BBQ - A Simple and Effective Approach to Backward Branch Predictions for Embedded Processors

Lei Wang* and Qiong-Xian Zeng

Department of Electrical Engineering,

Feng Chia University,

Taichung 407, Taiwan, ROC

{leiwang, M9301474}@fcu.edu.tw

Received 1 March 2009; Revised 21 May 2009; Accepted 3 June 2009

Abstract. The problem of control hazard induced by branch instructions is important for the modern processors. Although there are many solutions been proposed for the problem, most of these solutions may not suitable for the embedded processors. Our research tries to find a *smart* way to provide a cost-effective function of branch prediction for embedded processors. By classifying branch behaviors into forward and backward branches, this paper first focuses on backward branches to develop a favorable solution for the instructions. A novel approach named *Backward Branch prediction Queue*, or BBQ is proposed to predict the outcomes of backward branches efficiently. This study shows that BBQ is able to retain good prediction accuracies at a small fraction of the hardware costs and complexities. Although a hardware-frugal approach like BBQ is inevitably less accurate than the luxurious prediction mechanisms, it nevertheless creates new tradeoff points of costs and performance that best suit the application domains of embedded processors. Moreover, the prediction mechanism will trace the execution flow to identify the current position in a nested loop. The information of current position will be contributive to the prediction of forward branches.

Keywords: branch prediction, embedded processor, pipeline, loop prediction

1 Introduction

Since the introduction of pipelined instruction executions, branch predictions have been critical to reduce the penalty of control hazards. There are many different methods to implement the prediction of branch instructions. Most solutions to the problem achieved by hardware enhancement use some form of special-purpose associative memory as caches to keep track of past branch behaviors, including, but not limited to, the most likely outcomes of branch conditions and the associated target addresses. The most popular solution of this type of methods is called as *Branch Target Buffers*, or BTBs [1]. In modern processors, it is not uncommon to find 1K or more entries in a BTB (in the case of multi-core processors, each core will have its own BTB of similar sizes). For high performance processors, this can be considered a small price to pay for the sake of performance. The uses of deep pipelines (from 10 up to more than 30 stages) and parallel execution among instructions demand highly accurate branch predictions to avoid the costly penalties of wrong perdition, and performance is the most important issue for the processor after all.

Departing from the high performance processors, we investigate in this research the branch prediction problem for primitive embedded processors, which find their applications in the microcontrollers of embedded control systems, personal mobile devices (cell phones, personal digital assistants, GPS navigation devices, etc.), and many Internet appliances (802.11 access points, cable/DSL modem/gateways, and so on). Since the application features of embedded systems, some critical design constraints of embedded processors distinguish them from their mainstream counterparts: (1) Cost, measured in silicon areas and/or the numbers of transistors, is an important factor for embedded processor design. Applications of embedded processors mandate low-cost, high-yield designs, inevitably limiting the hardware resources available to branch prediction mechanisms; (2) Power consumption is another issue that will affect the design. Processors in mobile devices rely on batteries as the energy source and must be as energy efficient as possible to prolong battery life [2]. BTBs and other sophisticated branch predictors, if used, could be a major power drain for embedded processors.

It is really required by embedded processors a more smart solution to handle the branch prediction problem cost-effectively. In this work, we take on the challenge of devising extremely simple, low-cost and yet effective branch prediction solutions. The solution should not only be suitable for the use of a primitive embedded proc-

* Correspondence author

essor individually, but also can be used to cooperate with other prediction schemes to achieve satisfactory prediction accuracy with lower hardware cost. Not surprisingly, the research path that we follow is a return to the RISC philosophy: Focus on the most common cases that have the biggest impact on performance, and use simple solutions to support the common cases well.

We classify the behaviors of instructions that will cause control penalty into three types: they are Backward Branches, Forward Branches, and Call/Return instructions. Each type has its own character and corresponding solutions. For example, the address of the next instruction that following a Call instruction is always the target address of a Return. For Call/Return instruction pairs which may also induce the pipeline delays in the execution, there is a hardware mechanism called *Return Buffer* been proposed in a simple and effective way [3]. Unlike Call/Return instructions, the behavior of conditional branch instructions that jump forward is most unpredictable since most of them are controlled by an uncertain condition. Predicated instructions are then been proposed to be scheduled by the compiler technique named *If-Conversion* to reduce the appearance of this type of branches [4]. Furthermore, several local/global prediction strategies that have been proved can predict the behavior of this type of branches effectively by using the past history of branches.

In particular, we single out backward branches used in loops as the most important case of the branch prediction problem. Our rational is threefold. First, loops typically make up a significant portion, if not the major portion, of program execution times. Improving the performance of loops, especially nested ones, offers the best potential of performance improvements. Second, although there are already some hardware mechanisms been proposed such as *Loop Predictor* [5]or *Loop Termination Buffer* [6], the function of these mechanisms are work based on the feature of associative memories and act as an extra enhancement hardware to help the prediction of other mechanisms such as BTBs. It means that the efficiencies of this type of solutions are produced by increasing more hardware overhead, they are not suitable to be used individually either. Third, nested loops have a clear structure of program control flows. To illustrate this structure, we show in Figure 1(a) a nested loop structure. The three branch target addresses X, Y and Z mark the ingress points of the three loops, respectively. In execution, address Z is computed first when the branch instruction BRz is executed, followed by Y when BRy is executed, followed in turn by X when BRx is executed. The three addresses in this way form a queue (Z, Y, X), where Z is at the front and X at the rear, according to the times at which they become known to the processor. As shown in Figure 1(b), the later uses of the three branch addresses also reflect a similar pattern: inside the inner most loop, address Z is most likely to be the next branch target. Once the program flow exits the inner most loop, address Y becomes the most likely target of the next control flow branching. Once outside Loop Y, address X becomes the most likely branch target. As we will show later, the BBQ approach takes advantages of this simple, queue-structured behavior in effective branch predictions.

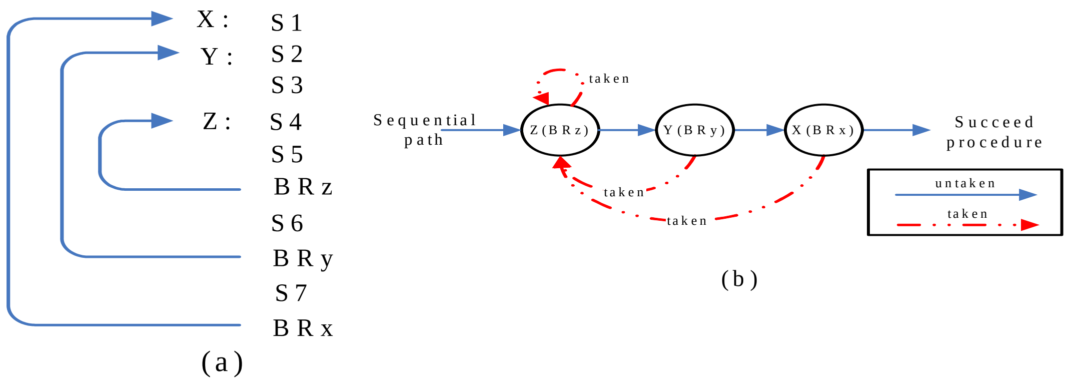


Fig. 1. A nested loop and its queue-structured branch targets

We acknowledge that not all loops are perfectly nested as the one in Figure 1. However, by taking into account the small number of exception cases which exist in applications (e.g., loops that are not perfectly nested, loops that include forward branches, etc.), we show that it is possible to preserve the simple, queue-like behaviors for the majority of backward branches. The result is a novel approach to the branch prediction problem, called *Backward Branch prediction Queue*, or BBQ. By using a standard benchmark suite for embedded processors, our performance results show that with merely 4 entries in the queue by always taken prediction strategy, the BBQ gets 88.37% prediction rate for backward branches and then achieves 42% of the performance benefits of a 128 entries, 4-way set-associative BTB by the hardware cost with less than 3.2% of the cost for the BTB. Moreover, BBQ can not only be considered as a practice of BTFN prediction, but also can be used to work by cooperating with other prediction mechanisms to achieve more accuracy with lower hardware cost. The simulation shows the hybrid usage can achieve a higher improvement by combining the BBQ with a smaller BTB. For a high-end embedded processor, the hybrid prediction mechanism can effectively reduce the hardware cost for implementation and achieve a cost-effective design.

The reminder of this paper is organized as follows. Previous approaches to branch prediction problem are first reviewed in section 2. In Section 3, we present the basic concept of BBQ and its handling of forward branches within nested loops that disrupt perfectly nested looping. We then describe in Section 4 one implementation (hardware circuit design) of BBQ that uses the popular ARM-9 micro-architecture as the reference platform. The use of ARM9 as the reference platform reflects its wide popularity rather than any bias on the part of BBQ for or against particular platforms. The reference platform also serves as a baseline for performance comparison between competing branch prediction technologies. In Section 5, we give the simulation results of our performance study. Conclusion and future work are given in Section 6.

2 Related Work

With the constant, rapid progress in microprocessor designs, the problem of control hazard becomes increasingly important. There are many methods for reducing the number of pipeline stalls caused by branch hazards. Since the study focuses on the stalls introduced by branch instructions for embedded processors, the methods proposed for predicting the execution of Call/Return instructions are not included in this paper. For general conditional branch instructions, we can classify the prediction methods into two major categories as described in the following subsections. The researches about branch prediction for the design considerations of embedded processors are also introduced in the end of this section.

2.1 Static Methods

By identifying branch instructions as forward or backward in compile time, there are several compiler-based optimization techniques have been proposed for eliminating the penalties produced by these branch instructions. For forward branches, the compiler technique named If-conversion is proposed to eliminate the appearance of this type of branches. If-conversion can convert conditional branches into predicate defining instructions, and instructions along alternative paths of each branch into predicated instructions. The predicated instructions need to be executed under the circumstance of special hardware support. It is noted that although the mainstream of embedded processors, the Advanced RISC Machine(ARM), which instruction set architecture include all instructions with the fully predicated execution capability, yet the conditional control only adopts simple flags for the control. Once if a condition becomes more complicated, the condition cannot be represented by a single compared N, C, V, or Z flag, and thus the efficiency of predicated execution is still limited because of the hardware constraint.

For backward branches, a static prediction scheme can be realized by software to reschedule the execution sequence in binaries. The scheme named BTFN, Backward Taken Forward Non-taken, is a simple but reasonable strategy for compiler to schedule instructions for branch prediction. By focusing on the feature of regular loops, the most popular compiler scheduling technique named *Software Pipelining* was proposed and turn into the basic feature of modern ILP compilers [7]. Software pipelining can reduce the number of iterations and create more instructions for compiler to reschedule in one basic block by unrolling loop codes for several times.

The other popular scheduling method is called *delayed branch* [8]. In a delayed branch, compiler will move one or more instructions that will be executed in the predicted execution path to the sequential successor(s) of the branch instruction to fill the branch delay slot(s). The drawbacks on delayed branch arising from: (1) The strict restrictions on the instructions that are scheduled into the delay slots. For example, the instructions that cannot be compensated when the prediction is proved to be wrong will not allowed to be scheduled into the slot. The instructions that may induce interrupt can not be scheduled either, or the instructions may make the processor to be interrupted by a situation that should be not happened. (2) The inability of processor architecture to handle the different execution flows of branches. Processors always can flush the execution in pipeline for the instructions that follow a branch. However, delayed branch make the situation be complicated because there will be one or more instructions be scheduled into the delay slot(s) from the predicted execution path. It means that the execution of these instructions should not be flushed when the branch is confirmed to be taken since the instruction(s) is scheduled from the taken path. On the contrary, these instructions should be flushed in the pipeline if the branch is confirmed to be not taken. Delayed branch also makes many NOP instructions be inserted into object codes and make the processor cannot be improved since any improvement that change the stages of pipeline will lead the binaries be invalid.

2.2 Dynamic Methods

To allow instruction fetch to continue without stalling, each cycle a traditional branch prediction circuit must determine: (1) the information stored about the fetched instruction if it is a branch instruction, (2) the direction of the branch and the predict address to be fetched in the next cycle. We can classify the various prediction mechanisms from two points of views as described below.

How to determine the information about the current branch instruction. The simplest dynamic branch-prediction scheme is a *Branch Prediction Buffer*. The buffer contains one or several bits for each branch instruction been executed recently to indicate whether the branch was recently taken or not. A branch prediction buffer can be implemented as a special associative memory accessed with the instruction address during the IF pipe stage. A branch prediction buffer that stores the predicted address for the next instruction after a branch is called a branch target buffer, abbreviated as BTB. The buffer is accessed during the IF stage to offer the predicted target address of the branch instruction if the address of fetched instruction cause a hit with an entry in the buffer. BTB is the most popular technique for branch prediction that used in modern processors. Even in modern high-end embedded processors, BTB acts as the standard equipment to improve the performance of execution. For example, a BTB with 128 entries is equipped in the Xscale embedded processor. There are some variances from the idea of branch prediction has been proposed in the studies of the elimination of control hazard. Branch folding can be used to obtain zero-cycle unconditional branches, and sometimes zero-cycle conditional branch [9]. The function of branch folding is somewhat like BTB except that it provides the machine code of the predicted target instruction but not the address.

Most of the proposed prediction schemes such as BTB are work under the organization of associative memory to store the information about the executed branch instructions for predictions. The organization of associative memory represents higher hardware complexity and longer access time, thus several prediction methods based on table lookup are proposed. A *bimodal* branch predictor has a table of two-bit entries, indexed with the least significant bits of the instruction addresses [10]. Unlike the associative structures, a bimodal predictor entry may be mapped to several branch instructions, in which case it is less accurate than the prediction methods made by associative structures. For enhancing the prediction accuracy provided by table lookup method, global branch predictors such as *gselect* is proposed to make use of the fact that the behavior of many branches is strongly correlated with the history of other recently executed branches. A global shift register is adapt to record the recent history of every branch executed, and the value of the register is used to index into the global table of bimodal counters. The prediction accuracy created by the gselect is little better than the bimodal scheme for large table sizes. A further improvement method that can do slightly better than gselect is gshare predictor which XOR the branch instruction address with the global history, rather than concatenating [11]. The prediction accuracy is a little better than gselect when the size of table is large.

How to determine the direction of branch instruction. Although the execution history of a branch instruction can be read out by the associative structure to make the prediction immediately, incorrect guesses of branches incur sever performance penalty by collapsing the execution flow in the processor datapath, consequently forcing the execution states to be flushed for re-warming the succeeding executions. Thus most of the researches about branch predictions concentrate on the strategies of branch predictions.

The basic branch prediction strategies are characterized by extremely straightforward heuristics that make “reasonable” predictions with very low hardware costs. The basic strategy, named *Backward Taken, Forward Not-Taken (BTFN)* [1], takes branch directions into account: the target instructions of backward branches are assumed “taken,” while those of the forward ones “not taken”. It is noted that this straightforward approach always be made as the prediction strategy for software techniques as described above. The prediction accuracy are also be created by simulation to be the comparison baseline for other prediction methods. There is no dedicated hardware mechanism that is designed to achieve the prediction strategy in hardware. The most simple prediction method introduced by hardware is to take advantages of the profiling of previous runs. A simple prediction strategy modified by the simple strategy is ‘*Two-Bits Counter*’ that record the prediction history of each with two bits like bimodal does, this elegant solution is able to handle effectively the common cases of “repeat the loop body many times and leave the loop once” behaviors of backward branches. In terms of hardware simplicity, the above solutions are generally acceptable to embedded processors [12]. However, the performance gained by the hardware is relative disappointed by checking the cost paid for the hardware.

Sophisticated branch prediction solutions generally keep track of the outcomes of recent branch instructions as the basis for predictions. In [13], the *two-level adaptive branch prediction* is proposed to use one or more branch history register (BHR) at the first level of history to record the outcomes of *k* most recent branches. The second level history is recorded in one or more *pattern history tables* (PHTs) of two bits saturating counters. In essence, the BHR is used to index the PHT to select respective two-bit counters.

In general, large amounts of history and behavioral information need to be maintained to order achieve the high prediction accuracies. Recent research of two-level predictions mainly focus on avoiding the exponential growth of table sizes and/or new hashing function to eliminate the conflictions in the tables [14-15], or propose some variants by acceding other dynamic states such as branch pattern probabilities to the hierarchical history tables to facilitate prediction accuracy [16].

Neural-network based approaches to branch predictions give rise to interesting "learning" (from past) potentialities stemming from the research in artificial intelligence. A dynamic branch predictor using neural networks, called *Learning Vector Quantization* (LVQ) was proposed in [17]. In 2001, Jiménez and Lin introduced a perceptron branch predictor [18], where neural networks are used as a more powerful alternative to the commonly used two-bit counters. Later improvements and other variants of the perceptron predictor can be found in [19].

2.3 Power-Saving Branch Predictors

In summary, branch prediction solutions involving, large history caches, hierarchical tables, and/or neural networks achieve high prediction accuracies at the expense of hardware costs. They are suitable for mainstream processors as the rapid advancements of VLSI technologies accommodate steadily increasing numbers of transistors per chip over time. The embedded processors however face drastically different design criteria, especially in the areas of complexity and power consumption constraints. In order to save power dissipation by a power hungry BTB, the most instinctive way is to reduce the accesses of BTB. A prediction probe detector (PPD) is proposed to use predecode bits to eliminate unnecessary predictor and BTB accesses in [20]. By adding an instruction filter cache in [21], the same goal is achieved by reducing the fetching operations from the instruction cache and the subsequent decoding. Some researches try to reduce the power consumption by modifying the structure of predictors. For example, the study in [22] claims that by looking up two predictions at a time by increasing the width of the PHT, and by accessing the PHT in advance. The reduction for the unnecessary BTB accesses can be achieved. In [23], the reduction is made by utilizing the hardware to buffer the control-flow structure of the executed program. It is noted that the related techniques focus the attention on the techniques of power saving and achieve their goal by adding extra hardware. The acclivity of hardware cost makes the solutions be suitable for the high performance processors, but not for low-cost embedded processors.

3 The BBQ Approach

As introduced in previous Section, we classify the behaviors of instructions that will cause control penalty into three types. Each type has its own character and corresponding solutions. In particular, we single out backward branches used in loops as the most important case of the branch prediction problem.

For backward branches, most of the proposed prediction schemes such as Loop Termination Buffer and Loop Predictor are all work under the same hardware circumstance like BTB. The organization of associative memory is used to store the information about the executed branch instructions for predictions. It is really required by embedded processors a more smart solution to handle the branch prediction problem cost-effectively. In this work, we show that it is possible to preserve the simple, queue-like behaviors for the majority of backward branches.

3.1 Basic Concept

We use the nested loop shown in Figure 1 to illustrate the operations of the BBQ. When a program starts its execution and an innermost backward branch BRz is encountered for the first time in loop Z, the BBQ discovers that it is a backward branch instruction by checking the OP-code and offset fields in the instruction, and thus the PC value and the target address of BRz are stored in the BBQ first. As shown in Figure 2(a), the content *Z* in the front of the BBQ storage stands the stored information about BRz such as the PC value and Target address. Although the BBQ encounters BRz for the first time and cannot immediately provide a target address, but thereafter if the same innermost loop, Z, is executed, the BBQ will read the front pointer by the BBQ to locate the correct predicted address for each time.

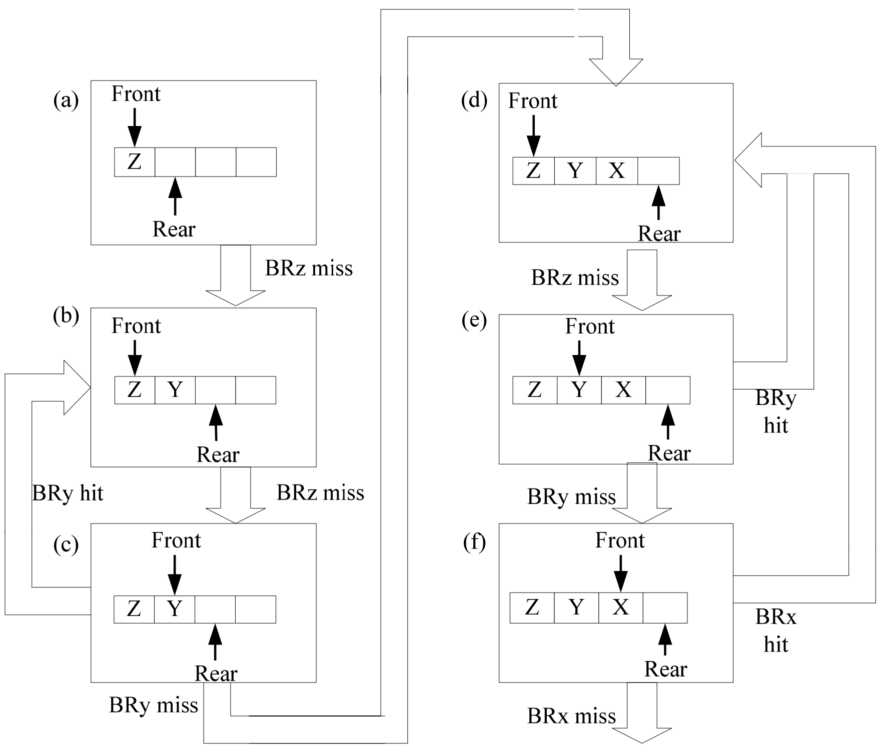


Fig. 2. BBQ in operation

If the execution of the program exits such innermost loop and enters into a middle loop Y, the BBQ will induce a wrong prediction on BRz. However, the BBQ will not clear its content until the program execution encounters the middle backward branch BRy. BRy is also a backward branch. Its target address is in front of the target address of the innermost backward branch BRz, and the PC value of BRy is greater than the value of BRz, thus the BBQ will store BRy in the BBQ as shown in Figure 2(b). Thereafter, the execution of BRy will jump back to repeat the execution, and the read front point of the BBQ is reset to zero (pointing at the jump information of a backward instruction of the innermost loop stored in the BBQ) to quickly provide the target address of BRz from the innermost loop Z, until the last jump prediction fails. By then, the read front pointer will enter into the next prediction, and adjust the prediction to the next prediction for BRy as shown in Figure 2(c). After the BRy instruction successfully predicts the middle loop Y, the read front pointer of the BBQ will return to the starting point automatically as shown in Figure 2(b), so that the next prediction will be an execution of the innermost backward branch BRz. The BBQ will repeat the foregoing operation and continue changing the process between the innermost loop Z and the middle loop Y alternately. By then, the BBQ will record a “Double-level loop status” and such status will remain until the execution of the middle loop Y no longer has a backward jump, that is, there is a miss occurred for BRy and the execution flow into the succeeding section in the outermost loop X.

The program continues executing the outermost loop X and encounters the branch BRx. Similarly, the loop X is a backward branch and constitutes a nested loop (the target address of BRx is less than or equal to the target address of BRy and the PC value of BRx is greater than the PC value of BRy). Therefore, the BBQ will be added directly with the record of the outermost loop X as shown in Figure 2(d), the BBQ is set to predict the next encountered backward branch and jump back to the innermost loop Z, and the BBQ will store a “Three-level loop status” and make changes as shown Figures. 2(d), 2(e) and 2(f).

In Figure 2(f), the execution continues until the outermost loop X no longer jumped, and then the prediction ends and gets ready to exit this nested loop, but the content in the BBQ will not be cleared yet until the execution encounters another backward instruction, say BRw, later. The BBQ compares and finds an unmatched condition, the target address of BRw is greater than the target address of BRx or the PC value of BRw is less than the PC value of BRx, then the BBQ will be cleared, and BRw will be stored in the BBQ, similar to the situation of BRz as shown in Figure 2(a).

3.2 Handling Forward Branches

The basic concept of BBQ is to focus on the predictions of backward branches used in nested loops. However, there cases, as shown in Figure 3, where forward branches interfere with the control flow of nested loops. If not handled with care, such forward branches could cause confusion to the BBQ circuits.

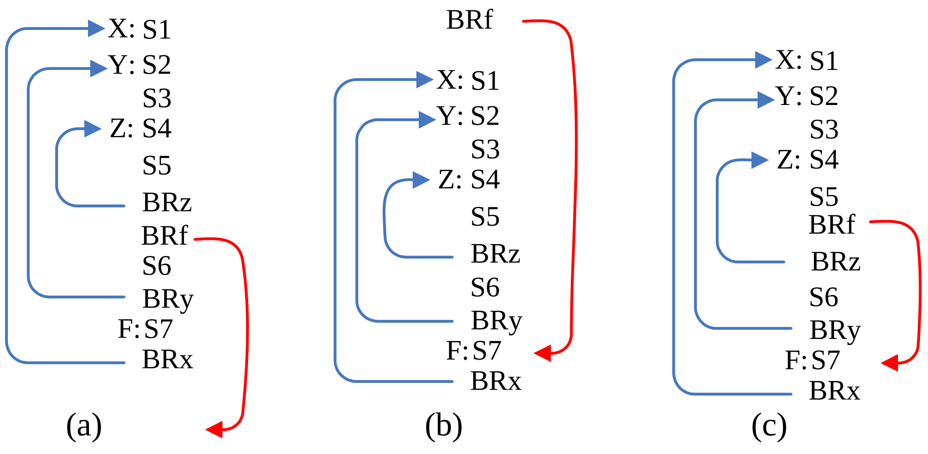


Fig. 3. Conditions of forward branches

The way of the forward branch behavior interfere the prediction of the BBQ is described in detail as follows. Although the BBQ does not store the information of a forward branch, the flow running from the interior to the exterior of a nested loop will be ruined after the forward branch instruction jumps. Therefore, the prediction has to take the effect of the forward branch instructions into consideration. According to the observation from the behavior of simulated applications, the forward branches of this sort that will alter the regular behaviors of nested loop are divided into three types as shown in Figure 3.

The situations as shown in Figures 3(a) and 3(b) will not ruin the existing prediction mechanism of the BBQ and at most it may confuse the BBQ to store unnecessary information only. As the loop continues, the BBQ will rearrange the predicted information of the foregoing mechanism, so as to eliminate the interference of the jumps of this sort.

The situation as shown in Figure 3(c) is more complicated. If the forward branch, BRf, occurs in the nested loop, and its target address is situated in another level of nest loop, the target address will exceed the innermost backward branch/branches of the nested loop. Refer to the figure, the target address of BRf exceeds the backward branches, BRz and BRy. If the forward branch jumps, it will exit the range of the innermost loop Z. Since the forward branch instruction BRf jumps and the flow enters directly into the outermost loop X. Meanwhile, the BBQ retains the original prediction that is pointed to loop Z but not loop X. The condition will make the prediction for the following backward branch to be missed even the loop structure has already built completely. Because the forward branch instruction BRf in the nested loop will be repeated continuously, the interfering misses caused by the repeated executions would be seriously. Based on the analysis of the dynamic execution of the benchmark programs, we discovered that the situation of this sort occupies about 0.9139% of the total number of executed instructions. Particularly in certain specific applications such as the jpeg and dijkstra shortest path occupy 5.773% and 16.839% of the total executed instructions respectively. The interfering condition must be distinguished by the BBQ for prediction accuracy.

To overcome the ruin of this sort of forward branch as described above, some comparators are added to compare the target address of the executing branch instruction and the PC values stored in each entries of BBQ to determine the location where the forwarding branch jumped. The result of comparisons will adjust the front pointer to catch the progress of execution flow for the following execution.

4 The BBQ Implementation

In this section, we present the hardware design of the proposed BBQ approach by using the ARM9TDMI architecture as the reference platform. The ARM-9 architecture is part of the extremely (if not the most) popular ARM processor family in embedded applications. It is widely used as the processing core of cellular phone, PDA, and hand-top game player. The use of the ARM-9 as the implementation platform of BBQ makes our design details presented below readily applicable to real world applications. The instruction pipeline of ARM-9 is shown in Figure 4. We can find from the figure that the execution of branch instructions will induce 2 cycles penalty to change the execution flow.

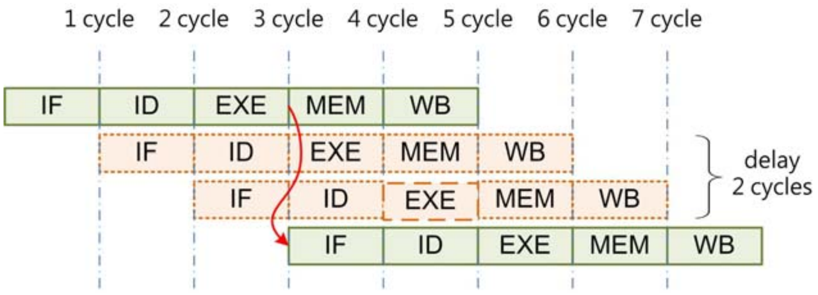


Fig. 4. Branch delay in ARM-9 instruction pipeline

Based on the 5 stage pipeline of ARM-9, the function of BBQ is added into the three leading stages: instruction fetch (IF), decode (ID) and execution (EXE). The control flow of BBQ is shown in Figure5. The behavior of BBQ can be observed from the figure. It is noted that when the BBQ is full with nested loops, a new entry will be insert in the first entry as a circular structure does. This situation will destroy the contents in the BBQ and induce extra cycles to rebuild the nested loop in the following execution. According to the simulation results, we found that a BBQ with 4-entries can satisfy most of nested loops and sustain a good prediction hit rate. The hit rate of a BBQ with 8-entries is only slightly higher than the smaller BBQ gained. Thus this study determined to design the BBQ circuit with 4-entries and simulate the BBQ with the same capacity for performance evaluation.

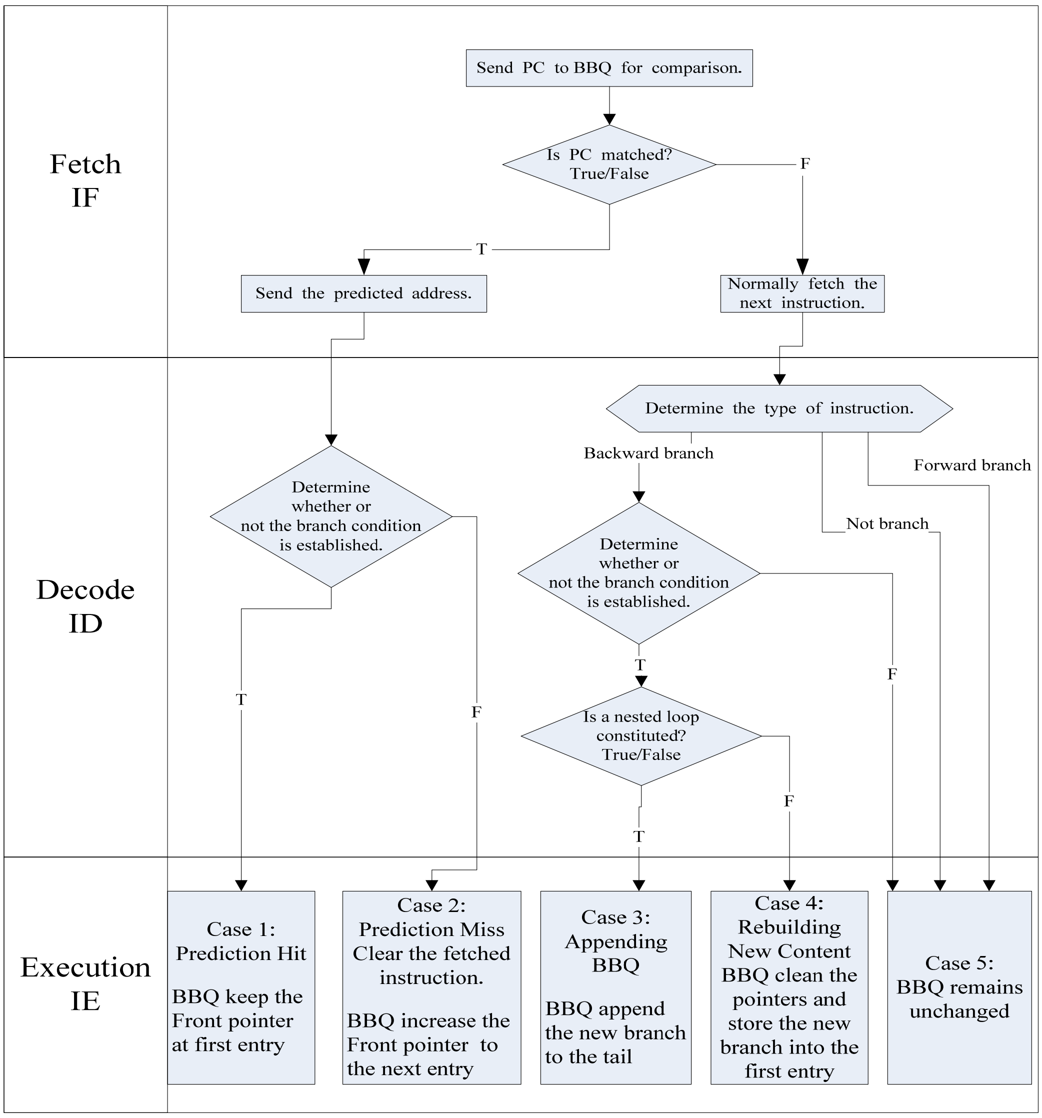

Fetch IF
Send PC to BBQ for comparison.
Is PC matched? True/False
T
F
Send the predicted address.
Normally fetch the next instruction.
Decode ID
Determine the type of instruction.
Backward branch
Forward branch
Not branch
Determine whether or not the branch condition is established.
Determine whether or not the branch condition is established.
T
F
Is a nested loop constituted? True/False
Execution IE
Case 1: Prediction Hit BBQ keep the Front pointer at first entry
Case 2: Prediction Miss Clear the fetched instruction. BBQ increase the Front pointer to the next entry
Case 3: Appending BBQ BBQ append the new branch to the tail
Case 4: Rebuilding New Content BBQ clean the pointers and store the new branch into the first entry
Case 5: BBQ remains unchanged

Fig. 5. The control flow of BBQ in pipeline

4.1 BBQ Actions in ARM-9 Pipeline Stages

The block diagram of the BBQ circuits is shown in Figure 6. The core part of BBQ circuit is the unit named *BBQ Element*. Since the update of the element is done in EXE stage, the detail circuit of the BBQ Element is then described in the stage, too.

The circuits for the IF stage uses a NPC multiplexer to select an address to write to the *next program counter* (NPC) as the address for the instruction be fetched in the next cycle. Besides selecting the original cumulative PC values from the arithmetic logic unit (ALU) or memory access, a new data line, BTAR, is added to the multiplexer as the target address of the predicted backward branch. If the instruction being fetched is a backward branch and its PC matches the content of BBQ front element, BPC, the comparator will select the target address by means of EQU signal to NPC multiplexer for indicating the next address being fetched will be the BTAR. It means that the value of BTAR will be written into the program counter to fetch the predicted instruction in the next cycle.

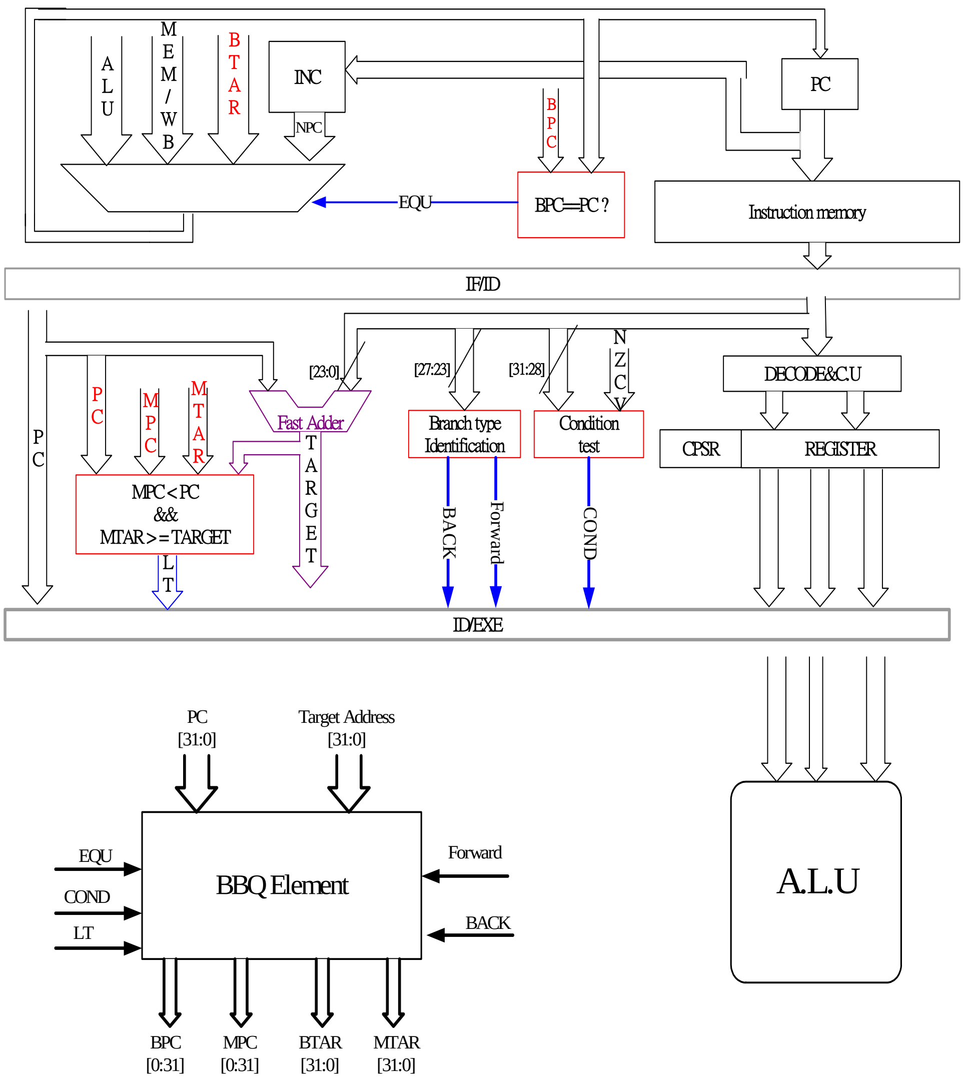


Fig. 6. The block diagram of BBQ design for ARM9 datapath

When an instruction enters the decode stage, the decode circuit will use its first level OP-code and the sign bit of displacement (bits [27:23]) to determine whether the instruction is a branch instruction or not, and identify the type of the branch instruction such as a forward branch or a backward branch, producing BACK and FORWARD control inputs to the BBQ Element when the instruction enters the EXE stage. Another control signal COND is produced in the decode stage by comparing the conditional fields of the decoded instruction ([31:28] bits in the instruction) with the value of NZCV flags by a comparator.

It is noted that in the original ARM-9 architecture, branch target addresses are calculated by ALU at the EXE stage. This arrangement delays the updates of the BBQ Element to the 4th stage as described in Figure 4. If the address is calculated at the ID stage, the branch instruction can be completed one stage earlier, and the branch delay can be reduced to one. This optimization technique applies equally to both forward and backward branches. We therefore modify the ARM-9 pipeline by adding a dedicated branch address adder into the ID stage (see Fig. 5) and in this way obtain the branch target address one stage earlier than the standard ARM-9 pipeline. By comparing the target address with the MTAR issued from the BBQ Element, and the original PC value with the MPC of the BBQ Element, the outcome of the comparator, LT, can be made to determine whether the backward branch is a new one or not, and decide whether the new branch can constitute an outer loop of the nested loops structure stored in BBQ already. The LT is then latched and send to the next stage for the use of BBQ Element as BACK/Forward/COND does.

When the execution enters into the execution stage, the BBQ Element will update its content according to the comparison results in this stage.

4.2 BBQ Element

Besides the control lines such as COND, LT, and EQU produced in the first two stages. There are other input/output lines of BBQ Element defined below.

PC[31:0]. A 32-bit input that provides the address of the instruction being fetched in the present cycle. It should be written into the BBQ if the instruction is a new backward branch.

Target Address[31:0]. A 32-bit input that provides the target address of a branch in ID stage. The input is used for updating the BBQ Element.

BPC[31:0]. A 32-bit output that gives the PC value of the backward branch instruction predicted by the front pointer.

BTAR[31:0]. A 32-bit output that gives the branch target address read out by the front pointer.

MTAR[31:0]. A 32-bit output that gives the target address of the branch instruction stored in BBQ as the outermost branch of the nest loop, the one stored in the rear entry in the queue.

MPC[31:0]. A 32-bit output that gives the PC address of the branch instruction stored in BBQ as the outermost branch of the nest loop.

The circuit design of the BBQ Element is depicted in Fig. 7. The BBQ Element can be divided into three components:

BBQ Element control circuit. The signals be used to control the action of BBQ Element are all issued from the control circuit. It’s a combination circuit for translating the input lines to the control signals.

BBQ Element storing circuit. It is the main storage of the BBQ element organized in Flip/Flops and related decoders and counters (pointers). The front pointer, BBQF, and the rear pointer, BBQR, are two counters used to address the entry that should be read out for prediction or be written for new branch. Another counter, BBQM, is used as the pointer which points to an entry in the BBQ that contains the last valid PC/target address pair for the outer most loop be recorded.

BBQ pointer adjust circuit. The circuit can adjust the prediction entry in the BBQ Element to eliminate the interfering caused by the forward branches described in section 3. It is also a combination circuit.

For verifying the design and evaluating the efficiency of BBQ, this research adopts Xilinx Foundation 4.2i and Xilinx ISE as the design tools to implement the circuit in gate level. By incorporating the circuit with an ARM-9 equivalent circuit that implemented by the formal research [24]. The whole circuit has been verified and the hardware characters can be concluded below.

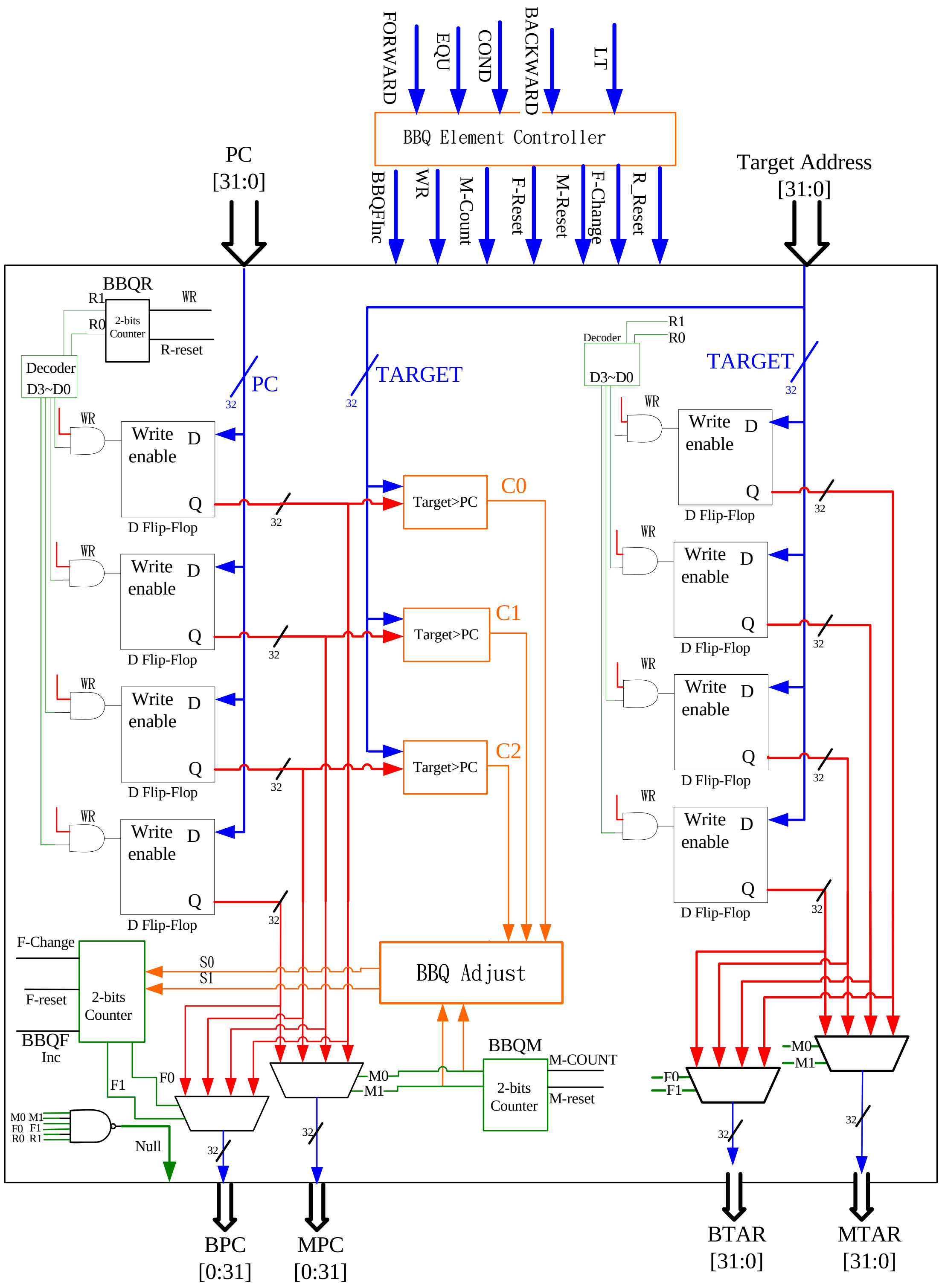


Fig. 7. The circuit in BBQ Element

The total cost of a 4-entry BBQ in gate counts is 5013 gates. It is noted that the whole register bank built in the processor cost 22756 gates (including the related decoder circuits) and other 6850 gates been used for a barrel shifter. The number proved that the hardware cost is much low for implementation. Furthermore, the study has also built a fully associative BTB with 4 entries for the architecture, too. The hardware comparison between BBQ and the BTB shows that BBQ is smaller than the BTB (it is designed in the same technology with 5114 gates). For access speed, BBQ can issue the prediction in 0.89 ns, and the BTB will read out the predict data by 1.324 ns. Since the simplicity of the circuit, the prediction speed of BBQ is much faster than BTB, too. It must be emphasized that the prediction speed is an important issue for the design of a microprocessor with deep pipeline and high clock frequency. The prediction made by long access time may become the critical path of the pipeline and make the execution be postponed.

5 Performance Evaluation

We use 12 benchmarks that contribute a representative part of Mibench [25] as a standard performance testing programs. They are Bitcount and Quicksort used in industrial applications, Jpeg encode/decode and Tiff2bw used in consumer programs, Dijkstra, CRC32, and FFT for telecommunication usage, SHA, Blowfish encrypt/decrypt, and Rijndael encrypt/decrypt for security functions. All of these benchmarks are popular and widely used in the various applications of embedded systems. The testing platform selected for simulation is Simplescalar [26], which is a well-known and reliable simulation tool used for architecture studies. Simplescalar can behave as the architectures of MIPS or ARM9 processors for simulation. We can also modify the target architecture to fit the features of the study and then simulate the benchmarks to get the data about simulation conveniently.

The simulation results reveal the truth that BBQ is a cost-effective technology for embedded processors. Since a successful prediction include the hit of a stored entry that can provide the target address and the correct direction provide by the prediction strategy. The simulation gathers statistics in term of prediction rate to stand the successful prediction with correct target address.

We simulated with the same benchmarks for the backward branch prediction rate made by different BTBs for comparison. The rates about these BTBs are statistic from the execution of backward branches. The predictions for forward branches are omitted in the simulations. As shown in Figure 8, the simulated results show that even predicted by a 128-entry direct mapped BTB with bimodal prediction strategy, which is the BTB equipped in XScale processor, the prediction rate for backward branches is only 82.33%.

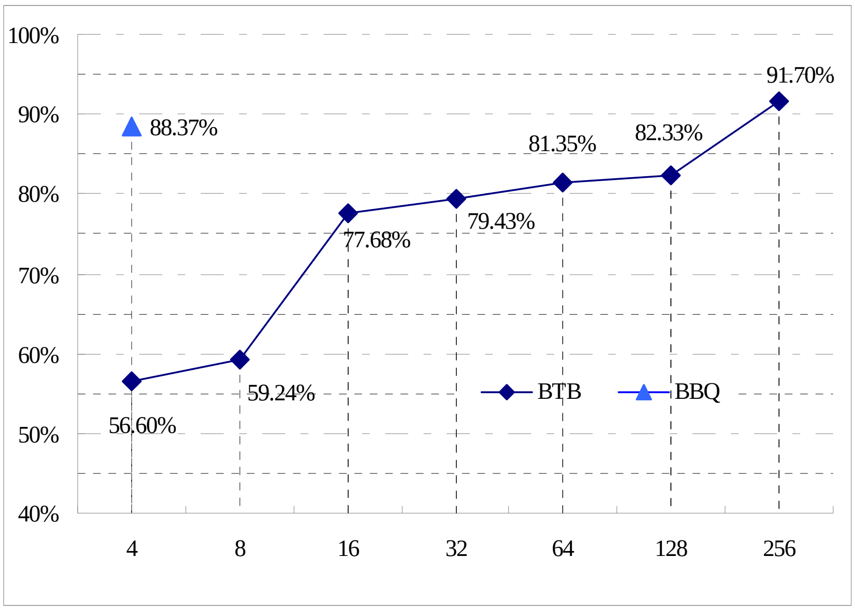


Fig. 8. The backward branch prediction rates of direct mapped BTBs and 4-entries BBQ

The BTB with direct mapped structure must be equipped with more than 256 entries to gain a prediction rate higher than 90%. Further, a BTB with set-associative structure can achieve higher prediction rate by complex hardware. A 128-entry 4-way set associative BTB with bimodal prediction strategy can predict backward branches in 91.77% prediction rate. The simulation results exhibited in Figure 9 shows that our BBQ could achieve 93.66% of the prediction rate of a 128-entry 4-way set associative BTB.

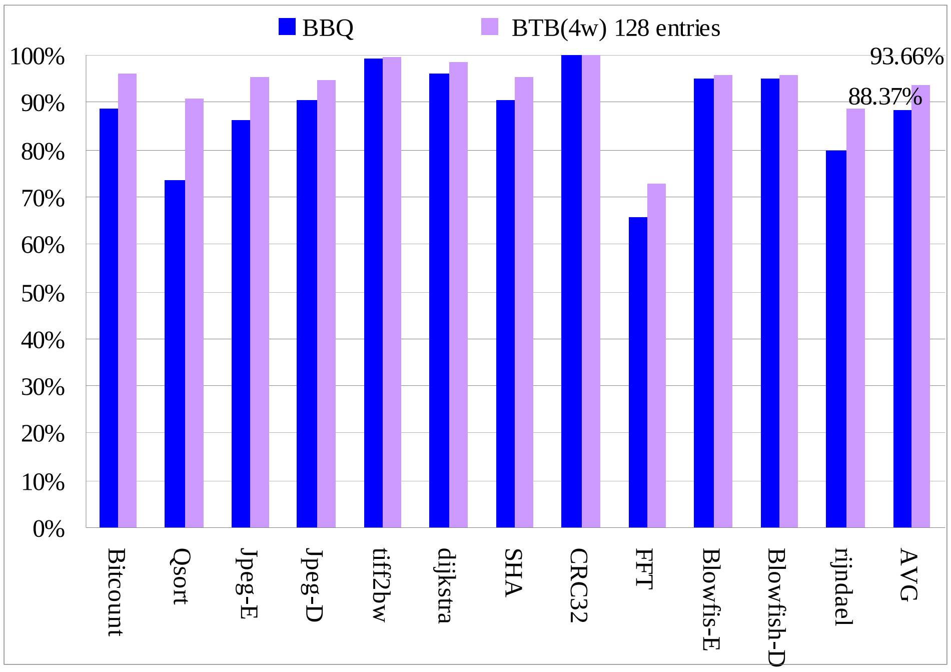
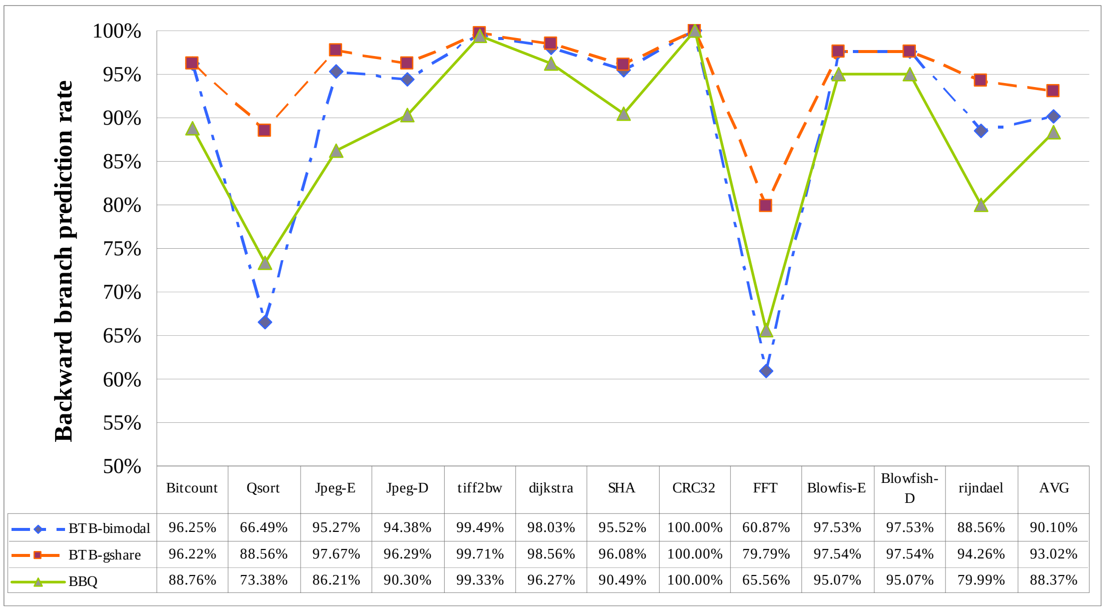


Fig. 9. The comparison of backward branch prediction rate between 4-entries BBQ and set associative BTB with 128 entries

It is interesting to observe the prediction efficiency of a small but intelligent BTB to serve backward branch prediction as BBQ does. We simulated a fully associative BTB with 4 entries to store the backward branches recently executed. By equipping the BTB with two different prediction strategies, one is bimodal model and the other is gshare model, for simulation. The prediction rates are shown in Figure 10. The curves in Figure 9 show that although BBQ acts as a simple prediction mechanism that predict backward branch by the assumption of regular loop structure, which is not as flexible as fully associative BTB that can find the correct branch instruction from the 4 entries without any limitation. The prediction strategy of BBQ is also very simple that always predict the executed backward branch will be taken. The prediction rate of BBQ is just a little lower than the fully associative BTBs. The 4-entry BBQ can achieve 95% of prediction accuracy of a smart BTB. We must remind that the smart BTB simulated is equipped with a global history table indexed by a 10-bits shift register. The hardware is more complex than the BBQ as introduced in the previous section. Moreover, it should be emphasized again that the gate delay is also another important issue for the design. According to the implementation circuit, we can find that the BBQ can provide the prediction in the speed that is 1.5 times faster than the BTB.



	Bitcount	Qsort	Jpeg-E	Jpeg-D	tiff2bw	dijkstra	SHA	CRC32	FFT	Blowfis-E	Blowfish-D	rijndael	AVG
BTB-bimodal	96.25%	66.49%	95.27%	94.38%	99.49%	98.03%	95.52%	100.00%	60.87%	97.53%	97.53%	88.56%	90.10%
BTB-gshare	96.22%	88.56%	97.67%	96.29%	99.71%	98.56%	96.08%	100.00%	79.79%	97.54%	97.54%	94.26%	93.02%
BBQ	88.76%	73.38%	86.21%	90.30%	99.33%	96.27%	90.49%	100.00%	65.56%	95.07%	95.07%	79.99%	88.37%

Fig. 10. The comparison of prediction rate between BBQ and set associative BTB

Although BBQ can achieve the high prediction accuracy by a little cost, the performance improved by the design should be evaluated more carefully since BBQ only work in the execution for backward branches. Figure 11 shows the performance improved by a 4-entry BBQ over the execution without branch prediction for the benchmarks. From the figure we can conclude that the BBQ can improve the performance to 8.62% in average. It is noted that the performance improved by a 128-entry 4-way set associative BTB is 20.45%, the BBQ achieves 42% of the performance benefits of the BTB by the hardware cost with less than 3.2% of the cost for the BTB. The improvement is gained by adding only 6% of the gate counts of the ARM-9 processor core. It means that the idea of BBQ can be used in low-end embedded processors individually to improve the efficiency in a much cost-effective way.

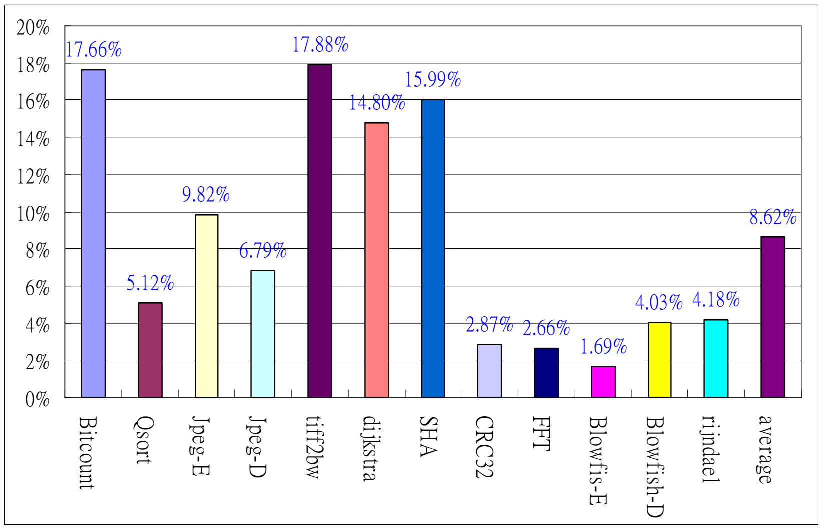


Fig. 11. Performance improved by a 4-entry BBQ

The original idea of BBQ is not proposed to replace the role of other branch prediction mechanisms. It can also be equipped to cooperate with other prediction mechanisms to achieve a more cost-effective design, too. By equipping a BTB to handle the prediction of forward branches and a 4 entry BBQ to predict backward branches, the prediction rates shown in Figure 12 exhibit the prediction rates achieved by the cooperation of a BBQ and direct mapped BTB, the lines show that the hybrid usage can achieve a higher improvement by combining the BBQ with a smaller BTB. For a high-end embedded processor, the hybrid prediction mechanism can effectively reduce the hardware cost for implementation and achieve a cost-effective design. However, it should be noted that the improvement by combining the features of BBQ and set associative BTBs is comparatively small than the values shown for direct mapped BTBs. Because of the higher prediction accuracy of set associative structure, the hybrid feature can benefit prediction evidently only in the conditions that the capacity of the BTB is less or equal than 32 entries.

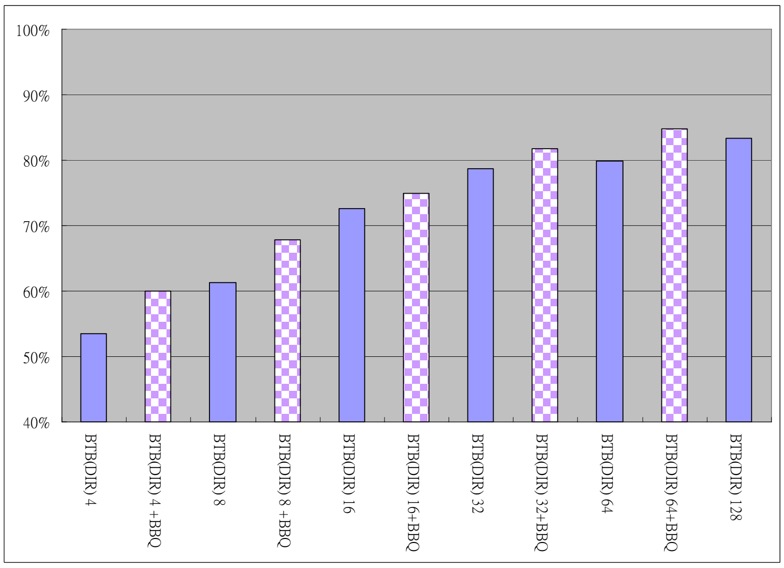


Fig. 12. Prediction rates of direct mapped BTBs with BBQ

6 Concluding Remarks

In this paper, we propose a novel mechanism, BBQ, for the branch prediction. By focusing only on backward branches used to create loops, we show that BBQ is able to sustain good prediction accuracies in a most cost-effective way. The idea of BBQ creates new tradeoff points of costs, performance, and power consumptions that best suit embedded processors.

There is an important issue for BBQ been observed from the simulation results, we found that the behavior of function call and the condition of multiple nested loops exist in a same outer loop confuse BBQ and introduce some unnecessary misses. We are now improving the BBQ design by modifying the update algorithm of BBQ to maintain the prediction data more efficiently. A BBQ with circular replacement strategy is proposed. According to the preliminary simulation, we expect that the modification can promote the prediction rate of BBQ to over 92%. The detail of this improvement will be proposed when the hardware verification is completed.

The goal of our research is to find a more smart way to achieve branch prediction in less complexity for embedded processors. BBQ does not only provide a cost-effective prediction for backward branches, it also maintains the execution state by tracing the execution flow to identify the current position in a nested loop. The information of current position can be used to help the prediction for forward branches. For example, the structure of associative memory can be modified by means of the information to manage the placement and replacement of a BTB. We expect that the modification can promote the prediction rate and reduce the hardware complexity of a BTB. The design of a smart forward branch prediction mechanism is the next design target of our research.

7 Acknowledgement

This work was supported by National Science Council in Taiwan under grants 94-2213-E-035-038 and 95-2221-E-035-009. We would like to thank Dr. Yih Huang for his advice and discussions.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 1996.

[2] M. B. Kamble and K. Ghose, "Analytical Energy Dissipation Models for Low Power Caches," *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, pp. 143-148, 1997.

[3] V. E. Hummel and H. Sharangpani, "Return Register Stack Target Predictor," United State Patent Number 6,560,696, 2003.

[4] J. A. Fisher and S. M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Pprogram," *Proceedings of the 5th international conference on Architectural support for programming languages and operating systems*, pp. 85-95, 1992.

[5] M. Alba and D. Kaeli, "Runtime Predictability of Loops," *Proceedings of IEEE International Workshop on Workload Characterization (WWC-4)*, pp. 91-98, 2001.

[6] T. Sherwood and B. Calder, "Loop Termination Prediction," *Proceedings of the 3rd International Symposium on High Performance Computing. Springer-Verlag*, pp. 73-87, 2000.

[7] V. H. Allan, R. B. Jones, R. M. Lee, S. J. Allan, "Software Pipelining," *ACM Computing Surveys*, Vol. 27, No. 3, pp. 367-432, 1995.

[8] T. R. Gross and J. L. Hennessy, "Optimizing Delayed Branches," *Proceedings of the 15th annual workshop on Microprogramming*, pp. 114-120, 1982.

[9] L. H. Lee, J. Scott, B. Moyer, J. Arends, "Low-cost Branch Folding for Embedded Ppplications with Small Tight Loops," *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pp. 103-111, 1999.

[10] J. E. Smith, “A Study of Branch Prediction Strategies,” *Proceedings of the 8th International Symposium on Computer Architecture*, USA, pp. 135-148, 1981.

[11] J. W. Kwak and C. S. Jhon, “High-performance Embedded Branch Predictor by Combining Branch Direction History and Global Branch History,” *IET Computers & Digital Techniques*, Vol. 2, No. 2, pp. 142-154, 2008.

[12] *Intel XScale^{TM} Technology*, http://www.intel.com/design/ intelxscale/.

[13] T.Y. Yeh and Y. Patt, “Two-level Adaptive Training Branch Prediction,” *Proceedings of the 24th annual international symposium on Microarchitecture*, pp. 51-61, 1991.

[14] A. Fern, R. Givan, B. Falsafi, T. N. Vijaykumar, “Dynamic Feature Selection for Hardware Prediction,” *Journal of System Architecture*, Vol. 52, No. 4, pp. 213-234, 2006.

[15] Y. Ma, H. Gao, H. Zhou, “Using Indexing Functions to Reduce Conflict Aliasing in Branch Prediction Tables,” *IEEE Transactions on Computers*, Vol. 55, No. 8, pp. 1057-1061, 2006.

[16] J. W. Kwak, J. H. Kim, C. S. John, “The Impact of Branch Direction History Combined with Global Branch History in Branch Prediction,” *IEICE Transactions on Information and Systems*, Vol. E88-D, No. 7, pp. 1754-1758, 2005

[17] L. Vintan and M. Iridon, “Towards a High Performance Neural Branch Predictor,” *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, pp. 868-873, 1999.

[18] D. A. Jimenez and C. Lin, “Dynamic Branch Prediction with Perceptrons,” *Seventh International Symposium on High-Performance Computer Architecture*, pp. 197-206, 2001.

[19] V. Desmet, L. Eeckhout, K. De Bosschere, “Improved Composite Confidence Mechanisms for a Perceptron Branch Predictor,” *Journal of Systems Architecture*, Vol. 52, No. 3 , pp. 143-151, 2006.

[20] D. Parikh, K. Skadron, Y. Zhang, M. R Stan, “Power-Aware Branch Prediction: Characterization and Design,” *IEEE Transaction on Computers*, Vol. 53, No. 2, pp.168-186, 2004.

[21] W Tang, R Gupta, A Nicolau, “Power Savings in Embedded Processors through Decode Filer Cache,” *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, pp. 443-448, 2002.

[22] S. W. Chung, G. H. Park, S. B. Park, “A Low-Power Branch Predictor for Embedded Processors,” *IEICE TRANSACTIONS on Information and Systems*, Vol. E87-D, No. 9, pp. 2253-2257, 2004.

[23] P. Petrov and A. Orailoglu, “Low-power Branch Target Buffer for Application-specific Embedded Processors,” *IEE Proceedings-Computers and Digital Techniques*, Vol. 152, No. 4, pp. 482-488, 2005.

[24] H.Y. Lo and L. Wang, “SmartARM- An Improved Microarchitecture Design for ARM Processor,” *National Computer Symposium*, Taipei, 2005.

[25] *EDN Embedded Microprocessor Benchmark Consortium*, http://www.eembc.org.

[26] T. M. Austin and D. C. Burger, “SimpleScalar Version 4.0 Release,” *Tutorial in conjunction with 34th Annual International Symposium on Microarchitecture*, 2001.