Lazy BTB: Reduce BTB Energy Consumption Using Dynamic Profiling

Yen-Jen Chang

Department of Computer Science
National Chung-Hsing University, Taichung, 402 Taiwan
Tel : 886-4-22840497 ext.918
e-mail : ychang@cs.nchu.edu.tw

Abstract- In this paper, we propose an alternative BTB design, called *lazy BTB*, to reduce the BTB energy consumption by filtering out the redundant lookups. The most distinct feature of the *lazy BTB* is that it dynamically profiles the taken traces during program execution. Unlike the traditional design in which the BTB has to be looked up every instruction fetch, by introducing an additional field to record the trace information, our design can achieve the goal of one BTB lookup per taken trace. The experimental results show that with a negligible performance degradation the *lazy BTB* can reduce the BTB energy consumption by about 77% on average for the *MediaBench* applications.

I. Introduction

It is well known that the *control hazards* caused by the branch instructions are the major bottleneck in developing high performance processors. The most common solution to the control hazards is to introduce a specific hardware table, called *branch target buffer* (BTB). A BTB is a small associative memory that caches recently executed branch addresses and their target addresses. The purpose of the BTB is to provide early branch identification and its target address before the instruction is decoded. Thus, traditionally, the BTB has to be always looked up during instruction fetch stage. Because the BTB is actually a set-associative cache which is usually implemented using arrays of densely packed SRAM cells for high performance, the energy consumption of the BTB is considerable. For example, the Pentium Pro consumes about 5% of the total processor energy in the equipped 512-entry BTB [1].

The related techniques for BTB energy savings can be classified into two categories. One is to reduce the energy consumption per BTB lookup [2] [3], and the other is to reduce the number of BTB lookups [4] [5]. Based on the observation that reveals most BTB lookups are redundant, in this paper we propose an alternative BTB design, called *lazy BTB*, which aims to reduce the number of redundant BTB lookups. The key idea behind our design is to look up the BTB only when the instruction is likely to be a taken branch. We augment the conventional BTB organization with an additional field, called *taken trace size* (TTS) field, to store the instruction number between the predicted target and the next taken branch, referred to as a *taken trace*. We have developed a dynamic taken trace profiling technique which can collect the sufficient taken trace information during program execution. According to the profiled data from the previous runs, our design can conditionally skip the BTB lookup to reduce the energy consumption.

The distinct features of our design are summarized as follows. First, the lazy BTB is a software independent technique. Without any compiler instrument, it can dynamically profile the taken traces during program execution. Second, the lazy BTB can achieve the goal of one BTB lookup per taken trace. It is more energy efficient than other related work [4][5] that achieve one BTB lookup per basic block, because a taken trace contains more than one basic block. We use *SimpleScalar* [6] to perform the execution-driven simulation of *MediaBench* [7], and the BTB energy consumption are estimated by using *CACTI* [8] configured with 0.18μm technology. The results show that by eliminating a large amount of redundant lookups, our design can reduce the total energy consumption of BTB lookups by 56%~88% with a 1.7% IPC penalty.

The rest of this paper is organized as follows. Section 2 presents our motivation and the characteristics of BTB lookups, which reveals most BTB lookups are redundant. In Section 3, we describe the proposed lazy BTB in detail, including the necessary hardware augmentations. Then, the experimental results, including the impact of our design on energy reduction and performance, are given in Section 4, and Section 5 offers some brief conclusions.

II. Branch Target Buffer (BTB)

Pipelining is the key implementation technique to the high performance processors. As introduced in [9], for most RISC processors the widely used pipeline model is the typical five-stage pipeline, which is composed of instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write back (WB) stages. When a *taken* branch is executed, the branch target address is normally not determined until the end of ID. This implies that the pipelined processor needs to know the path of the branch (in order to fetch the next instruction) before it has been determined. There are two possible solutions to this problem. One is waiting for the branch to finish the target address calculation and the other is to continue fetching the instructions, possibly from the wrong path. Either solution would interrupt the steady pipeline flow, called *control hazard*, which has been shown to cause a great pipeline performance loss.

To eliminate the control hazard, the processor must perform the following jobs by the end of IF stage: identifying the instruction as a branch, deciding whether the branch is taken or not, and the target address calculation. This requirement can be achieved by using the *branch target buffer* (BTB). The BTB is a set-associative memory that caches several types of information, including recently executed branch addresses, their corresponding target addresses, and the prediction information. Fig. 1 shows a

0-7803-9451-8/06/$20.00 ©2006 IEEE.

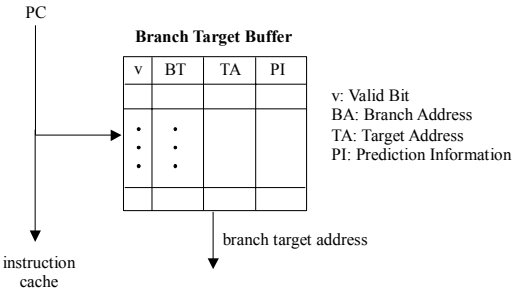

Fig. 1. A typical instruction fetch integrated with the BTB lookup.

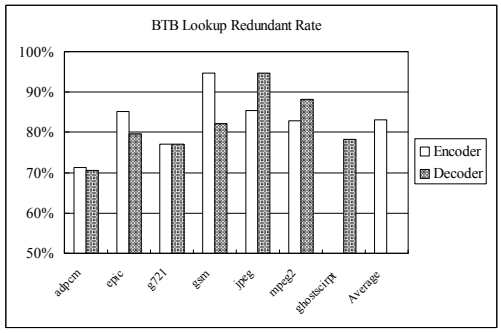

Fig. 2. BTB lookup redundant rate measured from *MediaBench*.

typical instruction fetch integrated with the BTB lookup. During IF stage, the instruction address, i.e., *program counter* (PC) value, is concurrently issued to the instruction cache and BTB. If a valid BTB entry is found for that address, then the instruction is a branch. According to the cached prediction information, if the branch is predicted taken, the BTB would output the corresponding target address to be used as the next PC. If the branch is predicted not taken, the processor continues fetching sequentially after the branch.

After the processor finishes executing the branch, it checks to see if the BTB correctly predicted the branch. If it has, all is well, and the processor can continue sequentially. If the branch was predicted incorrectly, the processor must flush the pipeline and begin fetching from the correct branch path. Then, the branch prediction information and branch target address (if changed) must be updated.

A. Characteristics of the BTB Lookups

Note that the BTB only caches the information regarding the recently executed branch instructions. Thus the BTB lookup is necessary only for the branch instructions. In the traditional BTB lookup mechanism, because the fetch engine has no sufficient information to distinguish the branch instructions, the BTB has to be looked up every instruction fetch, such that an overwhelming majority of the BTB lookups are redundant (or unnecessary). As indicated in [9], the branch instructions account for about 20% of the total executed instructions. It means that at least 80% of the BTB lookups are redundant. Fig. 2 shows the proportion of the non-branch instructions to the total executed instructions (referred to as *redundant rate*) measured from the execution traces of *MediaBench* benchmarks [7]. From this figure, the BTB lookup redundant rate is around 83% on average.

Unlike the conventional design where the BTB is always looked up every instruction fetch, motivated by most BTB lookups are redundant, we propose an alternative BTB design, called *lazy BTB*. The lazy BTB can dynamically profile sufficient information during program execution, and then use these profiled data to skip the BTB lookup conditionally. The goal is to look up the BTB only when the lookup is necessary. By filtering out most redundant BTB lookups, our design can effectively reduce the total energy consumption of the BTB.

B. Related Work

As described previously, we only survey the related work which target on reducing lookups to save energy dissipated in BTB. Petrov and Orailoglu [4] proposed *application customizable branch target buffer* (ACBTB), which is a software profiling technique. By utilizing the precise control-flow information of the application, the ACBTB is accessed only when a branch instruction is to be executed. Because the control-flow information must be extracted during compile/link time, their method is static and not applicable to the existing executable programs. In addition, a large hardware modification is necessary.

We can use predecode technique to test if the instruction is a branch, but the drawback is that the predecode bits only become available at the end of the instruction fetch stage. This would result in a significant performance penalty. In [5], Parikh et al. proposed a small hardware table, called *prediction probe detector* (PPD), to reduce unnecessary predictor and BTB accesses. The PPD can use compiler hints and predecode bits to recognize when lookups to the direction-predictor and BTB can be avoided. The drawback of this approach is that the PPD lookup must be performed before accessing the predictor and BTB. That would result in the extra power consumption and possible performance penalty.

III. Lazy BTB

This section gives the detailed description of the proposed lazy BTB design. We first discuss the BTB management, and then develop a dynamic profiling technique, which is critical to the lazy BTB. In addition, the necessary hardware augmentations are also provided.

A. BTB Management

The BTB management is concerned with the issue of entry allocation and replacement. For most microprocessors, the BTB is a valuable resource with limited size. Thus, instead of allocating entry for each branch, we only cache the branches which have the potential for improving performance. Because caching the untaken branches does not improve the performance and they are unlikely to be taken in the future [10], the allocation policy used in our lazy BTB is that we only allocate a new entry for a branch on its first taken execution. If no entry is available, then the replacement is necessary. As indicated in [10], LRU is good enough. It achieves the similar performance gain to their proposed MPP algorithm which is an elaborate replacement policy. Thus, the entry replacement used in the lazy BTB is the simple LRU.

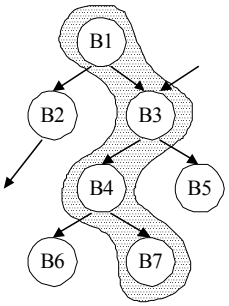

Fig. 3. An example of control flow graph (CFG). The shaded area is a taken trace.

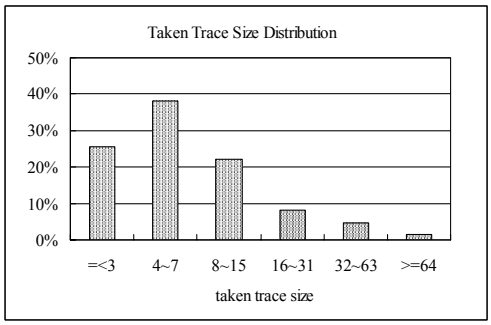

Fig. 4. Taken trace size (TTS) distribution measured from *MediaBench*.

B. Basic Block vs. Taken Trace

Fig. 3 shows a control flow graph (CFG), in which one node corresponds to one basic block. The *basic block*, by definition, is a sequential code that has no branch in except at the entry and no branch out except at the exit. Therefore, the branch instruction must be the last instruction of the basic block. Previous studies have shown that the average basic block size is usually small, especially for integer codes, it is around four to six instructions. As shown in Fig. 3, each basic block has two possible successors (caused by the *taken* and *untaken* path), but the correct path does not be determined until the codes are executed. Consequently, the basic block flow only depicts the static control structure of a program. It cannot reflect the dynamic behavior of a program.

In contrast to the basic block, we define a *taken trace* as the instruction stream between the two consecutive *taken* branches. A taken trace illustrates a snapshot of program execution. It can reflect the dynamic behavior of a program. A taken trace, by definition, contains more than one basic block. As shown in Fig. 3, the shaded area is a taken trace that is composed of basic blocks *B1*, *B3*, *B4* and *B7*. It means that the last instructions of *B1*, *B3* and *B4* are all untaken branches during program execution. Instead of *one BTB lookup per basic block*, the goal of our design is to achieve *one BTB lookup per taken trace*.

C. Hardware Augmentations

The lazy BTB design relies on the profiled taken trace from previous runs to skip the BTB lookup. A key issue in the realization of our design is how to profile the taken trace during program execution. Unlike the *ACBTB* technique presented in [4], which is based on the compiler profiling, our method is a hardware implementation without any software supports, including compiler. Before describing our design in detail, we first provide the necessary hardware augmentation.

(1) The conventional BTB has to be augmented with an extra field for each entry, called *taken trace size* (TTS) field, which is used to record the size of the following taken trace. The width of the TTS field must be large enough to accommodate most taken traces. Of course, the appropriate TTS field width depends on the dynamic behavior of the applications. Fig. 4 shows the average distribution of the TTS for *MediaBench* benchmark. For the best tradeoff between the energy reduction efficiency and hardware cost, the TTS field width is determined to be fixed 6-bit throughout this paper.

(2) Our design only performs the BTB lookup while the instruction is likely to be a taken branch. We need a counter, called *remainder trace length* (RTL), to indicate whether the currently fetched instruction locates within a taken trace or not. The initial RTL value is 0. When a BTB hit occurs, the RTL counter is set to the TTS value which is retrieved from the hit entry. Before looking up the BTB, if the RTL value is not equal to zero, then the currently fetched instruction is within a taken trace and is not a taken branch. Therefore, the BTB lookup can be skipped for energy saving. If the instruction is actually not a taken branch, then the RTL value is decreased by 1. In the other case, the RTL value is equal to zero, which implies that the currently fetched instruction is likely to be a taken branch. The BTB lookup is necessary for branch prediction and target address retrieval.

(3) An additional counter, called *trace size accumulator* (TSA), is needed to accumulate the taken trace size during program execution. The initial TSA value is 0 and increased by 1 every non-branch instruction execution. Until a taken branch is encountered, the TSA value is restored to the TTS field of the previous taken branch indexed by TE value (described below), and then it is reset to 0 to be accumulated until the next taken branch.

(4) Finally, in order to restore the TSA value to the corresponding taken branch, a temporal register, called *target entry* (TE), is needed to remember the index of the previous hit/allocated BTB entry during program execution. The initial TE value is 0. There are two cases where the TE value would be set. First, when we allocate a BTB entry for a new coming taken branch, the TE value has to be set to the allocated entry number. Second, if a BTB hit occurs and its prediction is correct, then the TE value has to be set to the hit entry number.

The hardware augmentations include an extra 6-bit field in BTB, three additional counters, and the necessary control circuitry. Except for the first one, the energy overheads caused by the remainder two are negligible to the energy consumption per BTB lookup.

D. Dynamic Taken Trace Profiling

Unlike the cache whose output must be accurate for correct program execution, the output of BTB is allowed to be inaccurate. The system can recover and continue by flushing any instructions fetched from the incorrect path before their results have been committed. This is the most important feature that guarantees our design can work well. Fig. 5 illustrates the dynamic taken trace profiling developed for the lazy BTB, which covers from the IF to EX stage. The

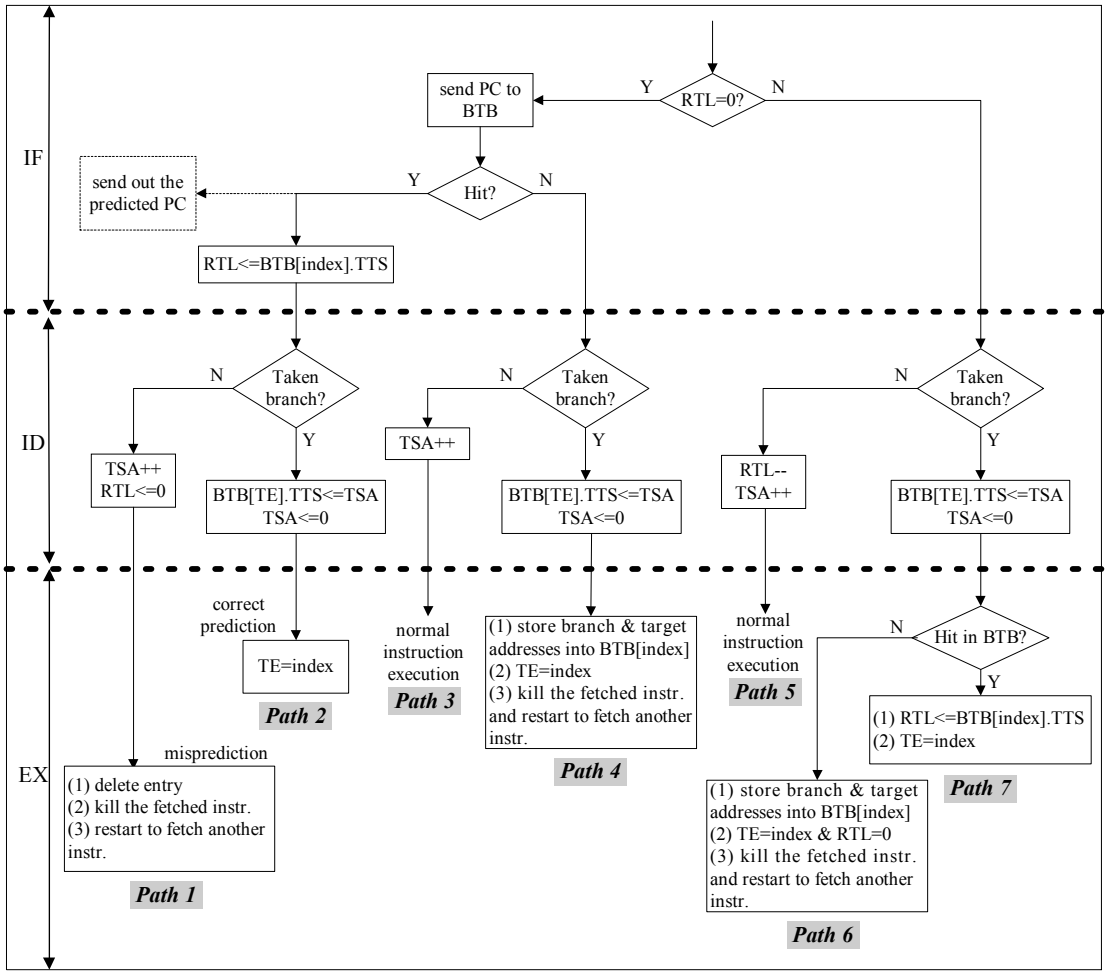

Fig. 5. The dynamic taken trace profiling technique developed for the lazy BTB design.

Table 1. The seven possible paths in the lazy BTB scheme.

Possible Paths	BTB Lookup	Hit/Miss	Prediction	Actual Branch	BTB Loop in EX	Penalty Cycles
Path 1	Y	Hit	taken	not taken	-	2
Path 2	Y	Hit	taken	taken	-	0
Path 3	Y	Miss	-	not taken	-	0
Path 4	Y	Miss	-	taken	-	2
Path 5	-	-	-	not taken	-	0
Path 6	-	-	-	taken	Y/Hit	3/4
Path 7	-	-	-	taken	Y/Miss	1/2

BTB lookup is performed (or skipped) during IF stage, and the actual branch result, i.e., the path and target address, would be determined in ID stage. If the prediction is correct, the execution continues with no stall. Otherwise, the recovery procedure for misprediction would be executed in the EX stage, which costs the performance penalty. From this figure, we can break the entire dynamic profiling scheme into seven possible paths. Their characteristics, including penalty cycles incurred by misprediction, are summarized in Table 1, and the detailed descriptions are provided as follow.

Path 1: In this path, because the instruction is found in the BTB and predicted taken, we can retrieve the corresponding TTS from the hit entry and set RTL to it during the IF stage. Next, in the ID stage, the branch is resolved and actually not taken. It is a misprediction case. The RTL value has to be reset to 0, and the TSA continues to accumulate the taken trace size. Note that the ID stage would be overlapped with the IF stage in the pipeline. In order to avoid hardware conflict the RTL value changes only in the second phase of IF stage, and the first phase of ID stage. In the EX stage, due to the misprediction, we have to kill the fetched instruction, delete the BTB entry, and restart to fetch the instruction from the correct path. The penalty cycles are 2 for this path.

Path 2: Unlike the path 1 which is a misprediction, the BTB prediction is correct in this path. As shown in Fig. 5, the RTL is set to the retrieved TTS value during the IF stage. Next, in the ID stage, the TSA value has to be restored to the previous taken branch entry indexed by TE, and then be reset to 0 to accumulate the following taken trace size. Finally, the TE value is set to the index of the hit entry in the EX stage. Due to the correct prediction, the penalty cycle is 0.

Path 3: This path is the execution flow of the non-branch instructions. Thus, we only increase the TSA value by 1 to accumulate the taken trace size during the ID stage. Of course, the penalty cycle is 0.

Path 4: Due to the BTB miss, the instruction is predicted as non-branch (or not taken), but it is resolved as a taken branch in the ID stage. Consequently, the TSA value has to be restored to the previous taken branch entry indexed by TE, and then be reset to 0 to accumulate the next taken trace size. Next, in the EX stage, we allocate a BTB entry for this taken branch. After storing the branch address and its target addresses, the TE value is set to the index of the allocated entry. Finally, we have to kill the fetched

Table 2. Major processor and penalty parameters used in our processor model.

Processor Configuration	
Issue width	1 intr. per cycle
Intruction window	2-RUU, 2-LSQ
Function units	1 Int ALU, 1 Int Mult/Div
	1 FP ALU, 1 FP Mult/Div
L1 instruction cache	16KB, 32-way, 32B blocks
L1 data cache	16KB, 32-way, 32B blocks
TLB (iTLB & dTLB	128-entry, 4-way
Branch perdictor	2-Level 1K-entry
BTB	512-entry, 4-way
Return address stack	8-entry
Penalty Parameters	
L1 hit latency	1 cycle
Branch misprediction	2 cycles
Memory access latency	8 cycles for the first chunk
	2 cycles for the rest of a burst access
TLB miss penalty	30 cycles

instruction, and restart to fetch the instruction from the correct path. The penalty cycles are 2.

Path 5: Similar to the path 3, this path is also the execution flow of the non-branch instructions. The only difference between the paths 3 and 5 is that the BTB lookup can be skipped in this path due to RTL<>0. Note that, besides increasing the TSA value by 1, the RTL has to be decreased by 1 in the ID stage. The penalty cycle is also 0.

Path 6: Due to RTL<>0 the BTB lookup can be skipped in the IF stage, and then the instruction is resolved as a taken branch in the ID stage. Thus, we first restore the TSA value to the previous taken branch entry indexed by TE, and then reset TSA to 0 to accumulate the next taken trace size. Next, in the EX stage, before allocating a BTB entry for this taken branch, in order to avoid duplicated allocation we have to check whether it is already in the BTB or not. In the path 6, because this taken branch is not found in the BTB, we have to allocate a BTB entry for this taken branch as the steps in the path 4. Note that the RTL has to be reset to 0. Because a BTB lookup is unavoidable in the EX stage, in the worst case it may be overlapped with the BTB lookup in the IF stage. Thus, the penalty cycles are 3 for the normal case, and 4 for the worst case.

Path 7: This path is almost the same as the path 6. The only difference is that the taken branch is already in the BTB. Thus, instead of allocating a BTB entry for this taken branch, we can retrieve the corresponding TTS from the existing entry and set RTL to it in the EX stage. The penalty cycles are 1 for the normal case, and 2 for the worst case.

We summarize the important features of the new BTB design. (1) In paths 1~4, due to RTL=0 the lookup is necessary as the conventional BTB design. In contrast, because RTL<>0, the BTB lookup can be skipped in paths 5~7, as shown in shaded columns in Table 1. (2) Compared to the conventional BTB, a significant energy savings come from the path 5 in our design. This is because we have enough information profiled during program execution to skip the BTB lookup conditionally. (3) The lazy BTB achieves one BTB lookup per taken trace. It is more energy efficient than the ACBTB [6], which realizes one BTB lookup per basic block.

Table 3. Path distributions for each benchmark.

Benchmark	path 1~4	path 5	path 6~7
adpcm_en	37.53%	59.41%	3.06%
adpcm_de	32.83%	64.63%	2.54%
epic_en	13.89%	85.68%	0.43%
epic_de	15.95%	83.39%	0.66%
g721_en	18.00%	81.11%	0.89%
g721_de	17.72%	81.42%	0.86%
gsm_en	15.00%	84.45%	0.56%
gsm_de	11.35%	88.50%	0.15%
jpeg_en	14.57%	84.92%	0.51%
jpeg_de	14.44%	85.07%	0.49%
mpeg2_en	30.79%	66.90%	2.31%
mpeg2_de	17.37%	81.81%	0.82%
ghostscirpt	13.17%	86.48%	0.35%
Average	19.43%	79.52%	1.05%

IV. Experimental Results

For the results presented in this study, we use *SimpleScalar* [6] toolset to model a baseline processor that closely resembles StrongARM processor [11]. It is a single-issue, in-order, pipelined machine with five stages. The major processor and penalty parameters are listed in Table 2. We use the execution-driven simulation to investigate the potential energy efficiency of the *lazy BTB* design, and its impact on performance.

A. Benchmarks

Because our baseline processor model is usually used in the embedded systems for multimedia or mobile applications, the input benchmark is *MediaBench* [7]. Unlike another popular benchmark, *SPEC2000*, which is a suit of general-purpose programs, the *MediaBench* is a suite of applications focus on multimedia and communications systems. Each benchmark of *MediaBench* has two separate programs: encoding and decoding.

B. Results and Discussions

Path Distributions: From Fig. 5, we know that the path distributions have a strong impact on the energy efficiency of the lazy BTB. Table 3 shows the path distributions for each benchmark. In this table, we divide the seven paths into three groups which are path 1~4, path 5 and path 6~7. Except for *adpcm_en*, *adpcm_de* and *mpeg2_en*, the percentage of path 5 is over 80% for all benchmark. This is because the BTB miss rates of *adpcm_en*, *adpcm_de* and *mpeg2_en* are higher than that of the other benchmarks. Because in our design the major energy savings come from the path 5, the large percentage of path 5 is preferred. In contrast, the BTB lookup is necessary in both path 1~4 and path 6~7, so their small percentages are favorable. Particularly, besides the energy consumption, path 6~7 further has a negative impact on the performance. This would be discussed below.

Total Energy Consumption of BTB Lookups: Because the organization of a BTB is essentially identical to that of a cache, we can use the *CACTI* tool [8], which is a widely accepted cache timing and power model, to estimate the BTB energy consumption. As listed in Table 2, the BTB organization is 512-entry 4-way. By using CACTI configured with 0.18μm technology, we obtained the energy

Table 4. Total energy consumption (measured in *mJ*) for both the conventional and lazy BTB designs.

	BTB_{Conv}	BTB_{Lazy}	Reduction
adpcm_en	290.8	126.9	56.35%
adpcm_de	239.2	90.7	62.09%
epic_en	25.3	3.7	85.25%
epic_de	3.2	0.6	82.73%
g721_en	131.9	26.1	80.22%
g721_de	128.5	25.0	80.56%
gsm_en	896.6	144.4	83.90%
gsm_de	305.7	35.6	88.35%
jpeg_en	48.6	7.6	84.41%
jpeg_de	12.3	1.9	84.58%
mpeg2_en	544.4	192.8	64.59%
mpeg2_de	82.2	15.6	80.99%
ghostscirpt	557.1	77.3	86.13%
Average	251.2	57.6	77.09%

consumption per lookup is about *0.484 nJ* and *0.492 nJ* for the conventional and lazy BTBs, respectively. Because in our design the BTB is augmented with an extra TTS (6-bit) field for each entry, the energy consumption per BTB lookup is slightly larger than that of the conventional BTB.

The metric used to evaluate the energy efficiency is the simple total energy consumption of BTB lookups. Table 4 shows the total energy consumption number in *mJ* for both the conventional and lazy BTB designs. Compared to the conventional BTB, one can immediately notice that the energy reduction would be an order of magnitudes. By filtering out most redundant BTB lookups, the lazy BTB can reduce the total energy consumption of BTB lookups by 56%~88% for *MediaBench*.

Performance Impact: The unit of performance measurement we use is *instructions per cycle* (IPC), calculated as the total number of execution instructions divided by execution cycles. Given that both the number of execution instructions and processor cycle time are constant, IPC is a direct measure of performance. Compared to the conventional BTB, from Fig. 5 we can see that only the paths 6 and 7 result in the extra penalty cycles. In this case, the BTB lookup is skipped, but the instruction is a taken branch actually. Thus, one BTB lookup has to be paid during the EX stage, that would decrease the overall performance. The paths 6 and 7 are, therefore, referred to as *unfavorable path*.

From previous discussion, we conclude that the negative impact of our design on the performance depends on the occurrence of unfavorable path. If most instructions follow the unfavorable path, then the lazy BTB would result in a significant decrease in IPC. Fortunately, the occurrence of unfavorable path is small enough. As shown in Table 3, the percentage of path 6~7 is about 1.05% on average, Therefore, the lazy BTB has a negligible degradation in performance. It can be seen from Fig. 6, which shows the IPC value for the conventional and lazy BTBs. Our design results in roughly 1.7% IPC degradation on average.

V. Conclusions

In this paper, we have proposed a low power BTB design, called *lazy BTB*. By using the developed dynamic taken trace profiling technique, the lazy BTB can achieve the goal of one BTB lookup per taken trace instead of one BTB lookup per basic block. The results show that without noticeable performance difference from the conventional BTB, our design can reduce the total energy dissipated in BTB lookups up to 88% for the *MediaBench* applications.

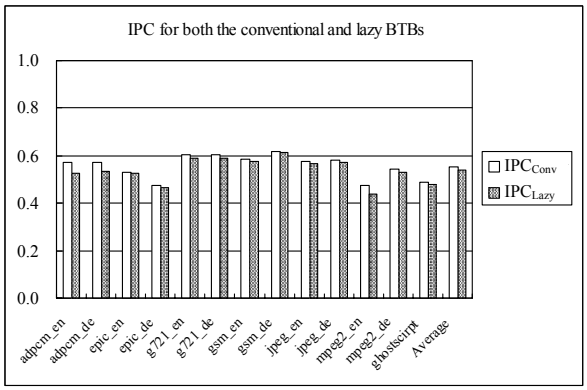

Fig. 6. The IPC value for the conventional and lazy BTBs.

References

[1] S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," in Proc. of International Symposium on Computer Architecture, 1998, pp. 132-141.

[2] B. Fagin, "Partial Resolution in Branch Target Buffers," IEEE Transactions on Computers, Vol. 46, No. 10, 1997, pp. 1142-1145.

[3] D. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," in Proc. of International Symposium on Microarchitecture, 1999, pp. 248-259.

[4] P. Petrov and A. Orailoglu, "Low-Power Branch Target Buffer for Application-Specific Embedded Processors," in Proc. of Euromicro Symposium on Digital System Design, 2003, pp. 158-165.

[5] D. Parikh, K. Shadron, Y. Zhang, and M. Stan, "Power-Aware Branch Prediction: Characterization and Design," IEEE Transactions on Computers, Vol. 53, No. 2, 2004, pp. 168-186.

[6] D.C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Computer Architecture News, 25 (3), pp. 13-25, June, 1997. Extended version appears as UW Computer Sciences Technical Report #1342, June 1997.

[7] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in Proc. of International Symposium on Microarchitecture, Dec. 1997, pp. 330-335.

[8] G. Reinman and N. P. Jouppi, "CACTI 2.0: An Integrated Cache Timing and Power Model," COMPAQ WRL Research Report, 2000.

[9] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," 3rd Ed., Morgan Kaufmann Publishers, Inc., 2003.

[10] C. H. Perleberg and A. J. Smith, "Branch Target Buffer Design and Optimization," IEEE Transactions on Computers, Vol. 42, No. 4, 1993, pp. 396-412.

[11] R. Witek and J. Montanaro, "StrongARM: A High-Performance ARM Processor," in Proc. of COMPCON, 1996, pp. 188-191.