JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 24, 1127-1142 (2008)

DOI:10.1688/JISE.2008.24.4.8

Filtering of Unnecessary Branch Predictor Lookups for Low-power Processor Architecture*

WEI-HAU CHIAO AND CHUNG-PING CHUNG
Department of Computer Science
National Chiao Tung University
Hsinchu, 300 Taiwan

Efforts to reduce power consumption of processors have gained much attention recently. Dynamic branch predictor, including BTB, is exercised in every instruction cycle, yet branch instructions only constitute a small percentage of instructions during program execution. This study proposes a novel method to collect the next branch distances of the recent branch instructions at runtime, in order to eliminate unnecessary branch predictor lookups. Simulation results show that the proposed design reduces the energy consumption in the dynamic branch predictor by an average of 56.95% without performance degradation. The proposed method is fully compatible with BPP and SEPAS, and saves more energy than PPD, Lazy BTB, and EIB does.

Keywords: branch prediction, BTB, lookup filtering, low power, pipelined processor

1. INTRODUCTION

Recently, reducing power consumption for processor has come to constitute one of the defining challenges for processor architecture design. Almost all modern processors are highly pipelined today. To reduce the number of stall cycles due to branches, most processor cores perform dynamic branch prediction at the first pipeline stage. However, since the fetched instruction cannot be identified as a branch at this stage, the dynamic branch predictor, which includes a branch direction predictor and branch target buffer (BTB), is always exercised. Moreover, the dynamic branch predictor is a large array structure. Many works have shown that the dynamic branch predictor dissipates a non-trivial amount of power – averaging to 5-10% of the total processor power [1-4]. The power-hungry nature of the above discourages the portable devices from using the dynamic branch prediction. Nevertheless, dynamic branch prediction is still very attractive to processors for power-miser applications owing to its success in performance improvement. Therefore, low-power issue for dynamic branch prediction becomes a significant research topic.

Because branch instructions constitute only a small portion of all executed instructions, most dynamic branch predictor lookups are useless. This work focuses mainly on eliminating these useless branch predictor lookups. The problem of skipping useless branch predictor lookups can be transformed into the problem of locating the next upcoming branch instruction. This work presents a next branch distance collection mechanism to record the number of non-branch instructions between the two adjacent branch

Received September 12, 2006; revised January 25 & June 25 & October 19, 2007; accepted November 22, 2007.
Communicated by Tei-Wei Kuo.
* This paper was partially supported by the National Science Council of Taiwan, R.O.C. under grant No. NSC 95-2221-E-009-065-MY3.

instructions on the execution path of program. These next branch distances can be used to locate the next upcoming branch instruction, making the elimination of useless branch predictor lookups become trivial.

The rest of this paper is organized as follows. Section 2 introduces the background and the related works in detail. Section 3 presents our design. Section 4 gives the experiments. The last section draws conclusions.

2. BACKGROUND AND RELATED WORKS

2.1 Dynamic Branch Prediction

Pipeline stalls due to branch instructions have great impact on processor performance. Dynamic branch prediction and target address caching can be a great help here. A typical dynamic branch predictor is composed of a direction predictor and a BTB. Various implementations of the direction predictor record the branch statuses in different ways, and use them to predict the branch direction [5-7]. Furthermore, hybrid implementations integrate several sub-predictors to improve the branch prediction accuracy [8]. BTB is used to record the branch target addresses of the recently executed branches. If a branch instruction is predicted taken and its target address is found in BTB, then the target address is used as the next program counter (PC). Otherwise, the next sequential PC is used.

2.2 Low-power Branch Prediction

The power consumption of a dynamic branch predictor can be adjusted in two ways:

1. Changing configurations

Reducing the BTB and direction predictor size can reduce the power consumption of branch prediction. However, the extra chip-wide energy due to the insufficient BTB and direction predictor size is more expensive than this localized energy saving energy [2].

2. Reducing the number of unnecessary predictor accesses

Reducing the number of predictor accesses is an obvious way to reduce power. Compiler-hinted approaches [3, 9, 10] have some limitations in nature. For example, [3, 10] could skip unnecessary branch predictor lookups in each hot spot of a program, but not the whole program. Moreover, these approaches rely on the compiler to insert extended instructions, inevitably increasing the program size, and the complexity for the instruction decoder.

Architecture-level approaches examine dynamic behaviors of programs to avoid unnecessary predictor accesses. BPP [11] and SEPAS [12] have been proposed for hybrid branch direction predictors. PPD [2] employs a hardware table to determine whether the current accessed L1 I-cache set (the cache lines with the same indexes) is control-free, so that the lookup in the branch predictor can be avoided.

The above three approaches [2, 11, 12] use on-chip filter table to determine whether the predictor access is performed or not. If the full cycle has insufficient time to allow the filter table to be accessed in-series with the branch predictor (serial scenario), then the power saving to dynamic branch predictor is limited. It implies that the access to the filter table can not complete in time to avoid the full branch predictor access, so the accesses of the filter table and branch predictor must be in parallel (parallel scenario). This situation means that only partial predictor access is terminated.

Lazy BTB [13] dynamically profiles the taken traces to avoid unnecessary predictor accesses. The number of instructions between a taken branch and its subsequent taken branch are collected into an additional BTB field. This collected information is then used to eliminate the predictor lookups. The simulation results in [13] show that Lazy BTB reduces the predictor access energy consumption by about 77% on average, with 1.7% performance degradations. Unfortunately, the extra chip-wide energy due to the performance degradations may be serious, especially in deep pipeline processors. However, no discussions about the extra chip-wide energy are found in [13].

3. DESIGN FOR ELIMINATING UNNECESSARY BRANCH PREDICTOR LOOKUPS

This section introduces the design for eliminating unnecessary branch predictor lookups. We first present the system overview in section 3.1. The function blocks of this design are then described in sections 3.2, 3.3 and 3.4.

3.1 System Overview

In this paper, we intend to develop a method for the elimination of the unnecessary branch predictor lookups without performance degradation. Through out this paper, the distance in terms of the number of non-branch instructions between a branch instruction and its subsequent branch instruction on execution path is defined as next branch distance (NBD). If NBD is revealed to the processor early, then the branch predictor lookups for these non-branch instructions can be avoided. Therefore, this work focuses on dynamically collecting NBDs and eliminating the unnecessary branch predictor lookups using these collected NBDs.

Fig. 1 displays the function blocks of this design integrated in a typical pipeline front-end which is composed of instruction fetch (IF), instruction decode (ID), and execution (EX) stages. An extra storage, NBD table (NBDT), is used to record the NBDs. During EX stage, an NBD counter (NBDC) collects the NBDs of the executed branches into NBDT. During IF stage, BTB, direction predictor (Dir-Pred), and NBDT are looked up simultaneously, where the BTB and Dir-Pred lookups are for dynamic branch prediction and the NBDT lookups are for NBD probe. If a NBD equal to m is found during NBDT lookup, the next lookup filter filters the following m lookups in BTB, Dir-Pred, and NBDT. The control bit EN is used for the lookup filtering for each fetched instruction. If the lookup to Dir-Pred and BTB is allowed, the predicted next PC is determined by the typical dynamic branch prediction. Otherwise, the static branch prediction (predicted not taken) is used.

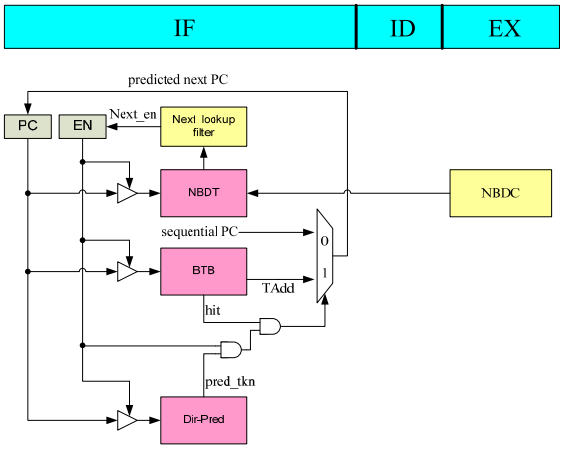


Fig. 1. The function blocks of this design integrated in a typical pipeline front-end.

3.2 NBD Collection and NBDT Management

NBDT records the NBDs of all branch instructions existing in BTB. Each BTB entry has its owned NBDT entry. Each NBDT entry contains two *n*-bit NBD fields and two single-bit valid fields, where tkn_NBD records the NBD of taken path, nt_NBD NBD of not taken path, tkn_v indicates if tkn_NBD is valid, and nt_v indicates if nt_NBD is valid. Initially, all valid fields are invalid.

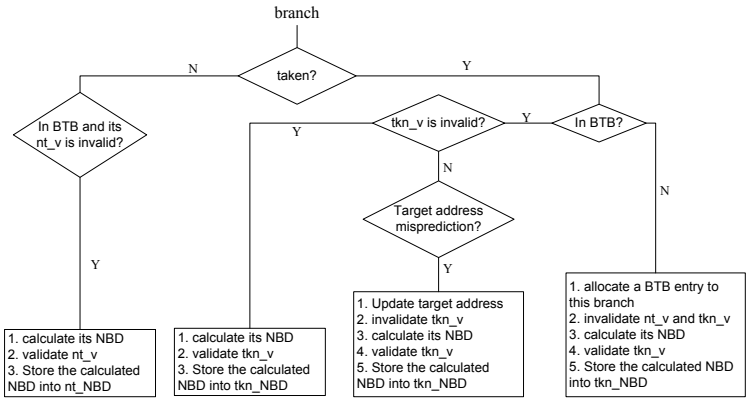


Fig. 2. NBD collection and NBDT management.

Fig. 2 shows the NBD collection algorithm for a branch instruction and its NBDT management, during EX stage. The detailed descriptions are:

- There is no operation if the branch is not taken and does not exist in BTB, since only the NBDs of the branch instructions existing in BTB are collected.
- If the NBD had been already collected, the repetitive collection operations are useless.

Therefore, NBD collection operation is performed only while the corresponding NBD field in NBDT entry is invalid.

- The collected NBDs in NBDT should be consistent with the branches in BTB. In other words, if a BTB entry is allocated to a new branch or an indirect jump updates its target address, the corresponding NBD fields should be invalidated. This invalidation prevents the next lookup filter from receiving the NBD of the replaced BTB entry or the replaced jump target. After the invalidation, the new NBD starts to be collected.

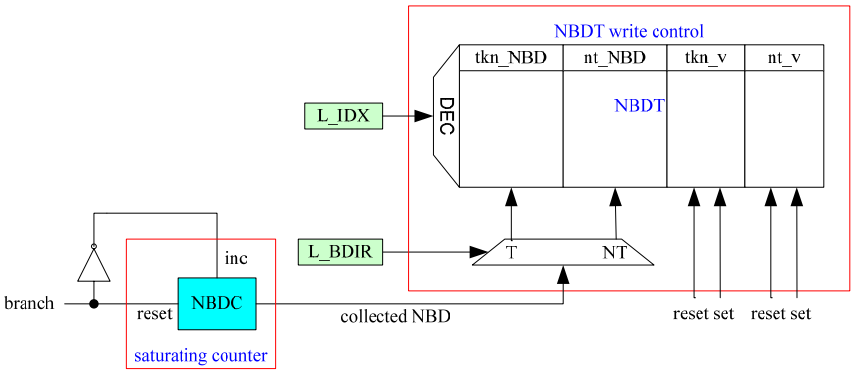


Fig. 3. The circuits of the NBD calculation and write controls of NBDT.

Fig. 3 shows the circuits of the NBD calculation and write controls of NBDT. The detailed descriptions are:

- The NBD calculation is performed using an *n*-bit saturating counter (NBDC). NBDC resets itself after a branch instruction (B1) is executed, and then, starts to increment itself for each following non-branch instruction. While the subsequent branch (B2) is executed, the value of NBDC is equal to that of B1's NBD, and then, this value is stored into NBDT if the NBD collection of B1 is allowed. Therefore, a last BTB index register (L_IDX) and a last branch direction register (L_BDIR) are added to preserve the BTB index and branch direction of the branch instruction until the subsequent branch is executed.
- The reset and set signals connected to valid fields of NBDT are used for valid field invalidation and validation, respectively.

Table 1 shows an example of the NBD collection process using a trivial loop code. During the first iteration, BNE is placed in BTB. During the second iteration, the NBD of BNE on taken path (2) is calculated and stored into NBDT.

Table 1. NBD collection process.

Executed Instructions	NBDC	tkn_NBD of BNE	tkn_v of BNE
L: ADD R0, R0, #1	1	X	0
CMP r0, #10	2	X	0
BNE L	0	2	1

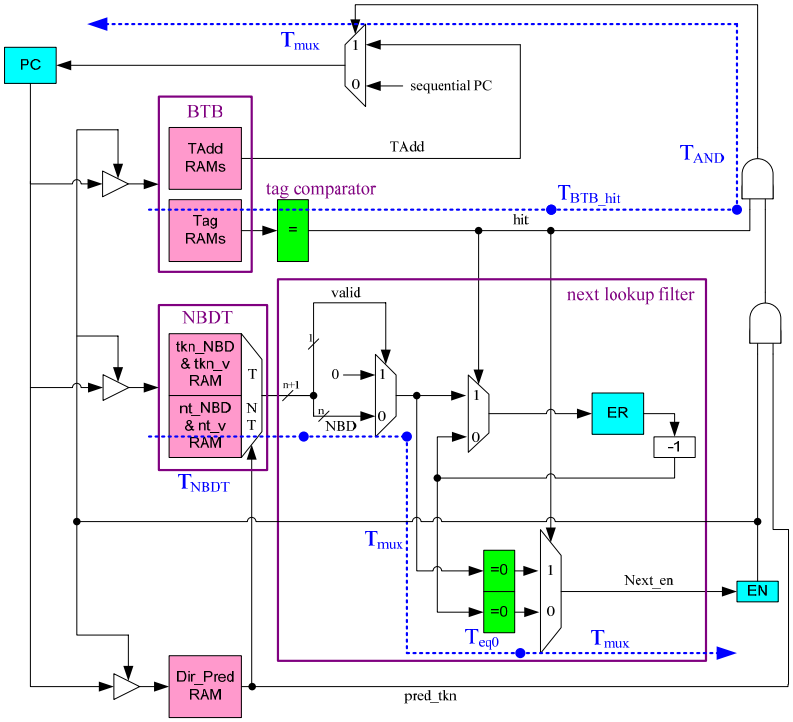


Fig. 4. Next lookup filter and its corresponding lookup filtering controls.

3.3 Filtering the Unnecessary Branch Predictor Lookups

Fig. 4 shows the next lookup filter and its lookup filtering controls. Its output signal (Next_en) determines whether the next lookup is filtered or not. In order to control the lookup filtering for each fetched instruction, Next_en is latched into EN every instruction cycle. In addition, an *n*-bit enable register (ER) is used to record the number of upcoming non-branch instructions before the subsequent branch instruction, on execution path. Therefore, with correct management of ER, Next_en can be simply generated by checking if ER is equal to zero.

The value of the ER is initialized with a default NBD value of zero. In order to find the NBD of the fetched branch instruction, NBDT and BTB are probed simultaneously. If the BTB lookup result is hit, then the fetched instruction is a branch. Both tkn_NBD and nt_NBD are read while NBDT is probed, yet only the one consistent with the predicted branch direction is selected. If the selected NBD is valid, then it is stored into ER; otherwise, ER is reset to the default NBD value of zero. In the non-branch instruction cycles, ER is decremented until its value reaches zero.

Table 2 shows the lookup filtering process using the same code example in Table 1. In this example, the NBD of BNE on taken path has already been collected in NBDT. While BNE is fetched, NBDT is probed. The NBD value of BNE on taken path (2) is selected (assume pred_tkn is 1), and latched into ER. Therefore, if BNE is actually taken, the branch predictor lookups of ADD and CMP are successfully eliminated.

Table 2. Lookup filtering process.

Fetched Instructions	ER	Next_en	EN
L: ADD R0, R0, #1	1	0	0
CMP r0, #10	0	1	0
BNE L	2	0	1

3.4 Handling of Incorrect Predictor Accesses

There are two possible impacts on performance and energy if the value of ER is incorrect. An insufficient ER value actuates the branch predictor too early and wastes energy, whereas an over-estimated ER value paralyzes the branch predictor while the next branch instruction is being fetched, degrading both performance and energy efficiency. Since the over-estimated ER case has more serious impact on energy and performance, it should be avoided first.

According to the proposed NBD collection and NBDT management, the collected NBDs in NBDT are consistent with the branches in BTB. In other words, if the predicted branch direction and the target address prediction is correct, ER receives a correct NBD value, a default NBD value of zero (the corresponding NBDT field is invalid), or an insufficient NBD value (the width of the corresponding NBDT field is insufficient). If a branch direction or target address misprediction occurs, then the instruction pipeline starts to fetch the new instructions from the correct branch direction or target. In this situation, the value of ER is incorrect and should be reset to avoid the performance loss due to the over-estimated ER.

4. EXPERIMENTS

The objective of these experiments is to evaluate the impacts of the proposed design on energy. The simulator and the benchmark are first described. The energy models are then defined. The simulation results are then provided. Finally, the proposed method and existing schemes are compared.

4.1 Simulator and Benchmarks

Simplescalar [14], an execution-driven, cycle-accurate simulator for modern processor core, is used as the processor and branch predictor simulator. Cacti 4.2 [15], an integrated cache access time, cycle time, area, leakage, and dynamic power model, is utilized for the power and energy estimation. The benchmark programs, comprising 12 integer and 13 floating point programs from SPECcpu2000 suite, are evaluated. All these programs are compiled using the Compaq Alpha compiler with the SPEC peak settings. Most of them could be executed within Simplescalar. The failed test programs are removed from the simulation. The provided reference inputs are used. All benchmarks are fast-forwarded past the first half-billion instructions, and full simulation is then performed for another 10-billion instructions.

Table 3 summarizes the detailed simulation configurations, where the width of tkn_NBD and nt_NBD (n) depends on the dynamic behavior of the benchmark programs.

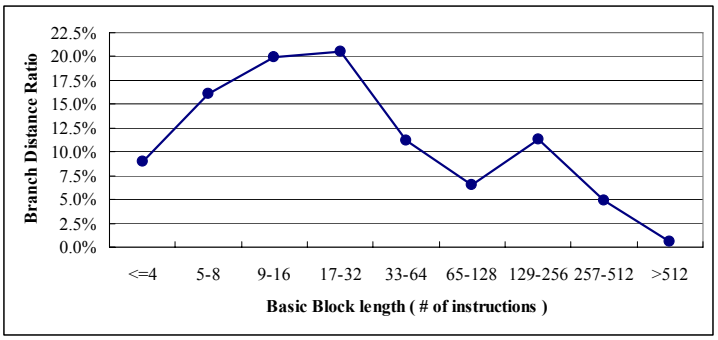


Fig. 5. Total branch distance ratio for each basic block size interval.

Table 3. Simulation parameters.

Processor Core	
Instruction Window	RUU = 64; LSQ = 32
Issue width	1 insts/cycle
Function Units	1 Int ALU, 1 Int mult/div 1 FP ALU, 1 FP mult/div, 1 mem ports
Memory Hierarchy	
L1 Instruction/Data Cache	32KB, 2-way, 32B blocks, 1cycle, wb
L2	Unified, 2MB, 4-way, 64B blocks, 11-cycle, wb
Memory latency	100 cycles
I/D TLB	128-entry, fly assoc.
Dynamic Branch Prediction	
BTB	512-entry, direct mapped
Direction predictor	Gshare 16K-entry
Branch Misprediction Penalty (B_MISP)	6 cycles
Our Design	
Default NBD	0
The width of tkn_NBD and nt_NBD (n)	9
Energy Estimations	
Technology	0.18μm
Clock Cycle Time	2ns

Fig. 5 shows the average total branch distance ratio (R_Total_BD) for each basic block length interval for SPECcpu2000. Typically, most basic blocks have the lengths shorter than 32. The results show that the total R_Total_BD of these short length basic blocks is only 66.29%, since a long basic block has more non-branch instructions than a short one. For the best tradeoff between the BTB energy reduction and NBDT energy, the value of n is determined to be fixed 9-bit. The detailed simulation results are shown in section 4.3.2.

4.2 Energy Model

Eqs. (1)-(11) give the energy models. Eq. (1) models the branch predictor energy with all possible impacts of the proposed design (E_{our_total}). E_{our_total} includes the branch

predictor energy ($E_{Predictor}$), NBDT energy (E_{NBDT}), extra logics and counters energy (E_{others}), and the chip-wide energy due to the skipped necessary predictor lookups (E_{stalls}). Eqs. (2)-(4) model the branch predictor energy with all possible impacts of PPD [2], Lazy BTB [13], and EIB [10], respectively. Since Lazy BTB use an array structure similar to NBDT, this array structure is denoted as NBDT_lazy here. Branch identification unit (BIU), which records the branch distance information of each hot spot, is also an array structure proposed in EIB.

Eq. (5) models $E_{Predictor}$ whose dynamic part can be modeled as the product of access counts (both lookups and updates are considered) of the accessed component and its average energy per access. Eqs. (6)-(10) model E_{NBDT}, E_{PPD}, E_{NBDT_lazy}, E_{BIU} and E_{others}, respectively. Since they are modeled in a similar way to $E_{Predictor}$, we ignore their descriptions here. Eq. (11) models E_{stalls} as the product of extra pipeline stall cycles and chip-wide energy per cycle, which is about 20 times of branch predictor access energy, since the branch predictor dissipates about 5% of the total processor power [1].

Table 4 lists the energy consumptions of single access of BTB, direction predictor, NBDT, NBDT_lazy, PPD, and BIU. The last four terms are the main power overhead sources of the proposed design, Lazy BTB, PPD, and EIB, respectively. The total energy comparisons among these designs are presented in section 4.4.

$$E_{our\_total} = E_{Predictor} + E_{NBDT} + E_{others} + E_{stalls} \quad (1)$$

$$E_{PPD\_total} = E_{Predictor} + E_{PPD} + E_{stalls} \quad (2)$$

$$E_{lazy\_total} = E_{Predictor} + E_{NBDT\_lazy} + E_{others} + E_{stalls} \quad (3)$$

$$E_{EIB\_total} = E_{Predictor} + E_{BIU} + E_{others} + E_{stalls} \quad (4)$$

$$E_{Predictor} = Count_{BTB\_acs} * E_{BTB\_acs} + Count_{dirp\_acs} * E_{dirp\_acs} + E_{Predictor\_static} \quad (5)$$

$$E_{NBDT} = Count_{NBDT\_acs} * E_{NBDT\_acs} + E_{NBDT\_static} \quad (6)$$

$$E_{PPD} = Count_{PPD\_acs} * E_{PPD\_acs} + E_{PPD\_static} \quad (7)$$

$$E_{NBDT\_lazy} = Count_{NBDT\_lazy\_acs} * E_{NBDT\_lazy\_acs} + E_{NBDT\_lazy\_static} \quad (8)$$

$$E_{BIU} = Count_{BIU\_acs} * E_{BIU\_acs} + E_{BIU\_static} \quad (9)$$

$$E_{others} = Count_{others\_acs} * E_{others\_acs} + E_{others\_static} \quad (10)$$

$$E_{stalls} = Count_{extra\_stalls} * 20 * (E_{BTB\_acs} + E_{dirp\_acs}) \quad (11)$$

Table 4. Single access energy (pJ) of the array structures used in this simulation.

BTB	Dir-Pred	NBDT	NBDT_lazy	PPD	BIU
12.43	4.31	5.41	2.50	2.09	4.68

4.3 Simulation Results

4.3.1 Analysis for the NBD predictions and remained predictor lookups

Table 5 lists all reasons for NBD mispredictions and the NBD misprediction ratio for each reason. The S1 mispredictions are unavoidable in any conservative design. The S2 and S3 mispredictions are dependent on the size, structure and replacement policy of BTB. In other words, the S1, S2, and S3 mispredictions are regarded as independent of our design effort. The energy trade-off between NBDT and the reduction for S4 mispredictions is shown in section 4.3.2.

Table 5. The reasons for NBD mispredictions.

Reasons	Descriptions	Ratio
S1	Upon first encountering of a branch instruction or a branch has not entered into BTB, there has been no NBD history in NBDT.	0.00017%
S2	After a branch instruction is replaced in BTB, it loses its NBD information.	22.83%
S3	Upon branch misprediction, the enable register is reset. It forces the branch predictor lookups for the following non-branch instructions.	6.64%
S4	The width of NBDT is not wide enough to record the full NBD.	0.26%

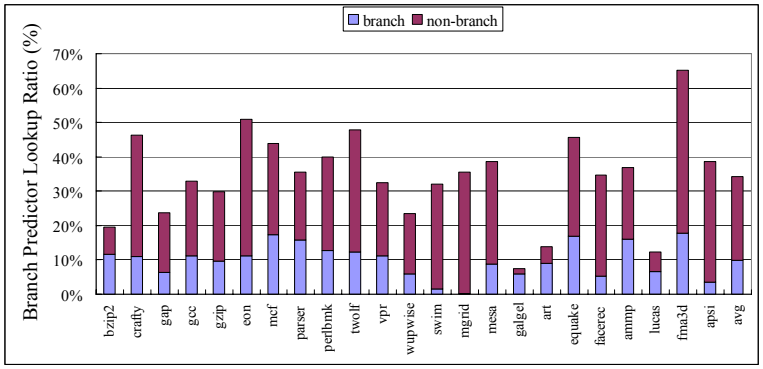


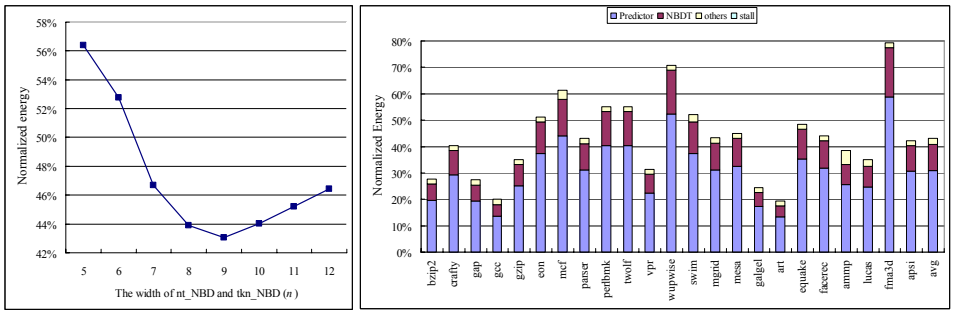
Fig. 6. Branch predictor lookup ratios.

On average, the total NBD misprediction ratio is 29.73%, where 0.00017%, 22.83%, 6.64%, and 0.26% are for S1, S2, S3, and S4 mispredictions, respectively. Some unnecessary branch predictor lookups are still remained due to the NBD mispredictions. Fig. 6 reports the predictor lookup ratios and their breakdown. The average lookup ratios are 9.54% and 20.58% for the branch and non-branch instructions, respectively.

4.3.2 Energy analysis

Fig. 7 (a) illustrates E_{our_total} with different values of n. All the energy numbers are normalized to that of a typical branch predictor. According to the evaluation result from CACTI, using one extra bit of tkn_NBD and nt_NBD consumes 2.87% more energy. This extra energy can be well compensated for by the saved energy in the basic blocks with branch distance shorter than 512. Therefore, analytical results show that $n = 9$ yields the largest energy reductions in branch predictor, on average 56.95%. The further partitioning of the E_{our_total} with $n = 9$ is shown in Fig. 7 (b). The results are:

- E_{our_total} is 43.05% on average where $E_{Predictor}$ occupies 31%.
- E_{NBDT} expends 9.88% on average. This is the main energy overhead of the proposed design, since the width of a NBDT entry is 20-bit.



(a) (b)

Fig. 7. (a) E_{our_total} with different values of n; (b) E_{our_total} distributions using $n = 9$.

- E_{others} occupies 2.17% on average due to the frequently set/reset the control registers and counters.
- E_{stall} is zero since the proposed design has no performance degradation.

4.3.3 Timing analysis

This section analyzes the timing effect of this design. As shown in Fig. 4, we suppose that the most time critical paths in the proposed filtering circuit and dynamic branch prediction are [NBDT → EN] and [BTB tag RAM → PC], respectively. Table 6 lists the delay of each component on the two paths, where the delay time is obtained from Cacti 4.2 and basic gate level analysis. The results show that the proposed filtering circuit fits into the processor pipeline without lengthening the critical paths of branch prediction.

Table 6. Results of delay analysis.

Critical path of dynamic branch prediction				
T_{BTB_hit}	T_{AND}	T_{mux}		Total
1.20 ns	0.06 ns	0.12 ns		1.38 ns
Critical path of lookup filtering circuit				
T_{NBDT}	T_{mux}	T_{eq0}	T_{mux}	Total
0.80 ns	0.12 ns	0.31 ns	0.12 ns	1.35 ns
Delay Evaluations				
1. T_{BTB_hit}, T_{NBDT} and T_{eq0} are obtained using Cacti 4.2 2. $T_{mux} = T_{eq0} * 2/5$ 3. $T_{AND} = T_{mux} * 1/2$				

4.4 Comparisons of the Existing Methods

4.4.1 High-level comparisons

Compared to the compiler hinted methods [3, 9, 10], the proposed method is a hardware implementation without any software support. The proposed method can easily be adopted in processor cores without the need to modify program codes, system soft-

ware, or ISA. The method proposed in [9] targets on VLIW processors. Therefore, we ignore its comparison here. The high-level comparisons between the proposed method and the methods in [3, 10] are:

- The methods in [3, 10] filter the unnecessary lookups only in hot spots, not the entire program, where the hot spots are identified by static profiling. However, which part of a program is hot spot is a runtime behavior which varies according to the program input data or user behavior. Changing the program input data or different user behavior may decrease the hot spot identification accuracy in [3, 10], resulting in lowering the power efficiency. This drawback does not exist in the proposed method due to its dynamic nature.
- The proposed method uses a runtime NBD collection, whereas [3, 10] uses a static one. The static method has the negative effect on program compatibility, compiler complexity, program size, instruction decoder complexity, and the overhead of an extra storage to record the NBDs, whereas the proposed design on the power overhead of NBDT and NBDC only.
- The NBD prediction coverage and accuracy of the methods in [3, 10] are mainly depending on the hot spot identification accuracy, whereas that of the proposed method on the branch direction predictor accuracy and BTB hit rate. A higher accuracy branch direction predictor and BTB benefit the proposed method achieving a better power reduction result.

Table 7. Comparisons of the low-power branch prediction techniques.

Tech.	Low-power schemes	Design targets	Lookup filtering
BPP	Avoid unnecessary sub-predictor accesses	Recently executed branches	Partial
SEPAS	Avoid unnecessary sub-predictor accesses and BTB updates	Well-behaved branches	Partial
PPD	Avoid unnecessary predictor lookups	Non-branches	Partial
Lazy BTB	Avoid unnecessary predictor lookups	Non-branches and not-taken branches	Full
Proposed	Avoid unnecessary predictor lookups	Non-branches	Full

Table 7 lists the comparisons of the proposed method and other hardware methods. The proposed method has the following features:

- For the low-power schemes, the proposed design filters the lookups to the entire branch predictor, not only to its partial component.
- The proposed method is fully independent of BPP and SEPAS. BPP and SEPAS target useless sub-predictor access filtering for most recently executed branches and well-behaved branches respectively, whereas the proposed method targets the predictor lookup filtering for non-branch instructions.

- The predictor lookup filtering operation (controlled by EN) is independent of the filtering signal generation. If the branch predictor lookup is not allowed, the full predictor access is stopped, not only the partial one.

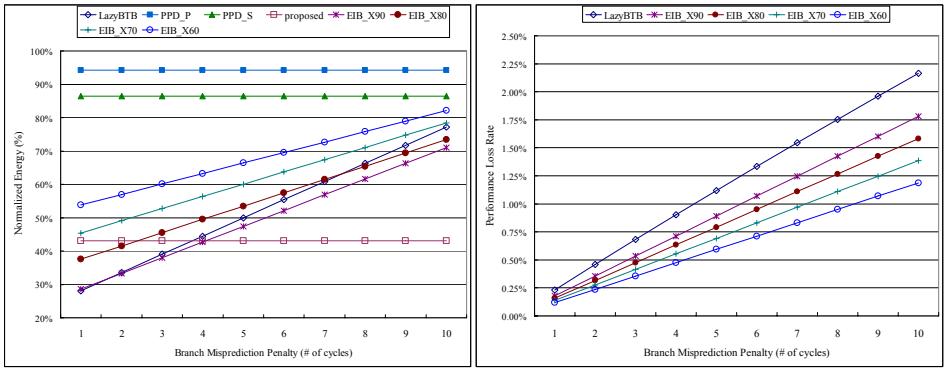
4.4.2 Simulation setups

PPD, Lazy BTB, and EIB are included in our simulation, since their goals are closer to those of the proposed method. The simulator includes a PPD table with the number of entries exactly identical to the number of I-cache entries and a NBDT-like table with the number of entries exactly identical to the number of BTB entries. In order to make a fair comparison between the proposed design and Lazy BTB, the width of NBDT_lazy is equal to an optimum value of 10 of the in Lazy BTB simulation. The simulation setups for EIB are:

- The most x% frequently executed basic blocks of each benchmark program are hot spots. According to the fact that 90% of the execution cycles are spent on 10% of the code, we use $x = 90$ as the best case of EIB (EIB_X90). The cases of $x = 80$, 70, and 60 (EIB_X80, EIB_X70, and EIB_X60) are also evaluated.
- In each hot spot, the BTB lookups are performed only for the predicted taken branches, whereas the direction predictor lookups are performed after the next branch address is calculated by BIU. In the other part of program, the BTB and direction predictor lookups are performed for each fetched instruction.
- We suppose branch distance is D, the number of instructions fetched per cycle is i, the latency to access the BIU and calculate the next branch address is t cycles, and the latency to access direction predictor is s cycles. According to the EIB design rule, if the time interval $D/i - t$ is shorter than s, a BTB lookup is conditionally performed based on the static branch prediction. According to the processor parameters listed in Table 3, the value of i is equal to 1. We assume $t = 2$ and $s = 1$. Therefore, in each hot spot, static branch prediction is used for the branches in the basic blocks with $D < 3$. A profile-based static branch predictor [16] is included for EIB simulation.
- The configuration of BIU is determined according to [10]. We assume this storage capacity of BIU is sufficient to store all the required information of the basic blocks in each hot spot. Moreover, we only consider the energy overhead of BIU lookup and the extra chip-wide energy due to the static branch prediction. The other energy overheads are ignored.

Both Lazy BTB and EIB have extra branch mispredictions. In a deep pipelined processor, the extra chip-wide energy due to the extra branch mispredictions is serous, since the deep pipelined processor has a heavy branch misprediction penalty (B_MISP). We intend to show the total energy effect of the extra mispredictions under different B_MISP. The steps to obtain the results of each design are:

1. Extract the number of the extra stall cycles from Simplescalar with various B_MISP.
2. Use the energy Eq. (6) to evaluate E_{stalls}.
3. Feed E_{stalls} into the total energy equation to get the results.



(a) (b)

Fig. 8. (a) Energy and (b) Performance comparisons of PPD, Lazy BTB, EIB and the proposed design.

Table 8. Branch predictor lookup ratio of PPD, Lazy BTB, EIB and the proposed design.

Lazy BTB	PPD	proposed	EIB_X90	EIB_X80	EIB_X70	EIB_X60
17.1%	73.39%	30.12%	18.59%	27.63%	36.68%	45.72%

4.4.3 Energy comparisons

Fig. 8 (a) displays E_{lazy_total}, E_{PPD_total} of PPD parallel scenario (PPD_P), E_{PPD_total} of PPD serial scenario (PPD_S), E_{EIB_total} and E_{our_total}. The corresponding branch predictor lookup ratios for these schemes are listed in Table 8. In PPD and our design, the energy values are independent to B_MISP, since both these two designs have no extra branch direction or target mispredictions. The results show that the proposed method out-performs PPD both on branch predictor energy and lookup ratio, since the proposed method targets on all non-branch instructions, whereas PPD on the non-branch instructions in branch-free cache lines only.

Lazy BTB saves more energy on $E_{Predictor}$ than the proposed BTB, since Lazy BTB not only skips the lookups that the proposed design does but also the lookups of not-taken branches. However, this aggressive design also skips some necessary lookups, resulting in an extra payment on chip-wide energy. Therefore, the results show that E_{lazy_total} is less than E_{our_total} only upon B_MISP is insignificant. While B_MISP is larger than 3, the proposed design becomes the better choice for low-power due to the characteristic of no performance degradation.

E_{EIB_total} and the corresponding branch predictor lookup ratio are mainly dependent on the ratio of the most frequently executed basic blocks (x) being hot spots and the value of B_MISP. Compared to the cases of $x \leq 70$, the proposed design has a better energy efficiency for all B_MISP. Even compared to EIB_X90, the proposed design is nevertheless better than EIB for the processors with B_MISP > 4.

4.4.4 Performance comparisons

Since both PPD and our method have no performance degradations, we show the relative performance loss rate only for Lazy BTB and EIB in Fig. 8 (b). The trends are

exactly as we would expect: A heavier B_MISP causes a higher performance loss rate in Lazy BTB and EIB. Therefore, both Lazy BTB and EIB are not suitable for deep pipeline processors.

5. CONCLUSIONS

This study addresses the issue of low-power dynamic branch prediction. A dynamic next branch distance generation and collection method is proposed to filter the useless branch predictor lookups. This method to reduce branch predictor energy can easily be adopted in processor cores without the need to modify program codes, system software, or ISA. Moreover, the proposed method is fully compatible with many other low-power branch prediction techniques such as banking [2], pipeline gating for low-confidence branches [2], BPP [11], and SEPAS [12].

High-parallelism computer architectures such as superscalar processor require very accurate branch prediction using large, multi-ported branch predictors. The waste of branch prediction energy in such systems is much more serious than in single-issue processors. Since the branches may be resolved out of order in such systems, implementing the proposed idea in is more challenging. Research of these topics is currently underway.

REFERENCES

1. S. Manne, *et al*., "Pipeline gating: speculation control for energy reduction," in *Proceedings of International Symposium on Computer Architecture*, 1998, pp. 132-141.
2. D. Parikh, *et al*., "Power-aware branch prediction: characterization and design," *IEEE Transactions on Computers*, Vol. 53, 2004, pp. 168-186.
3. P. Petrov and A. Orailoglu, "Low-power branch target buffer for application specific embedded processors," *IEE Proceeding Computers & Digital Techniques*, Vol. 152, 2005, pp. 482-488.
4. Y. C. Hu, *et al*., "Low-power branch prediction," in *Proceedings of the International Conference on Computer Design*, 2005, pp. 211-217.
5. S. T. Pan, *et al*., "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 76-84.
6. J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981, pp. 135-148.
7. T. Y. Yeh and Y. N. Patt,"Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, 1991, pp. 51-61.
8. P. Y. Chang, *et al*., "Alternative implementations of hybrid branch predictors," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995, pp. 252-257.
9. M. Monchiero, *et al.*, "Low-power branch prediction techniques for VLIW architectures: a compiler-hints based approach," *Integration*, *the VLSI Journal*, 2005, Vol. 38, pp. 515-524.
10. C. Yang and A. Orailoglu, "Power efficient branch prediction through early identifi-

cation of branch addresses," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006, pp. 169-178.
11. A. Baniasadi and A. Moshovos, "Branch predictor prediction: a power-aware branch predictor for high-performance processors," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, pp. 458-461.
12. A. Baniasadi and A. Moshovos, "SEPAS: a highly accurate energy-efficient branch predictor" in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2004, pp. 38-43.
13. Y. J. Chang, "Lazy BTB: reduce BTB energy consumption using dynamic profiling", in *Proceedings of the Conference on Asia South Pacific Design Automation*, 2006, pp. 917-922.
14. D. Burger and T. Austin, "The simplescalar tool set, version 2.0," *Computer Architecture News*, 1997, pp. 13-25.
15. D. Tarjan, *et al.*, "Cacti 4.2 web interface," in http://quid.hpl.hp.com:9081/cacti/.
16. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann Publishers, Inc., 2003, pp. 315.

Wei-Hau Chiao (喬偉豪) received the M.E. degree from National Chiao Tung University, Hsinchu, Taiwan, R.O.C. 2001. Currently, he is pursuing the Ph.D. degree in Computer Science and Information Engineering at National Chiao Tung University, Hsinchu, Taiwan, R.O.C. His research interests include computer architecture, Low-power techniques, AND java processors.

Chung-Ping Chung (鍾崇斌) received the B.E. degree from National Cheng Kung University, Tainan, Taiwan, R.O.C. in 1976, and the M.E. and Ph.D. degrees from Texas A&M University in 1981 and 1986, respectively, all in Electrical Engineering. He was a lecturer in electrical engineering at Texas A&M University while working towards the Ph.D. degree. Since 1986, he has been with the Department of Computer Science and Information Engineering at National Chiao Tung University, Hsinchu, Taiwan, R.O.C., where he is a professor. He was the visiting associate professor to the CS department of Michigan State University, the director of the Advanced Technology Center at the Computer and Communication Laboratories, the visiting professor to the School of Electrical and Electronic Engineering, Nanyang Technological University of Singapore, and the editor-in-chief in the information engineering section of the Journal of the Chinese Institute of Engineers. His research interests include computer architecture, parallel processing, and parallelizing compiler.