

國立交通大學

資訊科學與工程研究所

碩士論文

使用分支目標暫存器協助的可存放分支及副程
式的迴圈緩衝器以減少指令擷取能源消耗



Power Reduction in Instruction Fetch Using Forward-Branch
and Subroutine Bufferable Innermost Loop Buffer with
Assistance of BTB

研究生：田濱華

指導教授：鍾崇斌 博士

中華民國九十五年七月

使用分支目標暫存器協助的可存放分支及副程式的迴圈緩衝器以減少指令擷取能源消耗

Power Reduction in Instruction Fetch Using Forward-Branch and Subroutine Bufferable Innermost Loop Buffer with Assistance of BTB

研究生：田濱華

Student：Bin-Hua Tein

指導教授：鍾崇斌

Advisor：Chung-Ping Chung

國立交通大學



A Thesis

Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Computer Science

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

使用分支目標暫存器協助的可存放分支及副程式的 迴圈緩衝器以減少指令擷取能源消耗

學生：田濱華

指導教授：鍾崇斌 教授

國立交通大學資訊工程系

摘要

減少嵌入式處理器的能源消耗以增加使用時間變得日益重要。在嵌入式處理器中，指令擷取的能源消耗佔整個動態能源消耗很大一部分。近來提出一減少指令擷取能源消耗之設計，在指令快取及處理器間加入一更小的記憶體，利用指令的時間區域性，使得大部分的指令能從此小的記憶體中擷取。迴圈有很好的空間區域性，因此許多的迴圈緩衝器設計便被提出。然而，在設計複雜度的限制下，大多數的迴圈暫存器只存放迴圈內無向前分支指令及沒有副程式的最內層迴圈，或是存放迴圈內的指令直到向前分支指令及副程式的最內層迴圈。但許多的迴圈內包含向前分支及副程式，因此現有的設計仍有減少指令擷取能源的改善空間。我們提出一個簡單且有效率的方法，來存放迴圈內含向前分支及副程式的最內層迴圈：因為分支目的緩衝器在現今的嵌入式處理器設計中將普遍存在，如果我們加入一額外欄位在每項分支目的緩衝器內容後，用來指示這個在迴圈緩衝器存放的分支後的下一到指令是分支後接續的下一道或是分支目的，如此可避免掉設計複雜度而存放內含向前分支的最內層迴圈。含副程式的最內層迴圈且這個副程式內不包含迴圈也可用相似的方法處理。使用 MiBench 模擬的結果得到，我們的設計可比先前無法存放向前分支及副程式的迴圈緩衝器更減少 13.66% 的指令擷取能源消耗。

Power Reduction in Instruction Fetch Using Forward-Branch and Subroutine Bufferable Innermost Loop Buffer with Assistance of BTB

Student : Bin-Hua Tein

Advisor : Chung-Ping Chung

Dept. of Computer Science

National Chiao Tung University

ABSTRACT

Reducing power of embedded processors is becoming increasingly important for mobile applications. Much of the dynamic power of a typical embedded processor is consumed by instruction fetching. Recently, addition one tiny memory between CPU core and instruction cache had been proposed. Using the temporal locality of instructions, most of instructions can be fetched from this tiny memory to replace from instruction cache. Loops have temporal locality, so that many of loop buffer design had been proposed. Nevertheless, on design complexity dictates most loop buffer designs to store only innermost loops without forward branch or instructions within innermost loops before a forward branch. While program modeling shows that typical programs can best be represented with a simple loop model, many of them contain forward branches and subroutines in their innermost loops. Hence, existing designs lead to limitation in reduction of instruction fetch power. We propose a simple and effective way to cope with this complexity: since using BTB is a norm in most designs, if we add an extra bit in BTB, indicating if the loop buffer stores the fall-through or target trace after a within-the-innermost-loop forward branch, then much of the complexity can be avoided. The subroutine including no loop is also handled by using similar way. Results with MiBench indicate that up to 18% of further reduction in instruction fetch power compared with the design without forward branch and subroutine handling.

誌謝

首先感謝我的指導老師 鍾崇斌教授，在他的諄諄教誨、辛勤指導與勉勵下，而得以順利完成此論文。同時也感謝是我的口試委員的單智君、林正宗、楊竹星教授，由於他們的建議，使得這篇論文能更加完整。

感謝吳奕緯學長全程的協助，給我很多的幫助。也感謝 jojo 大學長、程式超強的林大頭、人超好又瘦的幾米學妹、強汪、陳 17、很帥的葉大帥跟黃士嘉、屁屁鄭、黃富群、莊富源的幫助，給我很多意見及鼓勵。

最後感謝我的家人和女朋友，謝謝你們在背後全心全意的支持我，關懷我、鼓勵我。讓我在這求學的路上走的更順利，使我能堅持追求自己的理想。

所有支持我、勉勵我的師長與親友，奉上我最誠摯的感謝與祝福，謝謝你們。



田濱華 2006.8

Contents

摘要.....	i
ABSTRACT.....	ii
誌謝.....	iii
Contents	iv
List of Figures	vi
List of Table	viii
Chapter 1 Introduction	1
1.1 Instruction Fetch Power Reduction Techniques.....	1
1.2 The Problems of Loop Buffer to Store Branch(es)	4
1.3 Motivations and Objectives	5
1.4 Organization of This Thesis	7
Chapter 2 Background and Related Work.....	8
2.1 Categorization of Execution Time of Program	8
2.2 Related Research and Technology	9
2.2.1 Dynamic Loop Cache	9
2.2.2 Hardware-Based Two-Way Loop Buffer	11
2.2.3 Pre-load Loop Cache.....	12
2.2.4 Cluster Loop Cache.....	13
Chapter 3 Design of the Forward-Branch and Subroutine Bufferable Innermost Loop Buffer.....	14
3.1 Features of Our Approach.....	14
3.2 Architecture and Design Issues of Our Approach.....	15
3.2.1 Innermost Loop Detection	16
3.2.2 Filling or Refilling an Innermost Loop Into Loop Buffer.....	17
3.2.3 Determining that CPU Core Should Fetch Intrusions From Loop Buffer or IL1	19
3.2.4 Handling Incorrect Instruction Filling and Fetching due to Branch Misprediction	20
3.2.5 No-loop-inside Subroutine Handling	21
3.3 Operation.....	22
Chapter 4 Simulation and Evaluation	27
4.1. Power Model.....	27
4.2 Simulation Environment	28
4.2.1 Simulator.....	28
4.2.2 Benchmark Programs.....	29
4.3. Simulation Results of Loop Buffer Policies	32

4.3.1 Loop Buffer Policies in Each loop Buffer state	32
4.3.2 Simulation Results of Loop Buffer Policies	35
4.3.2.1 Simulation Results of the Loop Buffer Policies in IDLE State	35
4.3.2.2 Simulation Results of the Loop Buffer Policies in FILL State	38
4.3.2.3 Simulation Results of the Loop Buffer Policies in ACTIVE State.....	42
4.3.2.4 Simulation Results of the Best Loop Buffer Policies	45
4.4 The Penalty of Miss-prediction in Loop Buffer Policies	49
4.5 The Reasons of Affecting the Loop Buffer Efficiency	51
4.5.1 The Nature of Benchmark.....	51
4.5.1.1 The Benchmark Has Less Execution Time in Innermost Loops	52
4.5.1.2 The Execution Paths are changed frequently.....	54
4.5.1.3 The Benchmark Has a Few of Loop Iteration.....	56
4.5.1.4 The Size of Innermost Loops is too large or too small than Loop Buffer.....	58
4.5.2 The Architecture of Processor.....	63
4.5.3 The Optimal of Compiler.....	64
4.6. Simulation Results and Evaluation.....	65
4.6.1 Strategies of Loop Buffer management as Comparison	65
4.6.2 Simulation Results of Different Loop buffer Designs	66
Chapter 5 Conclusions and Future Work.....	73
Reference	75
Appendix.....	78
A-1 Profiling of Innermost Loop.....	78

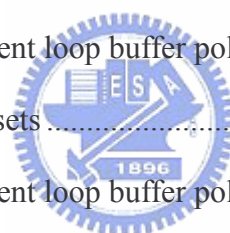
List of Figures

Figure 1.1: (a) Organization of CPU core, IL1, and filter cache;(b) Organization of CPU core, IL1, and loop buffer.....	4
Figure 1.2: A code segment without branch.....	4
Figure 1.3: A code segment with a branch.....	5
Figure 2.1: Profile the execution time of MiBench	9
Figure 2.2: the management of Dynamic Loop Cache	10
Figure 2.3: the management of Hardware-Based Two-Way Loop Buffer	12
Figure 2.4: the management of Pre-load Loop Cache	13
Figure 3.1: Architecture of our approach.....	15
Figure 3.2: An innermost loop consists of subroutine	22
Figure 3.3: Memory accessing in different states	23
Figure 3.4: State diagram of loop buffer controller	26
Figure 4.1: Access ratio of IL1 of different loop buffer policies sets (IDLE state)..	36
Figure 4.2: Access ratio of loop buffer of different loop buffer policies sets (IDLE state).....	37
Figure 4.3: Reduction in instruction fetch power of different loop buffer policies sets (IDLE state)	38
Figure 4.4: Access ratio of IL1 of different loop buffer policies sets (FILL state)...	40
Figure 4.5: Access ratio of loop buffer of different loop buffer policies sets (FILL state).....	40
Figure 4.6: Reduction in instruction fetch power of different loop buffer policies sets (FILL state)	41
Figure 4.7: Access ratio of IL1 of different loop buffer policies sets (ACTIVE state)	43

Figure 4.8: Access ratio of loop buffer of different loop buffer policies sets (ACTIVE state).....	44
Figure 4.9: Reduction in instruction fetch power of different loop buffer policies sets (ACTIVE state).....	45
Figure 4.10: Access ratio of IL1 of different loop buffer policies sets	47
Figure 4.11: Access ratio of loop buffer of different loop buffer policies sets	48
Figure 4.12: Reduction in instruction fetch power of different loop buffer policies sets.....	49
Figure 4.13: The penalty of miss-prediction of innermost loop detection.....	51
Figure 4.14: The ratio of different sizes (instructions) of innermost loop— tiff2bw	59
Figure 4.15: The ratio of different sizes (instructions) of innermost loop—susan-c	60
Figure 4.16: Access ratio of IL1 in different benchmarks	61
Figure 4.17: Access ratio of loop buffer in different benchmarks	61
Figure 4.18: Reduction in instruction fetch power in different benchmarks	62
Figure 4.19: The percentage of innermost loops with different sizes of innermost loops.....	63
Figure 4.20: Access ratio of IL1 of different designs	67
Figure 4.21: Access ratio of loop buffer of different designs	68
Figure 4.22: Loop buffer ACTIVE ratio	70
Figure 4.23: Reduction in instruction fetch power of different designs	72

List of Table

Table 4.1: Ratio of P_{LB}	28
Table 4.2: Parameters setting in SimpleScalar/ARM.....	29
Table 4.3: MiBench workloads	30
Table 4.4: Policies of loop buffer in each state of loop buffer	34
Table 4.6: Ratio of P_{ctrl} of different loop buffer policy sets (IDLE state).....	35
Table 4.6: Power of loop buffer controller of different loop buffer policy sets (IDLE state).....	37
Table 4.7: Loop buffer policies sets (FILL state).....	38
Table 4.8: Ratio of P_{ctrl} of different loop buffer policy sets (FILL state).....	41
Table 4.9: loop buffer policies sets (ACTIVE state).....	42
Table 4.10: Ratio of P_{ctrl} of different loop buffer policy sets (ACTIVE state)	44
Table 4.11: Loop buffer policies sets	48
Table 4.12: Ratio of P_{ctrl} of different loop buffer policy sets	48
Table 4.13: Loop buffer policies sets	50
Table 4.14: The relation between innermost loop ratio to ACTIVE ratio.....	53
Table 4.15: The relation between execution path ratio to the increase ratio.....	54
Table 4.16: the relation between execution path ratio to the increase ratio	57
Table 4.17: The best policies set of FSLB	66
Table 4.18: Ratio of P_{LB} and P_{ctrl}	70



Chapter 1

Introduction

Power consumption has become an increasingly greater concern in digital system designs, especially for battery powered devices. Much of the dynamic power of a typical embedded processor is consumed by instruction fetching, for example, 30-50%. Since instruction fetching happens on almost every cycle, involves switching of large number of high capacitance wires, and may involve access to a power hungry set-associative cache.

Loop buffering is an effective technique to reduce energy consumption in the instruction memory hierarchy. In any typical embedded application, significant amount of execution time is spent in small program segments. Hence, by storing them in a small loop buffer or an L0 buffer instead of the big instruction cache, energy can be reduced.



In this thesis, we propose a loop buffering mechanism which can store innermost loop with forward branch(es) or subroutine call(s) inside this innermost loop in loop buffer, so that the instruction fetch power can be reduced.

In this chapter, we will first instruction fetch power reduction techniques in section 1.1. In section 1.2, we describe the problems of loop buffer to store forward branch(es) and subroutine(s). The motivations and objectives are proposed in section 1.3. The organization of this thesis is described in section 1.4.

1.1 Instruction Fetch Power Reduction Techniques

Current embedded systems for multimedia applications like mobile and hand-held devices are typically battery operated. Therefore, low energy is one of the

key design goals of such systems. Power analysis of such processors indicate that a significant amount of power is consumed in the instruction caches during instruction fetch [1], [2] . For example in the TMS320C6000, a VLIW processor from Texas Instruments, up to 30% of the total processor energy is consumed in the instruction caches alone [1]. Since instruction fetching happens on almost every cycle, involves switching of large number of high capacitance wires, and may involve access to a power hungry set-associative cache.

Thus, several approaches have been proposed to reduce instruction fetch power. Some have focused on encoding the address and data bus signals to reduce bus switching [3-5]. Others have focused on compressing [6-7] or buffering instructions [8], also to reduce bus switching. Some researches have looked at reducing the power of the cache itself by deactivating several ways of a set-associative cache when deactivation does not heavily impact performance [9-10], or accessing items using multiple phases [11], thus trading off performance for reduced power.

Another class of approaches adds an unusually small instruction cache called instruction level zero cache as the first level of memory (level 0) in the instruction memory hierarchy, perhaps 16 to 128 word (typically 64 to 512 byte), between CPU core and instruction level one cache. The extremely low power per access for an instruction level zero cache is achieved because of very short wires inside the cache. Another reason for very low power per access to an instruction level zero cache comes from the fact that an instruction level zero cache can be integrated very close to or even inside a microprocessor, resulting in shorter and hence lower power bus lines. Two approaches of instruction level zero cache in recent years, one is called filter cache which is a cache-like architecture; the other is called loop buffer which is a simple instructions buffer without tag.

[12] proposes an instruction level zero cache design called a filter cache. A

filter cache is a tiny direct-mapped cache introduced as the first level of memory (level 0) in the instruction memory hierarchy as shown in figure 1.1 (a).

[12] shows that a 256 byte filter cache has a hit rate between 60–85% on MediaBench benchmarks. Using a 32 KB direct-mapped cache for the L1 cache, the filter cache could reduce instruction access power by over 50%, but at the expense of about 20% performance overhead. The energy*delay product related to memory accesses was reduced by about 50%. To reduce the performance overhead, [13] proposed using a profile-aware compiler to map frequent loops to a special address region recognized by the processor as loadable into the filter cache, resulting in less performance overhead along with improved energy savings.

In many researches [14-17], loop buffer has been proposed to reduce instruction fetch power. A loop buffer is a memory located between CPU core and L1 instruction cache, called IL1 hereafter, as shown in figure 1.1 (b). CPU core fetches instructions from either IL1 or loop buffer. Due to its limited, a loop buffer can provide instructions to CPU core at a very low power level. And the best pieces of code to be placed in a loop buffer will be innermost loops, since their executions tend to repeat many times. As an evidence, MiBench spends 71.22% of execution time on innermost loops.

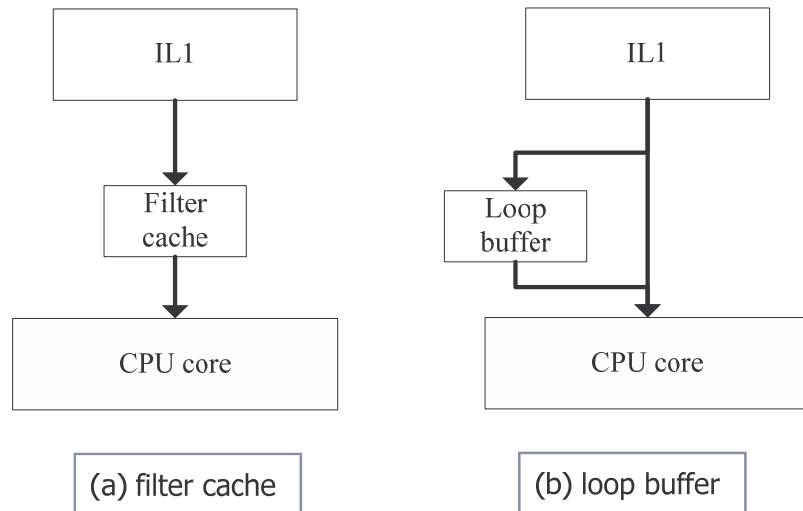


Figure 1.1: (a) Organization of CPU core, IL1, and filter cache;

(b) Organization of CPU core, IL1, and loop buffer

1.2 The Problems of Loop Buffer to Store Branch(es)

The loop buffer is a simple instruction buffer without tag, so that the management of loop buffer is different from cache. In general, loop buffer need an address generator to determinate whether the fetched instruction is in loop buffer and calculate where the fetched instruction address is in loop buffer.

Since the instructions fetch are usual in sequent order, we can place these sequent instructions into loop buffer in the same order, so that address generator of loop buffer can be implement by a counter to avoid area and delay overhead. An example for a sequent fetch code segment (i.e. a code segment without branch) is in figure 1.2.

On the other hand, the addresses of instruction fetch are not sequent between a branch and the next instruction after this branch, so that the address generator of loop buffer using a counter can not determinate whether this instruction after a branch has been stored in loop buffer at run time. An example for a code segment

with a branch is in figure 1.3.

Since the address generator of loop buffer can not handle the code segment with a branch, the utilization of loop buffer is limited.

Instruction	Address of next instructions	Address in loop buffer
Non-branch	PC+4	next (+1)
Non-branch	PC+4	next (+1)
Non-branch	PC+4	next (+1)
Non-branch	PC+4	next (+1)

Figure 1.2: A code segment without branch

Instruction	Address of next instructions	Address in loop buffer
Non-branch	PC+4	next (+1)
Branch	PC+4	next (+1)
Unknown	PC+4 / target	Unknown
Unknown	Unknown	Unknown

Figure 1.3: A code segment with a branch

1.3 Motivations and Objectives

To maximize the power advantage, a loop buffer should store innermost loops with forward branch(es) and subroutine(s). For example, MiBench spends 38.32% of execution time on innermost loops with forward branch(es) and subroutine(s) without backward branch. It is over half of total execution time on innermost loop.

However, to avoid design complexity, most loop buffer designs are capable of storing only innermost loops without forward branch [14, 15] or instructions within

innermost loops before a forward branch [15]. Since many applications consist of forward branch(es) in their innermost loops, utilization of loop buffer and reduction in instruction fetch power in [14, 15] is limited.

To increase utilization of loop buffer, [16, 17] propose a loop buffer consisting of an additional address generator to store any kinds of code segment. Before fetching instruction from loop buffer, address generator must generate an address and use this address to determine whether this instruction has been stored in loop buffer and if it is, where it is located.

Consequently, this address generator leads to a significant increase in power and fetch latency. In addition, most designs [14, 16, and 17] require compiler help to insert special instruction(s) in program to start filling instructions into loop buffer [14, 17] or to determine which code segments should be stored in loop buffer [16]. Recompilation and code compatibility issues hence arise.

To increase utilization of loop buffer without introducing much overhead, we use BTB to assist loop buffer in storing the innermost loops with following characteristics:

- (1) they can contain forward branch(es) and
- (2) they can call any number of subroutines as long as these subroutines have no loop inside.

In our design, an extra bit is added in branch target buffer (BTB) to record forward branch outcome. This bit indicates whether the loop buffer stores the fall-through or target trace after a forward branch. The subroutine including no loop is also handled by using similar way. Notice also that different from previous designs [14, 16, and 17], our approach does not need special branch instruction or compiler to assist loop buffer controller in innermost loop detection.

Results with MiBench indicate that our design can further reduce 18.00% and

14.61% instruction fetch power compared with only capable of innermost loop without forward branch and [15], respectively.

1.4 Organization of This Thesis

The remaining parts of this thesis are organized as follows. In Chapter 2, we examine some related the previous work. In Chapter 3, we present our proposed loop buffer design. In Chapter 4, we show the simulation results and discussion. Finally, in Chapter 5, we give the conclusions of this work.



Chapter 2

Background and Related Work

In Section 2.1 we divide the execution time of program into four categories that have different nature of execution flows. In Section 2.2, we survey the related researched in improving the loop buffer.

2.1 Categorization of Execution Time of Program

We divide the execution time of program into five categories which have different nature of execution flow:

Type A is innermost loop without forward branch and without subroutine. The addresses of instruction in *Type A* are sequent from the first instruction of innermost loop to the last instruction of innermost loop and the execution flow in this type is exclusive.

Type B is innermost loop with forward branch(es) but without subroutine. The addresses of instruction in *Type B* may not be sequent because of forward branch taken so that the execution flow in this type could not be exclusive, and the addresses of instruction in execution flow are within from the first instruction of innermost loop to the last instruction of innermost loop.

Type C is innermost loop with subroutine(s) consisting of no loop and it may contain forward branch(es) or not. If we inline the subroutine(s) in *Type C*, *Type C* will convert to *Type A* or *Type B*. Since an innermost loop with subroutine(s) consisting of loop(s) can not be called an innermost loop, we classify it into *Type D*. others. Unlike *Type B*, the addresses of instruction in subroutine(s) will beyond from the first instruction of innermost loop to the last instruction of innermost loop.

Type D is the outer loop of a nested loop.

Type E is others which we consider that the execution flow is not in any loop.

As an evidence, MiBench spends 71% of execution time on innermost loops which include *Type A*, *Type B* and *Type C*, and 79.85% of execution time on loops. MiBench is a benchmark suite for embedded processor. The profiling results of each category are shown in brackets at the right of figure 2.1. More detail profiling results of each benchmark are shown in Appendix A-1.

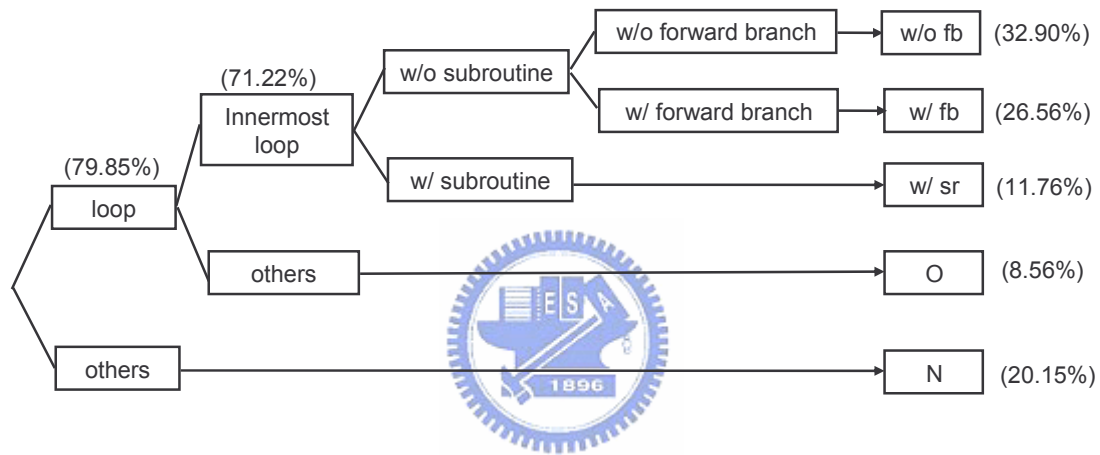


Figure 2.1: Profile the execution time of MiBench

2.2 Related Research and Technology

In this section, we study the research and technology related to our thesis including dynamic loop cache, hardware-based two-way loop buffer, pre-load loop cache, and cluster loop cache.

2.2.1 Dynamic Loop Cache

[14] is capable of storing only innermost loops without forward branch. To indicate where a backward branch exists, [14] uses a special branch instruction

“sbb”. If a “sbb” is detected and taken, loop buffer controller starts to fill instructions into loop buffer. Only if a “sbb” is detected and taken twice successively, CPU core starts to fetch instructions from loop buffer until this “sbb” is detected but not taken, meaning the loop is being exited. To reduce design complexity, [14] uses a counter to generate loop buffer addresses, called loop buffer program counter (LPC). The management of Dynamic Loop Cache is shown in figure 2.2.

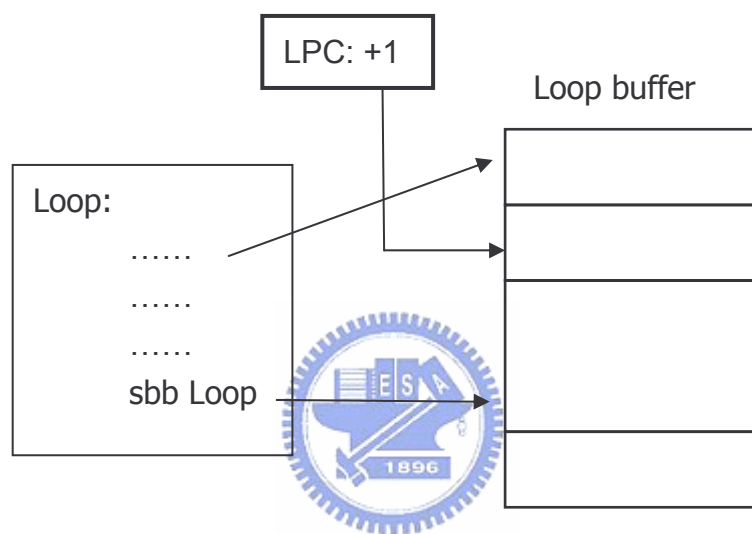


Figure 2.2: the management of Dynamic Loop Cache

If the entire loop does not fit in the loop cache, then the cache will be filled completely with the first part of the loop, so that CPU core fetch instructions will switch between loop buffer and IL1. [14] also explored a “warm-fill” version of the dynamic loop buffer that continually filled the loop buffer on every instruction fetch so that at any given time, the last N instructions were available in the loop buffer, where N is the loop buffer size. However, [14] showed that this design yielded little benefit—the power savings of being able to switch to loop buffer fetching immediately after detecting a “sbb” do not outweigh the power overhead for keeping the loop buffer filled.

Because filling of [14] is nonintrusive (i.e., no microprocessor stall occurs), a control of flow change (cof) which will cause the next instruction address is not the current address plus one, but the loop buffer may not get filled with the entire loop. Thus, in [14], a cof (other than the triggering sbb) encountered within a small loop would immediately terminate the loop cache filling or fetching. In other words, only loops without any cof were supported so that utilization of loop buffer and the reduction in instruction power are limited.

2.2.2 Hardware-Based Two-Way Loop Buffer

Instead of using a special branch instruction “sbb” in [14], [15] deploys a special register to record the address of backward branch. Once a backward branch is detected and taken twice successively, loop buffer controller starts to fill instructions into loop buffer. After successively filling, CPU core begins to fetch instructions from loop buffer. We also use this method to detect an innermost loop in this paper. Unlike [14], [15] can also store instructions within an innermost loop from first instruction of loop until a forward branch or a subroutine call. This is because instruction addresses from first instruction of loop until a forward branch or subroutine call are sequentially (i.e., there is no cof from first instruction of loop until a forward branch or subroutine call). To reduce design complexity, [15] also uses a counter to generate LPC so that [15] encounters the same limitation with [14]. The management of Hardware-Based Two-Way Loop Buffer is shown in figure 2.3.

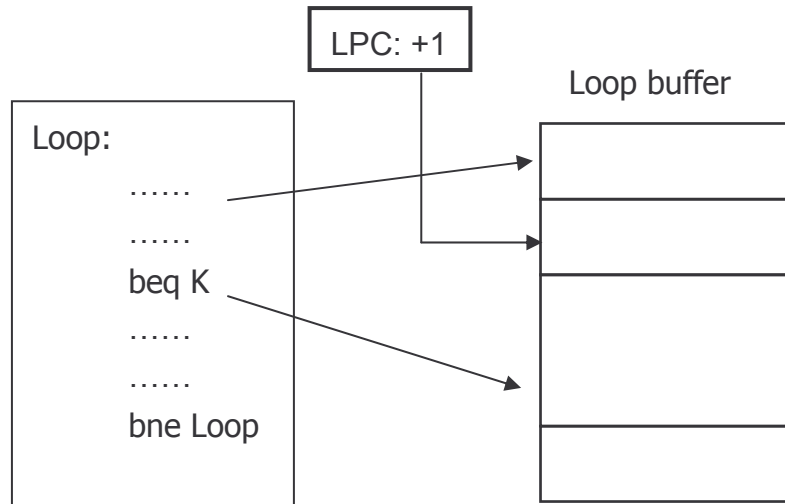


Figure 2.3: the management of Hardware-Based Two-Way Loop Buffer

2.2.3 Pre-load Loop Cache

[16] is capable of storing many kinds of code segment in which instruction addresses must be sequential. In [16], which code segment can be stored in loop buffer are analyzed statically. After the CPU booting, several code segments and their start address (`start_addr`) and end address (`end_addr`) are filled into loop buffer and special registers, called loop address registers (LARs), respectively. CPU core then continuously compares each instruction address with LARs to determine whether start and terminate to fetch instructions from loop buffer. This leads to inflexible usage of loop buffer. Since code segments stored in loop buffer may consist of forward branch, [16] uses an address generator to cope with non-sequential instruction fetch. Before fetching one instruction from loop buffer, loop buffer controller must wait for LPC calculated by address generator and use LPC to determine whether this instruction has been stored in loop buffer and where this instruction is. Consequently, this address generator leads to a great increase on instruction fetch latency and hardware cost. The management of Pre-load Loop Cache is shown in figure 2.4.

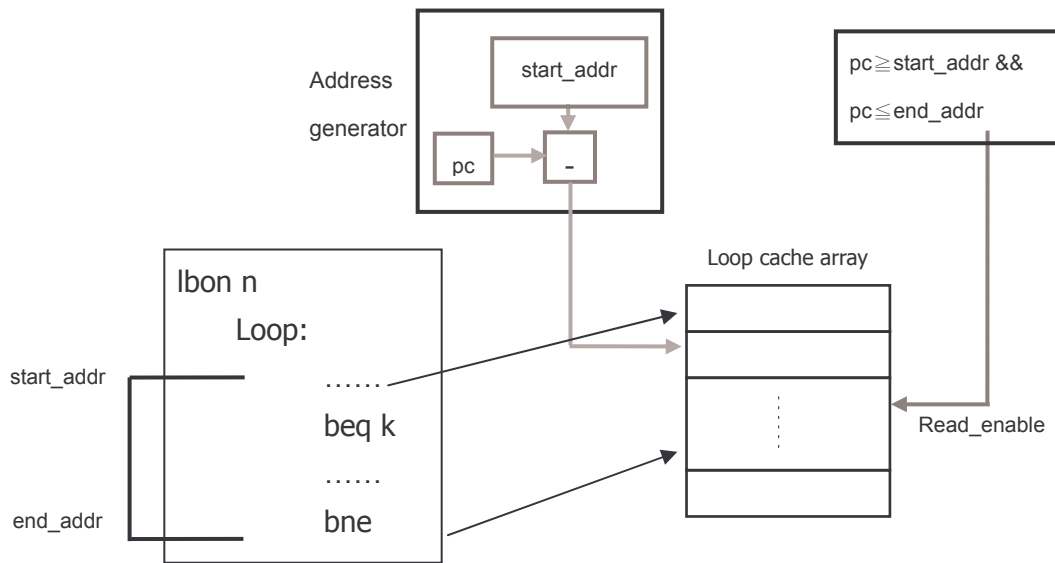


Figure 2.4: the management of Pre-load Loop Cache

2.2.4 Cluster Loop Cache

To dynamically fill code segments into loop buffer, [17] uses a special instruction “lbon n”, where n is number of instruction should be filled into loop buffer, to indicate where start to fill instructions into loop buffer. [17] also uses the same method to calculate LPC so that [17] encounters the same difficulties with [16]. The management of Cluster Loop Cache is similar to Pre-load Loop Cache which is already shown in figure 2.4.

Except for [15], all designs [14, 16, and 17] use compiler to assist in loop detection. This causes that program(s) must be recompiled in order to execute on a CPU containing one of these designs.

Chapter 3

Design of the Forward-Branch and Subroutine Bufferable Innermost Loop Buffer

In this chapter, we present the approach of our buffering mechanism which can store innermost loop with forward branch(es) and subroutine(s). In Section 3.1, we describe the features of our approach. In Section 3.2, we present the architecture and design issues of our approach in detail. In Section 3.3, we propose a possible design for our approach.

3.1 Features of Our Approach



In order to avoid the increase in the hardware design complexity, and can store innermost loop with forward branch and no-loop-inside subroutine, we add an extra bit in BTB to indicates whether the loop buffer stores the fall-through or target trace after a forward branch. No-loop-inside subroutine is also handled by the similar way. Here, we list several features of our approach as follows:

1. Loop buffer is a tagless memory unit.
2. Only one innermost storable.
3. Each entry in loop buffer only stores one instruction.
4. Instruction sequence in loop buffer is same with instruction trace, i.e. loop buffer only stores one execution path of innermost loop. This is because we does not employ extra address generator in our design, instructions must be fetched from loop buffer in sequence order.

5. Dynamically filling and refilling instructions according to runtime branch behaviors.

3.2 Architecture and Design Issues of Our Approach

In our approach, only one innermost loop can be stored in loop buffer. Since we does not employ extra address generator in our design, instructions must be fetched from loop buffer in sequence order such that only one execution path in innermost loop can be stored. To achieve these objects, we propose a design approach of loop buffer, as illustrated in figure 3.1. The main components consist of loop buffer, loop buffer controller and FD-bit (filled direction bit) in BTB.

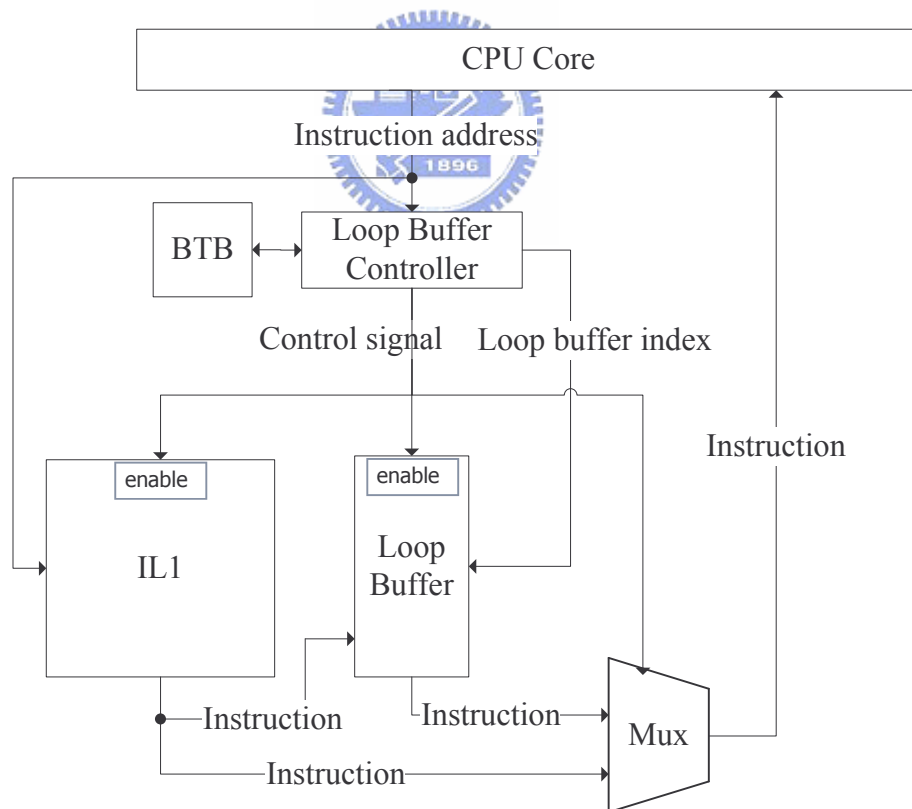


Figure 3.1: Architecture of our approach

Loop buffer controller is responsible for:

- (1) innermost loop detection;
- (2) filling or refilling an innermost loop into loop buffer;
- (3) determining that CPU core should fetch intrusions from loop buffer or IL1;
- (4) handling incorrect instruction filling and fetching due to branch miss-prediction; and
- (5) subroutine handling.

These five design issues will be introduced in detail later. FD-bit records the branch prediction result during filling or refilling instructions into loop buffer. It assist loop buffer controller in determining whether the loop buffer stores the fall-through or target trace after a forward branch.

3.2.1 Innermost Loop Detection

The objectives of innermost loop detection are as follow:

- (1) Detecting an innermost loop as early as possible to increase loop buffer utilization; and
- (2) Reducing the miss-detection rate such that energy overhead caused by miss-detection is low.

However, both objectives conflict each other. We therefore propose two innermost loop detection policies which are aimed at different objectives in this thesis. First, called FILL-1, if a backward branch is taken once, we identify that an innermost loop is detected. Second, called FILL-2, an innermost loop is detected only if a same backward branch is taken twice successively.

Since FILL-1 is more aggressive than FILL-2, it can increase loop buffer utilization. But FILL-1 has higher miss-detection rate such that it may cause higher power consumption. The power simulation of both policies will be shown in later

chapter.

We also can use a special instruction as “sbb” proposed in [1] to indicate that the execution flow is in an innermost loop at run time, but it need compiler to support.

During detecting an innermost loop, loop buffer controller also determines whether this innermost loop exists in loop buffer or not. If yes, the following instruction sequence has been stored in loop buffer, CPU core can immediately fetch instruction from loop buffer. If no, the current detected innermost loop is not same with one in loop buffer, and the following instruction sequence should be filled into loop buffer right now.

To determine whether this innermost loop exists in loop buffer or not, we add an extra register, called L_addr , to record the start or end address of an innermost loop been stored in loop buffer. If the program counter value is the same with L_addr , it indicates that the following instructions sequence has been stored in loop buffer. Otherwise, loop buffer controller should start to fill following instruction sequence into loop buffer.

Advantage of using start address is detecting existed innermost loop in one loop iteration in advance. Using end address has the advantage of fetching instructions from first one in innermost loop. However, using start address would cause that CPU core must start to fetch instructions from second one due to the delay of comparing L_addr with start address of an innermost loop. An innermost loop is detected until CPU core fetches the last instruction of one, if using end address.

3.2.2 Filling or Refilling an Innermost Loop Into Loop Buffer

Since only one execution path can be stored into loop buffer in our approach, we must determine which execution path should be stored into loop buffer.

Intuitively, the most frequently executed path should be stored. However, detecting the most frequently executed path would take a period of time such that loop buffer utilization is reduced. Therefore, the objective of filling or refilling an innermost loop into loop buffer has two:

- (1) filling or refilling instructions as early as possible; and
- (2) trying to fill or refill the most frequently executed path.

These are very similar with innermost loop detection.

To meet different objectives, we propose several filling or refilling policies. Loop buffer controller starts to fill instructions if first, partial or all forward branch(es) in innermost loop are predicted as strongly or weakly taken(non-taken), after detecting a innermost loop.

Refilling is executed when the execution path in innermost loop had changed. Similar to filling strategy, refilling is started only if first, partial or all forward branch(es) in innermost loop change its or their prediction result from strongly or weakly taken/non-taken to non-taken/taken.

During filling or refilling, the branch prediction result of each forward branch stored in loop buffer and the start or end address of innermost loop are recorded into FD-bit and L_addr respectively. After filling or refilling, loop buffer controller must count how many instructions are stored in loop buffer and record this result in an extra register, called L_leng.

During fetching instructions from loop buffer, loop buffer controller uses a counter to count how many instructions have been fetched. Once the counter counts down to zero, CPU core will fetch instructions from IL1 instead of loop buffer and loop buffer controller must start to detect innermost loop again.

In this paper, we employ the most aggressive policy for both filling and refilling, i.e. loop buffer controller starts to fill or refill instructions without

considering each forward branch's state (strongly or weakly taken/non-taken). In other word, loop buffer controller only follows instruction fetching sequence to fill or refill instructions. This is because that only 0.67% of branch prediction results in an innermost loop changes from weakly taken/non-taken to weakly non-taken/taken, i.e. the execution path in an innermost loop changes infrequently. Using the most aggressive policy can lead to higher loop buffer utilization such that more instruction fetch power is reduced.

3.2.3 Determining that CPU Core Should Fetch Intrusions From Loop Buffer or IL1

There are four situations that CPU core should fetch instructions from IL1 as follows:

1. No innermost loop is detected.
2. Loop buffer controller has detected an innermost loop, but not successfully fills or refills instructions into loop buffer.
3. Loop buffer is full due to a BIG loop which its size is larger than loop buffer's capacity.
4. The execution path has changed, but instructions located on new execution path do not be stored in loop buffer. We call this situation as loop buffer miss.

First and second situation can be determined according to the current status of loop buffer controller. The current status means current action of loop buffer controller, such as innermost loop detection, filling or refilling instructions ... etc. Comparing L_leng with the size of loop buffer, third situation can be solved. Fourth situation can be determined according to the comparing result of FD-bit with branch prediction result. This is because FD-bit can indicate that the loop buffer stores the

fall-through or target trace after a forward branch.

3.2.4 Handling Incorrect Instruction Filling and Fetching due to Branch Misprediction

The reason why we need to handle branch misprediction during fetching or filling instructions is that branch misprediction means the execution path has changed and instructions followed a mispredicted branch do not exist in loop buffer.

The simplest method which copes with forward branch misprediction during filling instructions into loop buffer is flushing all instructions which have been filled into loop buffer. However, since instruction sequence before a mispredicted forward branch is same, this method is an inadvisable one. We therefore propose second method that loop buffer controller just counts back M entries from current position and then refills instructions located at another control path. M is number of pipeline stages between instruction fetch (IF) and execution stage (EXE). Since an innermost loop has been successfully filled into loop buffer before a mispredicted backward branch, loop buffer controller just lets CPU core fetch instructions from IL1 without flushing instructions in loop buffer.

In another situation, i.e. during fetching instructions from loop buffer, a forward or backward branch misprediction indicates that the execution path has changed. Loop buffer controller therefore should let CPU core fetch instructions from IL1.

An exception happened when a new BTB entry created caused by misprediction and replaced the prediction result and FD-bit of branch in BTB entry during Loop detecting, which this branch is stored in loop buffer. Loss of the FD-bit record will make the loop buffer controller fail to determine CPU core should fetch instruction from loop buffer or IL1. If this situation occurred, we should flush loop

buffer.

Recognizing the FD-bit of branch which is stored in loop buffer at present of innermost loop or in the pass of innermost loop can not implement straightforward, because the FD-bit of branch which is stored in loop buffer in the pass of innermost loop would not be clean. To avoid the design complexity of detecting and handling this exception, we propose a simple approach. When a new BTB entry created caused by misprediction during loop detecting, L_addr will be clean so that loop buffer controller can not detect this innermost loop which already filled in loop buffer. Our simulation of MiBench is shown that this approach has decreased the utilization of loop buffer only 0.03% compared by the case of exception has never happened.

3.2.5 No-loop-inside Subroutine Handling

Subroutine call and subroutine return can be automatically handled by BTB and return stack, respectively. To store no-loop-inside subroutine in loop buffer, we only need to handle two situations:

- (1) CPU core has no return stack; and
- (2) return stack is full.

In fact, case 1 and 2 is same due to a full return stack means CPU core can not store any return address into return stack, i.e. return stack is useless at this moment. Therefore, we only need to handle one of situations.

The basic idea of handling subroutine return is to fill but disregard these invalid instructions followed with subroutine return. Since a subroutine return is an always-taken branch, CPU core without return stack always fetches G invalid instructions. According to when the subroutine return is detected by CPU core, G has two different values:

(1) G is number of pipeline stages between IF and EXE; and

(2) G is number of pipeline stages between IF and instruction decoder (ID).

However, these invalid instructions would be automatically flushed by CPU core and do not affect the correctness of program. If we store G invalid instructions into loop buffer, the instruction fetch sequence is held.

We use the figure 3.2 to explain this idea. Here, a five pipeline stages CPU core without return stack is assumed in this example. When Sub_A returns to Loop_A, the instruction fetch sequence is o (subroutine return), p, q, r, and s in which p and q are invalid instructions. If we also follow the same instruction fetch sequence, i.e. o, p, q, r and s, to fill instructions into loop buffer, the fetch sequence is held.

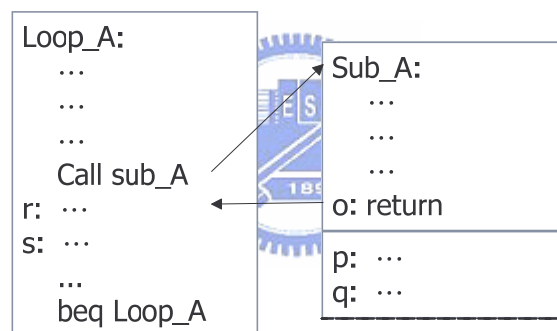


Figure 3.2: An innermost loop consists of subroutine

3.3 Operation

In this section, we propose a possible design for our approach and describe it as follows. The operation status of our design which is similar with [1] consists of three states: IDLE, FILL and ACTIVE. In the figure 3.3, the gray rectangle and the solid black line are currently accessing block and bus respectively during different states.

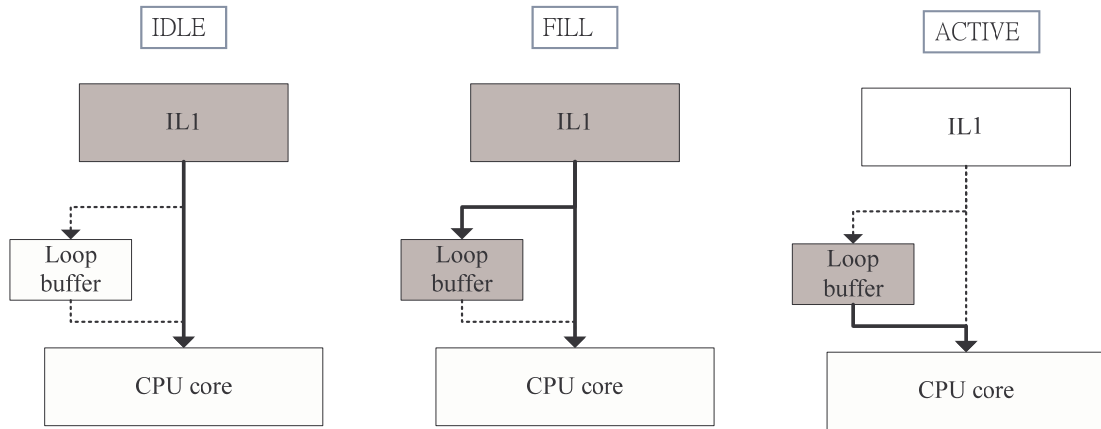


Figure 3.3: Memory accessing in different states

The state diagram of the loop buffer controller's finite state machine (FSM) is shown in the figure 3.4. When CPU core initializes or resets, loop buffer controller enters IDLE state first. During IDLE state, loop buffer controller continuously detects an innermost loop, as action A. Action C is taken if and only if an innermost loop has been detected and this innermost loop has been stored in loop buffer. Otherwise, loop buffer controller would enter FILL state, as action B. When BTB entry created caused by misprediction, L_addr will be clean.

There are two possible policies for detecting the innermost loop by a backward branch taken once or twice, respectively, denoted "FILL-1" and "FILL-2"; and two possible policies for detecting the innermost loop which has been stored in loop buffer are recording the start or end address of this innermost loop, denoted "START" and "END". The policies "FILL-1" and "START" are more aggressive than "FILL-2" and "END", respectively. The best choice of those policies in our simulation is "FILL-1" and "END". We will show the simulation result at next chapter.

During FILL state, instructions are sequentially filled into loop buffer from first entry, as action D. In the meanwhile, the branch predicted result of each forward

branch stored in loop buffer and the start or end address of innermost loop are also recorded into FD-bit and L_addr respectively. After successfully filling all instructions, loop buffer controller counts how many instructions are stored in loop buffer and record this result in L_len and enters ACTIVE state, as action E. Loop buffer is full caused by a BIG loop would let loop buffer controller return to IDLE state, as action F. Action G would give up this fill iteration and returns IDLE state when a forward branch miss-prediction occurs.

There two possible policies for handling miss-prediction of a forward branch. First policy is giving up this fill iteration and returns IDLE state, denoted “GOTO IDLE”. Second policy is counting back M entries of loop buffer and continuing filled instructions, denoted “CONT FILL”. “CONT FILL” is more aggressive than “GOTO FILL”. The best choice of those policies is “GOTO IDLE” (i.e. Action G). We will show the simulation result at next chapter.

During ACTIVE state, CPU core fetches instructions from loop buffer instead of IL1, as action H. When loop buffer miss, CPU core fetches instructions from IL1 and return to FILL state as action I. When CPU core has already fetched last instruction of loop buffer (action K) or loop buffer controller encounters a branch miss-prediction (action J), loop buffer controller would enter IDLE state. First situation occurs when loop buffer encounters a BIG loop. Since loop buffer dose not store a whole BIG loop, loop buffer controller must enter IDLE state to let CPU core fetch other instructions of BIG loop from IL1 after the last instruction in loop buffer has fetched. The second situation, a branch miss-prediction occurs, indicates that the execution path has changed and loop buffer does not store instructions on this execution path. Hence, loop buffer controller must also enter IDLE state.

There are two possible policies for handling loop buffer miss. First policy is entering FILL state to store another execution path immediately, denoted “aFILL”.

Second policy is entering FILL state when the branch prediction result is in strong state (i.e. strong taken or strong not taken), denoted “pFILL”. “pFILL” can avoid the invalid loop buffer writing caused by prediction result changing frequently. “aFILL” is more aggressive than “pFILL”. The best choice for handling loop buffer miss is “aFILL” (i.e. Action I).

There are three possible policies for handling miss-prediction of a forward branch. First policy is giving up this fill iteration and returns IDLE state, denoted “aIDLE”. Second policy is counting back M entries of loop buffer and continuing filled instructions, denoted “aFILL”. “aFILL” may cause more loop buffer miss, because not all miss-prediction will cause the branch prediction result changed prediction direction. Third policy is counting back M entries of loop buffer and continuing filled instructions when the branch prediction result is in weak state (i.e. weak taken or weak not taken), denoted “pFILL”. “pFILL” can improve the disadvantage of “aFILL”, because the branch prediction result changed prediction direction when the branch prediction result is in weak state and miss-prediction happened. The best choice of those policies is “pFILL” (i.e. Action J). We will show the simulation result at next chapter.

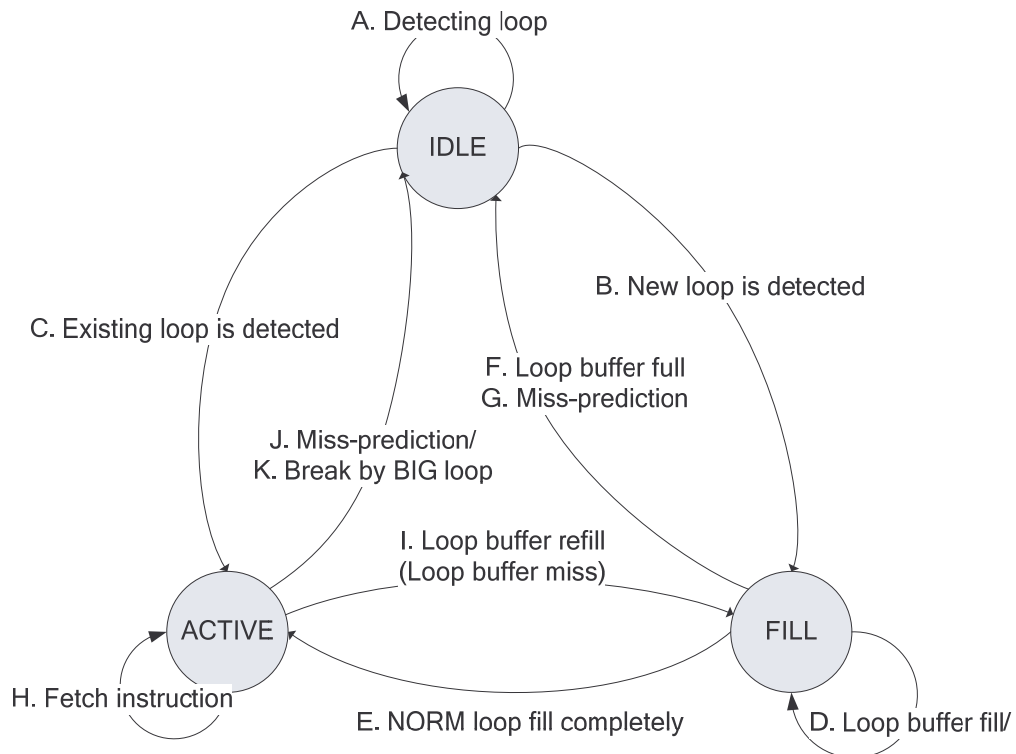
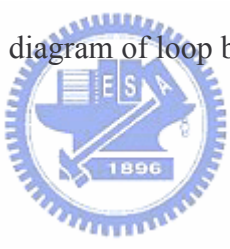


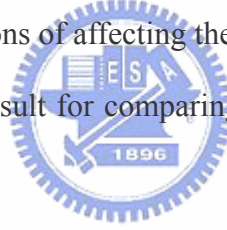
Figure 3.4: State diagram of loop buffer controller



Chapter 4

Simulation and Evaluation

In Chapter 3, we proposed the mechanism and architecture of our buffering mechanism which can store innermost loop with forward branch(es) and subroutine(s). In this chapter, we present the simulation result and evaluation of our design. In Section 4.1, we present the power model of instruction fetch power. In Section 4.2, we describe the simulation environment and the benchmark programs that we used in the simulation. In section 4.3, we present the simulation results for finding the best choice of loop buffer policies in each state of loop buffer. In section 4.4, we present the penalty caused by miss-prediction of loop buffer policies. In section 4.5, we analyses the reasons of affecting the loop buffer efficiency. In section 4.6, we present the simulation result for comparing other loop buffer managements and our analyses on them.



4.1. Power Model

The instruction fetch power (P_{IF}) per fetch dissipated by the loop buffer and lower level instruction memories can be expressed by equation (1):

$$P_{IF} = P_{IC} * R_{IC} + P_{LB} * R_{LB} + P_{ctrl} \quad (1)$$

where P_{IC} and P_{LB} are fetch power of lower level instruction memories and loop buffer respectively, R_{IC} and R_{LB} are access ratio of lower level instruction memory and loop buffer, respectively. In this paper, R_{IC} (R_{LB}) is defined as the ratio of number of instruction fetch from of lower level instruction memory (loop buffer) to total number of instruction fetch, as expressed by equation (2):

$$R_{IC}(R_{LB}) = \frac{\text{number of instrucion fetch from lower level memory (loop buffer)}}{\text{total number of instruction fetch}} * 100\% \quad (2)$$

Since P_{IC} is not affected by operation strategy and size of loop buffer, we assume P_{IC} as a constant in this paper. Comparatively, P_{LB} , R_{IC} and R_{LB} highly depend on the size of loop buffer. P_{ctrl} is the power consumed by loop buffer controller. Since loop buffer controller is always active, we assume that each instruction fetch would consume P_{ctrl} .

The loop buffer controller of each design is synthesized using Synopsys design tools in 0.18 μm TSMC CMOS technology. The power consumptions of loop buffer and IL1 are calculated by Wattch power modeling tool. The ratios of P_{LB} and P_{IC} are shown in table 4.1. The power consumption of FD-bit is 0.13% of power consumption of P_{IC}

Table 4.1: Ratio of P_{LB}

Size of loop buffer	64B	128B	256B	512B	1KB	2KB
$(P_{LB}/P_{IC}) * 100\%$	8.26%	8.92%	10.35%	13.56%	21.40%	38.34%

4.2 Simulation Environment

In Section 4.2.1, we describe the simulation tools that we used. In Section 4.2.2, we describe the benchmark programs that we used in the simulation.

4.2.1 Simulator

We use SimpleScalar/ARM simulator [18] to evaluate each design. SimpleScalar is an execution-driven simulator and used to simulate modern processor architectures.

The power evaluation of loop buffer, BTB and IL1 is based mainly on the

Wattch [20] power modeling tool. The loop buffer controller of each method is synthesized using Synopsys design tools.

In this thesis, we experimented with the different sizes of loop buffer, included 64, 128, 256, 512, 1024 and 2048 bytes (B). Other parameters used in SimpleScalar are shown in the table 4.2.

Table 4.2: Parameters setting in SimpleScalar/ARM

Parameter	Value
Loop buffer	64B, 128B, 256B, 512B, 1KB and 2KB
IL1	8KB, direct-mapped, 32B line
Branch predictor	Bimodal
BTB	512-set, 4-way
Return stack	No return stack

4.2.2 Benchmark Programs

The benchmark programs that we use are from MiBench suite [19]. These benchmarks are divided into six suites with each suite targeting a specific area of the embedded market. The six categories are Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. All the programs are available as standard C source code so that it focuses on portable applications written in high-level languages as processor architecture and software developers are moving in this direction. Where appropriate, there provide a small and large data set. The small date set represents a light-weight, useful embedded application of benchmark, while the large data set provides a more stressful, real-world application. We use the large data set in this thesis. All the programs of Mibench are publicly available and widely used on general purpose

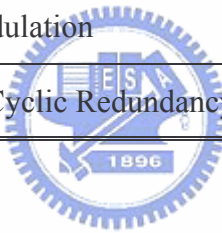
processors. Table 4.3 gives the description of the benchmark programs that we used.

Table 4.3: MiBench workloads

Benchmark name	Description
Automotive and Industrial Control	
bitcount	Testing the bit manipulation abilities of a processor by counting the number of bits in an array of integers.
qsort	Sorting a large array of strings into ascending order using the well known quick sort algorithm.
susan (edges)	Susan is an image recognition package. It was developed for recognizing corners and edges in Magnetic Resonance Images of the brain and it can smooth an image and has adjustments for threshold, brightness, and spatial control.
susan (corners)	
susan (smoothing)	
Consumer Devices	
jpeg encode	JPEG is a standard, lossy compression image format and it is commonly used to view images embedded in documents.
jpeg decode	
tiff2bw	Tiff2bw converts a color TIFF image to black and white image.
tiff2rgba	Tiff2rgba converts a color image in the TIFF format into a RGB color formatted TIFF image.
tiffdither	Tiffdither dithers a black and white TIFF bitmap to reduces the resolution and size of the image at the expense of clarity.
tiffmedia	Tiffmedian converts an image to a reduced color palette by taking several medians of the current color palette.
lame	A GPL'ed MP3 encoder that supports constant, average and

	variable bit-rate encoding.
mad	A high-quality MPEG audio decoder.
typeset	Typeset is a general typesetting tool, that has a front-end processor for HTML.
Office Automation	
ghostscript	Ghostscript is a postscript language interpreter without its graphical interface.
stringsearch	This benchmark searches for given words in phrases using a case insensitive comparison algorithm.
ispell	Ispell is a fast spelling checker that is similar to the Unix spell, but faster.
rsynth	Rsynth is a text to speech synthesis program that integrates several pieces of public domain code into a single program.
Networking	
dijkstra	Calculating the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm.
patricia	Patricia tries are used to represent routing tables in network applications.
Security	
blowfish encrypt	A symmetric block cipher with a variable length key.
blowfish decrypt	
sha	SHA is the secure hash algorithm that produces a 160-bit message digest for a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures.

rijndael encrypt	Rijndael was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks.
rijndael decrypt	
Telecommunications	
FFT	A Fast Fourier Transform and its inverse transform on an array of data.
IFFT	
GSM encode	The Global Standard for Mobile (GSM) communications is the standard for voice encoding/decoding in Europe and many countries.
GSM decode	
ADPCM encode	The encoder and decoder of Adaptive Differential Pulse Code Modulation
ADPCM decode	
CRC32	A 32-bit Cyclic Redundancy Check (CRC) on a file



4.3. Simulation Results of Loop Buffer Policies

In the following sections, we will show our simulations and the analysis for finding the best choice of the loop buffer policies in each state. In section 4.3.1, we describe loop buffer policies in each state briefly. In section 4.3.2, we present the access ratio of loop buffer and lower level memory and instruction fetch power reduction for these loop buffer policies.

4.3.1 Loop Buffer Policies in Each loop Buffer state

There are three states of loop buffer, IDLE state, FILL state, ACTIVE state.

In IDLE state, we handle the loop buffer detection. To detect the innermost loop, we propose two policies “FILL-1” and “FILL-2”. “FILL-1” and “FILL-2” are

detecting innermost loop when a backward branch taken once or twice successive, respectively. “FILL-1” is more aggressive than “FILL-2”. The hardware cost of “FILL-1” is less than “FILL-2”, because “FILL-2” needs more one register to record the backward branch than “FILL-1”.

To detect the innermost loop which has been stored in loop buffer, we propose two policies “START” and “END”. “START” and “END” are detecting the innermost loop which has been stored in loop buffer by recording the start or end address of this innermost loop, respectively. “START” is more aggressive than “END”. The hardware cost of “END” is less than “START”, because “END” can use BTB to assist record the end address of this innermost loop.

In FILL state, we handle the execution path changed caused by branch miss-prediction. To handle miss-prediction, we propose two policies “GOTO IDLE” and “CONT FILL”. “GOTO IDLE” and “CONT FILL” are entering IDLE state or filling the changed execution path into loop buffer, respectively. “CONT FILL” is more aggressive than “GOTO IDLE”. The hardware cost of “GOTO IDLE” is less than “CONT FILL” because “CONT FILL” needs extra logic (for example, another counter) to count back the index in loop buffer to the next entry of this miss-prediction branch.

In ACTIVE state, we handle the execution path changed caused by loop buffer miss (i.e. branch prediction direction is not equal to FD-bit) or branch miss-prediction. To handle loop buffer miss, we propose two policies “aFILL” and “pFILL”. “aFILL” and “pFILL” are filling the changed execution path immediately or when the branch prediction result is in strong state, respectively. “aFILL” is more aggressive than “pFILL”. The hardware cost of “aFILL” is less than “pFILL”, because “pFILL” needs extra logic to determinate whether the prediction result is in strong state or not.

To handle the branch miss-prediction, we propose three policies “aIDLE”, “aFILL”, “pFILL”. “aIDLE” is entering IDLE state without any changing. “aFILL” and “pFILL” are filling the changed execution path immediately or when the branch prediction result is in weak state, respectively. “aFILL” is more aggressive than “pFILL” and “aIDLE”, but “aFILL” may cause more loop buffer miss because not all branch miss-prediction would cause prediction direction changed. The hardware cost of “aIDLE” is less than “aFILL” and “pFILL”, because “aFILL” or “pFILL” needs extra logic (for example, another counter) to count back the location in loop buffer to the next entry of this miss-prediction branch.

We summarize the policies of loop buffer in each state of loop buffer to table 4.4. There are 48 cases of loop buffer policies set totally.

Table 4.4: Policies of loop buffer in each state of loop buffer

State of Loop buffer	Handling event	Policy
IDLE state	Detecting the innermost loop	FILL-1
		FILL-2
	Detection the innermost loop stored in loop buffer	START
		END
FILL state	Branch miss-prediction	GOTO IDLE
		CONT FILL
ACTIVE state	Loop buffer miss	aFILL
		pFILL
	Branch miss-prediction	aIDLE
		aFILL
		pFILL

4.3.2 Simulation Results of Loop Buffer Policies

To find the best choice of loop buffer policies set, we divide the simulation into two parts. First, we simulate each loop buffer policies in the same state of loop buffer and analysis them to find the local optimal. Second, we simulate loop buffer policies set to find the best choice of loop buffer policies set.

4.3.2.1 Simulation Results of the Loop Buffer Policies in IDLE State

In IDLE state, we compare four loop buffer policies set list in table 4.5.

Table 4.5: Loop buffer policies sets (IDLE state)

IDLE state		FILL state	ACTIVE state	
Detecting the innermost loop	Detection the innermost loop stored in loop buffer	Branch miss-prediction	Loop buffer miss	Branch miss-prediction
FILL-1	END START	GOTO IDLE	aFILL	aIDLE
FILL-2	END START			

Figure 4.1 shown R_{IC} for different loop buffer policy sets in different loop buffer size. We examined the size of loop buffer from 64 to 2048 bytes, i.e. 16 to 512 instructions. Simulation results show that “FILL-1” and “SATRT” averagely further decrease R_{IC} by 2.20% and 0.41%, respective (“FILL-2” and “END”).

Ideally, the R_{IC} of (“FILL-1”, “START”) will decrease 8.33% than (“FILL-2”, “END”), because that average number of innermost loop iterations are 12.41. The effects of the aggressive policies (i.e. “FILL-1” or “START”) are limited, because of the nature of MiBench. If one benchmark has much execution time on innermost loop, these innermost loops also have a lot of loop iterations; on the other hand, if one benchmark has less execution time on innermost loop, these innermost loops have few loop iterations.

Figure 4.2 shown R_{LB} for different loop buffer policies sets in different loop buffer size. Simulation results show that “FILL-1” and “SATRT” averagely further increase R_{LB} by 10.32% and 0.39%, respective (“FILL-2” and “END”). R_{LB} of “FILL-1” is more than “FILL-2” because of the invalid loop buffer filled caused of miss-detection of innermost loop.

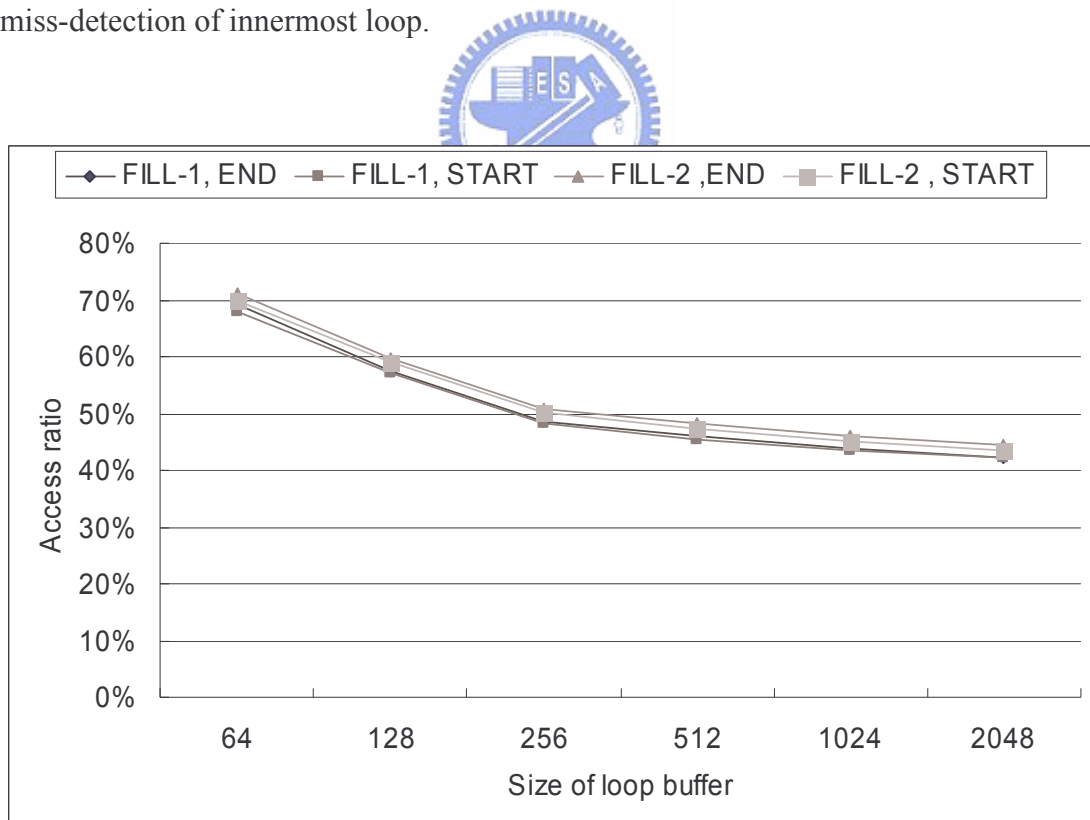


Figure 4.1: Access ratio of IL1 of different loop buffer policies sets (IDLE state)

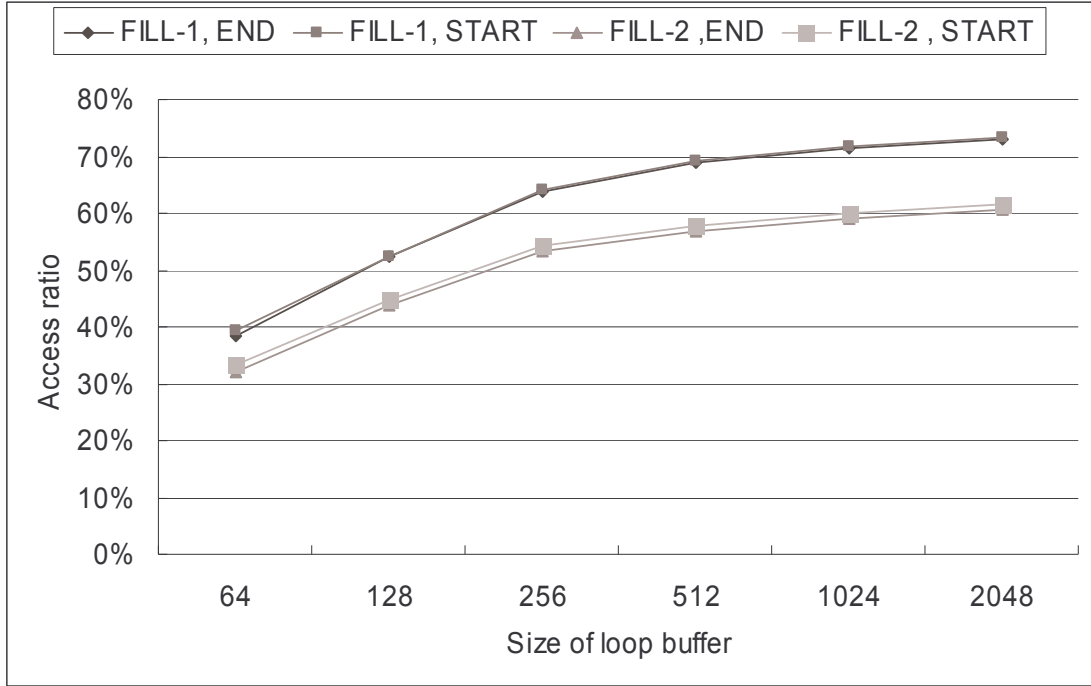


Figure 4.2: Access ratio of loop buffer of different loop buffer policies sets

(IDLE state)



Table 4.6 is the power consumption of loop buffer controller of different loop buffer policy sets.

Table 4.6: Ratio of P_{ctrl} of different loop buffer policy sets (IDLE state)

	64B	128B	256B	512B	1KB	2KB
FILL1, end	1.91%	1.95%	2.00%	2.08%	2.25%	2.34%
FILL1, start	2.75%	2.83%	2.90%	2.97%	3.14%	3.23%
FILL2, end	2.57%	2.62%	2.65%	2.74%	2.90%	2.98%
FILL2, start	3.41%	3.49%	3.55%	3.63%	3.79%	3.87%

The reduction in instruction fetch power of different loop buffer policy sets is shown in the figure 4.3. For each loop buffer policies sets, 256B or 512B has the maximum power reduction. According to figure 4.1 and 4.2, increasing loop buffer size can improve R_{IC} and R_{LB} but also increases P_{LB} . Hence, larger loop buffer may

be not beneficial for P_{IF} . The best loop buffer policies set of these four set is (FILL-1, END) because of the minimum power consumption of loop buffer controller.

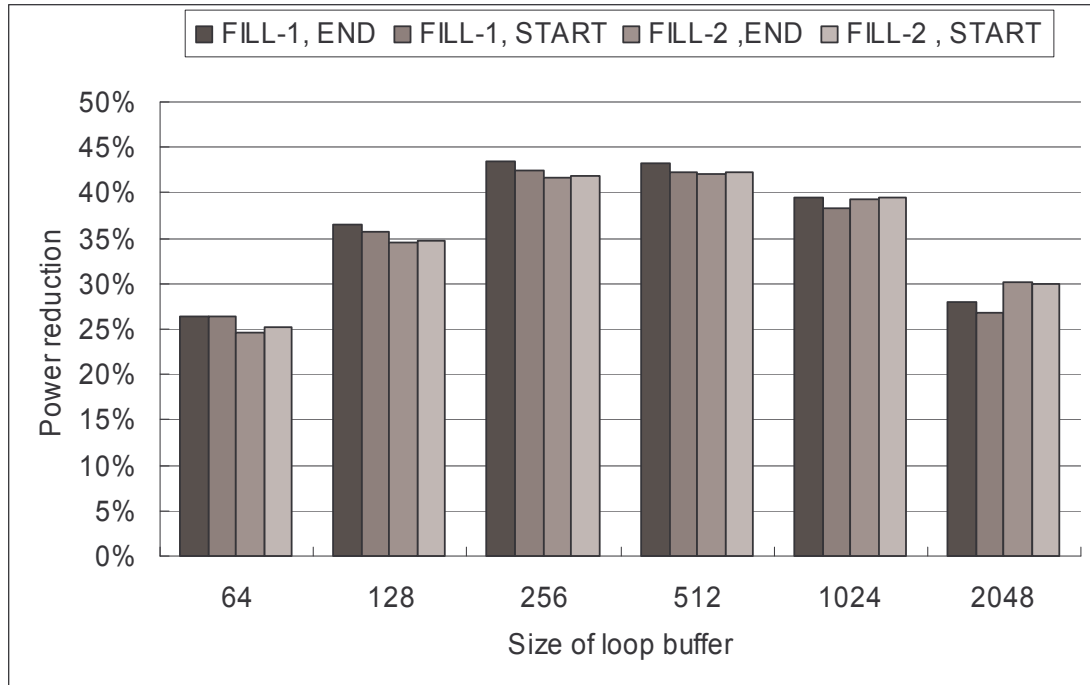


Figure 4.3: Reduction in instruction fetch power of different loop buffer policies sets (IDLE state)

4.3.2.2 Simulation Results of the Loop Buffer Policies in FILL State

In FILL state, we compare two loop buffer policies set list in table 4.7.

Table 4.7: Loop buffer policies sets (FILL state)

IDLE state		FILL state	ACTIVE state	
Detecting the innermost loop	Detection the innermost loop stored in loop	Branch miss-prediction	Loop buffer miss	Branch miss-prediction

	buffer			
FILL-2	END	GOTO IDLE	aFILL	aIDLE
		CONT FILL		

Figure 4.4 shown R_{IC} for different loop buffer policy sets in different loop buffer size. Simulation results show that “CONT FILL” averagely further decrease R_{IC} by 0.05% (“GOTO IDLE”).

The effect of the aggressive policy—“CONT FILL” are limited, because of the execution time in FILL state is limited, just about 4.37% averagely, and the execution path changed ratio in FILL state which has been defined as expressed by equation (3) is 18.79%.

$$\text{execution path ratio} = \frac{\text{the number of loop iteration which has execution path changed}}{\text{total number of loop iteration}} * 100\% \quad (3)$$

Figure 4.5 shown R_{LB} for different loop buffer policies sets in different loop buffer size. Simulation results show that “CONT FILL” averagely further increase R_{LB} by 1.66% (“GOTO IDLE”).

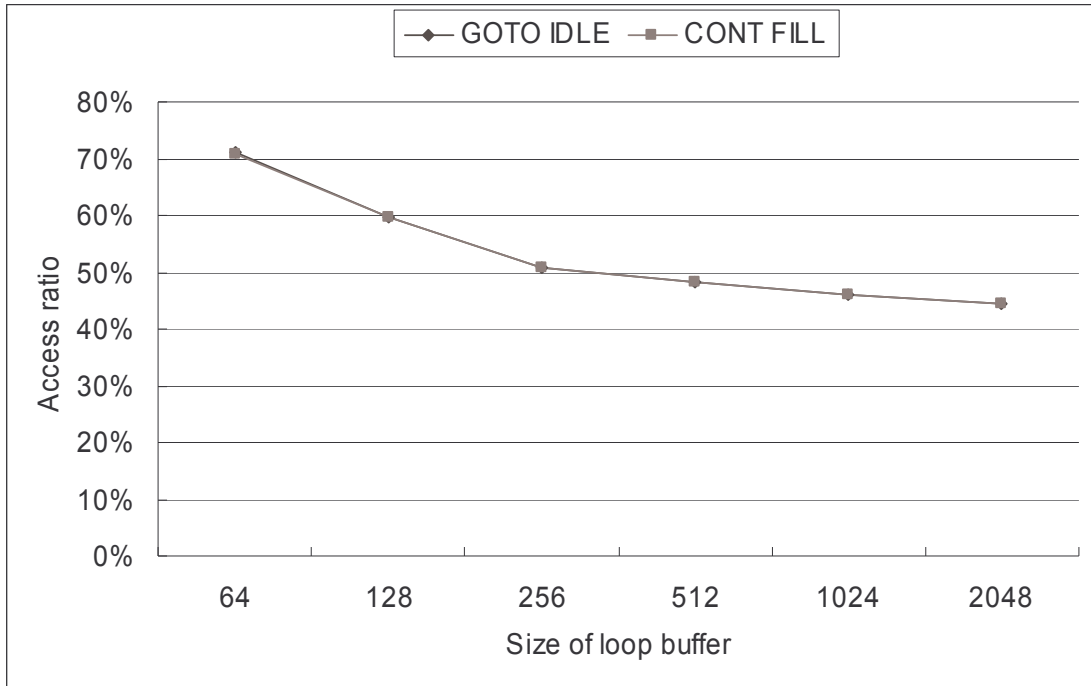


Figure 4.4: Access ratio of IL1 of different loop buffer policies sets (FILL state)

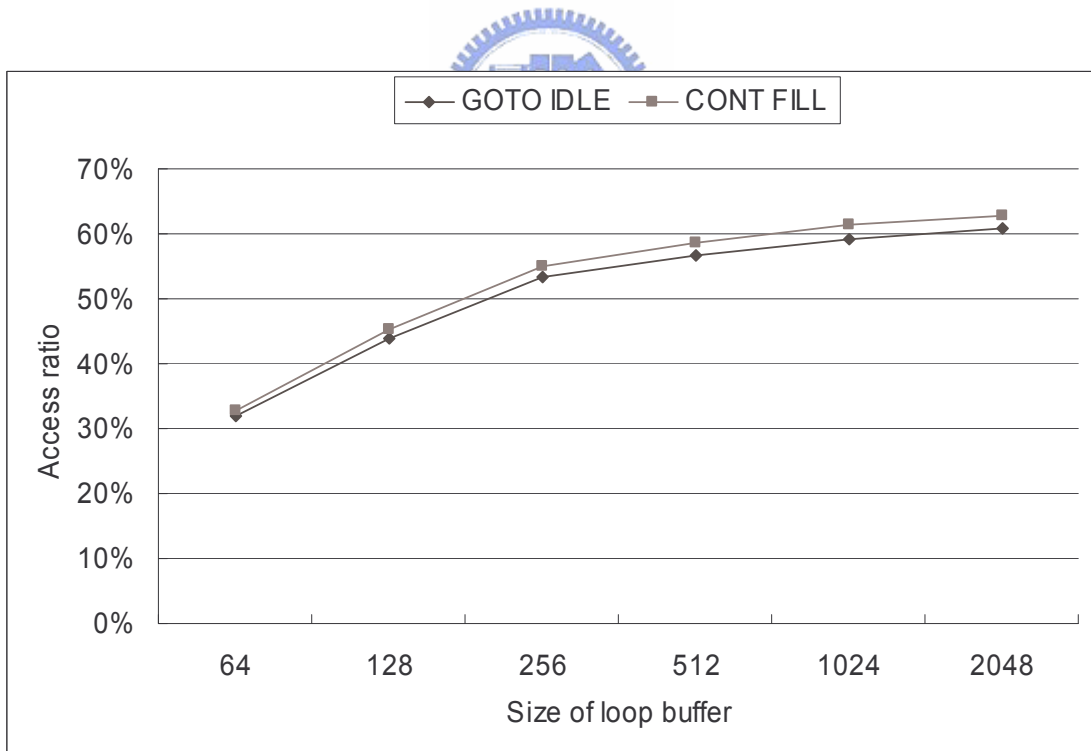


Figure 4.5: Access ratio of loop buffer of different loop buffer policies sets (FILL state)

Table 4.8 is the power consumption of loop buffer controller of different loop

buffer policy sets. To implement “CONT FILL”, we need add one counter to indicate the index of the next entry of the miss-prediction branch in loop buffer, and this counter will increase about 60% of power consumption of loop buffer controller.

Table 4.8: Ratio of P_{ctrl} of different loop buffer policy sets (FILL state)

	64B	128B	256B	512B	1KB	2KB
GOTO IDLE	2.57%	2.62%	2.65%	2.74%	2.90%	2.98%
CONT FILL	3.76%	3.92%	4.10%	4.28%	4.40%	4.56%

The reduction in instruction fetch power of different loop buffer policy sets is shown in the figure 4.6. The best loop buffer policies set of these two set is “GOTO IDLE”, because that in “CONT FILL”, the fetch power reduction of IL1 is less than the increasing of power consumption of loop buffer controller.

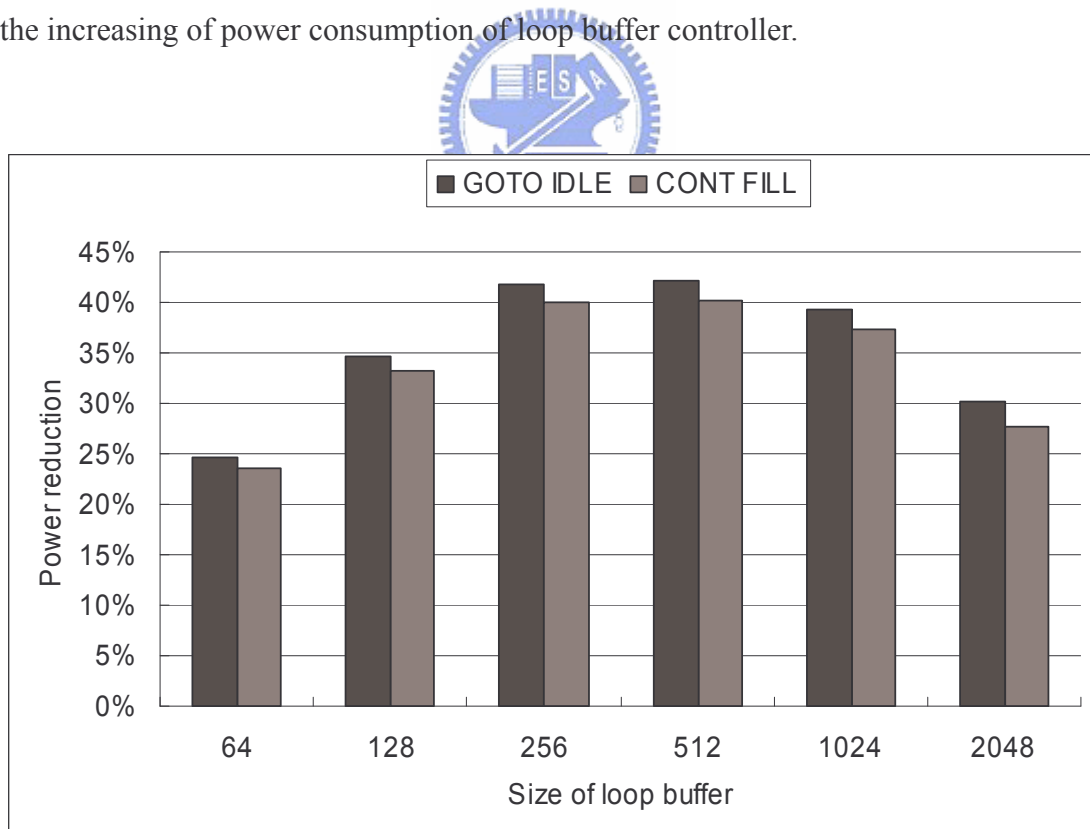


Figure 4.6: Reduction in instruction fetch power of different loop buffer policies sets (FILL state)

4.3.2.3 Simulation Results of the Loop Buffer Policies in ACTIVE State

In ACTIVE state, we compare six loop buffer policies set list in table 4.9.

Table 4.9: loop buffer policies sets (ACTIVE state)


IDLE state		FILL state	ACTIVE state	
Detecting the innermost loop	Detection the innermost loop stored in loop buffer	Branch miss-prediction	Loop buffer miss	Branch miss-prediction
FILL-2	END		aFILL	aIDLE
				aFILL
				pFILL
			pFILL	aIDLE
				aFILL
				pFILL

Figure 4.7 shown R_{IC} for different loop buffer policy sets in different loop buffer size. Simulation results show that the loop buffer miss handling policy—“aFILL” averagely further decrease R_{IC} by 0.12% (“pFILL”), and the branch miss-prediction handling policy—“pFILL” averagely further decrease R_{IC} by 0.13% / 0.45% (“aIDLE” / “aFILL”).

The effects of the aggressive policies are limited; because of the execution path changed ratio in ACTIVE state is 1.90%. And it implies that the execution path changed seldom in ACTIVE state.

Figure 4.8 shown R_{LB} for different loop buffer policies sets in different loop buffer size. Simulation results show that the loop buffer miss handling policy—“aFILL” averagely further increase R_{LB} by 0.04% (“pFILL”), and the branch miss-prediction handling policy—“aIDLE” averagely further decrease R_{IC} by 0.79% / 0.14% (“aFILL” / “pFILL”).

The results of R_{IC} and R_{LB} in branch miss-prediction handling policies show that “aFILL” will cause more invalid loop buffer filled but not decrease R_{IC} , and the best policy is “pFILL” which has least R_{IC} and limited increases R_{LB} .

The best policy of loop buffer miss handling is “aFILL” shown in the results, too.

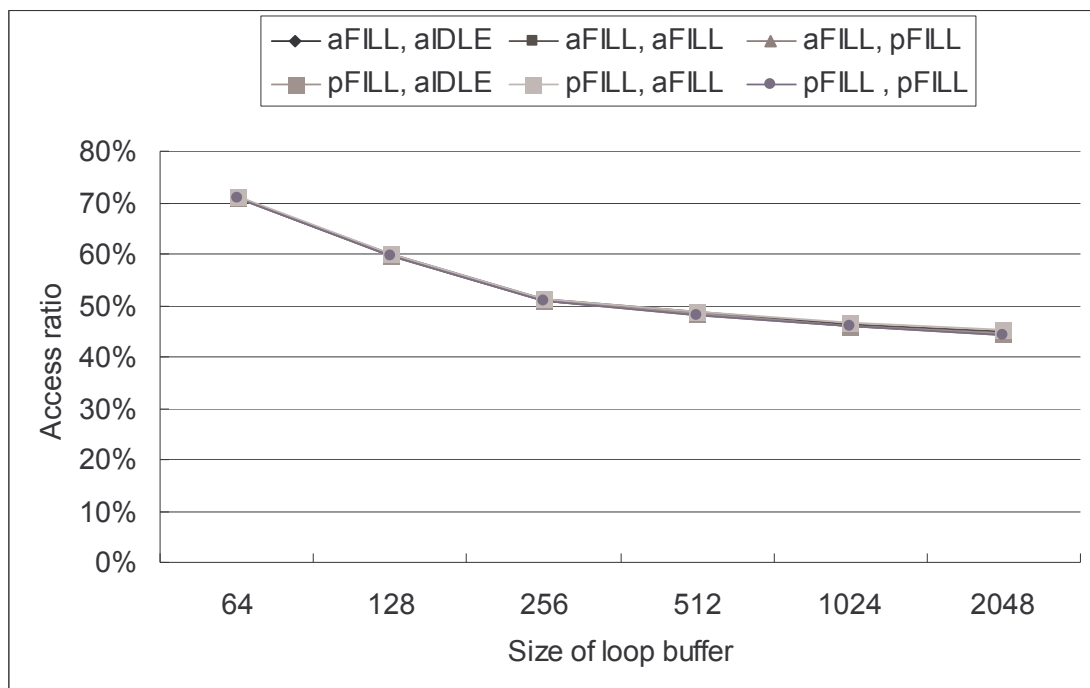


Figure 4.7: Access ratio of IL1 of different loop buffer policies sets (ACTIVE state)

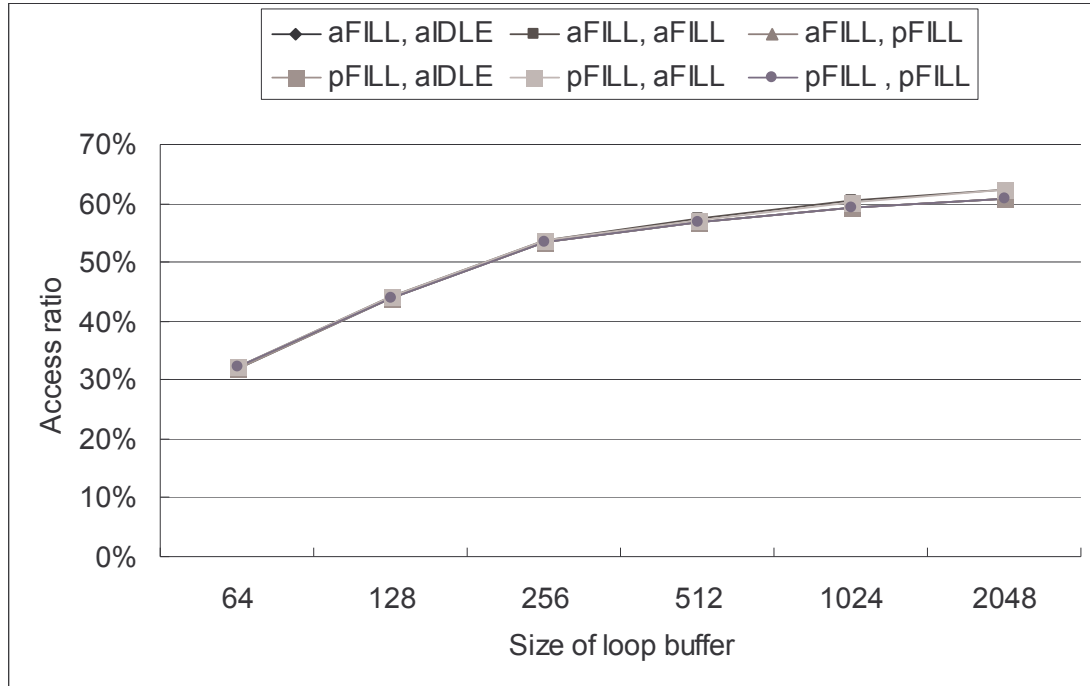


Figure 4.8: Access ratio of loop buffer of different loop buffer policies sets

(ACTIVE state)



Table 4.10 is the power consumption of loop buffer controller of different loop buffer policy sets. Like the policy—“CONT FILL” in FILL state, we need add one counter to indicate the index of the next entry of the miss-prediction branch in loop buffer.

Table 4.10: Ratio of P_{ctrl} of different loop buffer policy sets (ACTIVE state)

	64B	128B	256B	512B	1KB	2KB
aFILL, aIDLE	2.57%	2.62%	2.65%	2.74%	2.90%	2.98%
aFILL, aFILL	3.71%	3.87%	4.05%	4.23%	4.35%	4.51%
aFILL, pFILL	3.78%	3.94%	4.12%	4.30%	4.42%	4.58%
pFILL, aIDLE	2.67%	2.72%	2.75%	2.84%	3.00%	3.08%
pFILL, aFILL	3.81%	3.97%	4.15%	4.33%	4.45%	4.61%
pFILL, pFILL	3.88%	4.04%	4.22%	4.40%	4.52%	4.68%

The reduction in instruction fetch power of different loop buffer policy sets is

shown in the figure 4.9. The best loop buffer policies set of these two sets is (aFILL, aIDLE), because that in the branch miss-prediction handling policy—“aFILL” and “pFILL”, the fetch power reduction of IL1 is less than the increasing of power consumption of loop buffer controller, like the policy—“CONT FILL” in FILL state.

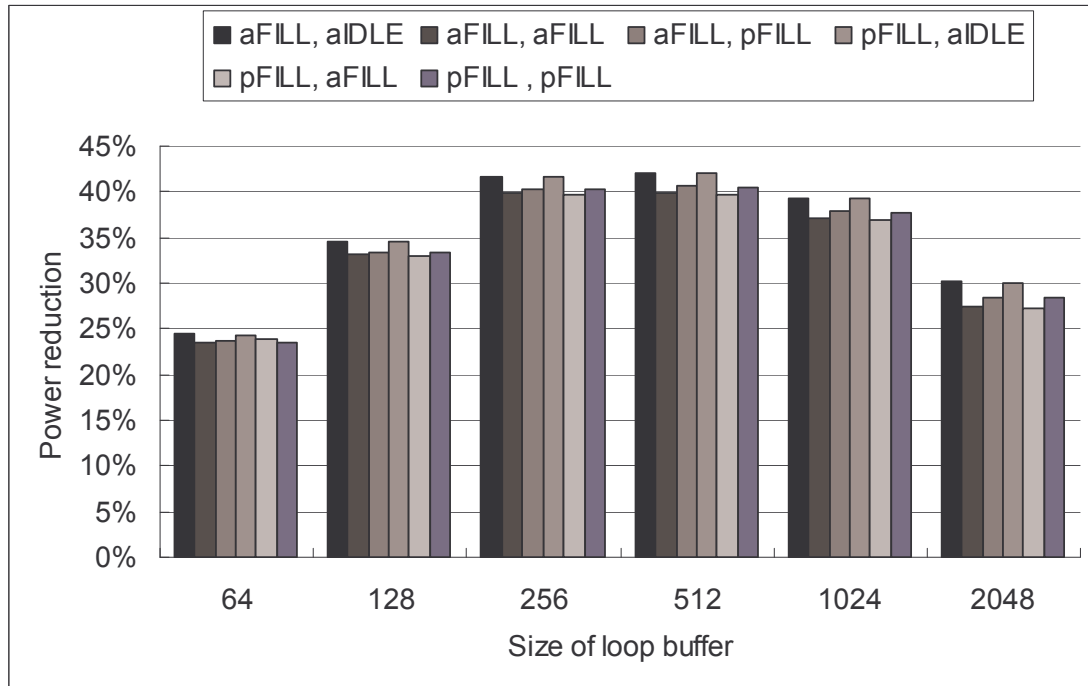


Figure 4.9: Reduction in instruction fetch power of different loop buffer policies sets (ACTIVE state)

4.3.2.4 Simulation Results of the Best Loop Buffer Policies

Now we have the best policy in each state of loop buffer. In IDLE state, the best policy is “(FILL-1, END)”; in FILL state, the best policy is “GOTO IDLE”; in ACTIVE state, the best policy is “(aFILL, aIDLE)”. The results show that the best policy in each state of loop buffer is the policy which has least power consumption of loop buffer controller, because that the power reduction of IL1 is less than the increase of power consumption of loop buffer controller in an aggressive policy like

“CONT FILL” in each state of loop buffer.

Since the power consumption of loop buffer controller is the most affecting of instruction fetch power reduction using loop buffer, we find that the hardware of branch miss-prediction handling in FILL state and ACTIVE state is the same, and it can be reused. So we propose another more aggressive loop buffer policy sets which handles branch miss-prediction in FILL state and ACTIVE state using “CONT FILL” and “pFILL”, respectively. The loop buffer policies sets have listed in table 4.11.

Table 4.11: Loop buffer policies sets

IDLE state		FILL state	ACTIVE state	
Detecting the innermost loop	Detection the innermost loop stored in loop buffer	Branch miss-prediction	Loop buffer miss	Branch miss-prediction
FILL-1	END	GOTO IDLE	aFILL	aIDLE
	START			
FILL-2	END			
	START			
FILL-1	END	CONT FILL		pFILL
	START			
FILL-2	END			
	START			

Figure 4.10 shown R_{IC} for different loop buffer policy sets in different loop

buffer size. Simulation results show that the aggressive policy averagely further decrease R_{IC} by 0.04%, 0.06%, 0.09%, 0.14% (“(FILL-1, END)”, “(FILL-1, START)”, “(FILL-2, END)”, “(FILL-2, START)”), total averagely further decrease R_{IC} by 0.08%. The effects of the aggressive policies are also limited, two.

Figure 4.11 shown R_{LB} for different loop buffer policies sets in different loop buffer size. Simulation results show that the aggressive policy averagely further increase R_{LB} by 4.82%, 4.83%, 1.71%, 1.70% (“(FILL-1, END)”, “(FILL-1, START)”, “(FILL-2, END)”, “(FILL-2, START)”), total averagely further decrease R_{IC} by 3.27%.

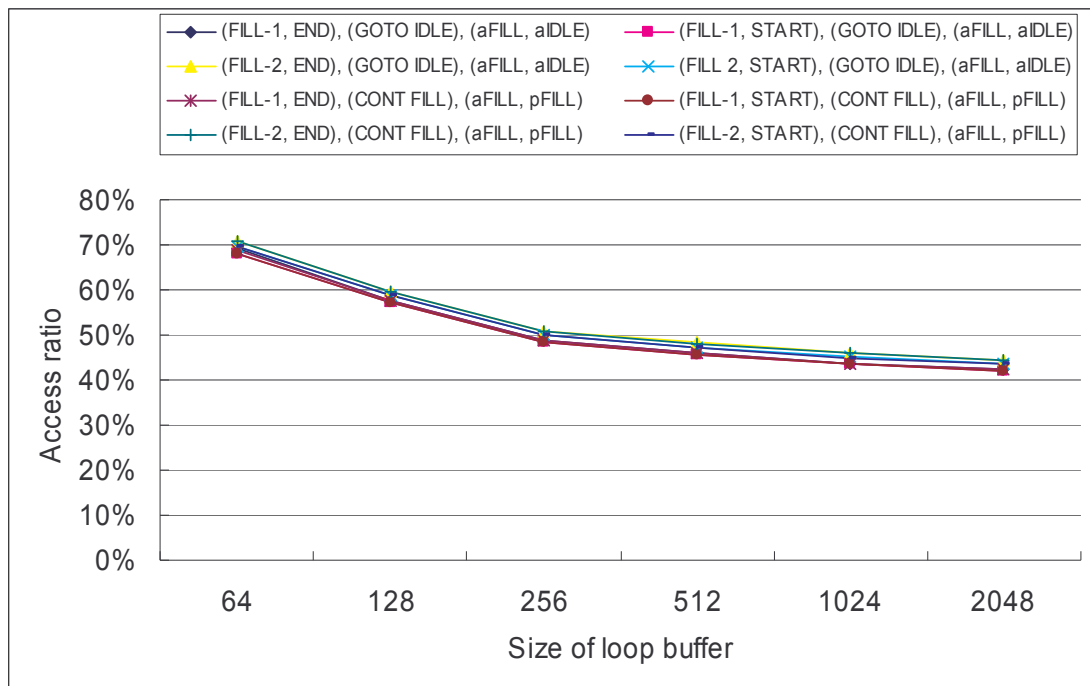


Figure 4.10: Access ratio of IL1 of different loop buffer policies sets

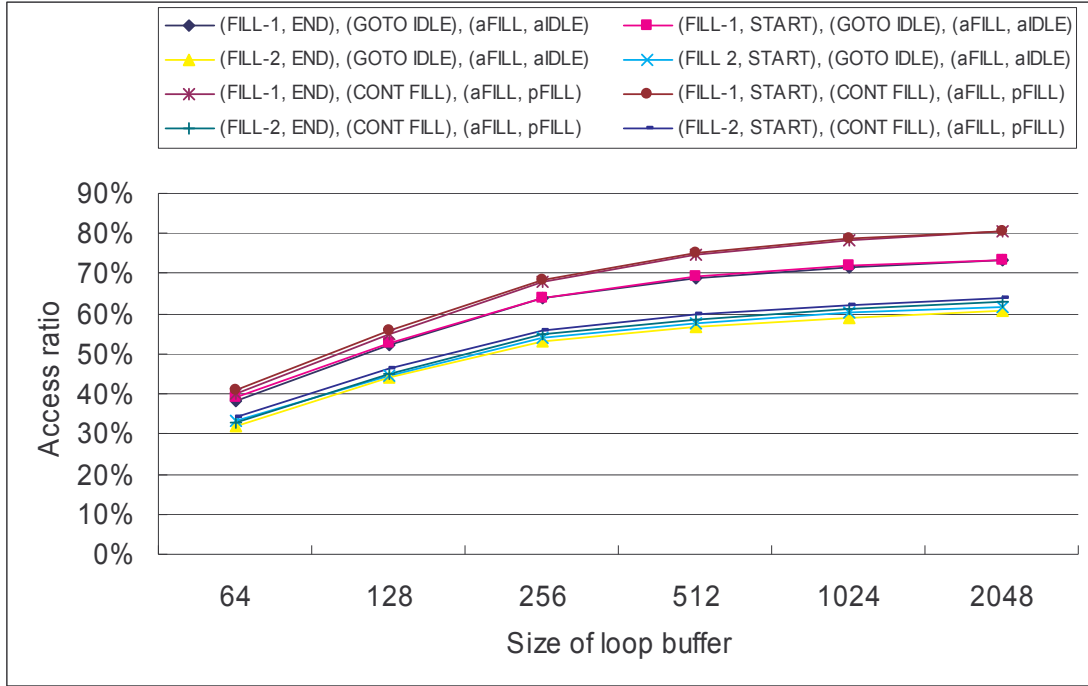


Figure 4.11: Access ratio of loop buffer of different loop buffer policies sets

The hardware of branch miss-prediction handling in FILL state can be reused in ACTIVE state, so that can saving the power consumption of loop buffer controller.

Table 4.12: Ratio of P_{ctrl} of different loop buffer policy sets

	64	128	256	512	1024	2048
(FILL-1, END), (GOTO IDLE), (aFILL, IDLE)	1.91%	1.95%	2.00%	2.08%	2.25%	2.34%
(FILL-1, START), (GOTO IDLE), (aFILL, IDLE)	2.75%	2.83%	2.90%	2.97%	3.14%	3.23%
(FILL-2, END), (GOTO IDLE), (aFILL, IDLE)	2.57%	2.62%	2.65%	2.74%	2.90%	2.98%
(FILL-2, START), (GOTO IDLE), (aFILL, IDLE)	3.41%	3.49%	3.55%	3.63%	3.79%	3.87%
(FILL-1, END), (CONT FILL), (aFILL, pFILL)	3.05%	3.20%	3.40%	3.58%	3.70%	3.87%
(FILL-1, START), (CONT FILL), (aFILL, pFILL)	3.89%	4.07%	4.30%	4.47%	4.59%	4.76%
(FILL-2, END), (CONT FILL), (aFILL, pFILL)	3.71%	3.87%	4.05%	4.23%	4.35%	4.51%
(FILL-2, START), (CONT FILL), (aFILL, pFILL)	4.55%	4.74%	4.95%	5.12%	5.24%	5.41%

The reduction in instruction fetch power of different loop buffer policy sets is shown in the figure 4.12. The best loop buffer policies set of these eight sets is

“(FILL-1, END), (CONT FILL), (aFILL, aIDLE)”, and the power consumption of loop buffer controller of this best policies set is the smallest in all loop buffer policies sets.

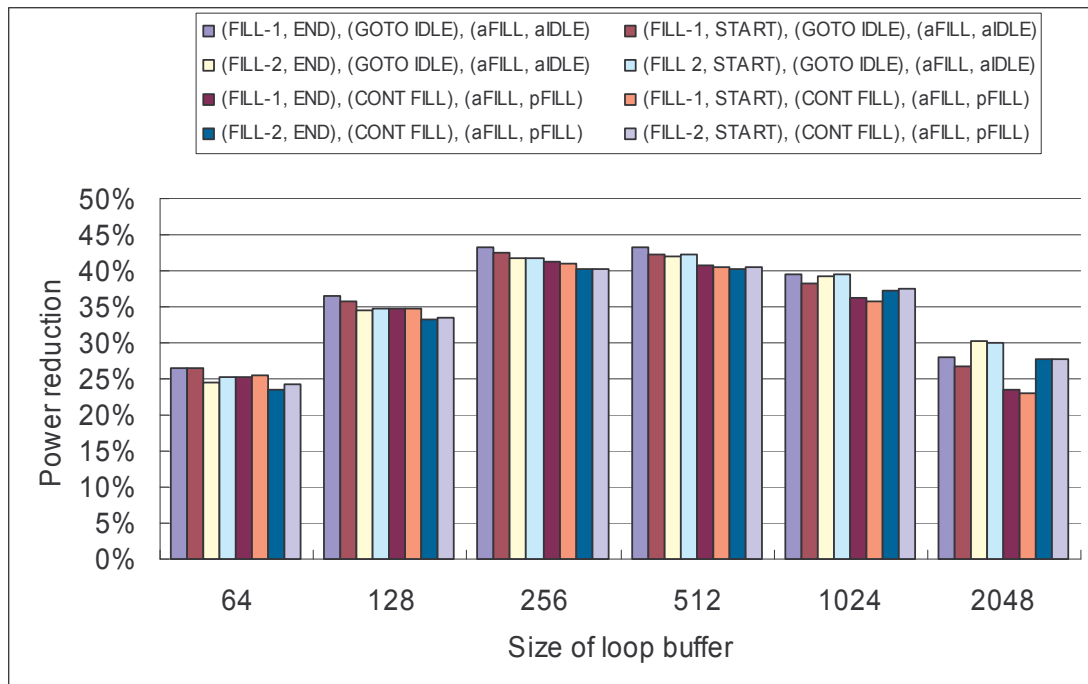


Figure 4.12: Reduction in instruction fetch power of different loop buffer policies sets

4.4 The Penalty of Miss-prediction in Loop Buffer

Policies

In this section, we discuss the penalty of miss-prediction of loop buffer policies. To detect the innermost loop is the only one guesstimate policy in IDLE state. Two possible miss-detection situations are that, we guess it is an innermost loop but it is not, or we miss the innermost loop.

Innermost loop detection policy—“FILL-2” can detect innermost loop correctly, expect the innermost loop only has one iteration; but the other Innermost loop

detection policy—“FILL-1” guesses innermost loop when meets a backward branch once. The penalty of miss-prediction of a innermost loop using “FILL-1” is increasing the invalid loop buffer filled. We can subtract “FILL-2” from “FILL-1” at the execution time in FILL state to get the penalty of miss-prediction of a innermost loop.

The loop buffer policies sets of “FILL-1” and “FILL-2” have listed in table 4.13.

Table 4.13: Loop buffer policies sets

IDLE state		FILL state	ACTIVE state	
Detecting the innermost loop	Detection the innermost loop stored in loop buffer	Branch miss-prediction	Loop buffer miss	Branch miss-prediction
FILL-1	END	GOTO IDLE	aFILL	aIDLE
FILL-2				

Figure 4.13 shown that penalty of miss-prediction of innermost loop detection averagely is 8.13%. It means that “FILL-1” averagely further increase loop buffer filled ratio by 8.13%. The hit-prediction rate of innermost loop detection is 50.08%.

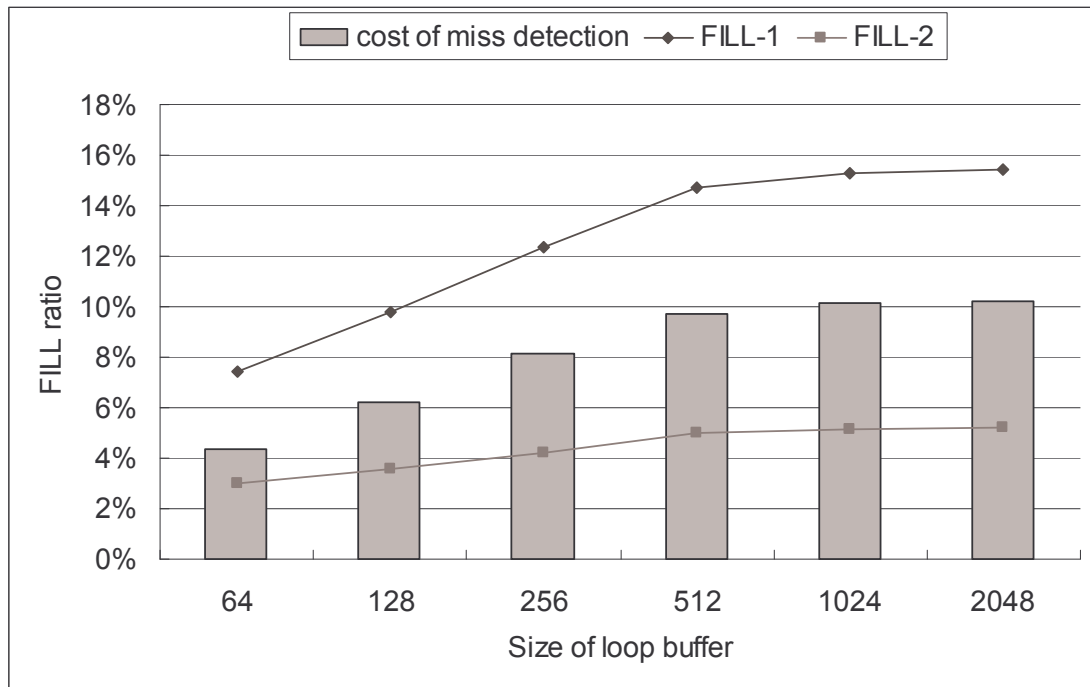


Figure 4.13: The penalty of miss-prediction of innermost loop detection

4.5 The Reasons of Affecting the Loop Buffer

Efficiency



In this section, we analysis the reasons of affecting instruction fetch power reduction using innermost loop buffer. We can divide these reasons into three parts: the nature of benchmark, the architecture of processor, and the optimal of compiler. The nature of benchmark is the most affecting part amount these three parts. We will describe these parts detail.

4.5.1 The Nature of Benchmark

Four characteristics of the nature of benchmark will cause the decrease of the instruction fetch power reduction using innermost loop buffer:

- (1) The benchmark has less execution time in innermost loop.
- (2) The execution paths are changed frequently in this benchmark.

(3) The benchmark has a few of loop iteration.

(4) The size of the innermost loop in benchmark is too large or too small than loop buffer.

We will describe four characteristics detail.

4.5.1.1 The Benchmark Has Less Execution Time in Innermost Loops

We define that the innermost loop ratio as expressed by equation (4):

$$\text{innermost loop ratio} = \frac{\text{execution time in innermost loop}}{\text{total execution time}} * 100\% \quad (4)$$

And a high innermost loop ratio is if one benchmark has more than 90% of innermost loop ratio, a low innermost loop ratio is if one benchmark has less than 60% of innermost loop ratio.

Then we define ACTIVE ratio as expressed by equation (5), and it is equal to $(1-R_{IC})$.



$$\text{ACTIVE ratio} = \frac{\text{time of loop buffer controller in ACTIVE state}}{\text{total execution time}} * 100\% \quad (5)$$

Higher ACTIVE ratio has less R_{IC} and more instruction fetch power reduction. We use ACTIVE ratio to measure the instruction fetch power reduction.

Table 4.14 is the relation between innermost loop ratio to ACTIVE ratio, the loop buffer policies set is “(FILL-1, END), (GOTO IDLE), (aFILL, aIDLE)”, size of loop buffer is 256 byte.

The benchmark which has higher innermost loop ratio has higher ACTIVE ratio, in general. On the other hand, the benchmark which has lower innermost loop ratio has lower ACTIVE ratio. Usually, the benchmark which has lower innermost loop ratio has lower BTB hit rate less than average (94.37%), except rsynth, rijndael-enc and rijndael-asc, these three benchmarks spend a lot of execution time

in the outer loop of a nest loop.

Table 4.14: The relation between innermost loop ratio to ACTIVE ratio

innermost loop ratio > 90%			
benchmark	innermost loop ratio	ACTIVE ratio	hit rate of BTB
bitcount	94.59%	74.16%	95.80%
susan-e	99.95%	96.61%	99.91%
susan-c	99.96%	57.37%	99.96%
djpeg	94.47%	79.22%	96.35%
cjpeg	90.01%	75.17%	94.13%
tiff2bw	98.67%	98.23%	99.64%
tiff2rgba	98.31%	97.53%	99.33%
tiffmedian	99.19%	97.81%	98.02%
dijkstra	90.11%	86.67%	99.10%
blowfish-e	94.21%	68.81%	99.03%
blowfish-d	94.23%	68.82%	99.03%
sha	95.76%	86.96%	95.75%
pcm	99.89%	99.67%	99.93%
adpcm	99.89%	99.67%	99.93%
CRC32	100.00%	99.91%	100.00%
gsm-untoast	96.40%	76.94%	98.71%
innermost loop ratio < 60%			
benchmark	innermost loop ratio	ACTIVE ratio	hit rate of BTB
qsort	56.56%	36.14%	91.89%
typeset	37.94%	15.95%	90.71%
patricia	34.86%	14.95%	89.17%
ghostscript	26.93%	5.59%	90.04%
ispell	52.06%	33.23%	90.95%
rsynth	34.24%	19.81%	96.07%
stringsearch	52.55%	44.18%	92.87%
rijndael-enc	11.96%	10.09%	94.67%
rijndael-asc	11.02%	9.70%	97.50%
FFT	44.14%	18.27%	90.00%
FFT-i	44.21%	18.37%	90.02%
gsm-toast	41.21%	33.33%	83.61%

4.5.1.2 The Execution Paths are changed frequently

The more aggressive policies of execution path changed handling, like loop buffer miss handling in ACTIVE state and branch miss-prediction handling in FILL state and ACTIVE state, will be more effective in the benchmark which has frequent execution paths changed, ideally.

The execution path changed ratio had been defined in previous section 4.3.2.2 as expressed by equation (3).

Then, we define the aggressive loop buffer policies set is “(FILL-1, END), (CONT FILL), (aFILL, pFILL)” and the conservative loop buffer policies set is “(FILL-1, END), (GOTO IDLE), (aFILL, aIDLE)”, size of loop buffer in both of two loop buffer policies set is 256 byte.

And an increase ratio is the ACTIVE ratio in aggressive loop buffer policies set subtracting the ACTIVE ratio in conservative loop buffer policies set dividing the ACTIVE ratio in conservative loop buffer policies set, as expressed by equation (6):

$$\text{increase ratio} = \frac{\text{ACTIVE ratio}(\text{aggressive}) - \text{ACTIVE ratio}(\text{conservative})}{\text{ACTIVE ratio}(\text{conservative})} * 100\% \quad (6)$$

We use the increase ratio to measure the efficiency of the aggressive loop buffer policies set.

Table 4.15 is the relation between execution path changed ratio to the increase ratio. The increase ratio is limited because the execution path changed ratio is too small to affecting the ACTIVE ratio.

Table 4.15: The relation between execution path ratio to the increase ratio

the execution path in innermost loop changed frequently (execution path change rate > 1.90%)				
benchmark	execution path	ACTIVE ratio	ACTIVE ratio	increase ratio

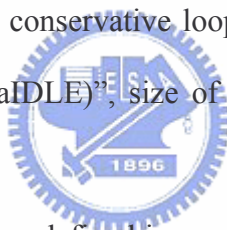
	changed ratio	-aggressive	-conservative	
cjpeg	3.66%	75.17%	75.70%	0.71%
lame	2.84%	52.12%	51.99%	-0.25%
tiffdither	3.04%	66.44%	66.69%	0.38%
typeset	3.69%	15.95%	16.97%	6.39%
patricia	5.11%	14.95%	14.85%	-0.67%
ghostscript	3.09%	5.59%	5.32%	-4.83%
ispell	3.64%	33.23%	33.17%	-0.18%
blowfish-e	3.58%	68.81%	68.86%	0.07%
blowfish-d	3.58%	68.82%	68.87%	0.07%
FFT	6.02%	18.27%	18.27%	0.00%
FFT-i	5.97%	18.37%	18.37%	0.00%
gsm-untoast	8.43%	76.94%	76.97%	0.04%
the execution path in innermost loop changed seldom (execution path change rate $\leq 1.90\%$)				
benchmark	execution path changed ratio	ACTIVE ratio -aggressive	ACTIVE ratio -conservative	increase ratio
bitcount	0.00%	74.16%	74.16%	0.00%
qsort	1.74%	36.14%	36.34%	0.55%
susan-s	0.09%	20.28%	20.46%	0.89%
susan-e	0.01%	96.61%	96.62%	0.01%
susan-c	0.03%	57.37%	57.40%	0.05%
djpeg	1.22%	79.22%	79.50%	0.35%
mad	1.59%	39.50%	39.74%	0.61%
tiff2bw	0.00%	98.23%	98.25%	0.02%
tiff2rgba	0.00%	97.53%	97.54%	0.01%
tiffmedian	1.17%	97.81%	98.01%	0.20%
dijkstra	0.34%	86.67%	86.69%	0.02%
rsynth	0.76%	19.81%	19.84%	0.15%
stringsearch	0.08%	44.18%	43.37%	-1.83%
rijndael-enc	0.00%	10.09%	9.70%	-3.87%
rijndael-asc	0.00%	9.70%	9.77%	0.72%
sha	0.00%	86.96%	87.20%	0.28%
adpcm-pcm	0.04%	99.67%	99.68%	0.01%
adpcm-adpcm	0.04%	99.67%	99.67%	0.00%
CRC32	0.01%	99.91%	99.96%	0.05%
gsm-toast	0.89%	33.33%	36.31%	8.94%

4.5.1.3 The Benchmark Has a Few of Loop Iteration

The more aggressive policy of innermost loop detection will be more effective in a benchmark which has a few of loop iteration than a benchmark which has a lot of loop iteration; but a benchmark which has a few of loop iteration has a few ACTIVE ratio, so that the increase of ACTIVE ratio using a aggressive policy might be limited.

Average of loop iteration in MiBench is 12.41, so the aggressive policy will increase the ACTIVE ratio by 8.33%, maximally.

We define the aggressive loop buffer policies set is “(FILL-1, START), (GOTO FILL), (aFILL, aIDLE)” and the conservative loop buffer policies set is “(FILL-2, END), (GOTO IDLE), (aFILL, aIDLE)”, size of loop buffer in both of two loop buffer policies set is 256 byte.



And an increase ratio has been defined in previous section 4.5.1.2 as expressed by equation (6), and we use the increase ratio to measure the efficiency of the aggressive loop buffer policies set.

Table 4.16 is the relation between execution path changed ratio to the increase ratio. The results show that a benchmark which has a lot of loop iteration has few of increase ratio and a benchmark which has a few of loop iteration has obvious increase ratio usually. It indicates that an aggressive loop detection policy has more effective (i.e. increase ratio) in a benchmark which has a few of loop iteration.

Beside, comparing table 4.14 with table 4.16, we can find that a benchmark which has a lot of loop iteration usually has higher innermost loop ratio, on the other hand, a benchmark which has a few of loop iteration usually has lower innermost loop ratio. It causes that, although an aggressive loop detection policy has more

increase ratio, the ACTIVE ratio of the aggressive loop detection policy is increase limited.

Table 4.16: the relation between execution path ratio to the increase ratio

the number of loop iteration ≤ 20				
benchmark	the number of loop iteration	ACTIVE ratio -aggressive	ACTIVE ratio -conservative	increase ratio
bitcount	7.054678	74.16%	74.16%	0.00%
qsort	4.795691	28.00%	36.63%	30.82%
susan-s	2.536506	11.28%	20.58%	82.45%
lame	5.912732	48.76%	52.15%	6.95%
mad	4.408253	34.85%	39.89%	14.46%
typeset	4.737559	14.03%	18.00%	28.30%
patricia	4.50309	12.04%	15.63%	29.82%
ghostscript	2.759617	3.12%	5.80%	85.90%
ispell	5.363204	26.39%	33.94%	28.61%
rsynth	3.82898	19.68%	19.81%	0.66%
stringsearch	15.40423	41.73%	44.45%	6.52%
rijndael-enc	15.08819	9.37%	10.15%	8.32%
rijndael-asc	15.0882	9.02%	9.77%	8.31%
FFT	3.156288	13.53%	18.72%	38.36%
FFT-i	3.166977	13.64%	18.82%	37.98%
gsm-toast	11.13197	30.38%	33.52%	10.34%
gsm-untoast	9.642393	76.39%	76.94%	0.72%
the number of loop iteration > 20				
benchmark	the number of loop iteration	ACTIVE ratio -aggressive	ACTIVE ratio -conservative	increase ratio
susan-e	365.208	96.59%	96.61%	0.02%
susan-c	353.5814	57.37%	57.37%	0.00%
tiff2bw	457.1911	98.02%	98.25%	0.23%
tiff2rgba	291.391	97.23%	97.54%	0.32%
tiffdither	104.3369	66.40%	66.44%	0.06%
tiffmedian	218.4292	97.72%	97.83%	0.11%
dijkstra	30.4933	85.98%	86.69%	0.83%
blowfish-e	39.10312	67.01%	68.84%	2.73%

blowfish-d	39.10303	67.02%	68.85%	2.73%
sha	24.59139	83.27%	87.21%	4.73%
pcm	743.1287	99.67%	99.68%	0.01%
adpcm	716.0261	99.67%	99.67%	0.00%
CRC32	1977128	99.91%	99.91%	0.00%

4.5.1.4 The Size of Innermost Loops is too large or too small than Loop

Buffer

When the size of innermost loops is larger than the size of loop buffer, instructions in the innermost loops can not be fetched from loop buffer completely, so that the instruction fetch power reduction is limited. On the other hand, the size of innermost loops is smaller than the size of loop buffer; the access power of loop buffer is increasing, so that the instruction fetch power reduction is limited, too.

Therefore, the choice of loop buffer size must be large enough to store most part of innermost loops to reduce R_{IC} . At last, the size of loop buffer trades off between power reduction by reducing R_{IC} with the access power of loop buffer increasing and the access power of loop buffer controller increasing.

The average size of innermost loops is 33.3 instructions, and about 71.82% of innermost loop's size is smaller or equal than average size of innermost loops.

We take two benchmarks for example —susan-c (average size of innermost loop is 220.715 instructions), tiff2bw (average size of innermost loop is 12.961 instructions), which has the biggest and the smallest size of innermost loops in the set of benchmarks which have higher innermost loop ratio. And the loop buffer policies set is “(FILL 1,end), (go to IDLE), (aFILL, IDLE)”, the size of loop buffer is 64, 128, 256, 512, 1024, 2048 bytes.

Figure 4.14 is the ratio of different size of innermost loops in tiff2bw, 87.11% of innermost loop's size is 12 instructions, 12.26% of innermost loops' size is 20 instructions.

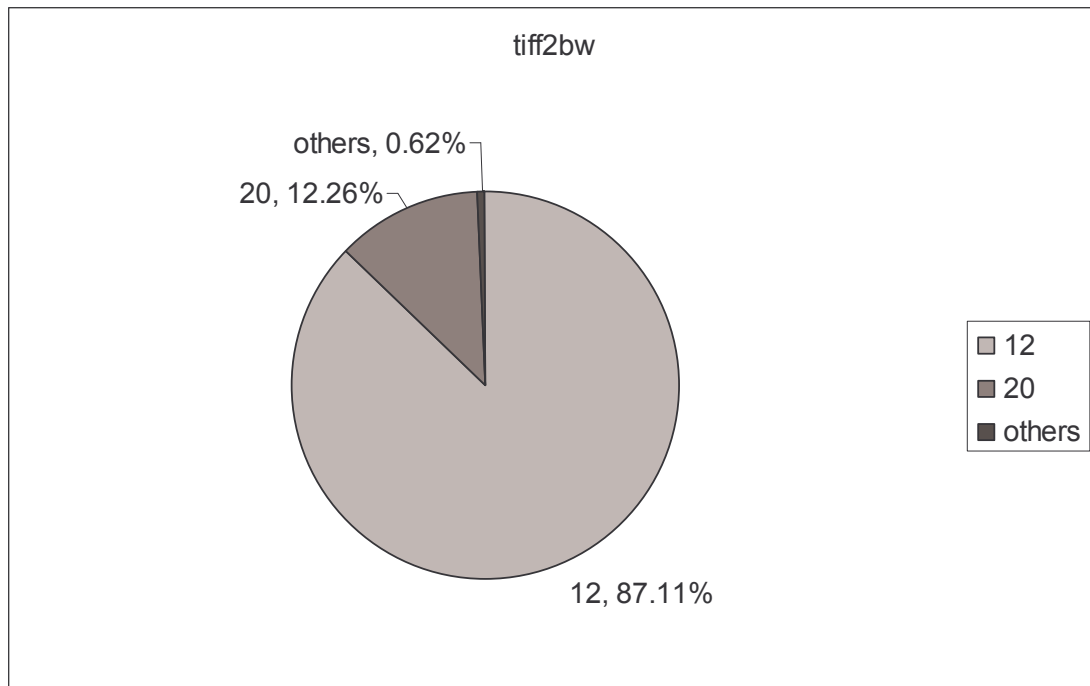


Figure 4.14: The ratio of different sizes (instructions) of innermost loop—
tiff2bw

Figure 4.15 is the ratio of different size of innermost loops in susan-c, 46.57% of innermost loop's size is 14 instructions, 6.20% of innermost loop's size is 16 instructions, 46.57% of innermost loops' size is 456 instructions.

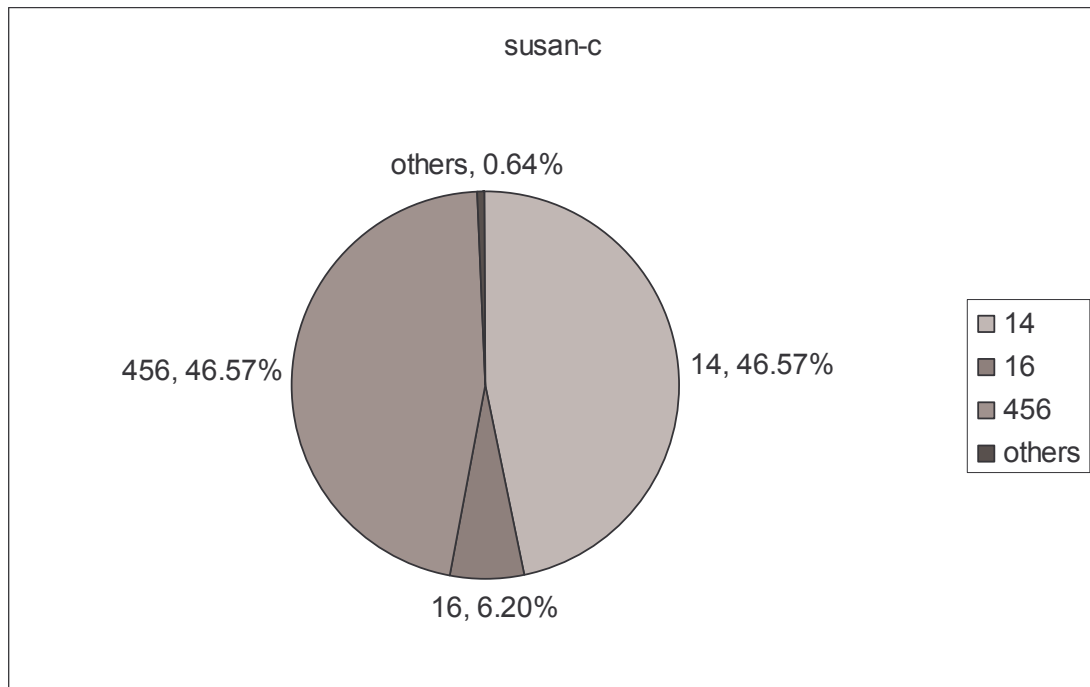


Figure 4.15: The ratio of different sizes (instructions) of innermost

loop—susan-c

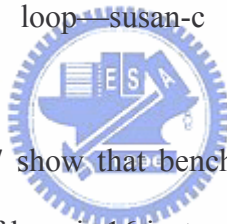


Figure 4.16 and figure 4.17 show that benchmark—tiff2bw can store all the innermost loops which the size of loop is 16 instructions and part of innermost loops which the size of loop is 20 instructions in the loop buffer which the size of loop buffer is 64 bytes (equal to 16 instructions). In the loop buffer which the size of loop buffer is 128 bytes (equal to 32 instructions), it can store all innermost loops, so that R_{IC} of the loop buffer which the size of loop buffer is 128 bytes is less than the loop buffer which the size of loop buffer is 64 bytes, and R_{IC} will not decrease if the size of loop buffer is larger than 128 bytes.

Benchmark—susan-c can store all the innermost loops which the size of loop is 14 and 16 instructions and small part of innermost loops which the size of loop is 456 instructions in the loop buffer which the size of loop buffer is 64 bytes (equal to 16 instructions). As the increasing of size of loop buffer, the ratio of innermost loops with 456 instructions can be accessed from loop buffer is increasing. In the loop

buffer which the size of loop buffer is 2K bytes (equal to 512 instructions), it can store all innermost loops, so that the R_{IC} in this case is minimum.

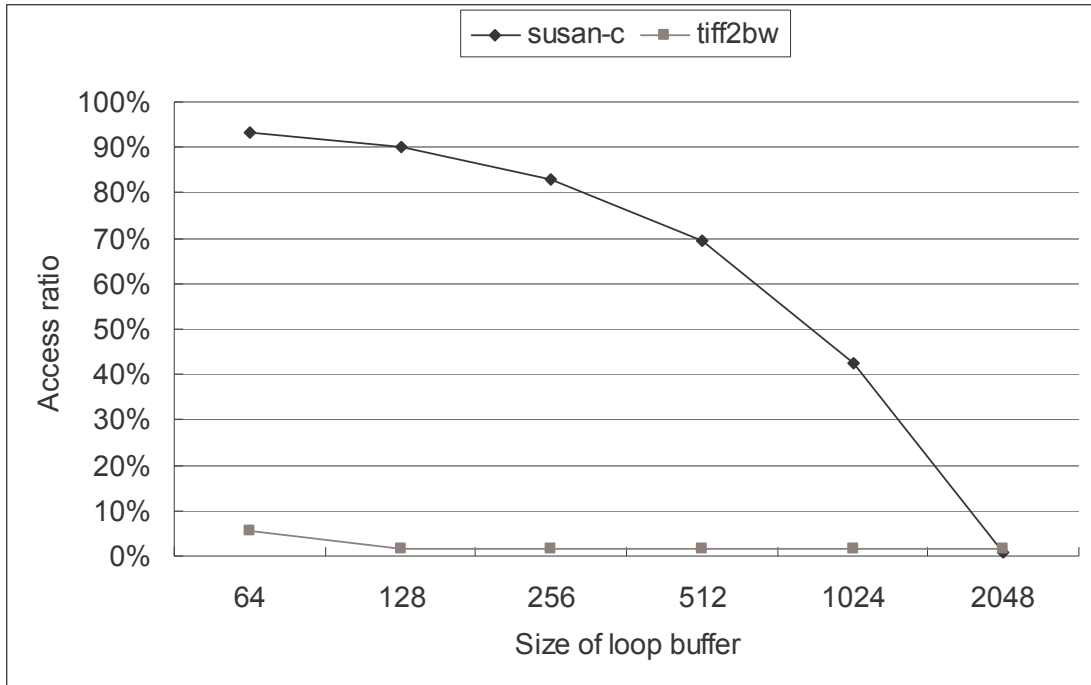


Figure 4.16: Access ratio of IL1 in different benchmarks

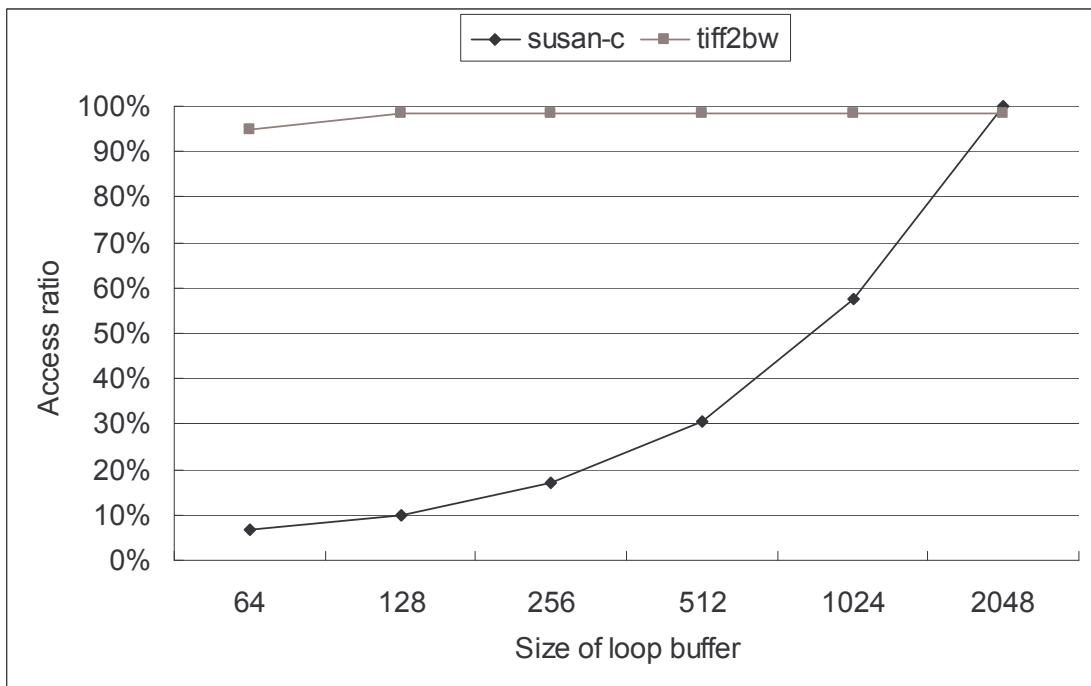


Figure 4.17: Access ratio of loop buffer in different benchmarks

The reduction in instruction fetch power of different loop buffer policy sets is shown in the figure 4.18. In tiff2bw, there is maximum instruction fetch power reduction when the size of loop buffer is 128 bytes; as the increase of size of loop buffer, the instruction fetch power reduction is decreasing because of the increasing of the access power of loop buffer. In susan-c, there is maximum instruction fetch power reduction when the size of loop buffer is 2K bytes; as the increasing of size of loop buffer, the instruction fetch power reduction is increasing because of the increasing of R_{IC} .

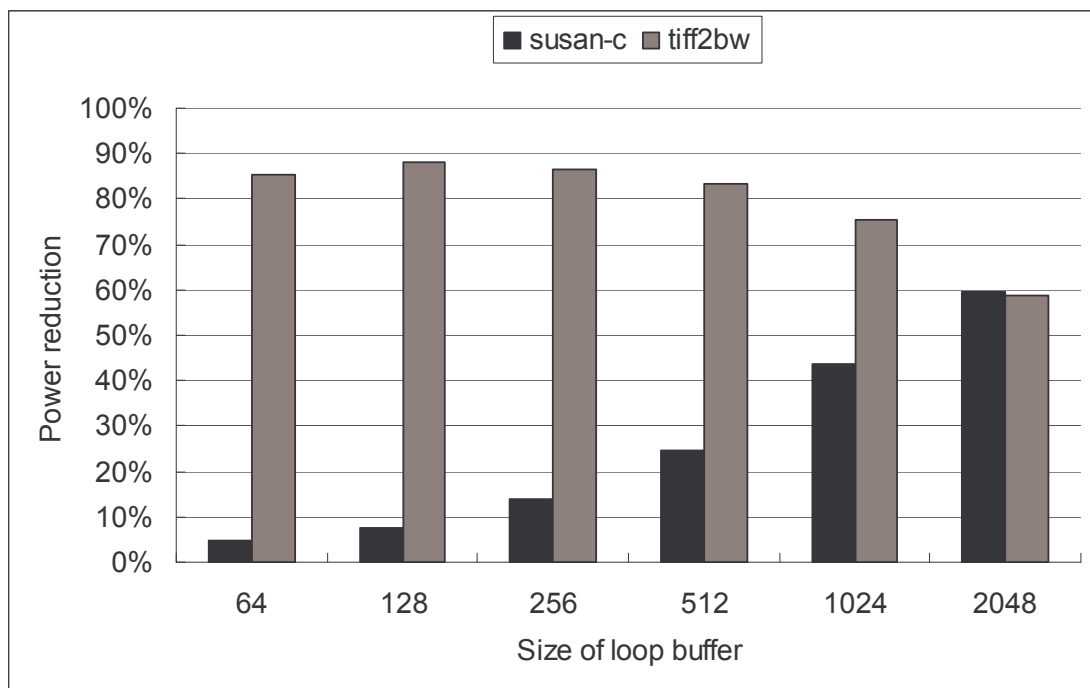


Figure 4.18: Reduction in instruction fetch power in different benchmarks

Figure 4.19 is the percentage of innermost loops with different size of innermost loops, and we can find the trend in this figure is likely to the trend of $(1-R_{IC})$

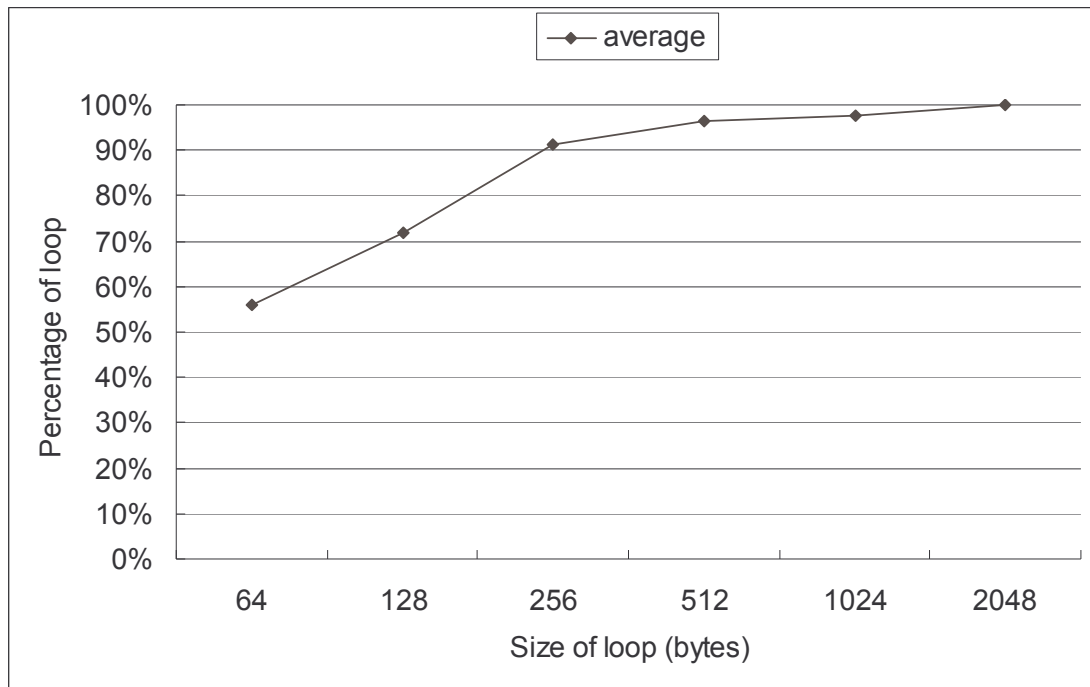


Figure 4.19: The percentage of innermost loops with different sizes of innermost loops

4.5.2 The Architecture of Processor

In the architecture of processor, branch prediction unit is only one component to affect the performance (i.e. instruction fetch power reduction) of our loop buffer.

Our policies of loop buffer are based on the branch predictor using saturating counter (for example, two-bit predictor). Other kinds of predictor like one-bit predictor or correlating predictor may decrease the instruction fetch power reduction.

In the case of using one-bit predictor, the miss rate has increased, so that the ACTIVE ratio decreases. Because there is no strong or weak state, the branch miss-prediction handling only has “aIDLE” or “aFILL” and loop buffer miss handling only has “aFILL”.

In the case of using correlating predictor, the accuracy of branch prediction is higher than using two-bit predictor, but it may cause more loop buffer miss. We

propose a possible method to solve the situation by adding a one-bit saturating counter in each BTB entry to count the frequent of loop buffer miss. This one-bit saturating counter will increase by one if loop buffer miss happened; otherwise this one-bit saturating counter will decrease by one; and this one-bit saturating counter will clean to zero when FD-bit changed. Hence, loop buffer stores new execution path only when loop buffer miss happed twice continued; otherwise, loop buffer will not store new execution path and just go to IDLE state from ACTIVE state.

However, in MiBench, the execution path changed is seldom. As the affecting of execution path changed handling policies is limited, the affecting of branch prediction policy is limited to be obvious.

4.5.3 The Optimal of Compiler

There are some optimal policies of compiler might affect the efficiency of loop buffer, for example:



- (1) Loop fusion. It is to fuse some adjacent loops into one loop to reduce loop overhead and improve run-time performance.
- (2) Loop fission. It splits a big loop into some small loops, as a reverse action of loop fusion.
- (3) Loop unrolling. It restructures loop constructs to expose possibilities for parallelism.
- (4) Loop peeling. It is a form of loop unrolling where the first iterations from a loop are unrolled. It removes i iterations from the beginning of a loop and adds i copies of the loop body directly before the start of the loop.
- (5) Loop interchange. It changes an outer loop for an inner loop
- (6) Subroutine (in loop body) inlining. The subroutine call in loop body will be replaced by this subroutine itself.

In (1), (3), these optimal policies might increase the size of loop body, we need to use a bigger loop buffer, and decreasing the efficiency of loop buffer (the instruction fetch power reduction). In (2), (4), it might decrease the size of loop body, so we can use a smaller loop buffer; the efficiency of loop buffer might increase. In (5), it might increase or decrease the efficiency of loop buffer. In (6), although it might increase code size of loop body, subroutine inlining can slightly decrease the size of loop buffer for storing this innermost loop with subroutine call, because subroutine inline will remove the subroutine call and subroutine return from loop body.

4.6. Simulation Results and Evaluation

In the following sections, we will show our simulations with other strategies of loop buffer management which had proposed in related works and the analysis. In Section 4.3.1, we describe four strategies of loop buffer as comparison. In Section 4.3.2, we present the access ratio of loop buffer and lower level memory and the instruction fetch power reduction for these strategies of loop buffer management.

4.6.1 Strategies of Loop Buffer management as Comparison

Four loop buffer designs, are only capable of innermost loop without forward branch, [15], [17], and our approach with the best policies set, are evaluated.

DLC, 2-way DLC and cluster LC is loop buffer design only capable of innermost loop without forward branch, [15] and [17], respectively.

Except for innermost loop without forward branch, 2-way DLC is also capable of storing instructions within innermost loops from first instruction of the innermost loop until a forward branch or a subroutine call. In both DLC and 2-way DLC, loop

buffer controller only starts to fill instruction after a same backward branch is taken twice successively.

Cluster LC can store a sequence of instructions which can be any kind of loop, included outer loop of a nested loop, and the loop detection is using a special instruction “lbon n” to indicate. Cluster loop cache can not store the loop and the subroutine called by the stored loop body at the same time, because that the addresses of the stored loop body and the subroutine call are usually discontinued. A possible disadvantage is that the delay overhead of the controller of cluster loop cache may be too large to fit the cycle time, so that it may increase the number of instruction fetch cycle and reduce the performance of processor.

FSLB (forward and subroutine loop buffer) is our best proposed design which has the maximum instruction fetch power reduction in our proposed design discussed in previous section 4.3.3.4. We list the policies set of FSLB in table 4.17.



Table 4.17: The best policies set of FSLB

IDLE state		FILL state	ACTIVE state	
Detecting the innermost loop	Detection the innermost loop stored in loop buffer	Branch miss-prediction	Loop buffer miss	Branch miss-prediction
FILL-1	END	GOTO IDLE	aFILL	aIDLE

4.6.2 Simulation Results of Different Loop buffer Designs

Figure 4.20 and 4.21 shows R_{IC} and R_{LB} for the different loop buffer designs in different size. We examined the size of loop buffer from 64 to 2048 bytes, i.e. 16 to 512 instructions. Simulation results show that FSLB averagely further decrease R_{IC} by 26.43% / 16.72%, respectively (DLC / 2-way DLC). We only compare FSLB with DLC and 2-way DLC, because that loop buffer in these designs can only store

innermost loop. Note that since both loop buffer and IL1 are accessed during filling instructions into loop buffer, the sum of R_{LB} and R_{IC} would exceed over 100% in each designs.

To reduce instruction fetch power, we hope that most instructions can be fetched from loop buffer, i.e. reduce R_{IC} and increase R_{LB} . In figure 4.20 and 4.21, our approach has significant improvement in R_{IC} and R_{LB} . Since larger loop buffer has higher opportunity to store more instructions, R_{IC} and R_{LB} would decrease and increase with the size of loop buffer. However, larger loop buffer may cause higher power consumption in P_{LB} .

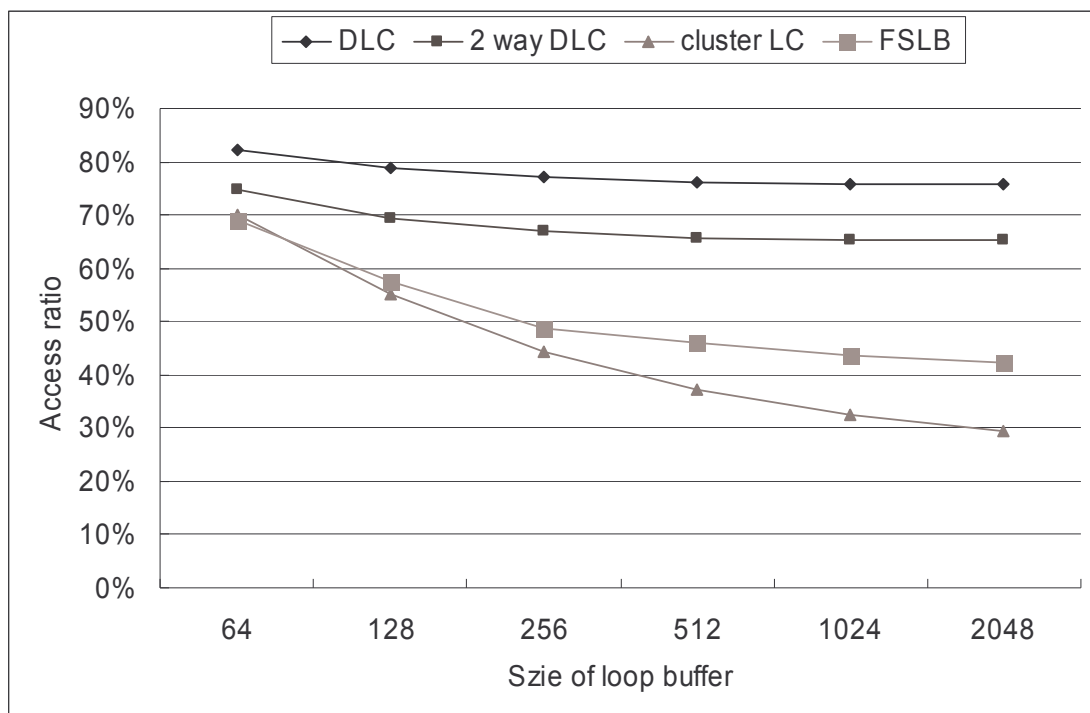


Figure 4.20: Access ratio of IL1 of different designs

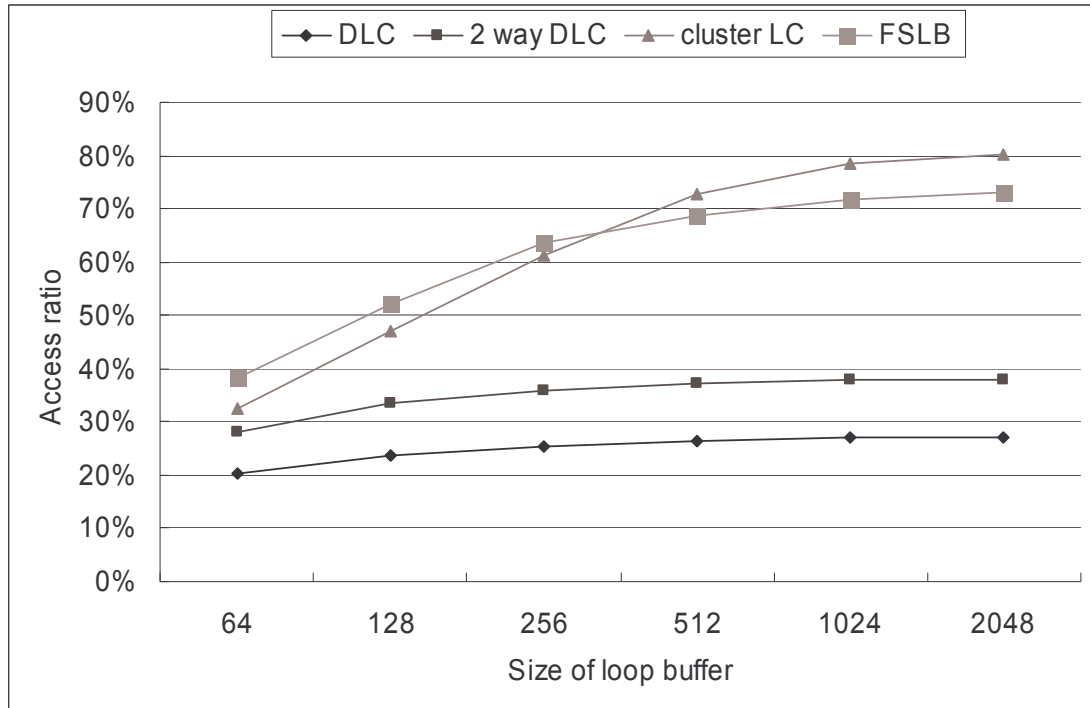


Figure 4.21: Access ratio of loop buffer of different designs



Figure 4.22 shows ACTIVE ratio in different designs, respectively, and had been defined in previous section 4.5.1.1 as expressed by equation (5).

Profiling results in MiBench show that 32.9%, 26.56% and 11.76% of execution time spends on innermost loop without forward branch, with forward branch(s), and with subroutine(s) including no loop, respectively. Using profiling result and ACTIVE ratio, we can easily explain the several interesting points in R_{IC} .

First, the optimal ACTIVE ratio of DLC, 2-way DLC, FSLB should be 32.9%, 58.46% and 71.22%, respectively. If we use compiler to assist in innermost loop detection, the optimal ACTIVE ratio could be achieved.

Second, the maximal ACTIVE ratio of FSLB is 57.70%, respectively. This result is not very close to optimal solution (71.22%), the reasons had been discussed in previous section 4.5.1. The main reasons are some benchmarks have less execution time in innermost loops and have few loop iterations. Since loop buffer

controller hardly catches their innermost loop in several benchmarks, average ACTIVE ratio is not very close to optimal one.

Third, 2-way DLC maximally has 10.48% improvement than DLC in ACTIVE ratio. Since innermost loop with forward branch(es) or subroutine call(s) averagely consist of 2.5 forward branches or subroutine call, each innermost with forward branch(es) or subroutine call(s) can be divided into 3.5 segments. 38.32% (sum of innermost loop with forward branch(s) and no-loop-inside subroutine(s)) divided by 3.5 equals to 10.95%. This value is very close to the improvement ratio using 2-way DLC.

Fourth, cluster LC has two advantages better than FSLB for increasing ACTIVE ratio; first, cluster LC can read loop buffer from the first loop iteration using a special instruction “lbon n” to indicate the start and end address of instructions sequence which stored in loop buffer; Second, the instructions sequence which stored in cluster LC can be any kind of loop, included outer loop of a nested loop. But cluster LC also has one disadvantage worse than FSLB for reducing ACTIVE ratio. Cluster loop cache can not store the loop and the subroutine called by the stored loop body at the same time, because that the addresses of the stored loop body and the subroutine call are usually discontinued. When the size of loop buffer is less or equal than 128 bytes, the loops stored in cluster LC are all innermost loop, so that the ACTIVE ratio of cluster LC is just a little larger than FSLB. As the size of loop buffer increasing, more outer loop of a nested loop can be stored in cluster LC, so that the ACTIVE ratio of cluster LC is much larger than FSLB.

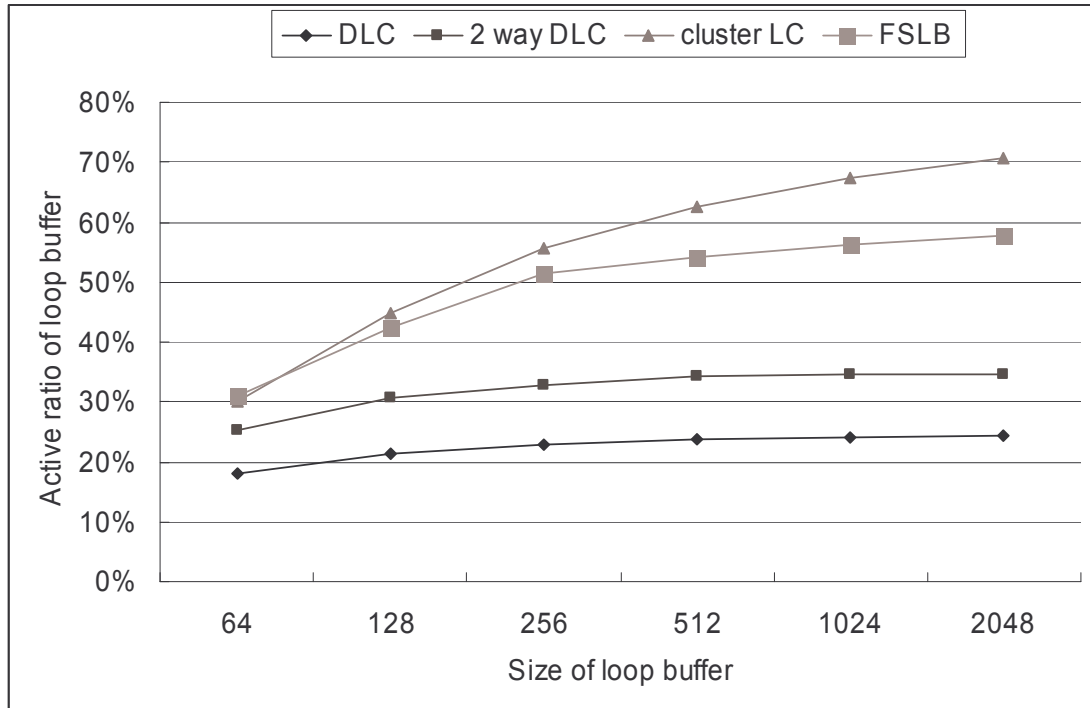


Figure 4.22: Loop buffer ACTIVE ratio



The loop buffer controller of each design is synthesized using Synopsys design tools in 0.18 μm TSMC CMOS technology. The power consumptions of loop buffer and IL1 are calculated by Wattch power modeling tool.

The ratios of P_{LB} and P_{IC} , P_{ctrl} and P_{IC} are shown in table 4.18. FD-bit is a 1-bit field which is attached to each entry of BTB, and the power consumption of FD-bit is 0.13% of power consumption of L1 instruction cache (P_{FD-bit}/P_{IC}). Power consumption of cluster LC controller is much larger than others, because that cluster LC controller includes two 32-bit subtractions and at least extra two 32-bit registers, but we do not include the power consumption of pipeline stall when cluster LC is in FILL state and the power consumption due to cycle penalty of cluster LC fetch.

Table 4.18: Ratio of P_{LB} and P_{ctrl}

Size of loop buffer	64B	128B	256B	512B	1KB	2KB

$(P_{LB}/ P_{IC}) * 100\%$	8.26%	8.92%	10.35%	13.56%	21.40%	38.34%
$(P_{ctrl-DLC\ w/o\ FB}/ P_{IC}) * 100\%$	0.39%	0.43%	0.49%	0.51%	0.53%	0.54%
$(P_{ctrl-2\ way\ DLC}/ P_{IC}) * 100\%$	4.82%	4.91%	4.98%	5.05%	5.23%	5.31%
$(P_{ctrl-cluster\ LC}/ P_{IC}) * 100\%$	10.74%	10.88%	11.07%	11.20%	11.32%	11.48%
$(P_{ctrl-FBLB}/ P_{IC}) * 100\%$	1.91%	1.95%	2.00%	2.08%	2.25%	2.34%

The reduction in instruction fetch power of different designs is shown in the figure 4.23. For each design, 256B or 512B has the maximum power reduction. According to figure 4.1 and 4.2, increasing loop buffer size can improve R_{IC} and R_{LB} but also increases P_{LB} . Hence, larger loop buffer may be not beneficial for P_{IF} . Compared to capable of innermost loop without forward branch and [15], FSLB has significantly improvement in instruction fetch power. And cluster LC has the maximum instruction fetch power reduction when the size of loop buffer is 512 bytes.

If we insert a special instruction in compiler phase, to indicate the stored instructions sequence like “lbon n” in cluster LC proposed in [17], we can also store outer loop, too; so that the ACTIVE ratio of FSLB can be increased and increasing the instruction fetch power reduction large than cluster LC, very possibility. We leave the issue to future work.

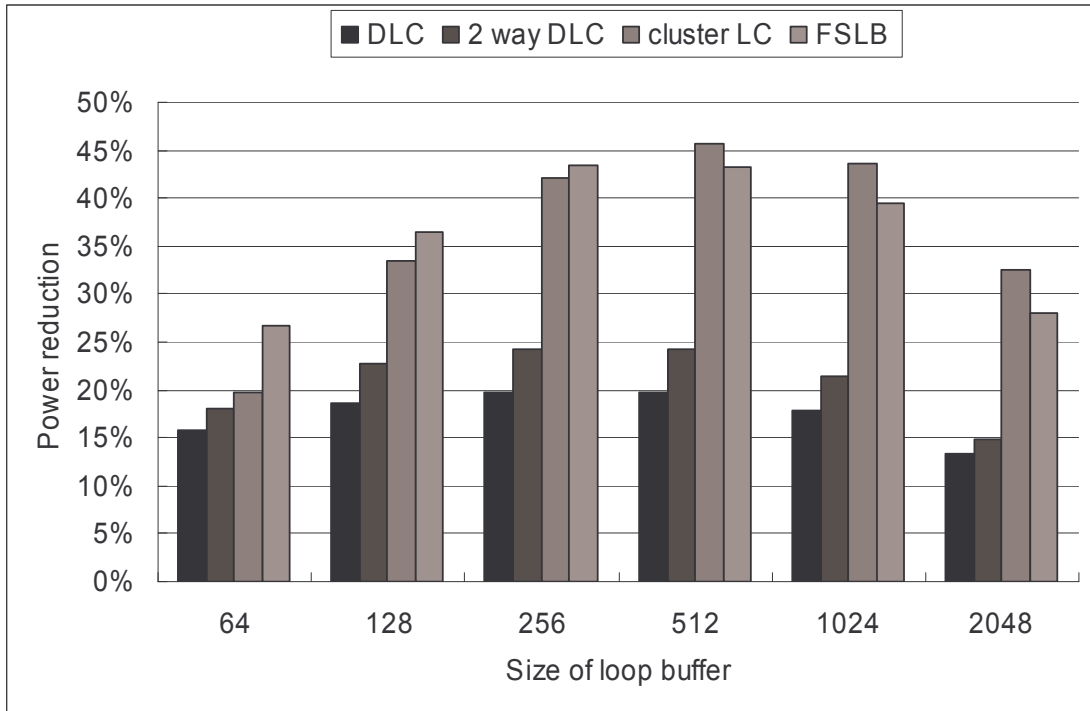


Figure 4.23: Reduction in instruction fetch power of different designs



Chapter 5

Conclusions and Future Work

Adding our design to the instruction memory hierarchy can significantly reduce instruction fetch power. Previous designs, although simple, fail to capture a large percentage of innermost loops, make power saving results unsatisfactory. We propose to correct this shortcoming by using BTB to assist loop buffer in storing innermost loops. The benefits of our design are:

(1) significant increase in loop buffer utilization and hence instruction fetch power saving;

(2) almost negligible hardware overhead; and

(3) no need for extra branch instruction or compiler to assist loop buffer controller in innermost loop detection.

Experiment results show that our design can further reduce up to 13.66% of instruction fetch power, and only introduce 1.8% of hardware overhead in BTB.

We can improve the instruction fetch power reduction by inserting a special instruction like “lbon n” in cluster LC proposed in [17] to store a sequence of instructions which can be any kind of loop, include outer loop. And it is possible to reduce instruction fetch power more than cluster LC.

Increasing utilization of loop buffer has another significance: it provides better changes for the lower level instruction memories to conserve static power leakage. With the advances in deep-sub-micro semiconductor processing, static power consumption is becoming dominant. Several researches show that the static power currently accounts for about 15%-20% of the total power consumption in the 130 nano process, and will exceed 50% in the 65 nano process. Although we only

address reduction in instruction fetch power, i.e. dynamic power, in this work, it will be an interesting if we study the static power effect of our design. We are already working on this topic.



Reference

- [1] Texas Instruments Inc., TMS320C6000 Power Consumption Summary, <http://www.ti.com>, Nov. 1999.
- [2] L. Benini, D. Bruni, M. Chinosi, C. Silvano, and V. Zaccaria, “A Power Modeling and Estimation Framework for VLIW-Based Embedded System,” *ST J. System Research*, vol. 3, pp. 110-118, Apr. 2002.
- [3] Aghaghiri, Y., Fallah, F., and Pedram, M. 2001. Irredundant address bus encoding for low power. *International Symposium on Low Power Electronics and Design*, 82–87.
- [4] Benini, L., Demicheli, G., Macii, E., Sciuto, D., and Silvano, C. 1998. Address bus encoding techniques for system-level power optimization. In *Design Automation and Test in Europe*.
- [5] Stan, M. R. and Burleson, W. P. 1995. Bus-invert coding for low-power I/O. *IEEE Transactions on Very Large Scale Integration Systems* 3, 1, 49–58.
- [6] Govindarajan, S. C., Ramaswamy, G., Andmehendale, M. 2001. Area and power reduction of embedded DSP systems using instruction compression and re-configurable encoding. In *International Conference on Computer Aided Design*.
- [7] Ishihara, Y. and Yasuura, H. 2000. A power reduction technique with object code merging for application specific embedded processors. In *Design Automation and Test in Europe*.
- [8] Bajwa, R. S., Hiraki, M., Kojima, H., Gorney, D., Nitta, K., Shridhar, A., Seki, K., and Sasaki, K. 1997. *Instruction buffering to reduce power in processors for signal processing. IEEE Transactions on VLSI Systems*, 417–424.
- [9] Albonesi, D. H. 2000. Selective cache ways: On-demand cache resource allocation. *Journal of Instruction Level Parallelism*.

- [10] Malik, A., Moyer, B., and Cermak, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *International Symposium on Low Power Electronics and Design*.
- [11] Hasegawa, A., Kawasaki, I., Yamada, K., Yoshioka, S., Kawasaki, S., and Biswas, P. 1995. SH3 high codes density, low power. *IEEE Micro*.
- [12] Kin, J., Gupta, M., and Mangione-Smith, W. 1997. The filter cache: An energy efficient memory structure. In *International Symposium on Microarchitecture*, 184–193.
- [13] Bellas, N., Hajj, I., Polychronopoulos, C., and Stamoulis, G. 1999. Energy and performance improvements in microprocessor design using a loop cache. In *International Conference on Computer Design*, 378–383.
- [14] L. Lee, B. Moyer and J. Arends, “Low-Cost Embedded Program Loop Caching – Revisited,” University of Michigan Technical Report CSE-TR-411-99, 1999.
- [15] T. Anderson and S. Agarwala, “Effective hardware-based two-way loop cache for high performance low power processors,” *International Conference on Computer Design*, 2000.
- [16] A. Gordon-Ross, S. Cotterell and F. Vahid, “Tiny Instruction Caches For Low Power Embedded Systems,” *ACM Transactions on Embedded Computing Systems*, 2003.
- [17] M. Jayapala, F. Barat, T. V. Aa, F. Catthoor, H. Corporaal and G. Deconinck, “Clustered Loop Buffer Organization for Low Energy VLIW Embedded Processors,” *IEEE Transactions on Computers*, 2005.
- [18] T. Austin, E. Larson and D. Ernst, “SimpleScalar: an Infrastructure for Computer System Modeling,” *IEEE Computer*, 2002.
- [19] D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, “MiBench: A free,

commercially representative embedded benchmark suite,” *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[20] D. Brooks, V. Tiwari and M. Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations,” *ISCA*.



Appendix

A-1 Profiling of Innermost Loop

Type A. w/o FB: innermost loop without forward branch and without subroutine

Type B. w FB: innermost loop with forward branch(es) but without subroutine

Type C. w FN: innermost loop with subroutine(s) consisting of no loop

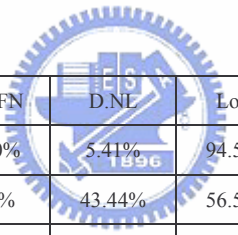
Type D. NL: not in an innermost loop

(w FB & FN): (Type B) and (Type C)

Loop: (Type A) and (Type B) and (Type C)

Loop size: average size of innermost loop

Hit_rate: hit rate of 512-set, 4-way BTB



	A. w/o FB	B. w FB	C. w FN	D.NL	Loop	(w FB & FN)	Loop size	Hit_rate
bitcount	74.30%	4.09%	16.20%	5.41%	94.59%	20.29%	12.58	95.80%
qsort	43.88%	11.24%	1.44%	43.44%	56.56%	12.68%	12.20	91.89%
susan-s	64.17%	0.01%	0.52%	35.30%	64.70%	0.53%	12.63	77.96%
susan-e	37.19%	62.42%	0.35%	0.05%	99.95%	62.76%	89.42	99.91%
susan-c	0.47%	99.14%	0.35%	0.04%	99.96%	99.49%	220.72	99.96%
djpeg	53.31%	41.00%	0.16%	5.53%	94.47%	41.16%	29.71	96.35%
cjpeg	52.78%	33.18%	4.05%	9.99%	90.01%	37.23%	19.96	94.13%
lame	58.51%	19.09%	1.66%	20.74%	79.26%	20.75%	60.92	90.92%
mad	44.23%	11.05%	12.99%	31.73%	68.27%	24.04%	29.19	83.38%
tiff2bw	98.65%	0.01%	0.01%	1.33%	98.67%	0.02%	12.96	99.64%
tiff2rgba	98.14%	0.02%	0.15%	1.69%	98.31%	0.17%	29.19	99.33%
tiffdither	7.93%	62.50%	3.00%	26.58%	73.42%	65.50%	29.25	94.61%
tiffmedian	96.16%	3.04%	0.00%	0.81%	99.19%	3.04%	23.31	98.02%
typeset	7.55%	12.46%	17.93%	62.06%	37.94%	30.39%	11.98	90.71%
dijkstra	14.83%	75.26%	0.02%	9.89%	90.11%	75.28%	14.30	99.10%
patricia	20.92%	11.16%	2.78%	65.14%	34.86%	13.94%	11.95	89.17%
ghostscript	8.62%	17.61%	0.70%	73.07%	26.93%	18.31%	17.23	90.04%
ispell	29.67%	15.96%	6.43%	47.94%	52.06%	22.38%	8.00	90.95%

rsynth	4.11%	27.13%	3.00%	65.76%	34.24%	30.13%	28.93	96.07%
stringsearch	47.35%	5.16%	0.04%	47.45%	52.55%	5.20%	4.60	92.87%
blowfish-e	0.00%	8.13%	86.09%	5.79%	94.21%	94.21%	55.18	99.03%
blowfish-d	0.00%	8.13%	86.10%	5.77%	94.23%	94.23%	55.31	99.03%
rijndael-enc	11.50%	0.31%	0.15%	88.04%	11.96%	0.46%	5.56	94.67%
rijndael-asc	11.02%	0.00%	0.00%	88.98%	11.02%	0.00%	5.50	97.50%
sha	95.76%	0.00%	0.00%	4.24%	95.76%	0.00%	15.03	95.75%
pcm	8.34%	91.56%	0.00%	0.11%	99.89%	91.56%	55.00	99.93%
apcm	8.34%	91.55%	0.00%	0.11%	99.89%	91.55%	44.00	99.93%
CRC32	0.00%	0.00%	100.00%	0.00%	100.00%	100.00%	48.04	100.00%
FFT	21.52%	13.65%	8.98%	55.86%	44.14%	22.62%	14.00	90.00%
FFT-i	21.49%	13.76%	8.96%	55.79%	44.21%	22.72%	14.00	90.02%
gsm-toast	12.09%	22.63%	6.49%	58.79%	41.21%	29.12%	8.64	83.61%
gsm-untoast	0.00%	88.68%	7.71%	3.60%	96.40%	96.40%	67.65	98.71%
average	32.90%	26.56%	11.76%	28.78%	71.22%	38.32%	33.34	94.34%

