Instruction Fetch Power Reduction Using Forward-Branch and Subroutine Bufferable Innermost Loop Buffer

I-Wei Wu, Bin-Hua Tein and Chung-Ping Chung
Dept. of Computer Science, National Chiao Tung University
gis93808@csie.nctu.edu.tw, binhua.cis93g@nctu.edu.tw, cpchung@cs.nctu.edu.tw

ABSTRACT

Recently, several loop buffer designs have been proposed to reduce instruction fetch power due to size and location advantage of loop buffer. 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 14.61% of further reduction in instruction fetch power compared with the design without forward branch and subroutine handling.

1: INTRODUCTIONS

Power consumption has become an increasingly greater concern in digital system designs, especially for battery powered devices. Among all power components within such a system, memory access power contributes the largest portion. In many researches [1-4], 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. 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.

To maximize the power advantage, a loop buffer should store innermost loops with and without forward branch(es) and no-loop-inside subroutine(s). For example, MiBench spends 26.56% and 11.76% of execution time on innermost loops with forward branch(es) and no-loop-inside subroutine(s), respectively. However, to avoid design complexity, most loop buffer designs are capable of storing only innermost loops without forward branch [1, 2] or instructions within innermost loops before a forward branch [2]. Since many applications consist of forward branch(es) or no-loop-inside subroutine(s) in their innermost loops, utilization of loop buffer and reduction in instruction fetch power in [1, 2] is limited. To increase utilization of loop buffer, [3, 4] propose a loop buffer consisting of an additional address generator to store many kinds of code segment in which instruction addresses must be sequential. Before fetching an instruction from loop buffer, address generator must generate a loop buffer 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 [1, 3, 4] require compiler help to insert special instruction(s) in program to start filling instructions into loop buffer [1, 4] or to determine which code segments should be stored in loop buffer [3]. Recompilation and code compatibility issues hence arise.

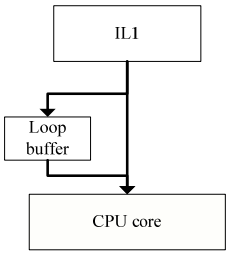

Figure 1: Organization of CPU core, IL1, and loop buffer

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 each entry of 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 no-loop-inside subroutine is also handled by using similar way. Notice also that different form previous designs [1, 3, 4], 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 [2], respectively.

The paper is organized as follows: Section 2 examines related the previous work. We present our proposed loop buffer design in Section 3. The simulation results and discussion are presented in Section 4. And finally, Section 5 concludes the paper.

2: RELATIVE WORKS

Using loop buffer to reduce instruction fetch power has been addressed in many researches [1-4]. [1] is capable of storing only innermost loops without forward branch. To indicate where a backward branch exists, [1] 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 and not taken. To reduce design complexity, [1] uses a counter to generate loop buffer addresses, called loop buffer program counter (LPC). Consequently, this causes that [1] is only capable of storing a loop without any forward branch so that utilization of loop buffer and the reduction in instruction power are limited.

Instead of using a special branch instruction "sbb" in [1], [2] 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 [1], [2] can also store instructions within an innermost loop before a forward branch or a subroutine call. This is because instruction addresses before a forward branch or a subroutine call are sequential. To reduce design complexity, [2] also uses a counter to generate LPC so that [2] encounters the same limitation with [1].

[3] is capable of storing many kinds of code segment in which instruction addresses must be sequential. In [3], 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 such that the reduction in instruction power is limited. Since code segments stored in loop buffer may consist of forward branch, [3] uses an address generator to cope with non-sequential instruction fetch. Before fetching a instruction from loop buffer, loop buffer controller must wait for LPC calculated by address generator and then uses LPC 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 great increase on instruction fetch latency and hardware cost.

To increase loop buffer utilization, [4] uses a special instruction "lbon n", where n is number of instruction should be filled into loop buffer, to dynamically fill code segments into loop buffer. [4] also uses the same method to calculate LPC so that [4] encounters the same problem with [3]. Except for [2], all designs [1, 3, 4] use compiler to assist in loop detection. Recompilation and code compatibility issues hence arise.

3: OUR APPROACH

3.1: LOOP BUFFER ARCHITECTURE

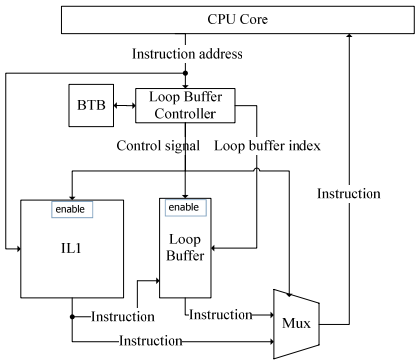

Figure 2: Architecture of our approach

Figure 2 shows a possible design of our approach in which mainly consists of loop buffer, loop buffer controller and FD-bit (filled direction bit) in BTB. 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 misprediction; and (5) no-loop-inside subroutine handling. These issues will be introduced in detail later. FD-bit records the branch perdition 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.

3.1.1: INNERMOST LOOP DETECTION

The objective of innermost loop detection is (1) detecting an innermost loop as early as possible to increase loop buffer utilization; and (2) reducing the innermost loop mis-detection rate such that energy overhead caused by mis-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 paper. 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

mis-detection rate such that it may cause higher power consumption. The power simulation of both policies will be shown in later section.

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 start or end address of an innermost loop is 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.

3.1.2: 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 may be limited. In conclusion, 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 the objective of innermost loop detection.

To meet these 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. When CPU core fetches 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 will 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.

3.1.3: DETERMINING THAT CPU CORE SHOULD FETCH INSTRUCTIONS 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.

3.1.4: 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.

3.1.5: 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 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.

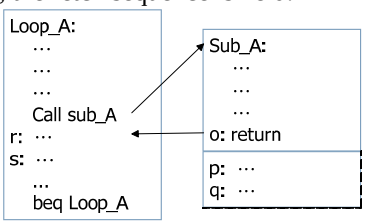

Figure 3: An innermost loop consists of subroutine

3.2: LOOP BUFFER OPERATION

In this paper, we also propose a possible design for our approach and describe it as follows. Operation states of our design which is similar with [1] consists of three states: IDLE, FILL and ACTIVE. In the figure 4, the gray rectangle and the solid black line are currently accessing block and bus respectively during different states.

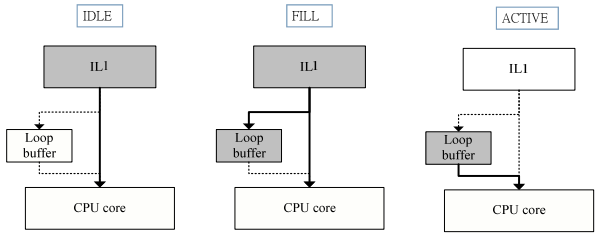

Figure 4: Memory accessing in different states

The state diagram of the loop buffer controller's finite state machine (FSM) is shown in the figure 5. 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.

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_leng 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 refill instructions when a forward branch misprediction occurs.

During ACTIVE state, CPU core fetches instructions from loop buffer instead of IL1, as action H. There are two actions to handle loop buffer miss, are action I and L. Action L is taken when branch prediction result changes from weakly taken (not-taken) to weakly not-taken (taken). Otherwise, we use action I to handle loop buffer miss. When CPU core has already fetched last instruction of loop buffer (action K) or loop buffer controller encounters a branch misprediction (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 misprediction 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.

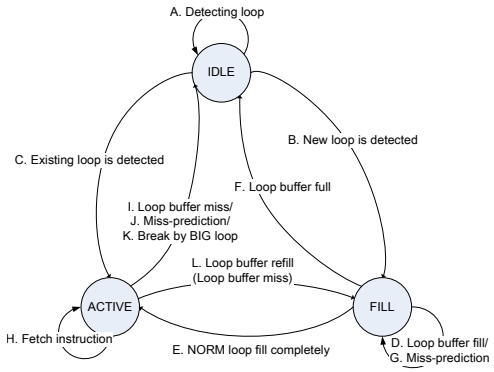

Figure 5: State diagram of loop buffer controller

4: PERFORMANC SIMULATION

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 the following equation:

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

where P_{IC} and P_{LB} are fetch power of lower level instruction memories and loop buffer, respectively, and 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):

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

Since P_{IC} is not affected by operation strategy and the size of loop buffer, we assume P_{IC} as a constant in this paper. 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}.

4.2: SIMULATION ENVIRONMENT

We use SimpleScalar/ARM simulator [5] and MiBench benchmark [6] to evaluate each design. SimpleScalar is an execution-driven simulator and used to simulate modern processor architectures. MiBench is a set of 35 embedded applications for benchmarking purposes. The power evaluation of loop buffer, BTB and IL1 is based mainly on the Wattch [7] power modeling tool.

In this paper, we experiment 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 1.

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

Table 1: Parameters setting in SimpleScalar

Four loop buffer designs, DLC, 2-way DLC, FSLB-1 and FSLB-2, are evaluated in this simulation. DLC and 2-way DLC is loop buffer design only capable of innermost loop without forward branch and [2], respectively. Both DLC and 2-way DLC employ FILL-2. FSLB-1 and FSLB-2 are our proposed designs and they are difference in instruction fill strategy. FSLB-1 and FSLB-2 employs FILL-1 and FILL-2, respectively.

Figure 6 and 7 shows RIC and RLB for the different loop buffer designs in different size, respectively. We examine the size of loop buffer from 64 to 2048 bytes, i.e. 16 to 512 instructions. Simulation results show that FSLB-1 and FSLB-2 averagely further increase RLB by 36.30%/26.27% and 25.98%/15.95%, respectively (DLC/2-way DLC). Note that since both loop buffer and IL1 are accessed during filling instructions into loop buffer, the sum of RLB and RIC would exceed over 100% in each designs. To reduce PIF, we hope that most instructions can be fetched from loop buffer, i.e. reduce RIC and increase RLB. In figure 6 and 7, our design has significant improvement in RIC and RLB. Since larger loop buffer has higher opportunity to store more instructions, RIC and RLB would decrease and increase with the size of loop buffer, respectively. However, larger loop buffer may cause higher power consumption in PLB. The tradeoff between the size of loop buffer and the reduction of instructions fetch power will be shown in later.

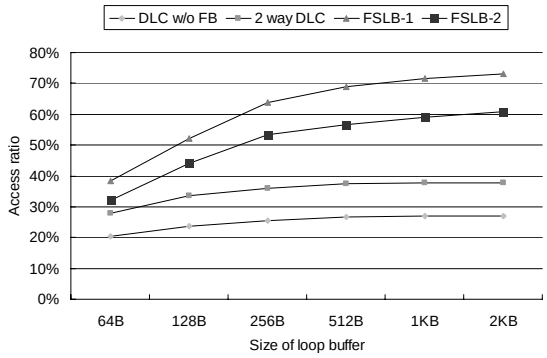

Figure 6: Access ratio of loop buffer of different designs

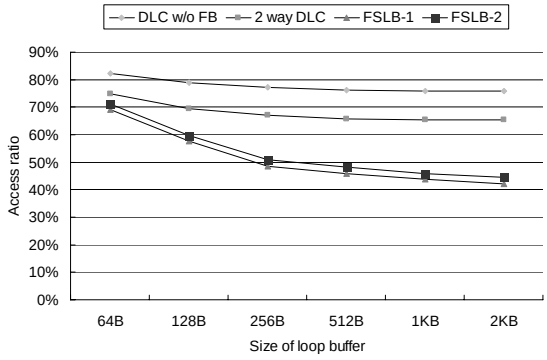

Figure 7: Access ratio of IL1 of different designs

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 2. Since FD-bit is a 1-bit field which is attached to each entry of BTB, the hardware overhead is about 1.8% such that the power consumption introduced by FD-bit is neglected.

Size of loop buffer	64B	128B	256B	512B	1KB	2KB
(P_{LB}/ P_{IC})	8.26%	8.92%	10.35%	13.56%	21.40%	38.34%
($P_{ctrl\text{-}DLC}$/ P_{IC})	0.39%	0.43%	0.49%	0.51%	0.53%	0.54%
($P_{ctrl\text{-}2\ way\ DLC}$/ P_{IC})	4.82%	4.91%	4.98%	5.05%	5.23%	5.31%
($P_{ctrl\text{-}FSLB\text{-}1}$/ P_{IC})	1.91%	1.95%	2.00%	2.08%	2.25%	2.34%
($P_{ctrl\text{-}FSLB\text{-}2}$/ P_{IC})	2.57%	2.62%	2.65%	2.74%	2.90%	2.98%

Table 2: Ratio of E_{LB} and E_{IC}

The reduction in P_{IF} of different designs is shown in the figure 8. For each design, 256B or 512B has the maximum power reduction. According to figure 6 and 7, increasing loop buffer size can R_{LB} and R_{IC} but also increases P_{LB}. Hence, larger loop buffer may be not beneficial for P_{IF}. Compared to DLC and 2-way DLC, both FSLB-1 and FSLB-2 has significantly improvement in P_{IF}.

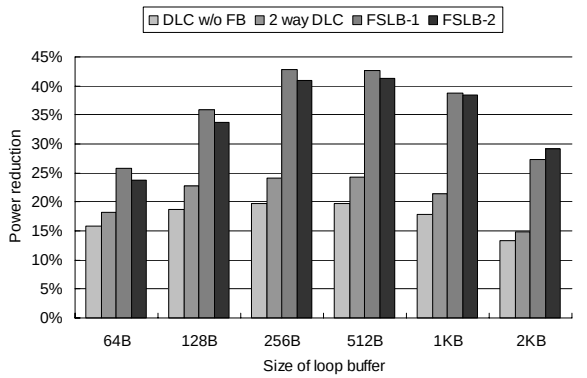

Figure 8: Reduction in P_{IF} of different designs

5: CONCLUSION AND FURTHER 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 sand hence instruction fetch power saving; (2) almost negligible hardware overhead; (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.

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.

6: REFERENCE

[1] L. Lee, B. Moyer and J. Arends, "Low-Cost Embedded Program Loop Caching – Revisited," University of Michigan Technical Report CSE-TR-411-99, 1999.

[2] T. Anderson and S. Agarwala, "Effective hardware-based two-way loop cache for high performance low power processors," *International Conference on Computer Design*, 2000.

[3] A. Gordon-Ross, S. Cotterell and F. Vahid, "Tiny Instruction Caches For Low Power Embedded Systems," *ACM Transactions on Embedded Computing Systems*, 2003.

[4] 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.

[5] T. Austin, E. Larson and D. Ernst, "SimpleScalar: an Infrastructure for Computer System Modeling," *IEEE Computer*, 2002.

[6] 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.

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