cache access. Once it is determined that a hit would occur in IB, the unfinished i-cache access can be aborted. As shown in Figure 2-2.

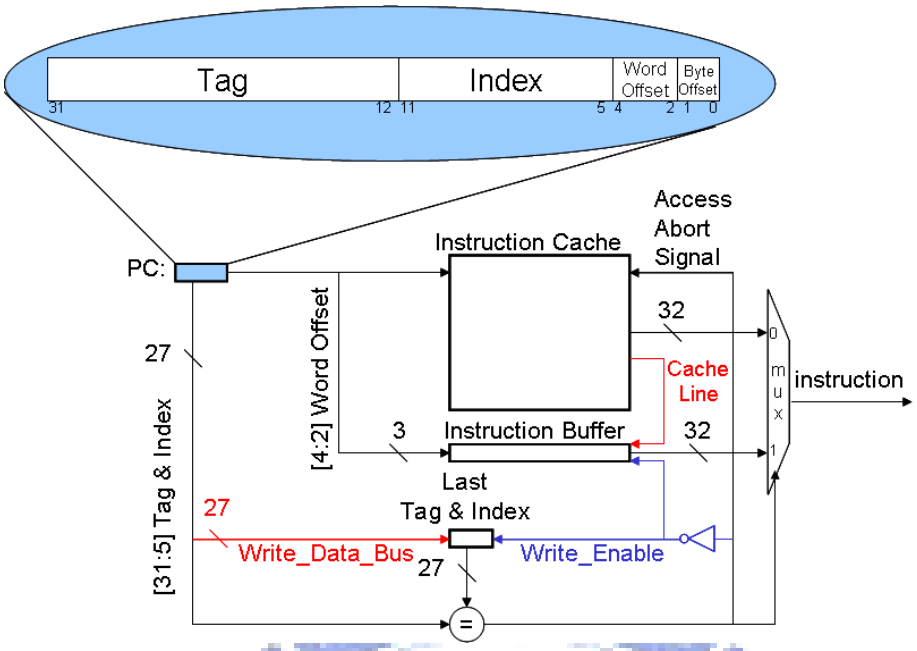


Figure 2-2: Instruction Buffer overview.

2.2 Related Works

There are many methods proposed to reduce BTB power. The nature of BTB, frequently accessed and large in structure, gives lot of opportunity when it comes to power saving, including:

1. BTB power management
2. Reduction of BTB access count
3. Reduction of BTB size

In this section, we focus on presenting previous works on BTB size reduction. Two related works are introduced here:

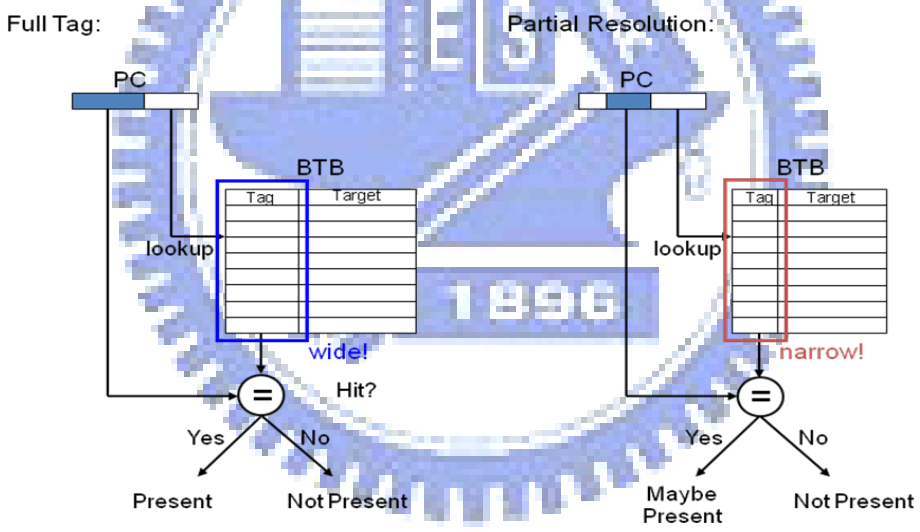
1. Partial Resolution in BTB, IEEE ToC, 1997. [5]
2. Cost-Efficient BTB, Euro-Par Conference on Parallel Processing, 2000. [6]

Both listed related works focus on reduction of each BTB entry width, while as we will learn later in this thesis, our method put effort on reducing number of BTB entries.

2.2.1 Related Work 1: Partial Resolution of BTB

In *partial resolution of BTB*, a technique of tag width reduction in BTB is proposed. By using the proposed technique, named Partial Resolution, tag field length of each BTB entries can be shorten to 3~8 bits in a direct-map BTB.

For a long time, tag comparison has been a very time and power consuming process in cache-like storage access, e.g., BTB. The comparator used for the process examines every bit of both tags to be equal for a hit, while a single bit mismatch can conclude a failure. By this characteristic, there is a chance to shorten the length of tag field in BTB. The shorten tag can unambiguously detect an absence in BTB, though it may falsely indicate a presence. Figure 2-3 shows the basic idea of Partial Resolution.



(a) (b)

Figure 2-3: (a) Conventional BTB (i.e. full tag) and (b) Partial Resolution in BTB.

As can be seen in Figure 2-3(b), a segment of least significant bits of tag are proposed to be used as a shorten version of tag. Intuitively, we can foresee that a partial tag may lead to possible **false hits**, which means mistaking a non-branch instruction as another branch instruction. Thus a non-branch instruction may wrongfully proceed with the target address provided by BTB, and a later necessary pipeline flush would take place for this misprediction.

Note that this particular kind of misprediction wouldn't occur in conventional BTB design, and it causes nothing but disturb to system pipeline flow and may even harm the accuracy of direction predictor. According to the experimental results presented in the paper, only 3~8 bits are required to provide 99% of the accuracy that full tag can deliver in a direct-map BTB. However, in a BTB with high associativity, longer tag bit field would be needed to maintain prediction accuracy.

2.2.2 Related Work 2: Cost-Effective BTB

Cost-effective BTB presented a technique to shorten target address field of BTB. By storing only the difference between branch address and branch target address, the target address field length of BTB can be reduced to essential.

Target address field shortening is based on the exploitation of **Branch Locality**. The fact is a branch doesn't tend to jump too far away from itself. So when we compare the address of a branch instruction and the address of its corresponding branch target, only a segment of low order bits would be different. Storing only the difference segment of branch and its target can help us reducing the target address field of BTB. The correct target address can be obtained by concatenating high order bits of PC with the difference segment stored in BTB when the branch is encountered again later on. Difference segment examination can be done by conducting bit by bit XOR on branch address and its branch target and then by finding the leading 1. The distance between the leading 1 and the least significant bit represents the difference segment length. Note that the difference segment length can vary greatly for each branch. And methods should be proposed for the variation in order to maintain correctness and accuracy. In this paper, two methods are proposed for this problem: **Paired-Entry BTB** and **Variable Entry Size BTB**.

Paired-Entry BTB

Paired-Entry BTB suggested that every entry in BTB has a shorten length than conventional BTB; while for long difference segment, two adjacent short entries can be **paired** together in order to form a longer entry. Extra control bit field should be attached to each BTB set to indicate the mode of the entry utilization: independent entries or as a long paired-entry. Note that in paired-entry mode, tag bit field of one of the entries is proposed to become a part of the target address field. Physically, this requires a specially designed BTB, where tag field can be programmed to function as data field. Figure 2-4 shows how the Paired-Entry BTB works.

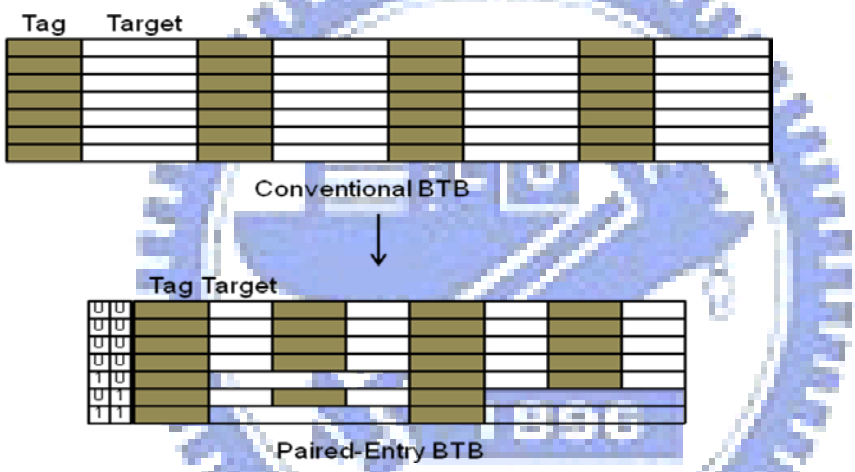


Figure 2-4: Conventional BTB vs. Paired-Entry BTB.

As can be seen in the figure, target address field in Paired-Entry BTB is noticeably shortened. Also, control bit field is attached to the BTB. A 1 indicates two entries are paired, and the original tag field between the two paired entries is therefore used as a part of new data field to contain the long address. A 0 indicates two entries function independently.

Variable Entry Size BTB

Variable Size Entry BTB applies a rather straightforward way of dealing with long difference segment. It is proposed to put entries into groups, usually each ways as different groups, and set different target field length to each group. Branches thus enter BTB group and

register for an entry according to the length of its difference segment. Figure 2-5 explains the idea.

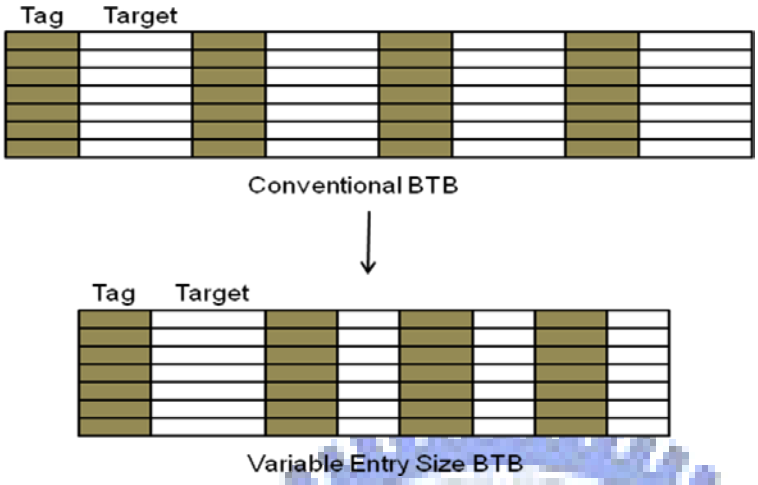


Figure 2-5: Conventional BTB vs. Variable Entry Size BTB.
Variable Entry Size BTB handle long difference segment by reserving certain number of entries with long target address field.

Paired-Entry BTB is more dynamically adjustable, since entries are only paired as needed. Besides, using tag field as data field requires hardware support, which leads to a more complex BTB structure in implementation. Variable Entry Size BTB shows less flexibility when it comes to dealing with long difference segment. The number of entries of different target address length should be decided precisely to provide optimal performance. Unbalanced utilization among groups is suspected for some cases where long difference segment branches use up all the reserved entries. Both method introduced experience a replacement policy complication in BTB, since a long difference segment branch may evict two short ones in Paired-Entry BTB; as for Variable Entry Size BTB, each groups of different target address length may have to maintain its own replacement policy.

2.3 Opportunity and Evaluation

Both the related works introduced in this chapter focus on BTB entry width shortening. None of them put effort on reduction of number of BTB entries. This is our first opportunity be working on something that has never been done before. Another issue worth mention is that both methods are likely to be accompanied with branch prediction accuracy and thus performance degradation. Our design principle, however, is set to implement a degradation-free BTB alternative. So that no system performance loss would occur and hence to system power overhead could harm our final power saving results.

Here we estimate the overall size reduction that is possibly accomplish by both methods introduced. For Partial Resolution, the author suggest that only 3~8 bits are required for a direct-map BTB to keep 99% of prediction accuracy of a full-tag BTB. Assume it is also true when the circumstances change to a 2-way or 4-way set associative BTB and also consider the accuracy does not get affected at all. For a 256-entry BTB, target address takes 30-bit, since the two least significant bits are always 0 in word address format; the index takes up 8 bits, so a full tag would be 22-bit long. That makes 30+22 bits for an original 256-entry BTB. After applying partial resolution tag is left to 3~8 bits, so that an entry is shorten to (30+3) ~ (30+8) bits. To sum up, partial resolution leads to approximately 27%~37% of size reduction for a 256-entry BTB. By the same measuring method, 25%~35% of area can be saved in a 512-entry BTB.

To evaluate cost-effective BTB, also let the performance wouldn't be harmed. Assume every entry's target field can be shortened to half size. For 256-entry BTB, index takes 8 bits, leaving 22 bits of tag. Now that the target address field only need 15bit, this gives a 29% of BTB size reduction. For 512-entry BTB the reduction is 30%. Based on the statistic we gain form experiment (see 3.4.2, adder length shortening), most branches only have difference segment of 20-bit length. So this drive us to make another estimation of only 10-bit target

field size is required. Thus for 256-entry BTB the reduction is 39% and for 512-entry BTB the reduction is 40%. In summary, by cost-effective BTB, the size reduction is 29%~39% for 256-entry and 30%~40% for 512-entry BTB. Table 2-1 gives a preview of our design compared to the related works. Note that the BHU design reduce the number of BTB entry, so the two methods in related works can still be applied to RBTB to achieve a further size reduction.

Table 2-1: Proposed design vs. existing methods.

	Partial Resolution (Related Work 1)	Cost-Effective BTB (Related Work 2)	BHU + RBTB (Proposed)
False Hit	Yes	No	No
BTB unbalanced utilization	No	Yes	No
BTB control and replacement policy complication	No	Yes	No
Performance degradation	Yes	Yes	No
Storage reduction Estimation	25% ~37%	29%~40%	More than 50%
Hardware Overhead	None	Little	One integer adder One partial decoder Three 32-bit buffers Control MUXs
Dynamic Power (from 9% to…)	8%	6%	14%+4%
Static Power (from 91% to…)	77%	64%	1%+14%
Total Power (from 100% to…)	85%	70%	Less than 40%

Chapter 3 Branch Handling Unit Design

In this chapter, we present two approaches of our Branch Handling Unit (BHU) design. Both with the final goal of reduce number of BTB entries (i.e. storage) and power consumption with the aid provided by BHU. The BTB shrunk in size with less number of entries is therefore referred to as Reduced BTB (RBTB). The two approaches basically share the same idea, while different implementations lead to different runtime behavior.

After the two approaches are introduced, a brief comparison would be presented and a general recommendation would be suggested base on some environment assumptions. Finally, two challenges encounter during the BHU design would be stated and possible solutions are introduced.

3.1 Architecture and BHU

The BHU design is targeted to be applied to general purpose processors, assuming running on RISC. BHU originally was meant to be designed for general purpose processors, where both performance and power should be concerned. However, this doesn't restrict the usage of BHU into certain specific environments. Application of BHU design in embedded system is also expected to be effective. The BHU design is also tightly coupled with Instruction Set Architecture (ISA), the nature of certain ISA can greatly change our implementation in practical. As for instruction sets, Alpha, MIPS, and ARM are all considered possible to become the final target machine ISA. While for simplicity and illustration, we make discussions about occasions under Alpha instruction set.

As mentioned in Chapter 1, there are basically two categories of branches: PC-relative Branches and Indirect Branches. In the aspect of addressing mode, they belong to PC-relative and Register Indirect addressing mode, respectively. PC-relative branches obtain effective

address by performing PC + 4 + offset, where offset can be extracted from instruction itself. Note that the PC + 4 + offset process is a simple integer addition, which in a traditional five-stage pipeline can be estimated to be done within one cycle time. Indirect branch accesses register file for target address. In addition, fixed register file entries are likely predefined in ISA to keep target address for certain indirect branch instruction to obtain from. In MIPS and ARM, $31 and $14 are used to hold the return address every time a call instruction is executed. In Alpha, more register entries are specified for target address holding: $26, $27, $28, respectively keeps target address for RETN (i.e. return), JSR (i.e. indirect call), and JMP (indirect jump). Table 3-1 gives the branch target generation method for branches of Alpha instruction set [7].

Table 3-1: Branch instructions and their target address generation methods in Alpha instruction set.

	Branch Instruction	Target Address
PC-relative Branches	BR, BSR, FBEQ, FBLT, FBLE, FBNE, FBGE, FBGT, BLBC, BEQ, BLT, BLE, BLBS, BNE, BGE, BGT	PC + 4 + offset
Indirect Branches	JSR, JMP, RETN, JSR_COROUTINE	($Rb) & ~3

The Alpha instruction set shows very organized branch and target generation characteristic. It follows a conventional PC-relative target generation method and well-defined register usage specification for indirect branches. As a matter of fact, the Alpha instruction set gives a good background environment to demonstrate how BHU is supposed to be implemented. Here in this thesis, we use Alpha instruction set as a specific example and explain the design of BHU.

3.2 Idea of BHU

BHU is intended to serve the same logical purposes as a BTB. By providing the same function and running under the same timing constrain, BHU can help reduce the information load, thus number of entries, of BTB.

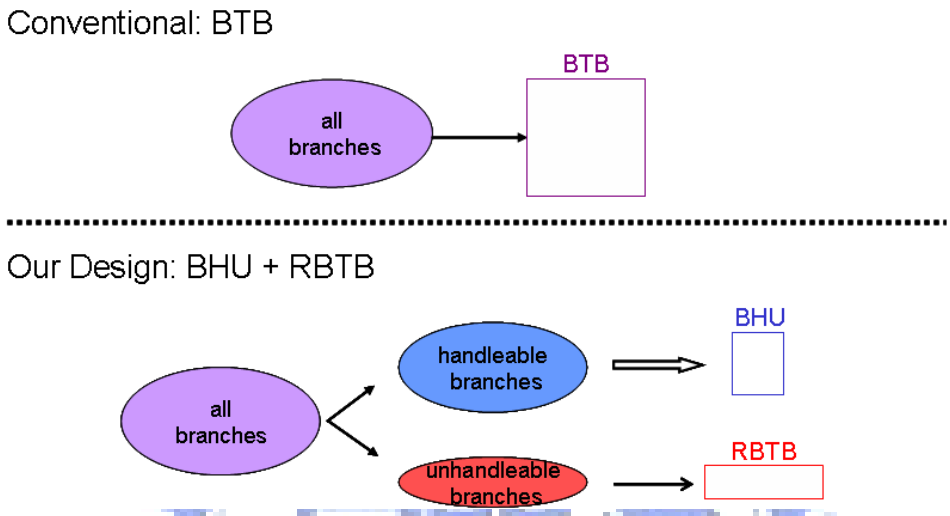


Figure 3-1: Branch handling policy comparison.

Figure 3-1 shows the idea of BHU+RBTB comparing to conventional BTB in the system. The ideal purpose of branch prediction is to incur no bubble to the system pipeline. Conventional BTB insures this by looking up for a hit. A hit would give a target address regarding the last target the branch has jumped to. A good target address together with a good direction prediction means a smooth execution flow without pipeline flush. Either it can be a direction prediction of **taken** with target address fetch as next PC or **not taken** with sequential PC used as next PC. Our design means for no different goal, but to split branches and deal with them by either BHU or RBTB. Target address of a branch may be provided by either one or both, while a decision would be made to choose out final target address. BHU and RBTB work as each other's complementary towards the same goal.

Branches which BHU can successfully provide target addresses in time are known as *handleable*, and handleable branches are the responsibility of BHU only. No allocation of

RBTB entries is needed for handleable branches. A branch can be *unhandleable* due to several timing reasons. In other words, a branch is unhandleable when BHU experience difficulties and fails to provide target address for it in time. Unhandleable branches are stored into RBTB in our system, hoping for future prediction correctness. By this policy, branches that are stored in the RBTB are the ones with a bad history record. So when it comes to the case that both BHU and RBTB are able to provide a target address for a certain branch, the one from RBTB is considered more appropriate. Figure 3-2 shows how BHU and RBTB cooperate. Conceptually, all branches dealt with in a conventional BTB system are also found solved under our design one way or the other. No branch is left untended. This is why, comparing to conventional BTB design, a performance degradation-free system is expected from our design; in fact, experimental results even shows that there is a possibility to slightly improve system performance.

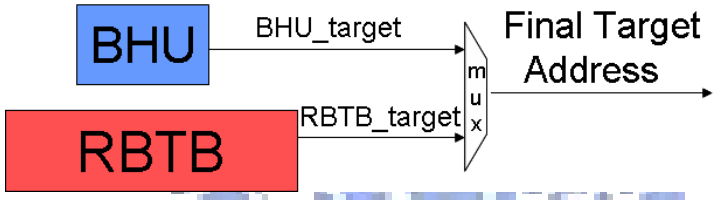


Figure 3-2: BHU and RBTB co-work scenario.
BHU and RBTB are both able of providing a target address, final target address is chosen from one of the two. RBTB holds branches with bad records with the BHU, so when these branches are encountered again, they should be trusted with RBTB's judgment prior to BHU's.

BHU design is closely related to Instruction Buffer (IB). BHU takes output of IB as input, including branch offset and operation code as well. We assume our system is a **single-issue**, **five-stage** pipeline. Also, the system is attached with an instruction buffer (IB) buffering the most recently accessed instruction cache line for fast and low power instruction fetch. Figure 3-3 shows a comparison between conventional BTB and BHU + RBTB.

Here we propose two practical implementation approaches of our BHU design. The two approaches slightly differ in the Branch Identifier (BI) part, while the Early Branch Target Generator (EBTG) part basically stays unchanged. However, due to divided strategies of branch identification, two approaches have various runtime behavior and applicable environments.

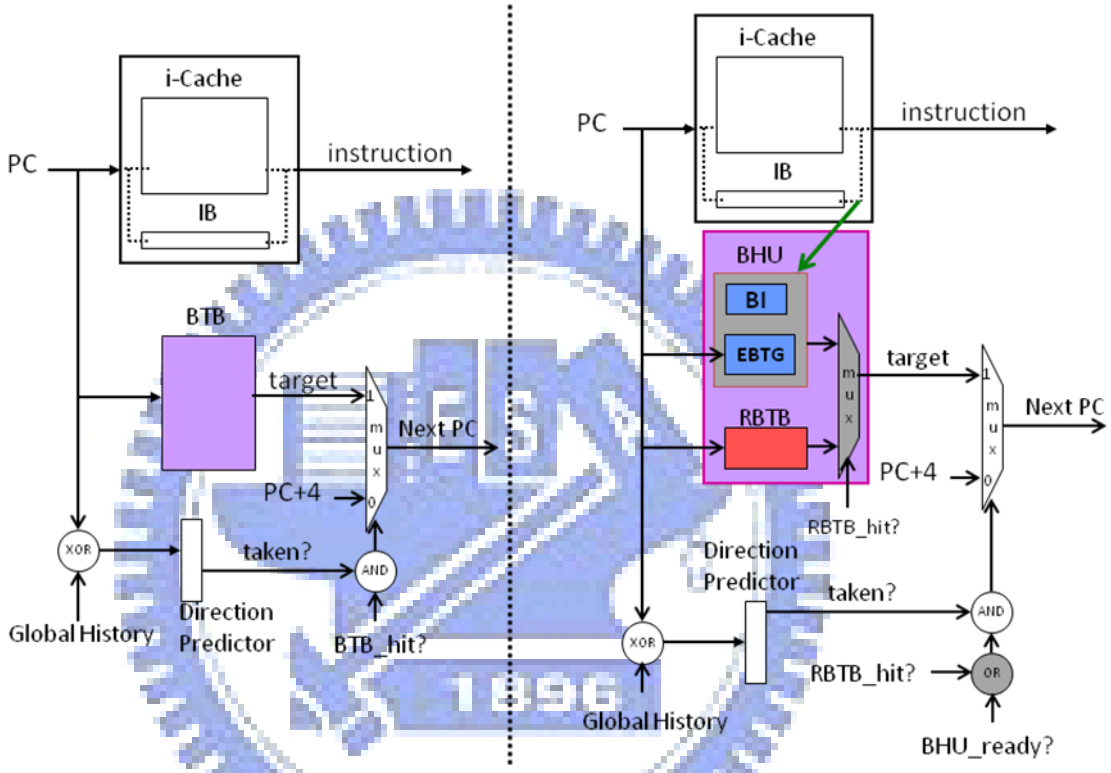


Figure 3-3: System overviews.
Comparison between conventional BTB and BHU + RBTB is shown. The purple colored zone in the right, including BHU and RBTB is expected to be as a counterpart of convention BTB. Overhead added in the system are colored grey.

3.3 Design of BHU

In this section, we would first present our design of Early Branch Target Generator (EBTG). And in second, Branch Identification (BI) is introduced along with how it should co-work with the EBTG.

BHU is designed to deliver two logical functions as BTB: **Identification of branches** and **target address providing for branches**. Thus it contains two parts for such purpose: **Branch Identifier (BI)** and **Early Branch Target Generator (EBTG)**. On one hand, EBTG is responsible of generating target addresses for each type of branches. Due to the parallelism and the various methods of branch target generation, there could be several target addresses ready to be selected as the output of BHU. On the other hand, BI provides the branch type information to indicate which target address (or none) is appropriate. Together, BI and EBTG provide a complete function of branch prediction: branch identification and target prediction, which are exactly what a conventional BTB stands purpose for traditional system. Figure 3-4 presents the idea of their work division.

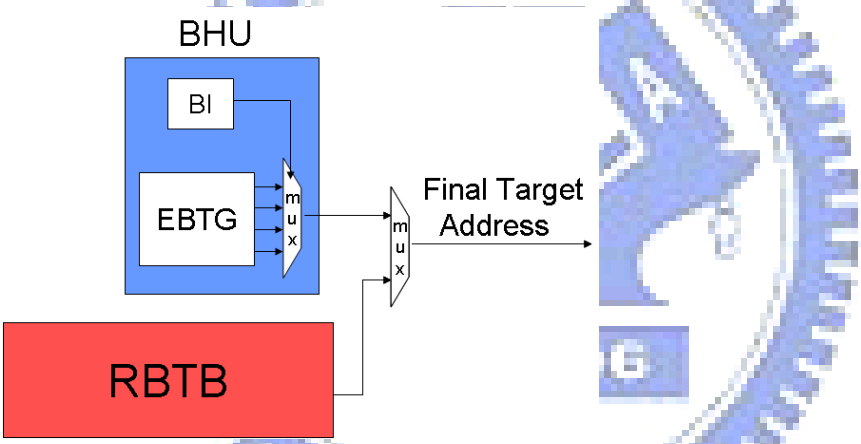


Figure 3-4: BI and EBTG working scheme.
Final target is provided from either BHU or RBTB. RBTB hit is considered prior to BHU generation.

3.3.1 Early Branch Target Generator (EBTG)

First we should take a look at the Early Branch Target Generator (EBTG). As mentioned before, the example is shown running Alpha instruction set. Recall Table 3-1, there are 4 different ways to generate target address for branches in Alpha instruction:

1. Perform (PC + 4 + offset) for all PC-relative branches.
2. Access $26 for RETN.

3. Access $27 for JSR.

4. Access $28 for JMP.

The actually instruction-set defined branch targets are described as above. Note that the other indirect branch, JSR_COROUTINE, is rarely used and in practical and its register undefined by instruction set; in fact, it does not appear at all in our benchmark set. Assume a single-issue, five-stage pipeline. Since our goal is to perform early branch target generation, we intend to move the above operations, which is meant to be done in EXE stage, to an earlier IF stage. This is when the fast instruction access from instruction buffer (IB) becomes handy.

Target Address Generation for PC-relative Branches

To generate target address for PC-relative branches, offset must be extracted for the addition to operate. Since instruction buffer (IB) is only a line size buffer, instructions inside can be available in a short access time. We propose to set an extra, dedicated adder for BHU to perform PC + 4 + offset, instead of sharing resource with the system pipeline functional unit. The nature of PC-relative branches is pointed out: once the (PC + 4 + offset) addition operation is properly finished, there is no chance that this target address can be wrong. The only thing that can still go wrong is the direction prediction, which is not the part we mean to focus on here.

In a traditional five stage pipeline, since EXE takes up one cycle, we have reason to believe that a simple integer addition operation can be properly finished to match the timing constrain of branch prediction: within the first pipeline stage. Evan in deeply pipelined systems, integer addition is still considered as a short latency operation, comparing to integer multiplication, integer division, floating point operations or memory operations. In addition, many existing techniques have suggested that integer adder can be well customized in timing or area aspect, and it can be easily tuned to incur very short latency. To sum up, integer adder

is likely to fit into one cycle time even in high clock rate machines. More timing details are presented in section 3.3.5.

Target Address Generation for Indirect Branches

As for the three specific indirect branches, JMP, JSR, and RETN, we propose to provide their target addresses by using three Register Buffers. Reading the register files could be a straightforward way for target addresses, but it may require another dedicated set of read ports to the register files to avoid hardware conflict. This drawback is unacceptable when it comes to a low power design like this work. So instead, buffering the register value becomes a reasonable alternative solution. It has the advantage of short access latency and low power consumption. Every time there is a write operation destine to the three specific register file entries, $26, $27, $28, a copy of the write value is duplicated into corresponding register buffer. Therefore when a certain indirect branch instruction is encountered, target address can be ready in a buffer access time. Note that the original register read was supposed to be done in the ID stage. Now the operation is done earlier in the IF stage by reading register buffer, there is a chance of suffering from **data dependency**. And also, conventional register usage may not always be followed. That is, compiler can set its own strategy of optimizing register usage, and violate the conventional rules in practical. Both facts mentioned above may result in a wrongful target address obtainment. However, according to our experiment, there are more than 70% of chances that this simple method provides a correct target address for indirect branches. As for the branches provided with bad target addresses, we propose to store them into RBTB for future correctness.

Some may argue that Return Address Stack (RAS) is another way to offer target addresses. Though RAS is indeed a useful design for **RETURN** instructions, it doesn't work for other indirect branches, in our case, the JMP and JSR instructions. And also a typical RAS takes up to 16 or 32 entries for reasonable prediction accuracy, while we proposed only takes

up 3 entries for all indirect branches. Recall that this is a research focus on storage reduction, it's not hard to understand why we preferred to left out RAS and go with register buffers.

Figure 3-5 shows our BHU design for Alpha instruction set. The extra dedicated adder has two inputs as PC and offset, which is available from the instruction buffer. Note that for a 32-bit system, target addresses can be provided in word address instead of byte address. So the (PC + 4 + offset) can actually be done by feeding PC and offset as inputs and 1 as the Carry In bit of the adder. This way, no extra latency but the adder itself should be incurred. The three register buffers are rather simple to implement. They share the (write) enable signals and write data bus with the physical register entries: $26, $27, $28. The only overhead is the extra bus lines and the three 32-bit buffers storage themselves.

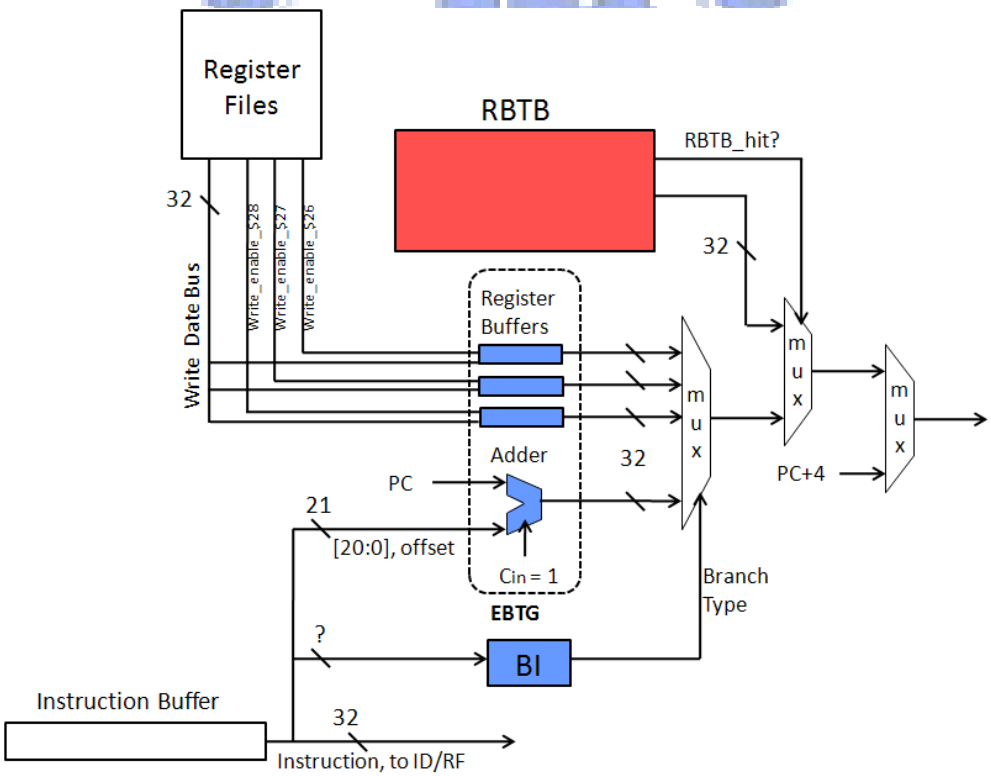


Figure 3-5: Overview of the implementation of a BHU in Alpha instruction set. Target addresses are generated by adder and register buffers. Final branch target decision is made by Branch Identifier (BI), of which two different proposals are introduced later in this chapter.

3.3.2 Branch Identifier

The Branch Identifier applies a dynamic way of branch identification by using a Partial Decoder (PD) for the job. Since the instruction is available from IB, we can also extract the operation code to perform a quick examination. Table 3-2 is the operation code of all branches in Alpha instruction set.

Table 3-2: PC-relative branches in Alpha.

Op name	Op code	Op name	Op code	Op name	Op code	Op name	Op code
BR	110000	FBNE	110101	BLT	111010	BGT	1111111
FBEQ	110001	FBGE	110110	BLE	111011		
FBLT	110010	FBGT	110111	BLBS	111100		
FBLE	110011	BLBC	111000	BNE	111101		
BSR	110100	BEQ	111001	BGE	111110		

Table 3-3: Indirect branches in Alpha.

Op name	Op code
JMP	000000
JSR	000001
RET	000010
JSR_C	000011

We mean to partially decode instructions to decide which target address of EBTG should be chosen. Therefore, **PC-relative branch** category along with three specific instructions: **RET**, **JSR**, and **JMP**, should be identified. According to Table 3-2, PC-relative branches occupy the encoding space of “11xxxx”, so a single AND of the highest two bits of the operation code (op code) would give a correct result. And according to Table 3-3, RET, JSR, and JMP can be decode in similar ways. A six-input AND can check out a match pattern for each of the three indirect branches. Note that the partial decoder would have a total of four outputs. Respectively indicates the current instruction being a PC-relative branch or one of the

three indirect branches. Only one of the outputs would be TRUE or none would be. Figure 3-6 shows the design of our partial decoder.

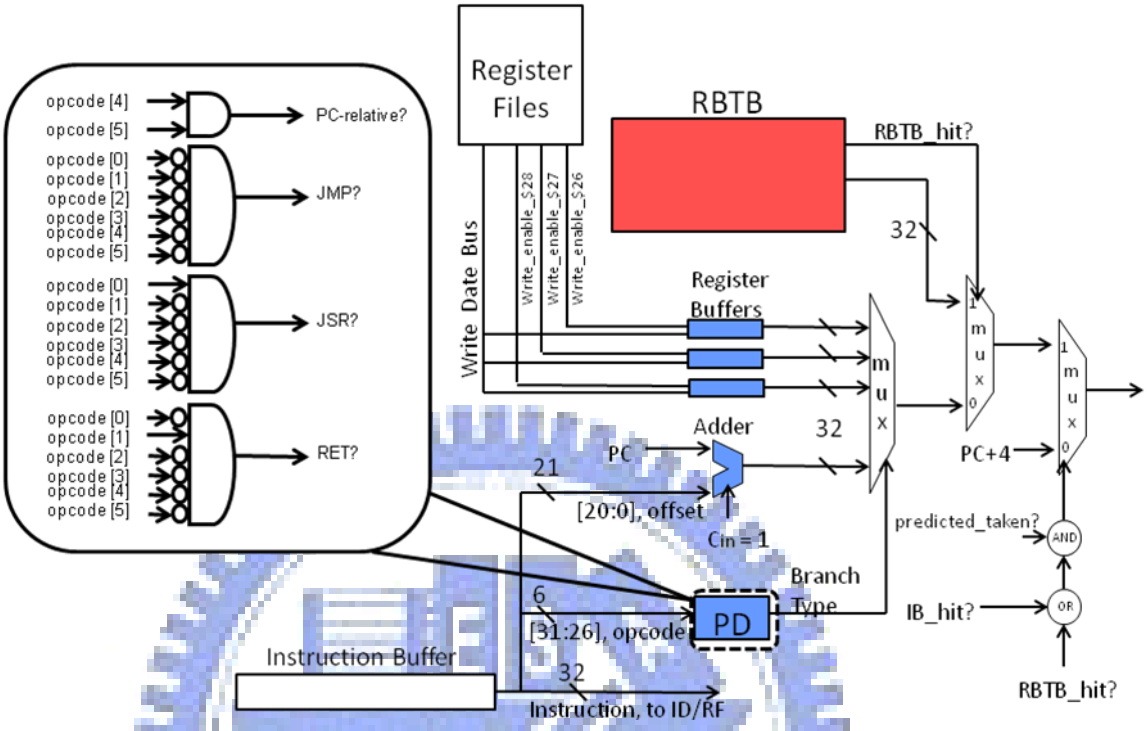


Figure 3-6: Implementation of Partial Decoder.
Overview of the whole PD + BHU mechanism in an Alpha instruction set system.

We must point out that with partial decoder, every cycle an exercise must be performed to know whether a branch instruction is being dealt with. The same thing happens to EBTG. Every cycle EBTG must blindly generate target address for instruction not yet identified, and the generated address is only a meaningful target when partial decoder confirms that it is a branch. A lot of generated results go to waste for non-branch instructions, since PD and EBTG are unlikely to work in sequence due to strict timing constrain.

3.4 Constrains of BHU

Ideally, we would like every branch to be identified and its branch target generated by BHU. However, this is not possible due to some constrains. These constrains cause timing issues, which lead some branch to become unhandleable and cannot be handled by BHU in desired timing. For branches unhandleable by BHU, our strategy is to deal with them by the traditional method: store them in BTB and look up later. This is the exact reason why a Reduced BTB (RBTB) still remains in the system. In this section, discussion and possible solutions are presented for the three constrains of BHU: instruction buffer refilling, BHU latency, and indirect branch misprediction.

3.4.1 Instruction Buffer Refilling

The first constrain is Instruction Buffer (IB) refilling. As can be seen in Figure 3 7, along with program execution, the PC is bound to stride from one cache line to another. No matter it is due to sequential step or a leap to target caused by a branch. When a line change occurs, IB miss will take place and refill must be carries out. Meanwhile, the instructions during refill are fetched from instruction cache instead. Before the refill is completed, BHU cannot be fed with any input signal to perform identification and generation. No meaningful output can come out in such circumstance, thus the output of BHU should be gated and discarded. Assuming a five stage pipeline with single cycle instruction cache access, we can expect the IB refill to be done in one cycle. This makes one unhandleable instruction every cache line change. These unhandleable instructions are stored into the RBTB, that is, of course, if they are taken branch instructions. Registries of these branches in RBTB are likely to lead to future prediction correctness.

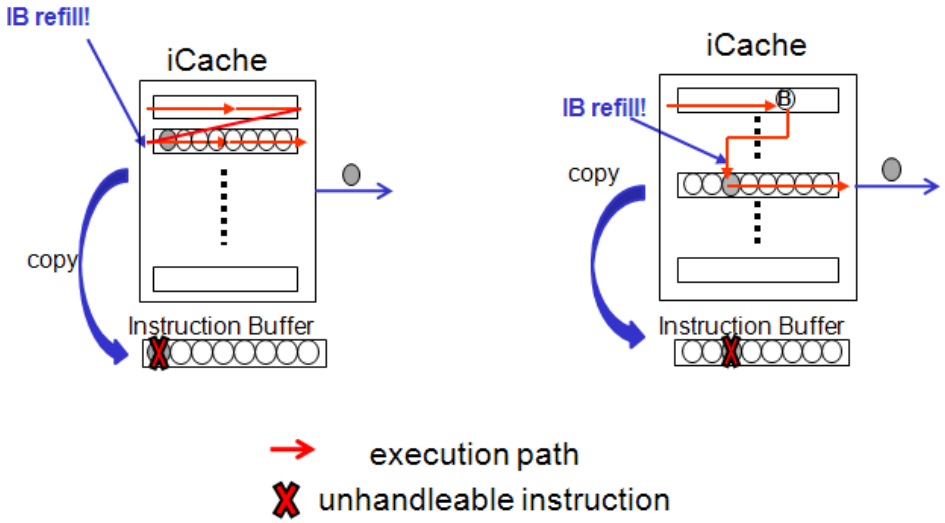


Figure 3-7: Two possible scenarios of Instruction Buffer miss.
On the left, program execution sequentially steps into a consequent cache line. On the right, control instruction, such as branch, changes the flow by leaping into the middle of another cache line.

Instruction buffer miss and the consequent refilling is an unavoidable issue. And every time its presence causes an inevitable unhandleable instruction to our design. The only thing that can be done here is to **hope** that these instructions are not branches. That way, the unhandleable ones causes nothing but a vain BHU exercise. Since the locations of instructions in a cache are adjustable with the help of the compiler, this problem can actually be transform to a new problem related to software co-design. This issue would be further discussed and advised in later chapter of this thesis.

3.4.2 BHU Latency

The second constrain involves circuit latency. As we know, BHU is a part of branch predictor, and as every branch predictor, it has timing constrain. Typically, branch predictor is designed to be properly accessed within the first cycle of the pipeline. So with a correct prediction, no bubble is incurred and branch penalty is completely avoided. BHU must not

have exercise latency exceeding one cycle time, or either the branch prediction mechanism would suffer a functional change, or the system pipeline cycle time would be strained for BHU to fit in. Both cases are unacceptable as long as performance is concerned. The BHU latency modeling is shown in Figure 3-8.

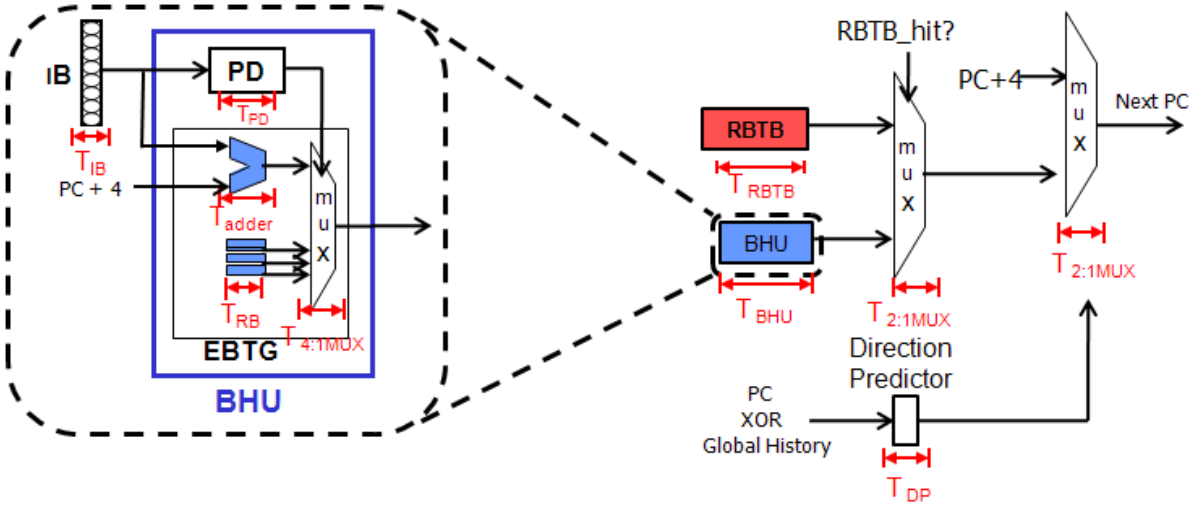


Figure 3-8: Latencies of every components of the Branch Prediction Mechanism.

As shown in the right of Figure 3-8, there are two paths for branch prediction mechanism to generate target address: from RBTB or BHU. If either path of branch prediction mechanism incurs latency that exceeds one cycle time, then the target address generated would be overdue and branch penalty cannot be avoided. We assume the critical path would be on the path of the BHU, since RBTB has smaller size and shorter access time comparing to the original BTB, which can be accessed in one cycle. More specifically, the critical path is assumed lying on the path of the integer adder within the BHU, as shown in Figure 3-9. To shorten the exercise latency of the BHU, we need to work on the estimated critical path by cutting the time requirement of the adder. Hence, we propose a possible solution to shorten BHU delay: ***Adder Length Shortening***. Adder length shortening is a rather practical guideline that aggressively adjusts adder latency to make generation fit into certain number of cycle(s). Most of the time, we tend to crunch the generation latency to fit into one cycle for a no-bubble branch prediction. Furthermore, for pipelines that even adder length shortening fails to make

the BHU work smoothly, we propose to apply ***Lookahead Pipelining*** technique to make a compromise between performance and applicability.

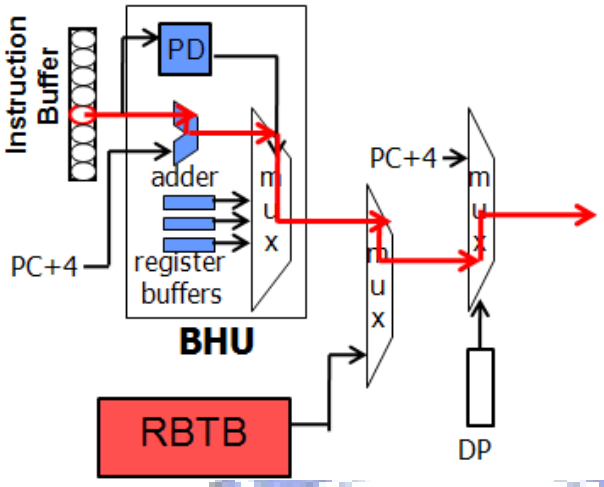


Figure 3-9: Critical path of the BHU

The critical path is assumed lying on the integer adder.

Adder Length Shortening

Adder shortening is a simple idea. We've discovered that most branches won't have target too far from themselves address-wise. That is why the (PC + 4 + offset) process wouldn't require a 32-bit full adder, since the carry will not ripple too far from the least significant bit. With a short adder, the output represents the low order bits of target address. The low order bits should be concatenated with the high order bits of PC (instruction address of the branch instruction) to form the correct target address. According to research experiments, latency of adders is approximately linearly reflected by the length of the adder. Hence this gives us a good opportunity to reduce BHU latency by shorten adder.

Figure 3-10 shows the result of percentage of PC-relative branch that carries far most into the i-th bit while performing (PC+4+offset), regarding the benchmarks in SPEC2K. Figure 3-11 shows the accumulative result of Figure 3-10. For an i-bit adder, we can determine whether the calculation is complete or not by examining the ***carry out bit*** of adder. A 0 carry out bit indicates a complete calculation, while a 1 carry out bit implies a failure.

Hence, carefully chose an adder with short latency and use the carry out as a hint of a correct calculation, we can reduce the delay on the adder path. Branches that cannot be correctly calculated by the chosen adder are again stored into RBTB. A compromise must be made between the length of the adder and the number of miscalculated PC-relative branches that need to be registered in RBTB. A long adder incurs longer latency and cost more area and dynamic power while exercising; while a short adder fails more target generation operations, leading to a RBTB with more entries needed.

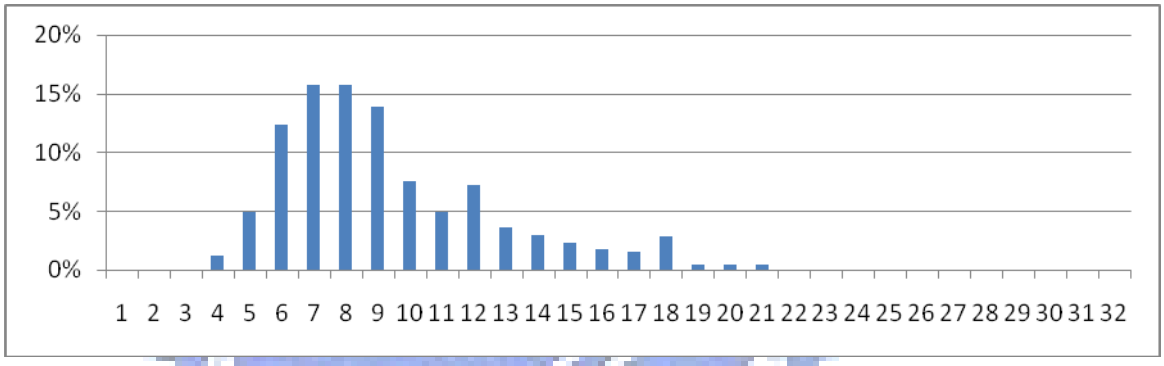


Figure 3-10: Difference segment length of PC-relative branches.

Percentage of PC-relative branch with their target address that ***carries far most into the i-th bit*** during calculation of (PC+4+offset). X-axis in the chart represents bit number, and y-axis represents the percentage of PC-relative branch.

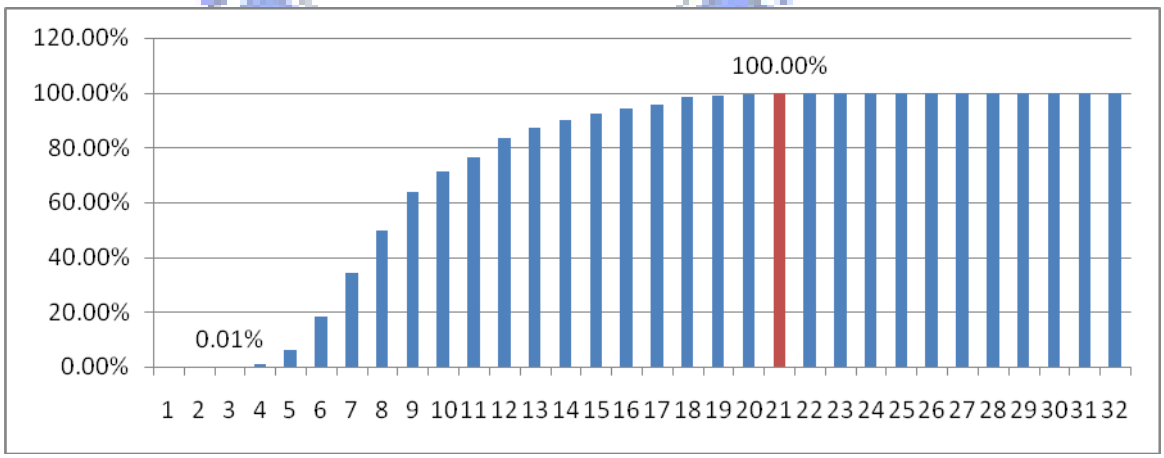


Figure 3-11: The accumulative result.

X-axis in the chart represents bit number, and y-axis represents the percentage of PC-relative branch.

Note that in Figure 3-11 adder of 1-bit or 2-bit length exhibit no accumulation at all. This is due to the fact that instruction addresses are presented in word address format. In a 32-bit system, every instruction is 32-bit long. So that in word address format, every last two bits of an instruction address should be “00”. This characteristic can actually be exploited to further reduce the requirement of adder length. There would be no reason for us to use a function unit to calculate bits that we’ve already known, thus another 2-bit length can be spared from the adder. In the end, our PC-relative branch target generator can be simplified as an 19-bit integer adder, putting its effort on only the middle part of the whole 32-bit address: outputting from bit 3 to bit 21, with bit 1 and bit 2 set to “0” and bit 22 to bit 32 copied from the branch address. Figure 3-12 shows the final chart of percentage of PC-relative branches comparing to the possible adder length.

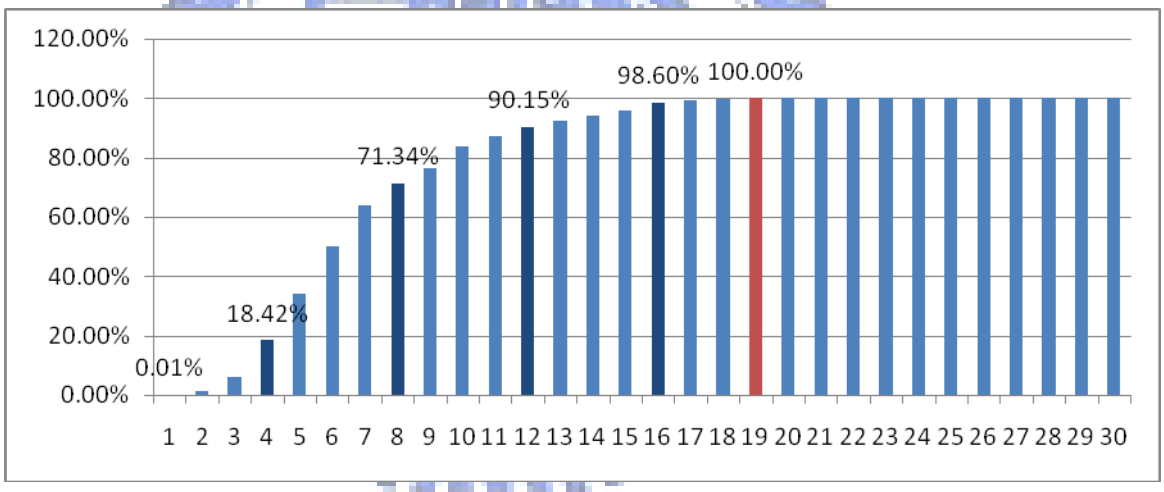


Figure 3-12: Actual length of adder required in implementation.
Most adders are implemented as a combination of 4-bit adders, so multiples of 4 are highlighted and labeled in this figure.

Lookahead Pipelining BHU

For systems of high clock rate where even adder shortening fails to suppress BHU latency within one clock cycle, we propose to apply lookahead pipelining technique. First, recall Figure 3-8, the latency of BHU is defined as:

$$T_{BHU} = \max (T_{IB} + T_{PD} + T_{4:1\ mux}, T_{IB} + T_{adder} + T_{4:1\ mux}, T_{RB} + T_{4:1\ mux})$$

, where

T_{BHU} = the delay of BHU

T_{IB} = the access time of instruction buffer

T_{PD} = the latency of partial decoder

$T_{4:1\ mux}$ = the latency of 4-to-1 mux

T_{adder} = the latency of adder

T_{RB} = the access time of register buffer

The BHU is pipelined according to its latency into N stages, where:

$$(N-1) \times cycle\_time < T_{BHU} \leqq N \times cycle\_time$$

Then lookahead is performed, sacrificing some instructions in order to keep others under the timing constrain of branch prediction mechanism. A target address is required at the end of the first system pipeline stage for all branches to have a bubble-free branch prediction. As BHU is pipelined, we can guarantee penalty free prediction for most branches by starting generation beforehand. The final expectation is: as an instruction enters the system pipeline, it also reaches the final stage of BHU generation. For the lookahead technique to work, a certain number of instructions would be skipped at the beginning. These instructions are unhandleable to BHU, and unhandleable branches are stored into RBTB. Figure 3-13 show the expected pipe flow and Figure 3-14 shows an example of a three-cycle BHU.

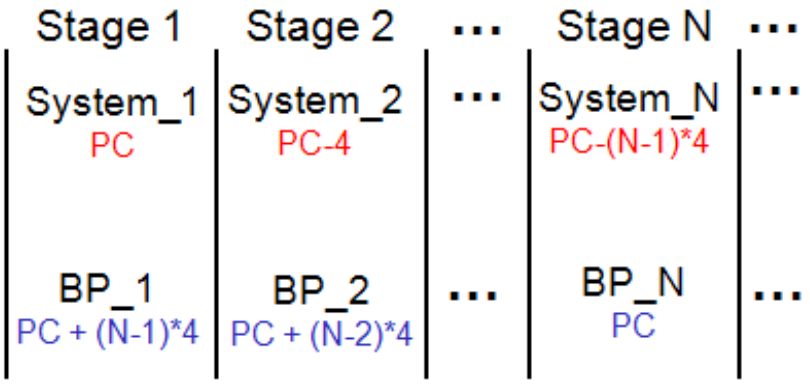


Figure 3-13: The expectation of Lookahead Pipelining BHU.

Instruction reaches the last stage of branch target generation as it enters first stage of system pipeline. In one cycle time, the target address would be ready and match the required branch prediction timing.

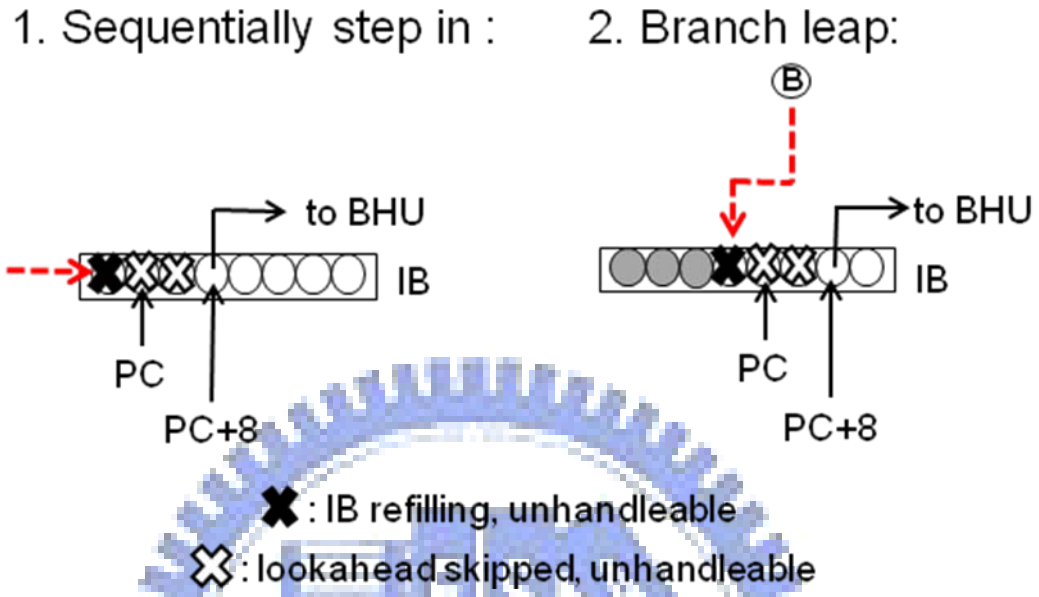


Figure 3-14: An example of a three-cycle target generation (i.e. N=3).

Here we elaborate the case of a three-cycle BHU to draw a clear picture of how lookahead pipelining works. As can be derived from Figure 3-14, in this N=3 scenario, for a prediction to be ready in time, the BHU should start 2 cycles before the branch instruction itself actually enters the pipeline. Two possible cases of a cache line change are shown above. Assuming instruction buffer refilling takes up one cycle, the first instruction that is supposed to be executed in the cache line always suffers as being unhandleable, marked as the black cross. In the consequent cycle, suppose the program flows sequentially, a hit would occur in the instruction buffer. This very moment's snapshot is presented in Figure 3-15. As PC pointed at the currently being executed instruction, a 2-cycle lookahead must be carried out, hence (PC+8) ought to get started with its target address generation for prediction to be ready in time. The lookahead brutally sacrifices PC and (PC+4) in order to keep the rest of the instructions in this cache line attended. Seemingly, PC and (PC+4) become ones that are unhandleable and are marked with white cross. All the crossed instructions, either black or

white, are out of BHU's hand. And if there are any branches among these unhandleable instructions, they should be taken care of by the RBTB.

It cannot be over emphasized that lookahead pipelining is a conceptual method proposed to for precaution and completeness of this research, and according to a lot of circuit design researches, which we may never have to put to practice.

3.4.3 Indirect Branch Misprediction

The third constrain lies in prediction for indirect branch. Once again, in Alpha instruction set, JMP, JSR, and RETN each has its own dedicated register to access in order to obtain target address. The register read operation is supposed to be done in ID stage, assuming a conventional five-stage pipeline. However, with our BHU design, the register buffers are read in IF stage to deliver branch prediction. This makes an one-cycle-early register read operation. Chances are that the value is not ready for access yet at this time. For example, there could be an in-flight LOAD or MOV instruction still working on the value that should be accessed. The problem, as can be seen, is in fact a data dependency hazard. Fortunately, true dependency like this is mostly well-handled by the compiler. So even if the data dependency causes a bad target prediction, it still belongs to one of the minority cases. And the mispredcited indirect branches would be put into RBTB for future reference.

The real challenge of indirect branch prediction actually comes from the **outlaws**. Although the instruction set defines register entries for indirect branches to keep their targets, sometimes compiler can violate the convention by keeping them elsewhere. For example, when sequential CALLs not interleaved by RETNs are being optimized. In such a scenario, compiler can selectively keep the return address in currently unoccupied register entries to achieve multiple return addresses tracking without pushing information into program stack in memory. These rule-breakers are the outlaws that cannot be controlled. The only thing we can

do about it is to monitor their every move by storing them into the RBTB or to endure the branch penalty and wait until their correct target addresses are verified in the EXE stage. Figure 3-15 shows the instruction format and describes how an indirect branch obtains its correct target address in the EXE stage.

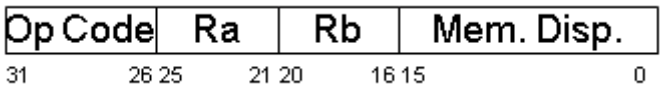


Figure 3-15: Instruction format of indirect branches in Alpha instruction set.

The correct target address is obtained by accessing the register entry indexed by field Rb, and then set the two least-significant bits of the value to 0 (taking only the word address).

To sum up, the cause of indirect branch misprediction can be traced back to two reasons: data dependency and unconventional register usage. Both can simply be dealt with using a compiler that takes these two issues into consideration. Likewise, this is a problem that can be transformed into a compiler co-design topic.

3.5 BHU in More Complicated Pipeline

The design of BHU is intended to target general purpose processors. We depicted the design in a conventional single-issue, five-stage pipeline, where the design works smoothly and efficiently. However such pipeline configuration can be considered old-fashioned and improvable through the performance hungry eyes of today. Actually, there are certain things we can do to apply our design to more sophisticated systems and exploit most functionality of our BHU design, if not fully. In this section, we would show how the BHU design adapts to different environment of more complicated pipeline.

Take a quick glance at the modern architecture. It's not hard to find that with the pursuit of higher performance, systems today have some major improvement implemented, which

leads to more complicated pipeline designs. There are many different ideas fulfilled in nowadays processor designs, the two mainstream aspects are to increase issue width and clock rate. By realizing multiple-issue, the pipeline can gain theoretical multiple through put; and driving up the clock rate shortens the time that pipeline periodically delivers the finished instructions. Both methods are common strategy implemented to increase the system performance and these techniques achieved major efficiency breakthrough for the past decades. In a system of wide issue width and high clock rate, changes can be made for our design to adapt. Note that out-of-order execution pipeline does not at all complicate the design of BHU. Since even the instructions can be executed in out-of-order style, the nature of human-written program is still sequential. So the pipeline frontend, where instructions are fetched and branch predictions are made, still goes in-orderly. In this section, we would focus on the polymorphism of BHU in wider issue width and higher clock rate systems.

3.5.1 Multiple Issue Systems

Firstly, we should take a look at multiple issue architecture. Assuming an n-issue system, n instructions are fetched from cache every cycle. As a guide of the fetcher, it is the branch predictor's task to identify branch instructions among these n instructions and provide possible target addresses and credible taken or not taken forecasts. It is obvious that the only thing differs from what has been depicted in aforementioned single-issue pipeline is the quantity. The number of prediction that should be made in one cycle is a not much of a deal to handle. A straightforward idea emerges as to duplicate the BHU for each pipe in such a system. Figure 3-16 shows the case of a 2-issue pipeline.

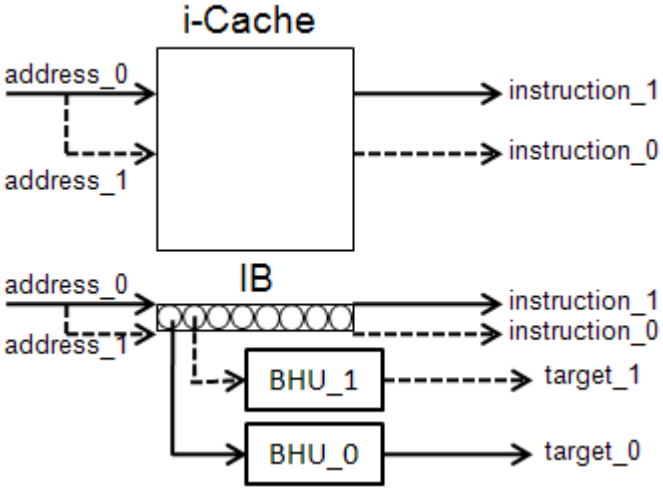


Figure 3-16: BHU adaption in a 2-issue pipeline.

Figure 3-16 seems to work fine, but actually there could be BHU hardware redundant in this scenario. Recall that the BHU contains two parts of hardware for both PC-relative and indirect branch. According to earlier assumption, the PC-relative route, which lies on an integer adder, would be on the critical path. Given the tight timing constrain, it would be risky for the two instructions to utilize one integer adder sequentially (adder works in its own clock and calculate two target addresses in one system cycle; buffer must be assigned to latch outputs). But for the indirect branch, however, we can save a duplication of the register buffers. The register buffers contain the values of corresponding register entries that should be ready to be accessed as target address by this moment. Hence one copy of the register buffers ought to be sufficient for this purpose. Figure 3-17 shows the version of this redundant elimination. In addition, the two Partial Decoders (PDs) may have a chance to be further simplified. Since partial decoding actually incurs very short latency (which is shown to be equal to three AND sequential gates delay), instructions of different pipes have a chance to utilize a common Partial Decoder (PD) in turn (same way as the adder works described above) without breaking any timing constrain. Of course, this is another decision that should be made based on the issue width, cycle time, and PD latency.

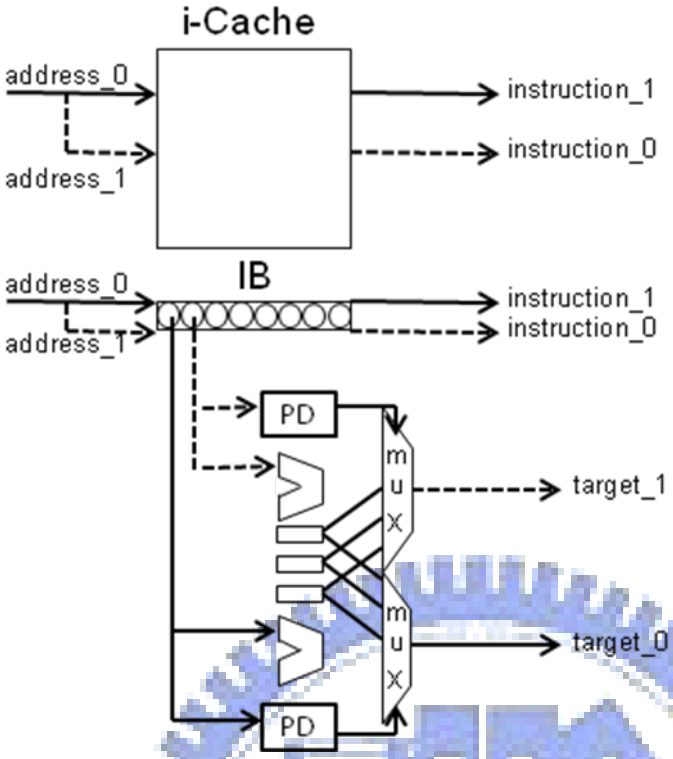


Figure 3-17: BHU adaption after eliminating the redundant parts.

3.5.2 High Clock Rate Systems

Secondly, the high clock rate issue. A system with high clock rate can significantly shorten the amount of time for the BHU to work within. Although this may seem terrifying given that the branch prediction is a now-or-never job at the very frontend of system pipeline, in practice, short cycle time is not as scary as it seems. By taking a closer look at the components of the BHU, we can find it to be a lightweight and short latency unit.

Figure 3-19 shows each and every part of the BHU design. As can be seen in the figure, instruction buffer can be accessed in a very short latency. The Partial Decoder (PD) consists of four parallel AND, one with a two-bit input and the other three with a six-bit input. The critical path of this part is clearly dominated by six-bit input AND, which in the worst case can be formed by three levels of two-bit AND gates. The integer adder, which can first be shorten in length and as mentioned early in this thesis, can be well customized and clocked up

to 10GHz in practical [8][9][10], is likely to meet the timing requirement. The set to zero and cascade operation following integer addition barely cause any delay. The three register buffers are even more accessible with short latency given each is of just 32-bit in size. And the MUX can be implemented by pass transistors to lower pass through latency.

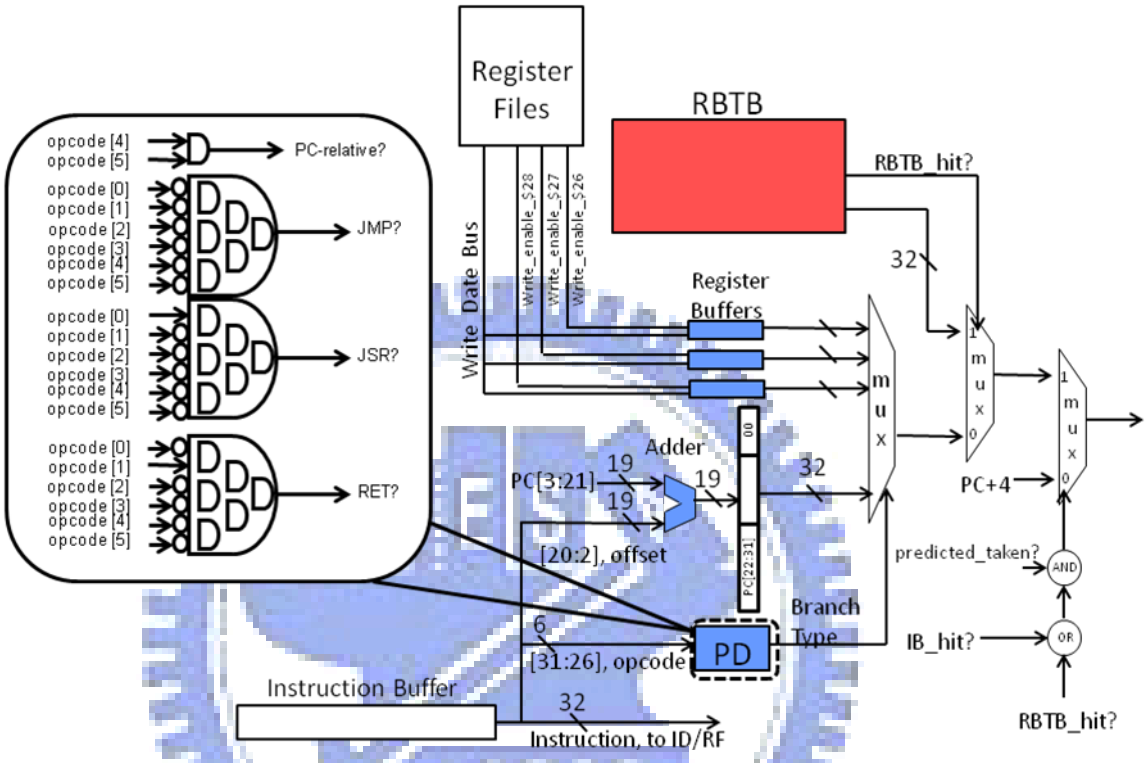


Figure 3-18: Components in BHU.

As a matter of fact, the true challenge of applying BHU to a high clock rate system lies in instruction buffer refilling. The pipeline design in such systems tends to be deeper than the conventional five stages. And each task is divided into more coarse grain subtasks spread over more pipeline stages. In this case, the assumption that instruction can be fetched from the instruction cache in one cycle may not stay true. Thus the number of cycles required to complete an IB refill may also be strained. Since every cycle needed to complete an IB refill causes an unhandleable instruction to BHU, it is evident how this is going to affect our design.

The number of cycles required to access instruction cache varies from processor to processor. Most state-of-the-art architecture design well-known today wouldn't take more

than three cycles to access instruction cache. For example, Intel Itanium processor only need one cycle for instruction cache access, clocking at around 1GHz; while ARM9 based processors often require two to three cycles to fetch an instruction. We intend to turn this into an experimental parameter, by simulating different settings and find out the overall applicability of the BHU design in modern pipeline systems. The results would be presented in the next chapter.

3.5.3 Summary of BHU Adaption

To sum up, BHU encounters two major challenges when it comes to sophisticated pipeline adaption. Issue width problem can be brutally handled by providing more hardware resource to the BHU. High clock rate issue is trickier due to the fact that it involves instruction buffer refilling, which is the most critical weak part of the BHU design. As the number of cycle required for instruction cache access is put into consideration, the problem can be transform to how many instructions would be unhandleable after an instruction cache line change. The Reduced BTB (RBTB) serves as a counterpart to complete the BHU functionality as the original expectation. And in system that unhandleable cases increase, heavier workload is anticipated for RBTB.

Chapter 4 Experiment

In this chapter, experiment methods are described. We implement our BHU design in a high level language simulator. The simulator is chosen to be the well-known and trusted SimpleScaler 3.0 [11]. For testing program, SPEC2K [12] is used as benchmark to give a general idea of how common programs behave in reality. To evaluate power, statistic from SimpleScaler 3.0 and CACTI 4.2 [13] [14] are used together to model the power consumption. The simulation environment is listed in Table 4-1:

Table 4-1: Simulation environment.

Simulator	SimpleScaler 3.0
Benchmark Set	SPEC2K, 23 benchmarks tested
Power Tools	CACTI 4.2, WATTCH
Instruction Set	Alpha instruction set
Pipeline Description	Conventional five-stage pipeline
Issue	In-order, single-issue
L1 I-Cache Size	16KB (line size = 32B)
L1 D-Cache Size	16KB
L2 Cache Size	2MB
Instruction Buffer Size	32B

The benchmark SPEC2K is a well-known and trusted evaluation tool for general purpose testing. The benchmark set measure the performance of the processor, memory and compiler on the tested system. The benchmark set can be put into integer and floating point parts, each of 12 and 14 individual benchmarks respectively. The detail description for each and every one of the benchmarks is listed in Table 4-2 and Table 4-3.

Table 4-2: Integer benchmarks in SPEC2000.

Benchmark	Language	Category	Descriptions
164.gzip	C	Compression	SPEC's version of gzip performs no file I/O other than reading the input. All compression and decompression happens entirely in memory.
175.vpr	C	FPGA Circuit Placement and Routing	VPR is an example of an integrated circuit computer-aided design program, and algorithmically it belongs to the combinatorial optimization class of programs.
176.gcc	C	C Programming Language Compiler	176.gcc is based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled.
181.mcf	C	Combinatorial Optimization	A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation.
186.crafty	C	Game Playing: Chess	Crafty is a high-performance Computer Chess program that is designed around a 64bit word. It runs on 32 bit machines using the "long long" (or similar, as _int64 in Microsoft C) data type.
197.parser	C	Word Processing	The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax.
252.eon	C++	Computer Visualization	Eon is a probabilistic ray tracer based on Kajiya's 1986 ACM SIGGRAPH conference paper. It sends a number of 3D lines (rays) into a 3D polygonal model.
253.perlbmk	C	PERL Programming Language	253.perlbmk is a cut-down version of Perl v5.005_03, the popular scripting language.
254.gap	C	Group Theory, Interpreter	It implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).

255.vortex	C	Object-oriented Database	VORTEx is a single-user object-oriented database transaction benchmark which which exercises a system kernel coded in integer C.
256.bzip2	C	Compression	256.bzip2 is based on Julian Seward's bzip2 version 0.1. SPEC's version of bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in memory.
300.twolf	C	Place and Route Simulator	The TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips.

Table 4-3: Floating point benchmarks in SPEC2000.

Benchmark	Language	Category	Descriptions
168.wupwise	Fortran 77	Physics / Quantum Chromodynamics	"wupwise" is an acronym for "Wuppertal Wilson Fermion Solver", a program in the area of lattice gauge theory (quantum chromodynamics).
171.swim	Fortran 77	Shallow Water Modeling	Benchmark weather prediction program for comparing the performance of current supercomputers. The model is based on the paper, by Robert Sadourny.
172.mgrid	Fortran 77	Multi-grid Solver: 3D Potential Field	172.mgrid demonstrates the capabilities of a very simple multigrid solver in computing a three dimensional potential field.
173.applu	Fortran 77	Parabolic / Elliptic Partial Differential Equations	Solution of five coupled nonlinear PDE's, on a 3-dimensional logically structured grid, using an implicit psuedo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix.
177.mesa	C	3-D Graphics Library	Mesa is a free OpenGL work-alike library. Since it supports a generic frame buffer it can be configured to have no OS or window system dependencies.

178.galgel	Fortran 90	Computational Fluid Dynamics	This problem is a particular case of the GAMM (Gesellschaft fuer Angewandte Mathematik und Mechanik) benchmark devoted to numerical analysis of oscillatory instability of convection in low-Prandtl-number fluids.
179.art	C	Image Recognition / Neural Networks	The Adaptive Resonance Theory 2 (ART 2) neural network is used to recognize objects in a thermal image.
183.equake	C	Seismic Wave Propagation Simulation	The program simulates the propagation of elastic waves in large, highly heterogeneous valleys, such as California's San Fernando Valley, or the Greater Los Angeles Basin.
187.facerec	Fortran 90	Image Processing: Face Recognition	This is an implementation of the face recognition system described in M. Lades et al. (1993), IEEE Trans. Comp. 42(3):300-311.
188.ammp	C	Computational Chemistry	The benchmark runs molecular dynamics (i.e. solves the ODE defined by Newton's equations for the motions of the atoms in the system) on a protein-inhibitor complex which is embedded in water.
189.lucas	Fortran 90	Number Theory / Primality Testing	Performs the Lucas-Lehmer test to check primality of Mersenne numbers 2^p-1, using arbitrary-precision (array-integer) arithmetic.
191.fma3d	Fortran 90	Finite-element Crash Simulation	FMA-3D is a finite element method computer program designed to simulate the inelastic, transient dynamic response of three-dimensional solids and structures subjected to impulsively or suddenly applied loads.
200.sixtrack	Fortran 77	High Energy Nuclear Physics Accelerator Design	The function of the program is to track a variable number of particles for a variable number of turns round a model of a particle accelerator such as the Large Hadron Collider (LHC) to check the

			Dynamic Aperture (DA) i.e. the long term stability of the beam.
301.apsi	Fortran 77	Meteorology: Pollutant Distribution	Program to solve for the mesoscale and synoptic variations of potential temperature, U AND V wind components, and the mesoscale vertical velocity W pressure and distribution of pollutants C having sources Q.

The simulation is set to run a fix number of instructions in the benchmark on the environment built by SimpeScaler to verify the functionality of our design. In our case, we fast forward 500-million instructions at the beginning of each benchmark and then run the 500-million consecutive instructions to reflect the overall general case during program executions. Different configurations of BTB are tested in order to find optimization. Information such as: total cycle used, BTB read/write count, prediction accuracy, and BHU logic switching are kept track of for our power and performance evaluation.

For accuracy evaluation, SimpeScaler provides all the statistics during the program execution. However, only results related to branch prediction are of our interest here. The one thing we care about is the accuracy change that may be caused by the BHU design to the system. The branch prediction accuracy is defined by:

Accuracy = (number of branches correctly predicted) ÷ (total number of branch)

A correct branch prediction should include two parts: a correct direction prediction and a correct target address prediction. With nowadays direction predictor providing very high direction prediction accuracy, the overall branch prediction accuracy can be used as an impartial measurement for our BHU + RBTB design.

For power evaluation, equations below are applied to calculation the final total power. Each term in the power equation is available using CACTI 4.2 power tool with correct configuration set. For conventional BTB:

$P_{Total} = P_{BTB\_Read} \times BTB\_Read\_Count + P_{BTB\_Write} \times BTB\_Write\_Count + P_{BTB\_Leakage} \times Execution\_Cycle$

, where

P_{BTB_Read} = power requirement for a read operation of conventional BTB

P_{BTB_Write} = power requirement for a write operation of conventional BTB

$P_{BTB\_Leakage}$ = leakage power of conventional BTB

For Reduced BTB (RBTB) with BHU as hardware overhead:

$P_{Total} =$

$P_{RBTB\_Read} \times RBTB\_Read\_Count + P_{RBTB\_Write} \times RBTB\_Write\_Count + P_{RBTB\_Leakage} \times Execution\_Cycle + P_{BHU\_switching} \times BHU\_Switching\_Count + P_{BHU\_Leakage} \times Execution\_Cycle + P_{Additional\_Control\_Overhead}$

, where

P_{RBTB_Read} = power requirement for a read operation of reduced BTB

P_{RBTB_Write} = power requirement for a write operation of reduced BTB

$P_{RBTB\_Leakage}$ = leakage power of reduced BTB

$P_{BHU\_switching}$ = the sum of power requirement to exercise partial decoder, adder and perform read operation of register buffer

$P_{BHU\_Leakage}$ = leakage power of conventional BHU (mostly from register buffers)

$P_{Additional\_Control\_Overhead}$ = power consumption of additional MUXs and AND/OR gates

The final power is calculated then compared in normalized form, given that CACTI should be used as power consumption reference. The normalized form eliminates any debatable factor by seeing only the relative value presented in ratio.

4.1 Baselines Determination

In this section, we determine the configurations of conventional BTBs as our baselines for comparison. There are two major parameters when it comes to configuration of a BTB: **number of sets** and **associativity**. Note that number of sets multiplied by associativity equals to number of total entries. In modern processor designs, we find BTBs in common systems tend to have more than 256 entries. Also, statistic shows that BTBs with higher associativities outperform ones with lower associativity. However, large number of entries and high associativity comes with an expensive price of hardware complexity and access power. Therefore in practical, most BTBs are implemented as 256-entry to 512-entry [15] [16], as 2-way or 4-way set-associative. For example, Alpha 21264 embedded a 512-set, 2-way BTB; and Pentium Pro embedded a 128-set, 4-way BTB [16].

In order to determine our baseline, we ran simulations to have a clear view of how different configurations affect the accuracy of branch predicting mechanism. All the configurations are simulated in our experiments, and the results of the prediction accuracy and IPC they delivered are shown from Figure 4-1 to Figure 4-4.

In the following experimental results presentation of this chapter, all BTB or RBTB configurations are shown in the form of "(R)BTB_<number of sets>_<associativity>". The prefix depends on the environment we're conducting experiments on. With prefix "RBTB", it is implied that a BHU also exist in the system to branch predictor's aid; and with prefix "BTB" indicates that this is a system with conventional BTB dealing with all the branches on its own.

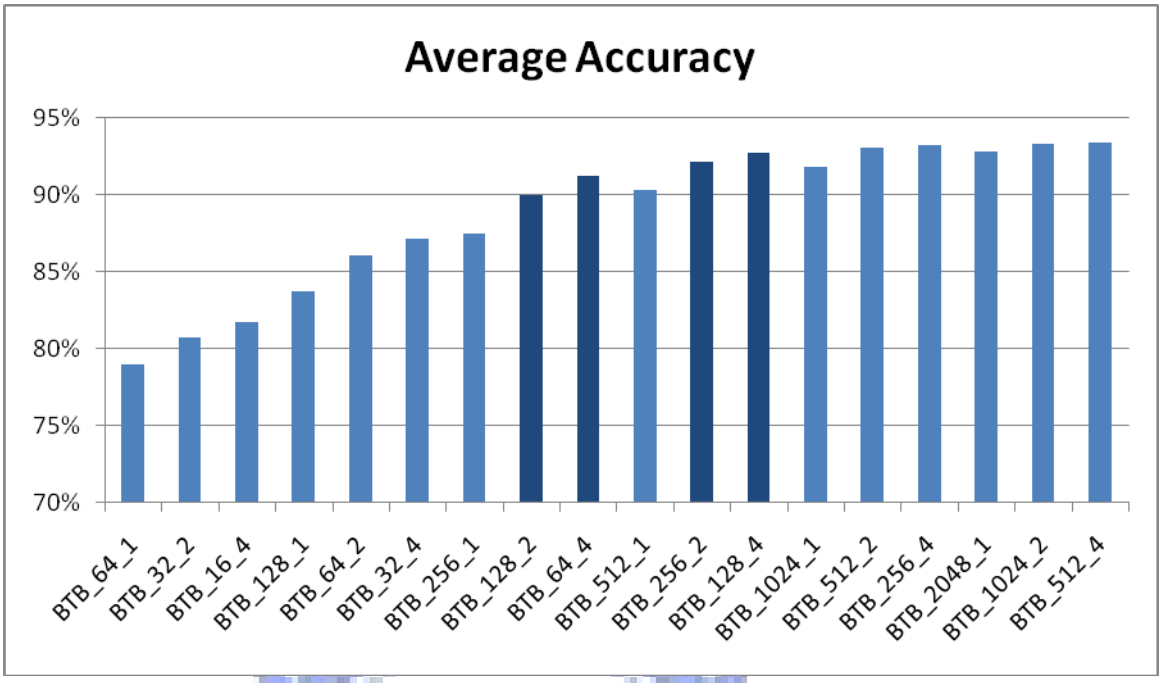


Figure 4-1: Average accuracy of different configurations of conventional BTBs.

The results in Figure 4-1 are presented in sets, each of three configurations setting to the same number of BTB entries but different associativity. Starting from the left is BTB of 64 entries configured in direct-map, 2-way, and 4-way set associative. Moving all the way to the right, the number of BTB entries increases in order of 2's power. For viewing convenience, results in Figure 4-1 are sorted according to accuracy and presented again in Figure 4-2 for a clearer view of how different size of BTBs perform in the benchmark set SPEC2K. Average IPC would be presented in the same way in Figure 4-3 and Figure 4-4.

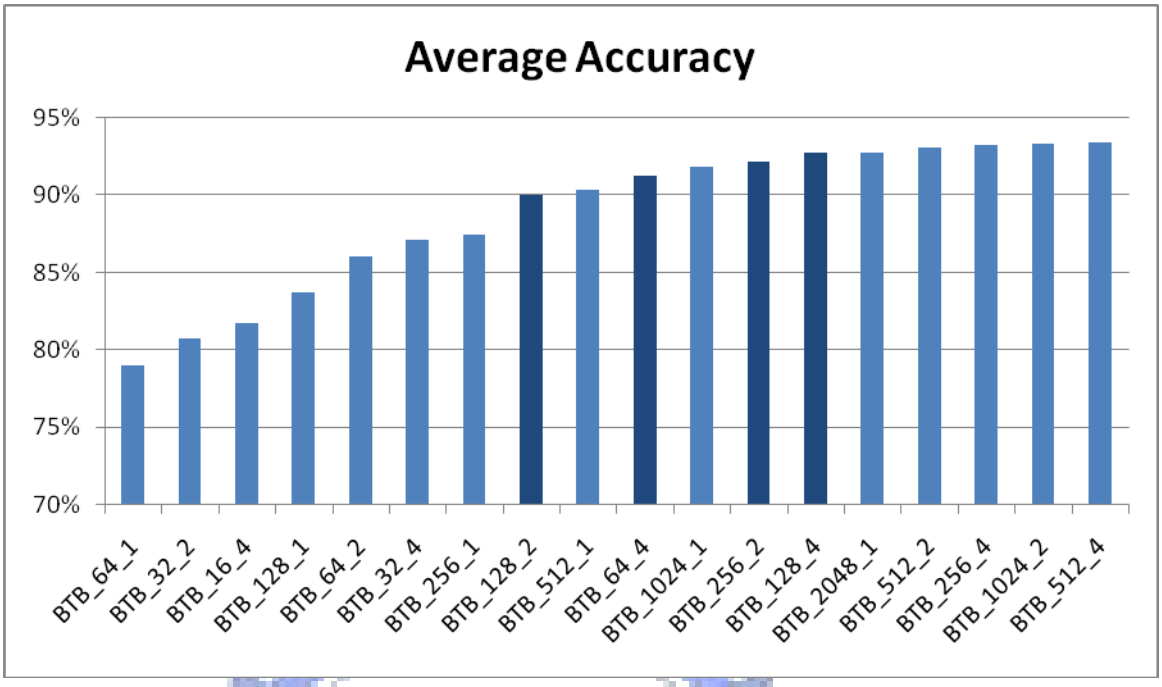


Figure 4-2: Average accuracy of different configurations of conventional BTBs in sorted order.

Average IPC is presented in the same way as average accuracy in the following. First grouped according to size and then in increasing order of IPC.

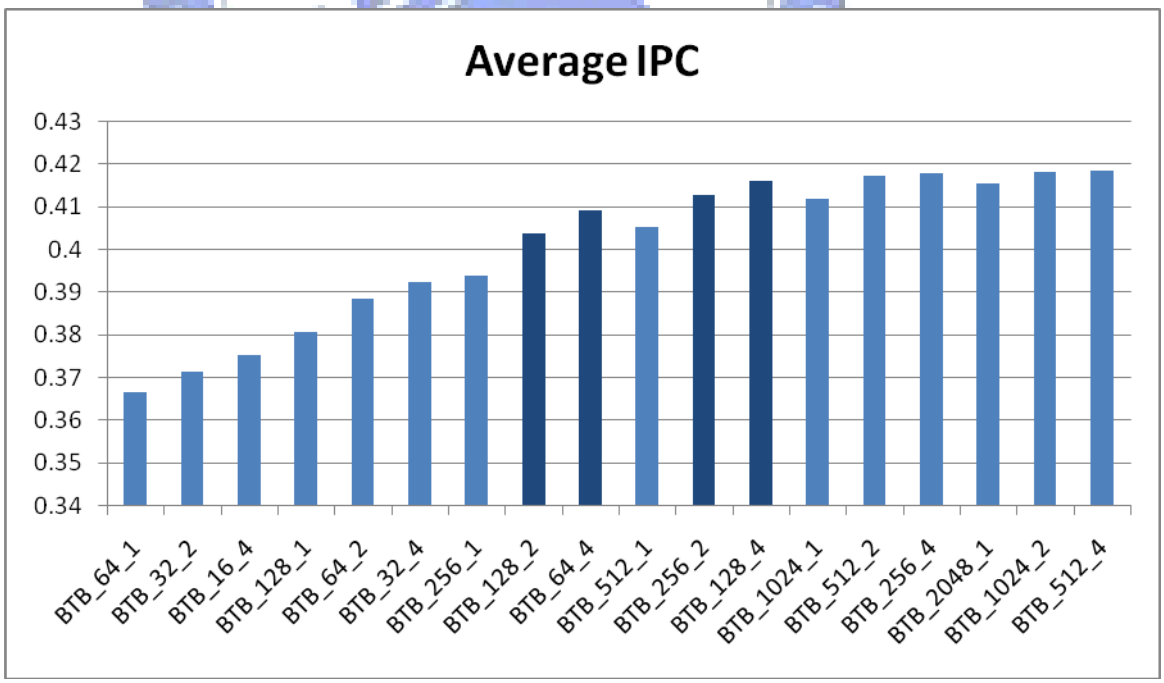


Figure 4-3: Average IPC of different configurations of conventional BTBs.

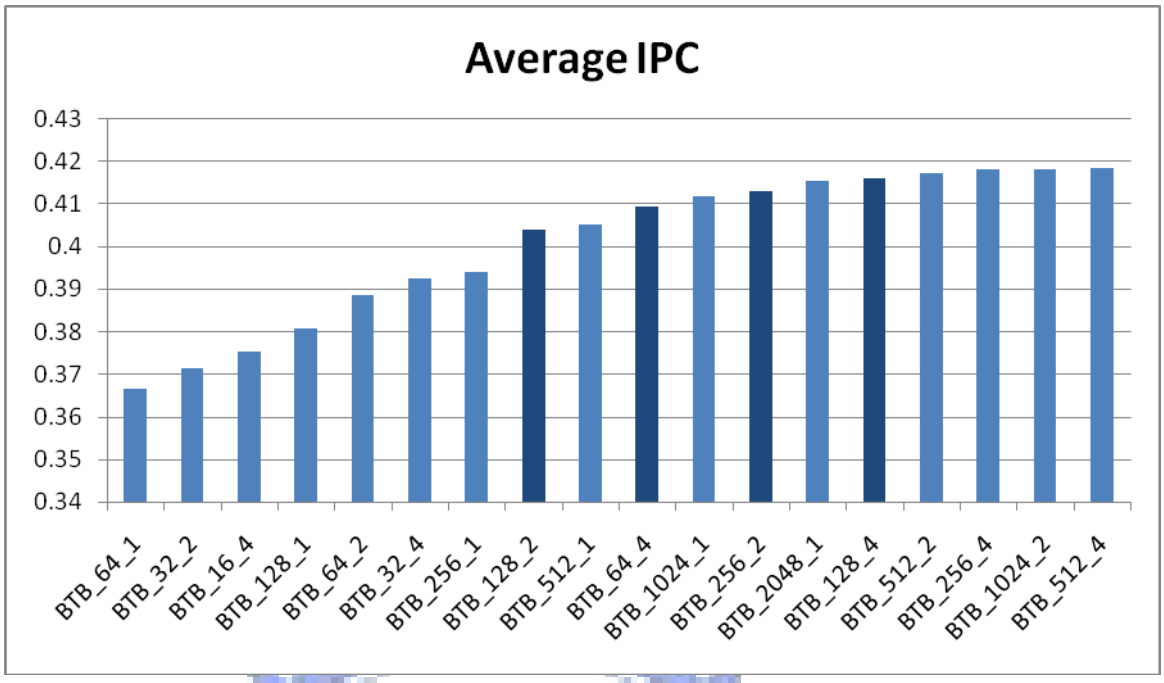


Figure 4-4: Average IPC of different configurations of conventional BTBs in sorted order.

The configurations of 256 and 512 entries, 2-way and 4-way associative BTBs are highlighted in above figures. The highlighted bars in the figure are mostly located at the points before the curve starts to fall drastically. It's not very hard to realize that these BTB configurations actually provide good balance between size and accuracy. Later on in this chapter, all these configurations would be the ones that are compared to as reference with our BHU + RBTB design.

4.2 Benchmark Evaluation Results

In this section, our simulation results are presented. SimpleScaler simulator is modified to follow expected working flow of our BHU + RBTB design. The most ideal scenario is shown in the first part. Then more complicated assumptions are made by passing in different parameter to model the circumstances for high clock rate pipelines where BHU can suffer from long delay of instruction buffer refilling. The outcome we care about here includes IPC, which reflects the overall performance; prediction accuracy, which reflects the functionality

and effectiveness of branch predictor; and last but not least, the power/energy consumption, which reflects the value of this research and was set to be the goal from the very start.

4.2.1 Case N = 1

Here we start with the most ideal case. Assuming we have a system, where the BHU can be fully functional within one cycle time and the refilling process also cost one cycle only. According to simulation description, instruction per cycle (IPC) is used as a criterion to measure overall performance. The first thing to do in order to filter the results is to compare the overall average IPC of our BHU + RBTB designs to conventional BTBs. We intend to target our work as a performance degradation-free design and to achieve as much BTB size reduction as possible, so only ones with similar performance compared to baseline configurations are presented here. Figure 4-5 shows the average IPC of this case.

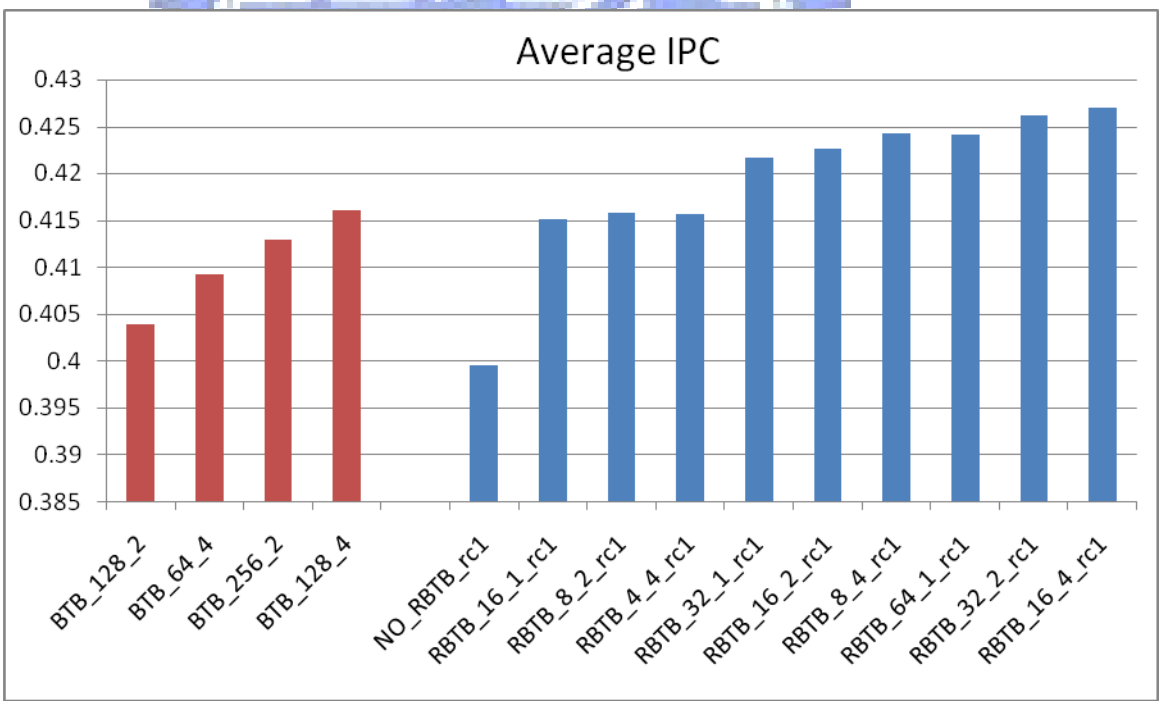


Figure 4-5: Average IPC in N =1 case.

The postfix “rc1” represents the value of N, in this case, 1.

As can be seen in Figure 4-5, by the help of BHU, performance what used to take up 256 or 512 BTB entries to deliver, now can be provided with only 16 or 32 entries. That is no

more than one-eighth of original BTB size. And even more, the 16-entry or 32-entry RBTB can be implemented in to a direct-map form which can both contribute to area and power reduction of the structure.

Figure 4-6 shows us exactly how much gain came from the BHU. In this scenario, BHU can deliver correct branch targets for about 70% of PC-relative branches, and about 7% of indirect branches. The RBTB in the system can deliver another 15% of the PC-relative branch prediction accuracy, and less than 2% of the indirect branch prediction accuracy. In total, our design achieved approximately 93% of prediction accuracy with a much smaller RBTB in size compared to convention.

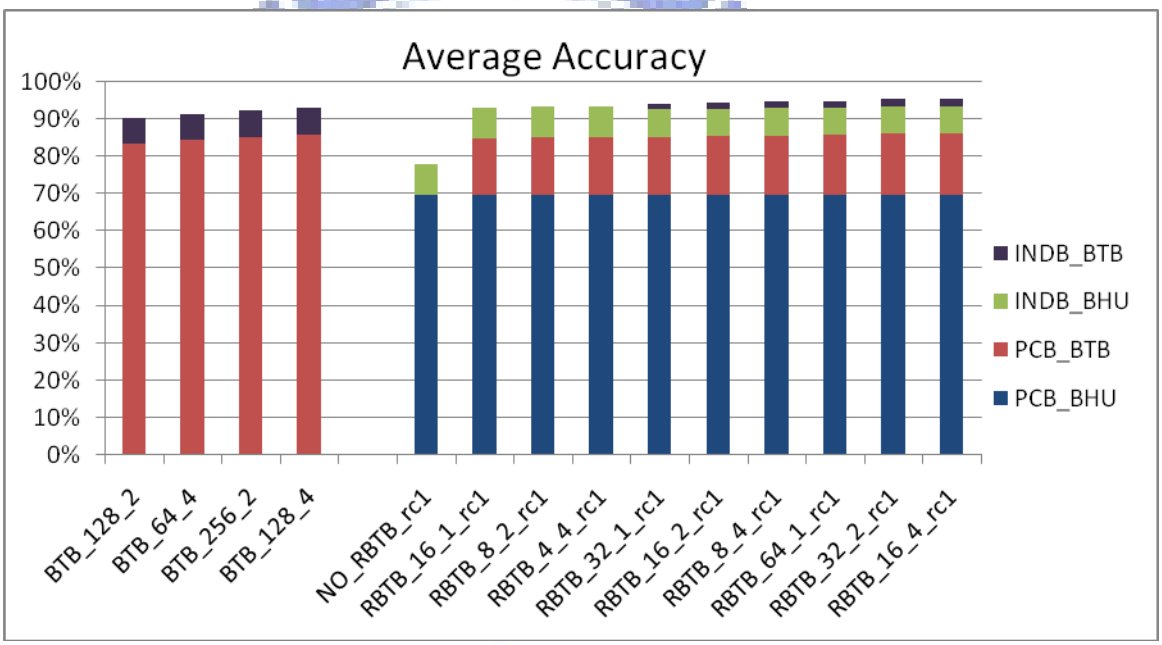


Figure 4-6: Average accuracy in N=1 case.

Note that in Figure 4-6, the order of the configuraiton in increasing is slightly different from Figure 4-5. It must be pointed out that the results presented here are the average of the whoel SPEC2K benchmark set. Thus some of the details may differ slightly due to this very reason. This stay true in all of the results presented in this chapter. Another fact worth mentioning is that for RBTB of 16-entry, no good prediction can be provided for indirect branches.

Now that we know with 16-entry or 32-entry of RBTBs together with BHU design, the average performance and accuracy outbeat conventional BTBs of 256 to 512 entries. Next thing we would like to show is exactly how much power is saved by all of these works.

The total power consumption is calculated according to the power equation mentioned above. Storage related power consumption including read power, write power and leakage power are available by using CACTI. But the power consumption of the overhead is a bit more complicated. Break down the hardware overhead we added, there are: an integer adder (adder legnth is actually adjustable down to 19 or 20 bits, but for modeling simplicity, it is treadted as a 32-bit full adder here), three 32-bit register buffer, a partial decoder with complexity of 19 2-input AND gates (6-input AND is measured by 6 2-input AND gates, and there's another 2-input AND gate, to a total of 19 2-input AND gates), one 4-input fast MUX, and one 2-input fast MUX. The ratio of BTB power consumption and integer adder power consumption can be aquired by WATTCH [11]. And the partial decoder and the MUXs are considered much less complex than an adder. For simplicity, the non-sotrage part overhead is modeled as two 32-bit integer adder, and each adder is modeled as system pipeline integer ALU, which power consumption is provided by WATTCH as a portion of BTB power consumption [7][8][9]. Note that this is obviously an over estimation, the actual power consumption should be less than this modeling. Figure 4-7 shows a normalized power consumption result. All the values are normalized to a conventional 128-set, 4-way (512-entry) BTB. The configurations are sorted in IPC order, the higher IPC to the right. Compared to a 512-entry conventional BTBs, our design can achieve more than 75% of power reduction with strictly no performance loss; and more than 65% power reduction when compared to 256-entry conventional BTBs.

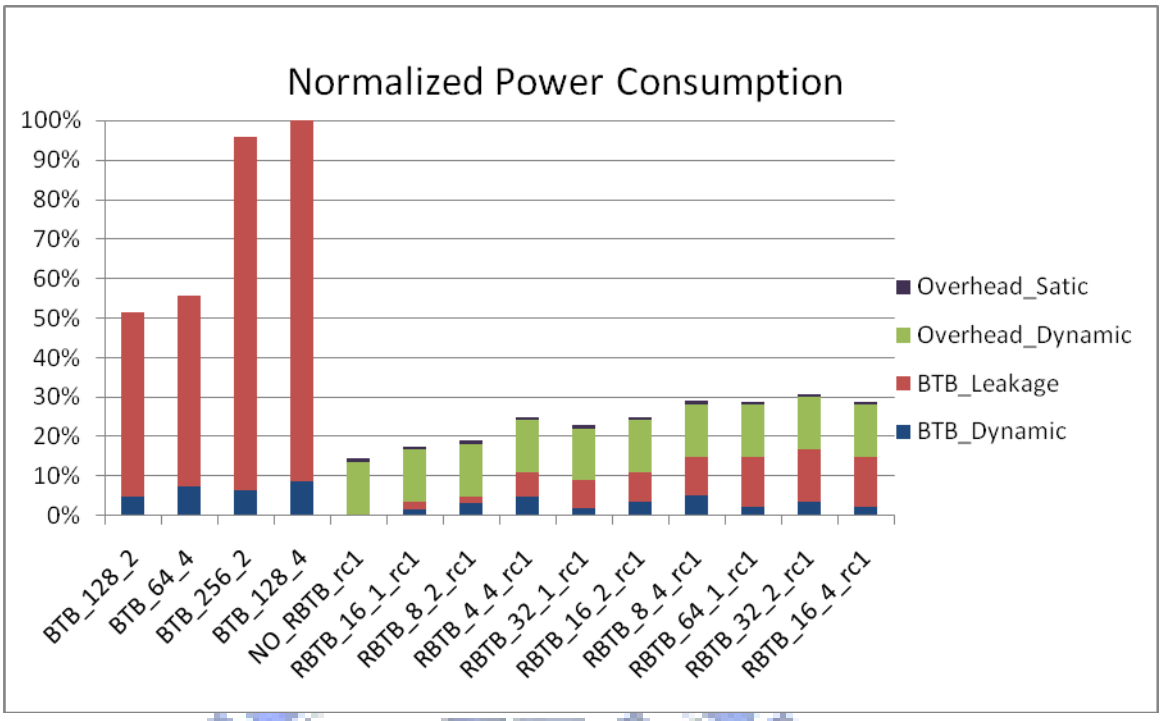


Figure 4-7: Normalized power in N=1 case.

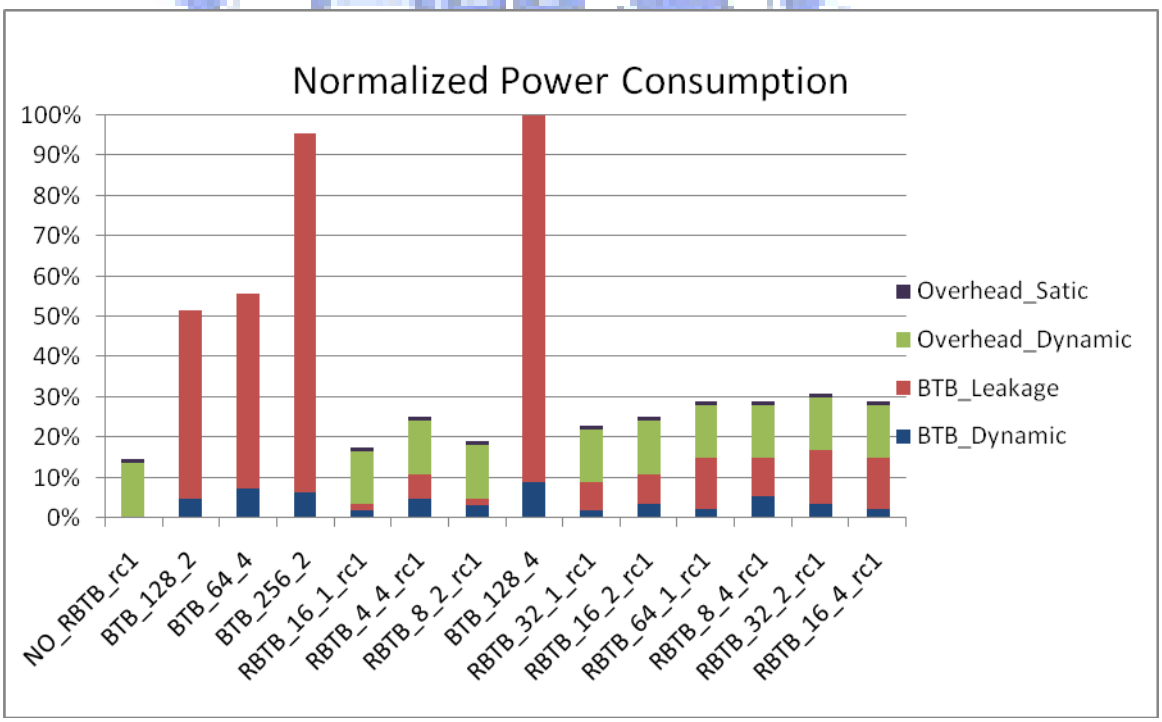


Figure 4-8: Normalized power sorted in increasing order of IPC in N=1 case.

4.2.2 Case N = 2

In the next part, simulation parameters are changed to model the BHU + RBTB design in more complicated pipelines. Here we present the N=2 case, which means there are two unhandleable instructions on every instruction cache change. In this case, we can imagine that number of unhandleable branches increse and information load in RBTB as well. More entries are needed for RBTB to keep the degradation-free performance. The results of IPC, accuracy, and power reduction are shown in Figure 4-8, 4-9, and 4-10, respectively.

Note that the N=2 case can be used to model a combination of situations. First, it can represent the system pipeline with single cycle cache access and a BHU with two cycle of latency. Second, it can also stand for a system pipeline with two cycle instrucion cache access while embedded with a single-cycle BHU. It can be understood that the N value means the total cycle of BHU delay and the number of cycles it takes to finish an instruction buffer refill. Generally, we assume that the BHU can fit into a single cycle given the short latency it incurrs mentioned above. And all the N parameter cases can be seen as an instruction buffer refill modeling for different implementations of instruction cache design.

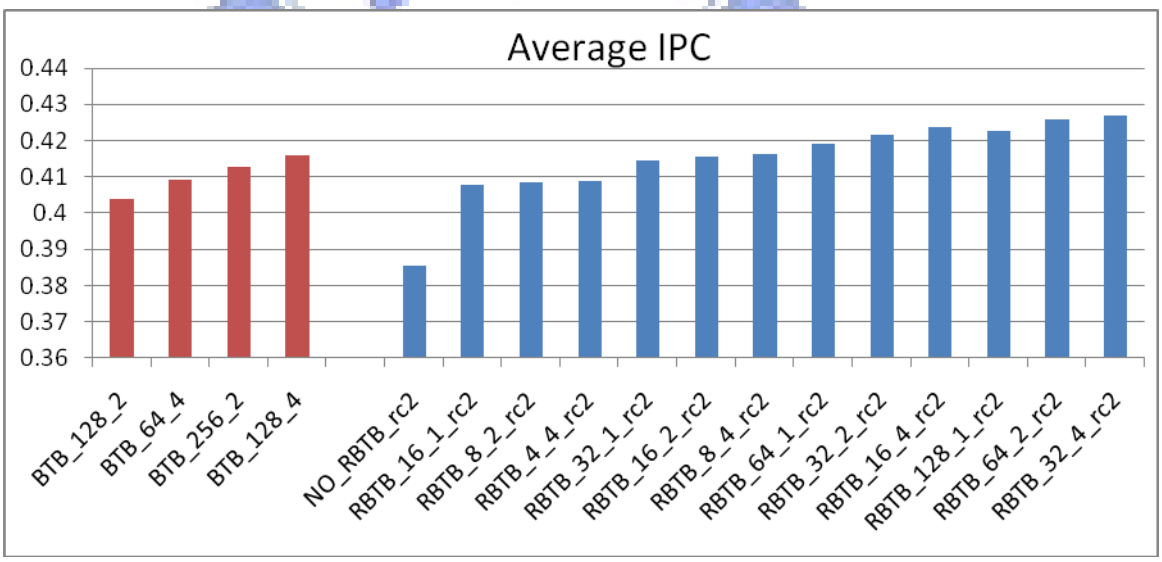


Figure 4-9: Average IPC in N=2 case.

The postfix “rc2” represents the value of N, in this case, 2.

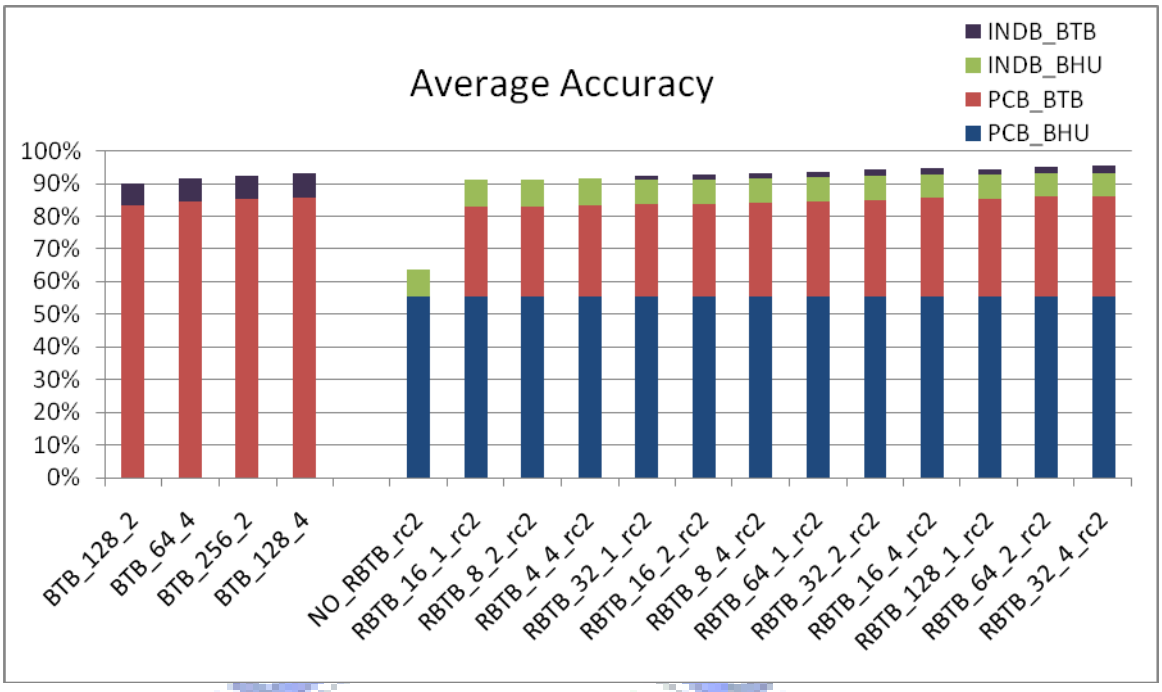


Figure 4-10: Average accuracy in N=2 case.

It can be seen that the percentage of PC-relative branches handleable by BHU decrease to about 55%. The accuracy lost of these unhandleable branches become the burden of RBTB. So now it takes more entries or higher associativity for our BHU + RBTB design to outbeat conventional BTBs.

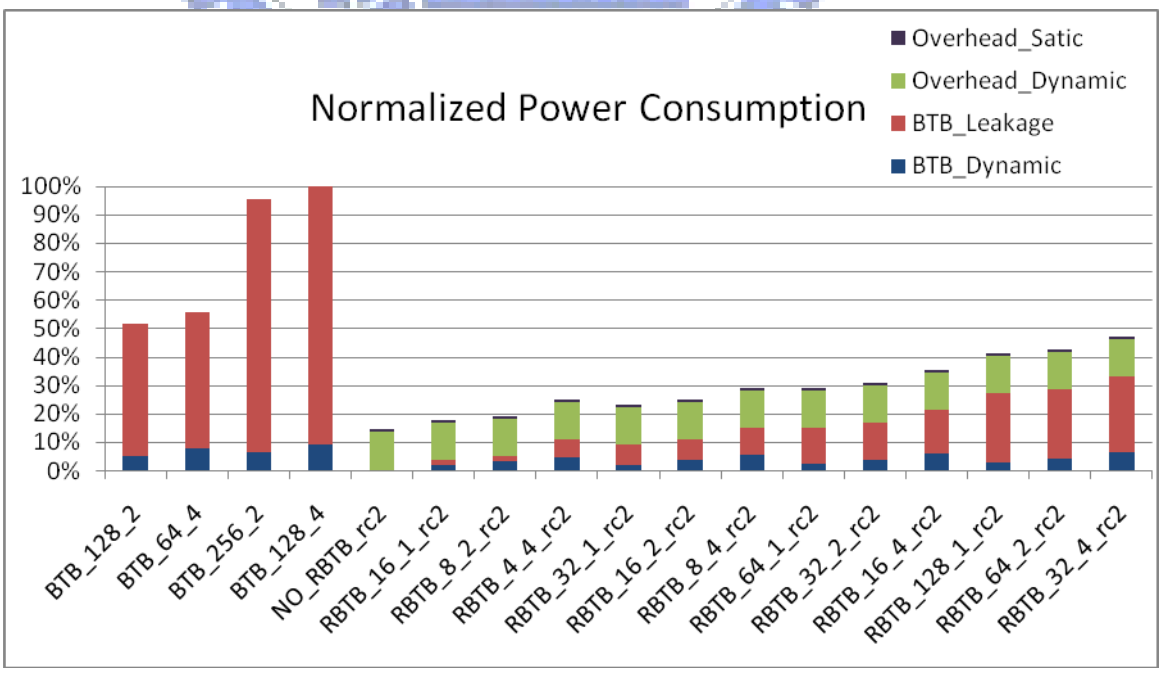


Figure 4-11: Normalized power in N=2 case.

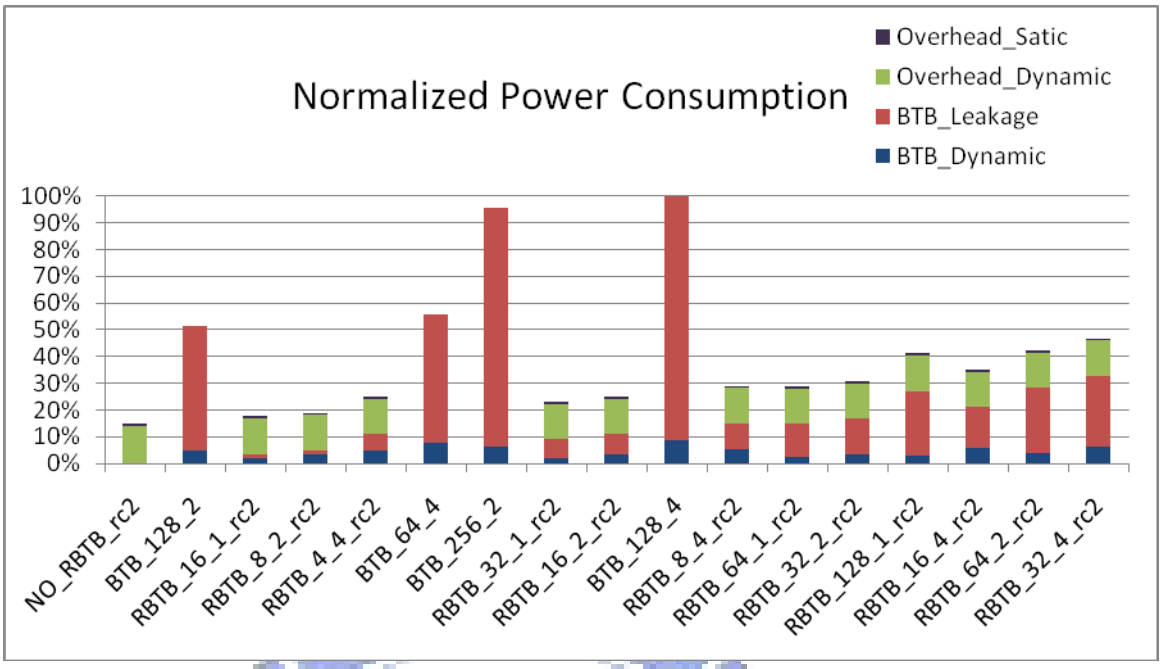


Figure 4-12: Normalized power sorted in increasing order of IPC in N=2 case.

As can be foreseen, with more number of entries required to keep the performance, the overall power saving faces greater challenge. In the N=2 case, to keep up with no performance loss, the power reduction can only reach 70% compared to 512-etnry conventional BTBs and 55% compared to 256-entry conventional BTBs.

4.2.3 Case N = 3

Since in common pipeline system, instruction cache access would not exceed three cycles, here we present the last part of our experiment results, the N=3 case.

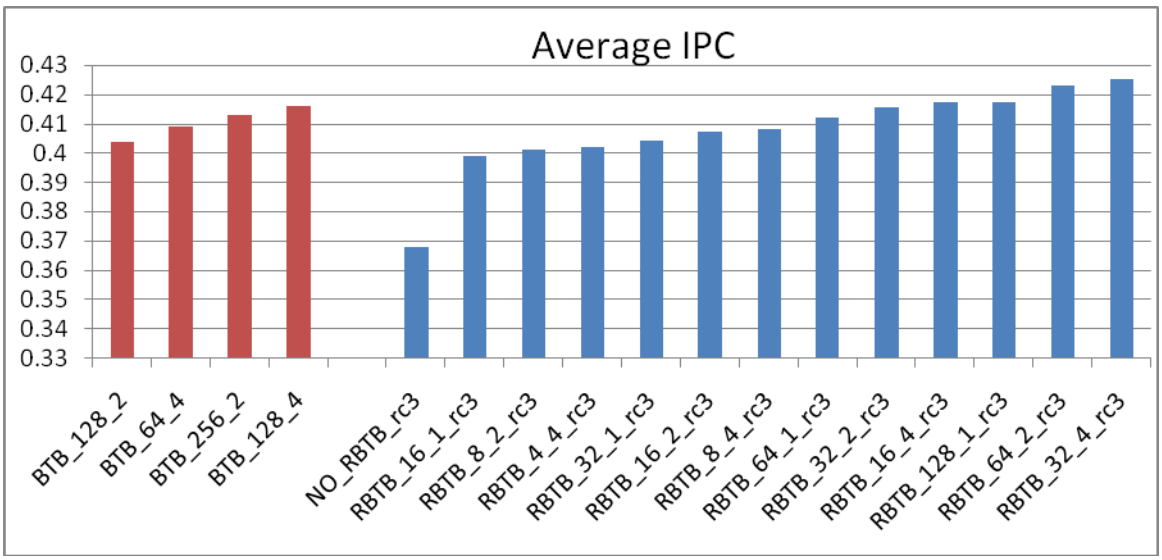


Figure 4-13: Average IPC in N=3 case.

The postfix "rc3" represents the value of N, in this case, 3.

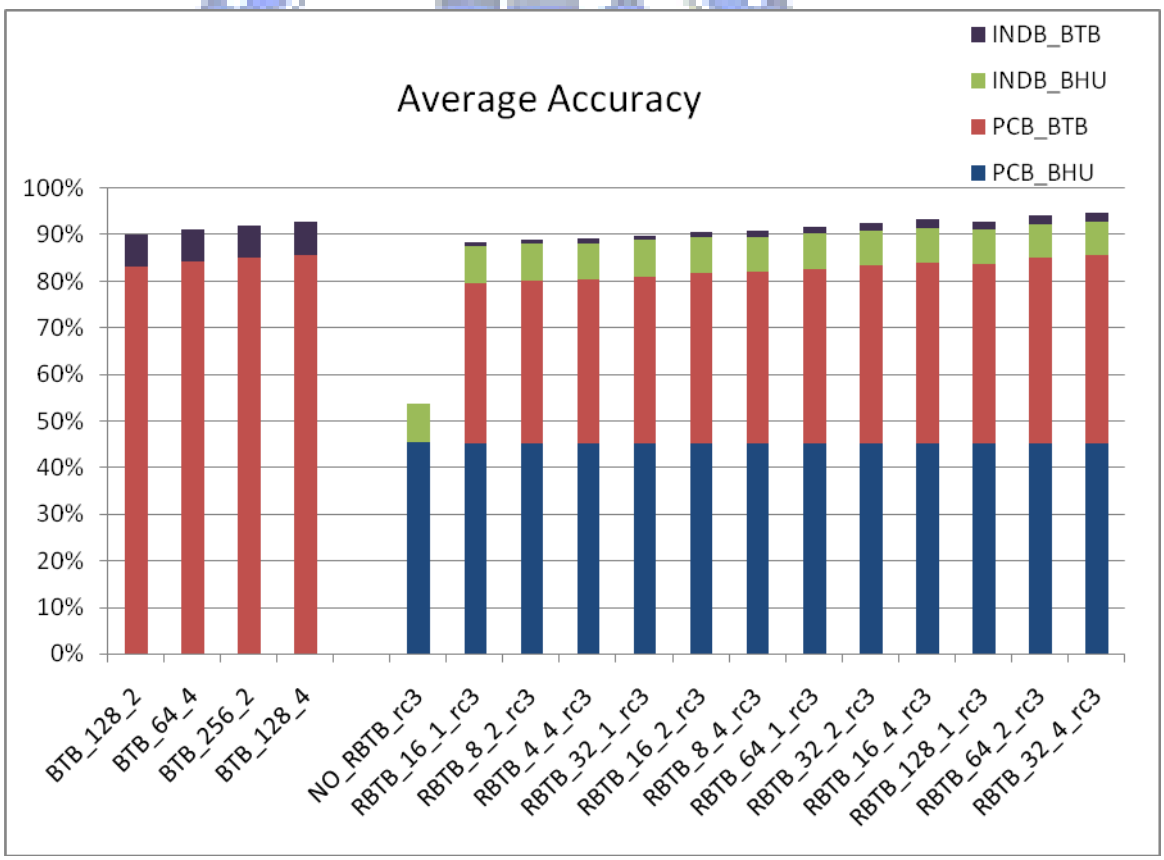


Figure 4-14: Average accuracy in N=3 case.

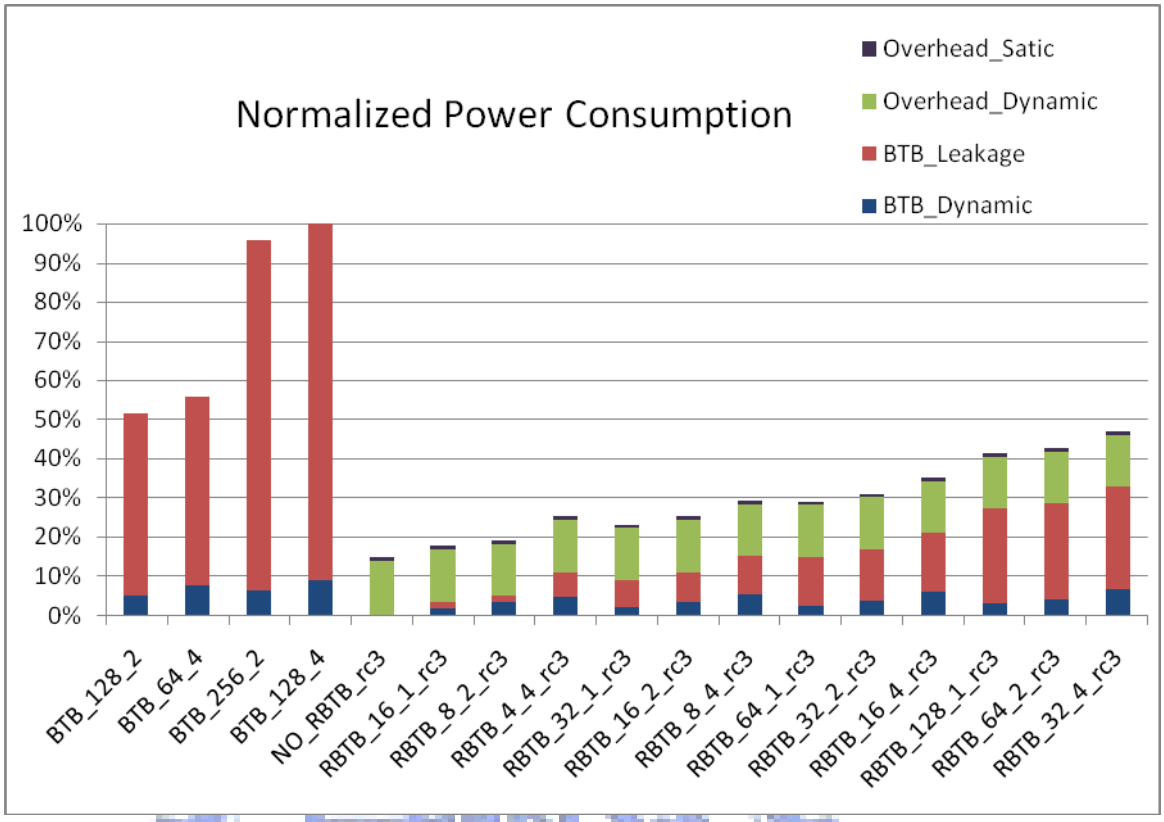


Figure 4-15: Normalized power in N=3 case.

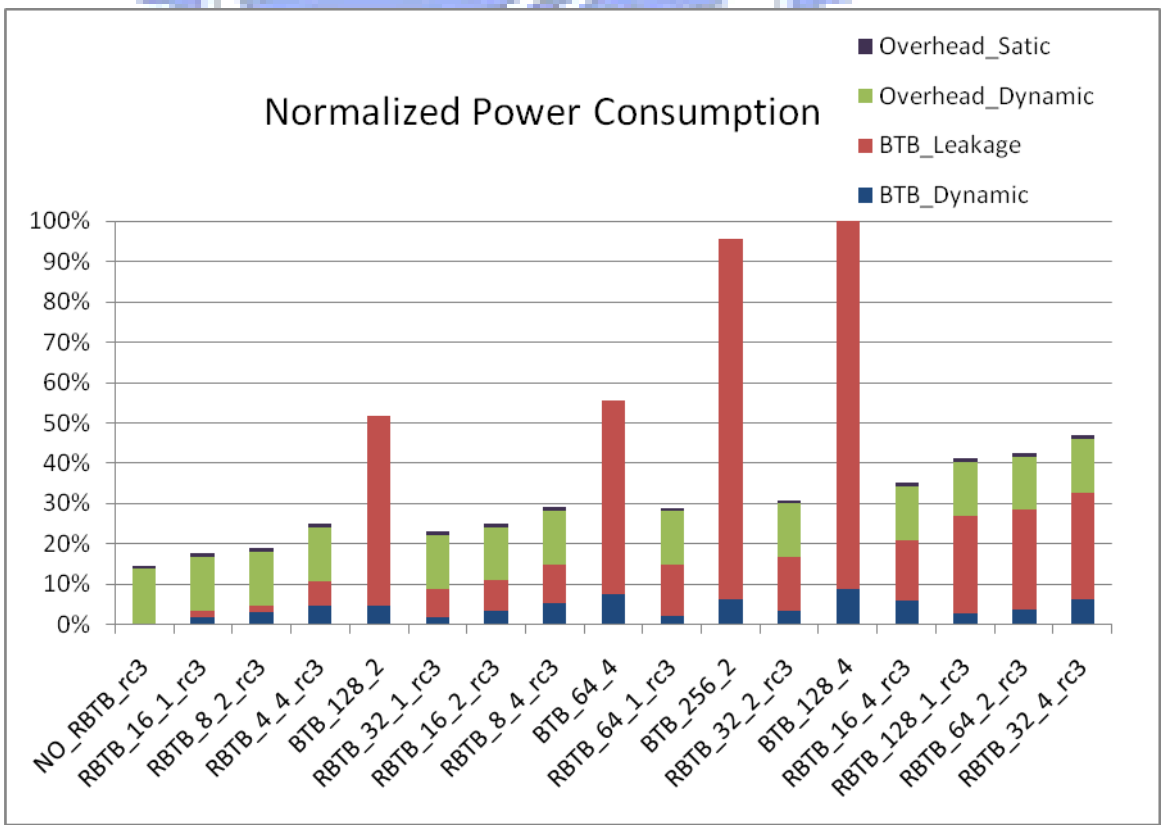
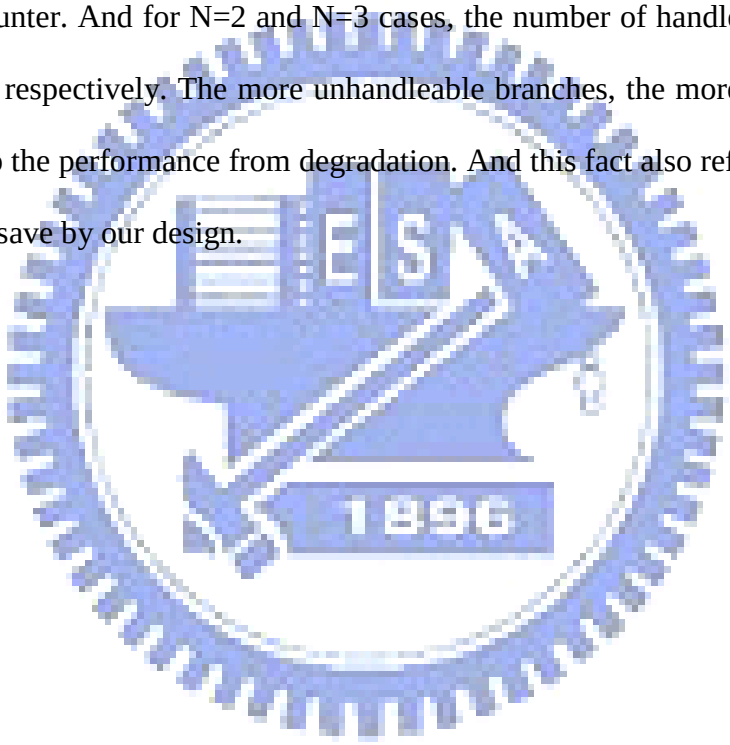


Figure 4-16: Normalized power sorted in increasing order of IPC in N=3 case.

4.3 Summary for Simulation Results

In this chapter we presented experiment results for N=1, N=2, and N=3 cases. The N value can be concluded as the sum of BHU exercise cycles and instruction buffer refilling cycles. According to many circuit implementation researches, we assume the BHU incurs very short latency and generally would not exceed one cycle time. So the N value reflects only the number of instruction loss due to the inevitably instruction buffer refill.

By the statistics, we've found that in the idea N=1 case, BHU can handle 77% of total branches encounter. And for N=2 and N=3 cases, the number of handleable branches drop to 62% and 53% respectively. The more unhandleable branches, the more entries RBTB would require to keep the performance from degradation. And this fact also reflects on the amount of power we can save by our design.



Chapter 5 Conclusions and Future Works

In the first part of this chapter conclusion are made. And in the second part possible future works are proposed.

5.1 Conclusions

In this thesis, we presented a Branch Handling Unit (BHU) that dynamically identify and generate target address for both PC-relative and indirect branches. The BHU is suppose to ease information load in the Branch Target Buffer. Target addresses that are able to be generated by BHU need not to register entries in BTB, thus the storage requirement can be lower. Due to some contrains, we still find the BTB cannot be completely eliminated from the system. In the end, the BTB can only be downsized with the help of BHU. In a nutshell, this research provides a way to have a trade-off. By exercising BHU, which is composed of a number of logic units and small size buffers, dynamic power is used to trade for leakage power. The overall outcome is deemed worthy in modern manufacture process, since the leakage power consumption today overwhelms the dynamic exercising power consumption.

Aside from power consumption reduction, the method in this thesis significantly lowered the area requirement of branch predictors. In the aspect of manufacturing, this can be a great advantage considering price and yield. And by reducing the information load, the number of updates in BTB is also decreased, leading to less evictions and replacements. Branches in the BTB stay longer, increasing the size of an abstract viewing window of braches for whom predictions should be made. This gives a good opportunity to improve the overall system performance.

The BHU is designed to be a low-latency, light-weighted unit in the system. For such a simple unit to reduce so many entries in the BTB, it’s not at all absurd to point out that

conventional BTB design wasted an inefficient amount of storage to keep track of the branches behavior. Our experimental results show that up to 85% of the BTB storage can be unnecessary and replaced by the BHU. This fact pushes us to review the nature of history-base predictors. With the chase of higher computational power and process stepping, is every piece of the hardware put to absolutely good places and being well-utilized to every gate? Or we're just abusing, brutally putting in more and more logic gates into a chip and gain less and less efficiency than they should really deliver?

5.2 Future Works

The future work of our BHU design can be put to three aspects. Each of these ideas targets to the same goal: further power reduction for BTB. The three aspects are: another downsizing for the RBTB, unnecessary BHU + RBTB access filtering, and compiler co-design.

First, the chances of further size shrinking of the RBTB lies in the width shortening of each entry. Since theoretically the information that can be spear from RBTB has been minimized by the means of BHU, the number of bit storage of each entry is the only part we can now attack. The two related works introduced in chapter 2 offer perfect solutions. The RBTB in our system operates exactly as a conventional BTB, thus the two independent methods would have no difficulty to co-exist with our design. By tag and address field shortening, the RBTB may become a buffer so lightweight and fully utilized, and so that every piece of it functions most of its cost.

The second part is the dynamic power reduction. Now that the BHU and RBTB are exercise every cycle, we know it's actually an overdriven state considering not every instruction is a branch. The unnecessary access filtering can be done separately on RBTB and BHU or as one task. There were a lot of researches proposed for BTB access count reduction

and as the two related works mentioned above, they can also be applied to our design. The method of access filtering can also come from the design itself. Assuming we have a loose timing constrain system, the partial decoding can then be serialized with following target generation or lookup process. Another alternative is to facilitate pre-decode, so that branches are identified even before they enter the pipeline.

Finally, a compiler co-design can improve the efficiency of our current BHU + RBTB design. Needless to say, data dependency has always been an issue that compilers are fighting against. By rearranging the instruction placement in the instruction cache, so that branches stay away from starting position in cache line, the number of unhandleable branches can be decreased. Avoid consecutive branches is another way to kill unhandleable branches. And setting constrains so that every indirect branch targets are kept in instruction-set-defined register can also improve the prediction accuracy in our design.

References

[1] Ram K. Krishnamurthy, Atila Alvandpour, Sanu Mathew, Mark Anders, Vivek De, Shekhar Borkar, “High-performance, Low-power, and Leakage-tolerance Challenges for Sub-70nm Microprocessor Circuits”, ESSCIRC 2002.

[2] Dharmesh Parikhy, Kevin Skadrony, Yan Zhangz, Marco Barcellaz, Mircea R. Stanz, “Power Issues Related to Branch Prediction”, HPCA 2002.

[3] David Brooks, Vivek Tiwari, and Margaret Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations”, ISCA 2000.

[4] Amirali Baniasadi, and Andreas Moshovos, “SEPAS: A Highly Accurate Energy-Efficient Branch Predictor”, ISLPED 2004.

[5] Barry Fagin, “Partial Resolution in Branch Target Buffers”, IEEE Transactions on Computer 1997.

[6] Jan Hoogerbrugge, “Cost-Efficient Branch Target Buffers”, Euro-Par 2000 Parallel Processing.

[7] Randal E. Bryant, “Alpha Assembly Language Guide”.

[8] Fatemeh Kashfi, and Nasser Masoumi, “Optimization of Speed and Power in a 16-Bit Carry Skip Adder in 70nm Technology”, 2006 IEEE.

[9] Fatemeh Kashfi, and S. Mehdi Fakhraie, “Implementation of a High-Speed Low-Power 32-BitAdder in 7Onm Technology”, ISCAS 2006.

[10] Debabrata Mohapatra, Georgios Karakonstantis and Kaushik Roy, “Low-Power Process-Variation Tolerant Arithmetic Units Using Input-Based Elastic Clocking”, ISLPED 2007.

[11] SimpleScaler, http://www.simplescalar.com

[12] SPEC, http://www.spec.org

[13] CACTI, http://www.hpl.hp.com/research/cacti

[14] CACTI V5.3, http://quid.hpl.hp.com:9081/cacti/index.y

[15] Yen-Jen Chang, “An Energy-Efficient BTB Lookup Scheme for Embedded Processors”, IEEE Transactions on Circuits and Systems 2006.

[16] David Brooks, Vivek Tiwari, and Margaret Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations”, ISCA 2000.