Xie ZC, Tong D, Huang MK. A general low-cost indirect branch prediction using target address pointers. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(6): 929–946 Nov. 2014. DOI 10.1007/s11390-014-1480-3

A General Low-Cost Indirect Branch Prediction Using Target Address Pointers

Zi-Chao Xie (谢子超), Dong Tong (佟　冬), *Member*, *CCF*, *ACM*, and Ming-Kai Huang (黄明凯)

Microprocessor Research and Development Center, Peking University, Beijing 100871, China

Engineering Research Center of Microprocessor and System, Ministry of Education, Peking University Beijing 100871, China

E-mail: xiezichao@gmail.com; {tongdong, huangmingkai}@mprc.pku.edu.cn

Received August 5, 2013; revised July 25, 2014.

Abstract Nowadays energy-efficiency becomes the first design metric in chip development. To pursue higher energy efficiency, the processor architects should reduce or eliminate those unnecessary energy dissipations. Indirect-branch prediction has become a performance bottleneck, especially for the applications written in object-oriented languages. Previous hardware-based indirect-branch predictors are generally inefficient, for they either require significant hardware storage or predict indirect-branch targets slowly. In this paper, we propose an energy-efficient indirect-branch prediction technique called TAP (target address pointer) prediction. Its key idea includes two parts: utilizing specific hardware pointers to accelerate the indirect branch prediction flow and reusing the existing processor components to reduce additional hardware costs and power consumption. When fetching an indirect branch, TAP prediction first gets the specific pointers called target address pointers from the conditional branch predictor, and then uses such pointers to generate virtual addresses which index the indirect-branch targets. This technique spends similar time compared to the dedicated storage techniques without requiring additional large amounts of storage. Our evaluation shows that TAP prediction with some representative state-of-the-art branch predictors can improve performance significantly over the baseline processor. Compared with those hardware-based indirect-branch predictors, the TAP-Perceptron scheme achieves performance improvement equivalent to that provided by an 8 K-entry TTC predictor, and also outperforms the VPC predictor.

Keywords microprocessor, indirect-branch prediction, energy-efficient, branch target buffer

1 Introduction

Power has become a first-class architectural design constraint[1], forcing the processor architects to make their great effort to reduce or eliminate those unnecessary energy dissipations as many as possible. Computer architecture research has pursued two themes for higher performance: exploiting parallelism and using speculation. Branch prediction is the best-known example of speculation. If the branch prediction was wrong, useless instruction executions would waste energy. Previous researchers have made great contributions to improve the prediction accuracy of conditional branches[2-6]. Some aggressive kinds of branch predictors have already been used in newly published processors. Unlike conditional branch prediction, it is more difficult to achieve high prediction accuracy of indirect branches, for they require predicting target addresses instead of the branch directions[7-9]. Nowadays the indirect-branch prediction has become more important, for a great many programs are developed using object-oriented languages, such as C++, Java. Indirect branches in those programs are used to implement several programming constructs, including virtual function calls, switch-case statements and function pointers[10]. As shown in Fig.1, they are more critical to processor performance compared to conditional branches.

Though recent commercial processors have implemented the indirect-branch predictors such as Cortex-A15[11], AMD K10 families[12], they are usually specific predictors, which require a large dedicated storage to store the indirect-branch targets. This kind of predictors[3,8-9,13], called dedicated-storage-predictors in this paper, can predict targets rapidly[3,8,13]. Their storage requirement, however, takes up significant die area, which translates into extra power consumption. In the light of energy efficiency, such techniques are inefficient especially when the processor runs the applications that have rare indirect branches. To solve this critical issue, Kim *et al.* proposed a novel way called

Regular Paper

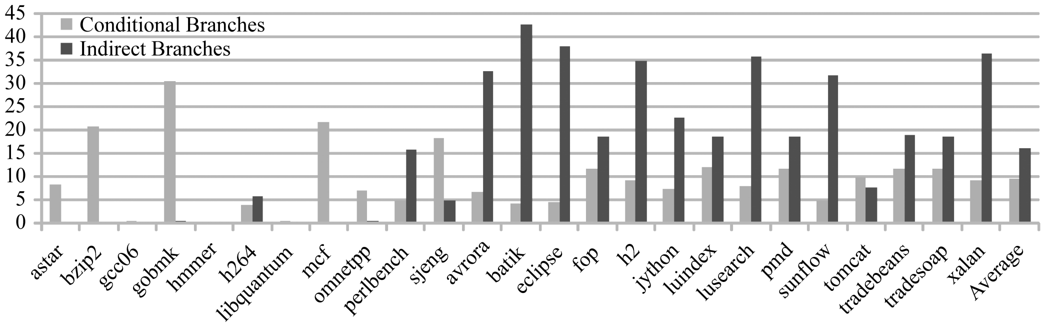


Fig.1. MPKI (mispredictions per kilo instructions) of conditional branches and indirect branches in SPEC CPU INT 2006 and DaCapo benchmarks using the conventional Perceptron branch predictor and the BTB. For DaCapo benchmarks, indirect branches are more critical to processor performance compared to the conditional branches[①].

VPC prediction[9] that reuses the existing branch predictor and the branch target buffer (BTB) to predict indirect branches. Though it has very low cost, it may take many cycles to finish one prediction, thus reducing its performance improvement[14] compared to that of the dedicated-storage-predictors. As the branch prediction has already consumed large amount of energy and is critical to the processor performance, what the architects need is an energy-efficient solution that spends similar time compared to that of dedicated-storage-predictors based on only existing branch prediction components. Such a solution should also be suitable to various kinds of branch predictors.

In this paper, we propose a fast, hardware-reusing indirect-branch prediction called target address pointer (TAP) prediction. Its key idea, shown in Fig.2, includes two parts: utilizing specific hardware pointers to accelerate the prediction flow and reusing the existing branch prediction components to reduce additional hardware costs.

• *Pointer Acceleration.* It constructs a certain set of specific hardware pointers, *target address pointers*, to establish the mapping from the indirect-branch occurrences to the stored indirect-branch targets.

• *Existing Hardware Reusing.* It reuses the existing branch predictor to distinguish various indirect-branch occurrences, and stores multiple indirect-branch targets in the existing hardware, such as the BTB.

When fetching an indirect branch, TAP prediction first uses the branch predictor to generate the bits of the target address pointer, and then selects the virtual address calculated from this pointer to index the predicted target. It is also a general solution to reuse various kinds of branch predictors. Our experimental results show that for four representative branch predictors: GShare[15], Perceptron[2], O-GEHL[4], and TAGE predictor[5], TAP prediction reduces the indirect-branch MPKI significantly over the commonly-used BTB-based prediction. The MPKI reduction results in performance improvement of 3.17%, 9.31%, 6.24% and 9.46%, respectively. Compared with previously proposed hardware-based predictors, TAP prediction with Perceptron predictor (TAP-Perceptron scheme) could achieve performance improvement equivalent to an 8 K-entry TTC predictor or a 2 K-entry Cascaded predictor, and also outperforms the VPC predictor by 2.39%.

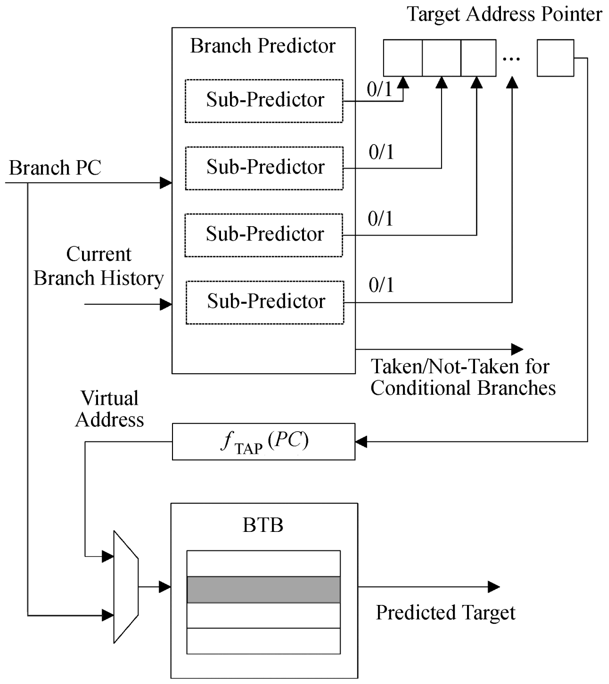


Fig.2. TAP prediction structure.

2 Related Work

BTB is used to predict indirect-branch targets conventionally[16]. However, the BTB records only the

①The running configuration is described in Section 5.

last-taken targets, which cannot distinguish various indirect-branch occurrences, resulting in poor prediction accuracy[8].

Previously proposed techniques of indirect-branch prediction have been proved to predict targets accurately. They can be classified into two types: pure hardware methods and hardware-software cooperative methods. The pure hardware methods use only hardware components, predicting indirect branch targets dynamically depending on the recorded branch histories. The hardware-software cooperative methods modify compilers to guide the indirect-branch prediction using related values or path information.

One sort of pure hardware methods is the dedicated-storage-predictor. Chang *et al.* first proposed Tagged Target Cache (TTC) to use branch history information to distinguish differences among indirect-branch occurrences[8]. Its concept is similar to that of a two-level branch predictor[15]. Each TTC entry contains a target address and a tag field. When fetching an indirect branch, the TTC predictor is indexed using the XOR result of the branch address (PC) and the global branch history register (GHR) to provide the predicted target. When the indirect branch retires, the corresponding TTC entry is updated with the actual target address. Driesen and Hölzle proposed another dedicated-storage-predictor, a cascaded predictor, by combining multiple target predictors[7,17]. A simple first-stage predictor is used for the easy-to-predict (single-target) indirect branches, whereas a complex second-stage predictor is used for the hard-to-predict indirect branches. The IT-TAGE predictor[5] proposed by Seznec and Michaud is conceptually similar to a multistage cascade structure. Its mechanism employs a base predictor and a number of tagged tables indexed by a very long GHR, PC, and path history. The predicted target comes from the table with the longest history, which the access hits. The dedicated-storage-predictors consume large amount of extra energy induced by the extra target-address storage.

An alternative to the dedicated-storage-predictor is VPC predictor that uses existing branch prediction components to reduce additional cost and energy consumption[9]. VPC prediction treats an indirect branch with T targets as T virtual direct branches, each with its own unique target address. When fetching an indirect branch, VPC prediction accesses the branch predictor iteratively. One of T virtual direct branches is predicted as well as a conditional branch in each time of iteration. This iterative process stops when a virtual direct branch is predicted to be taken, or a predefined maximum iteration number is reached. This technique, however, may take many cycles to accomplish the indirect-branch prediction, thus reducing its performance efficiency[14]. Our previous work[18] proposes to use pointers to accelerate the indirect branch prediction, but those pointers are quite different with the pointers from the generation to the usage in this paper. This paper uses branch predictor to generate the pointers, which is actually utilizing the prediction mechanism to make the indirect branch benefit from the high-performance branch predictors rather than simply storing them in the PHT. Thus the technique in this paper is more extensible and accurate.

With the help of the compiler and ISA modification, the hardware-software cooperative method improves processor performance significantly as well. Joao *et al.* proposed dynamic predication for hard-to-predict indirect branches (DIP)[14]. The compiler identifies the indirect branches that are suitable for predication along with their control-flow merge (CFM) points. When fetching a hard-to-predict indirect branch, the processor predicates the instructions between T targets of the indirect branch and the CFM point, thereby increasing the probability of fetching from the correct target path at the expense of executing more instructions. Farooq *et al.* proposed a Value Based BTB Indexing (VBBI) technique, a novel research done with compiler assistant[19]. For each static hard-to-predict indirect branch, the compiler identifies a hint instruction whose output value strongly correlates with the indirect-branch target. At run time, multiple indirect-branch targets are stored and subsequently accessed from the BTB according to different indices, which are computed using the branch addresses and the hint instructions' output values.

3 Motivation

As energy efficiency has become a first-class metric in processor designs, processor architects must consider the cost-benefit trade-offs, choosing those structures that achieve high performance per unit energy[20]. In such a way, improving performance based on the existing components is a good choice of energy-performance trade-offs.

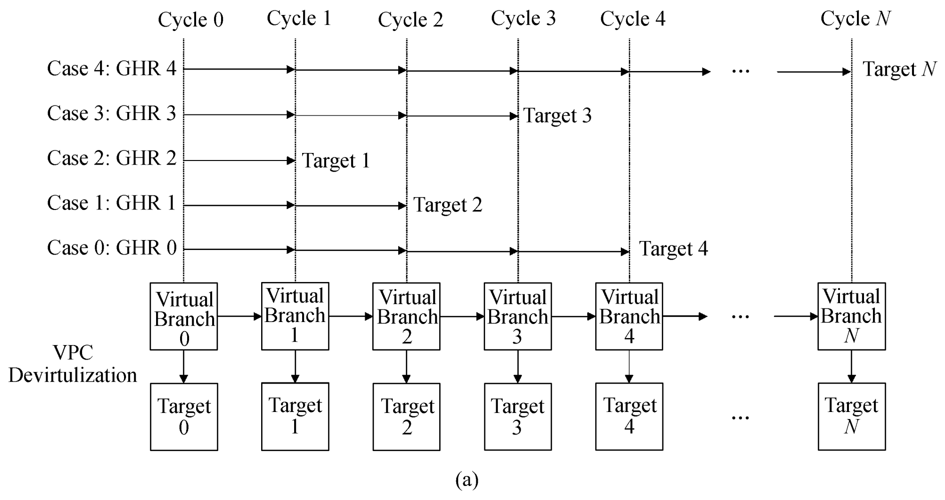
Let us first analyze VPC prediction, the indirect branch prediction that almost uses the minimal extra hardware costs. In fact, it organizes the targets of an indirect branch as a linked list. It adopts the same way as the linked list accessing to predict each target sequentially. VPC prediction differentiates various indirect-branch occurrences by recording their access sequence of the target linked list. Its effects, however, are partly reduced by such sequential access, especially in the case where an indirect branch has many target addresses.

In order to accelerate the prediction flow and maintain the low hardware costs, we propose a completely novel technique, which organizes the targets of an indirect branch as an array instead of a linked list. In such a way, when predicting an indirect branch, the target will be obtained easily using the index of the array, which is much faster than that of accessing a linked list sequentially. Its running time is $O(1)$ rather than $O(n)$. The concept comparison of these two methods is shown in Fig.3.

Our previous work called TAP prediction proposed to use pointers for indirect branch prediction[21]. In this paper, we make the following extensions.

1) We extend the TAP prediction mechanism to support the cases where an indirect branch has more than 16 targets. We propose to use iterative prediction algorithm for those situations, which usually occur in programs written in object-oriented languages.

2) We develop the TAP prediction to use the last-taken targets provided by the BTB, which has about 40% prediction accuracy of indirect branches. By using the BTB results, TAP prediction no longer changes the conditional-branch prediction flow or does not require any additional precoding techniques. In addition, this mechanism can accelerate the speed of TAP prediction flow and training flow as well.



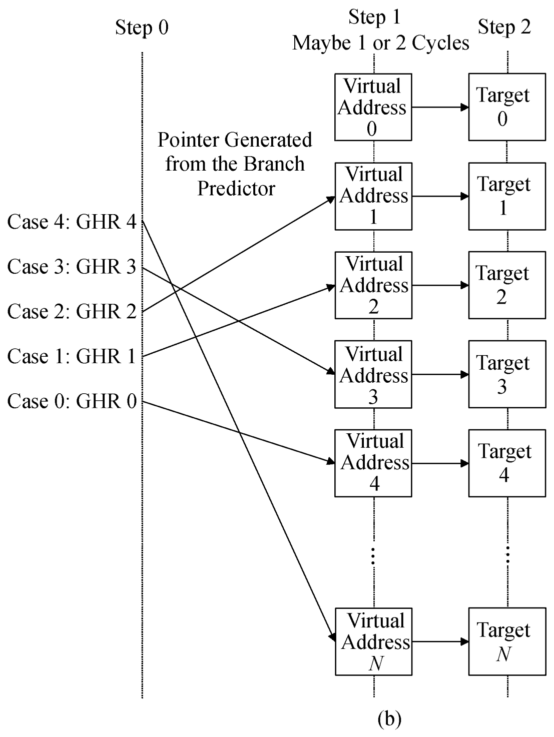


Fig.3. Concept of VPC prediction and TAP prediction. (a) VPC prediction procedure. (b) TAP prediction procedure.

3) We extend the TAP prediction to use TAGE predictor, which is one of the best high-performance branch predictors. It also achieves as good performance as that of other branch predictors.

4) We regard the TTC predictor as not only a comparison target, but an assistant structure for TAP prediction. By redefining TTC, it can further improve the TAP prediction performance.

5) We conduct the experiments in the gem5 simulator rather than the SimpleScalar simulator. The gem5 simulator can provide full-system evaluations. As a result, we can run various real applications, such as the DaCapo benchmarks, to fully exploit and evaluate our technique.

6) We change the ISA in our experiments from Alpha to x86, which makes TAP prediction compared with other techniques fairly.

7) We evaluate not only the indirect-branch-sensitive benchmarks, but other benchmarks in [21] as well. From the results, we show that TAP prediction has no adverse impacts on those indirect-branch-insensitive programs.

8) We evaluate the influence of energy consumption with TAP prediction. This shows that TAP prediction could not only improve performance, but reduce energy waste as well.

9) We evaluate the TAP prediction in TSMC 28nm technology rather than the 65 nm technology in that conference paper.

4 Target Address Pointer Prediction

4.1 Idea of TAP Prediction

The key idea of TAP prediction is to predict a pointer, which points to an indirect-branch target stored in the existing components such as BTB, rather than to predict a target address directly. In fact, such pointers can be treated as the indices of the indirect-branch target arrays that are organized by the TAP prediction. Using pointers has three benefits: first, it is easy to access the pointed data when the pointers are obtained; second, it is possible to store the pointed data in the existing storage, for those data can be stored in distributed memory locations; third, the pointer can be obtained in various ways, thus it can be adapted to different types of branch predictors. Based on such three advantages, TAP prediction uses target address pointers as the intermediate representations of those indirect-branch targets, separating the indirect-branch prediction flow into two steps, as shown in Fig.4: first generate a target address pointer according to each indirect-branch occurrence distinguished by branch histories (indirect-branch occurrences mapping), and then use this pointer to fetch the predicted target (targets mapping). As the way of occurrence mapping is partly determined by the target mapping, we explain the latter one first.

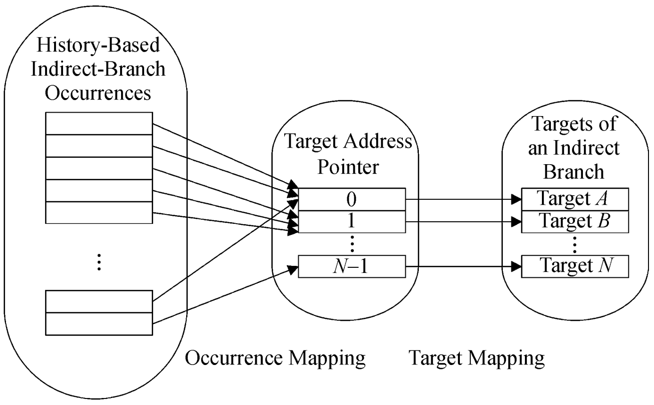


Fig.4. Mappings of TAP prediction.

4.1.1 *Target Mapping*

TAP prediction reuses existing processor components to store indirect-branch targets. The most common method is to save those targets in BTB. During a program executing, TAP prediction dynamically allocates some BTB entries, called target-entries, to store the encountered targets of indirect branches. Each target-entry is indexed by a virtual address calculated from the obtained target address pointer during the prediction. To the processors that already contain specific indirect-branch target storage such as TTC, TAP prediction can allocate the target-entries in TTC instead of BTB. TAP prediction also requires a sort of entries, called allocation-entries, to record the target-entry allocation information of each indirect branch. Each bit in such entries represents whether the corresponding target-entry has been already allocated for this indirect branch. Multiple allocation-entries can be employed to record a great number of target-entries. Allocation-entries are indexed by some special target address pointers, which are defined by hardware designers. An example of target mapping is illustrated in Fig.5.

The target address pointers are treated as the function pointers to call the function $f_{\mathrm{TAP}}(PC)$, which generates different virtual addresses to index the target-entries. $f_{\mathrm{TAP}}(PC)$ could have multiple simple implementations to distribute the generated virtual addresses widely, thereby reducing their interference with each other and with the conditional branches. In our paper, the virtual address is generated by $PC[\mathit{highest}:7]$ XOR $\mathit{Constant}$, $PC[6:0]$, where $\mathit{Constant}$ is “1010 . . . 1010”.

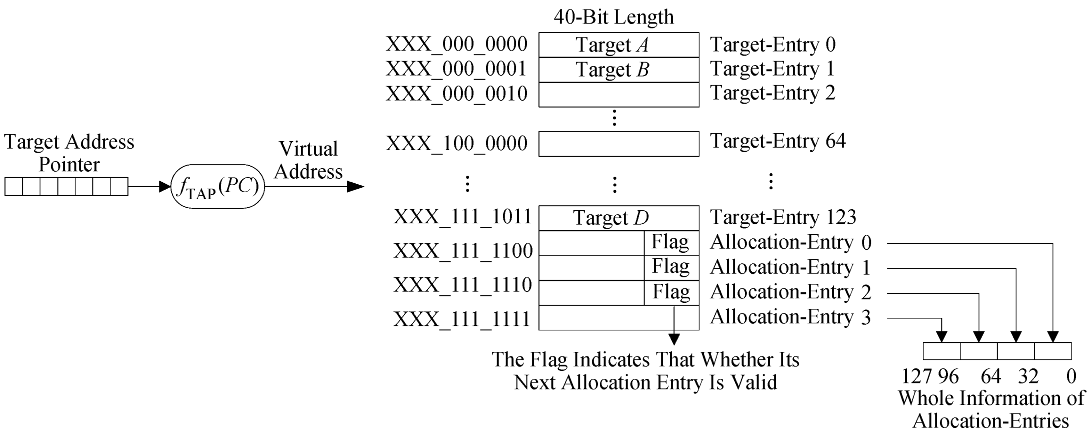


Fig.5. Example of target mapping where the target address pointer is defined as 7-bit. As a result, there are 124 target-entries and 4 allocation-entries. Both target-entries and allocation-entries are distributed in the BTB indexed by their virtual addresses. Here, the high part of the virtual address is computed by $f_{\rm TAP}(PC)$, which we use "XXX_" instead. Four allocation-entries are concatenated to form the whole allocation information of target-entries.

4.1.2 *Indirect-Branch Occurrences Mapping*

TAP reuses the existing branch predictor to generate the target address pointers. In such way, the indirect-branch occurrences can be recognized and differentiated by the existing high-performance branch predictors, reducing extra hardware costs. There are two issues which have to be addressed: 1) how to recognize indirect branches, and 2) how to differentiate various indirect-branch occurrences.

Recognizing an Indirect Branch[②]. Processor accesses the branch predictor and BTB simultaneously at the fetch stage conventionally, which provides the branch direction and the target address respectively. As TAP prediction also uses the last-taken target in BTB for indirect branches (please see the details in Subsection 4.2), it remains this traditional procedure, only adding a flag bit to each BTB entry indicating the encountered branch is an indirect branch. TAP prediction enters the indirect-branch procedure only when the corresponding flag is found to be set.

Differentiating the Indirect-Branch Occurrences. To adapt to various kinds of branch predictors, TAP prediction adopts a simple but efficient method to generate target address pointers, i.e., it reuses the existing branch predictor to construct several small predictors, each of which predicts one bit of the target address pointer, and accesses them iteratively if necessary. These small predictors, called sub-predictors, perform as well as the original predictor, but use fewer branch histories. The prediction result of each sub-predictor, "Taken/Not-Taken", is defined as one bit of the target address pointer, "1/0". For example, if we divide the original branch predictor into four sub-predictors, target address pointer which has more than 4 bits can be obtained by accessing the branch predictor twice. During the predictor updating, each sub-predictor treats the corresponding bit of the target address pointer as its training goal. Note that constructing sub-predictors does not change the conditional branch prediction flow.

This paper chooses four representative branch predictors — GShare[15], Perceptron[2], O-GEHL[4], together with TAGE Predictor[5-6] — to implement the occurrences mapping. Their ways of constructing sub-predictors are as follows (detailed implementations are listed in Table 1).

TAP-GShare. To this kind of predictor that uses SRAMs to construct PHTs, the SRAMs are divided into four groups. Each group constructs a sub-predictor with one-quarter of PHT entries. A multiplexer is added to each SRAM to select the corresponding index in indirect-branch prediction.

TAP-Perceptrons. Shown in Fig.6, for this computation-based predictor, four new adders are employed to calculate the results of the sub-predictors. The weights read from the predictor table are simultaneously sent to the sub-predictors to generate the target address pointers in indirect-branch prediction.

TAP-O-GEHL. As this predictor uses both PHTs and computations to predict branch directions, a sub-predictor is constructed with one-quarter of PHTs, additional adders, and corresponding multiplexers. According to different fetching cases, O-GEHL selects either the original mechanism to predict branch directions or the sub-predictors to produce the target address pointers.

[②]If predecode has been done during the ICache line fill and its result has been stored in the ICache, the implementation of this part can be ignored.

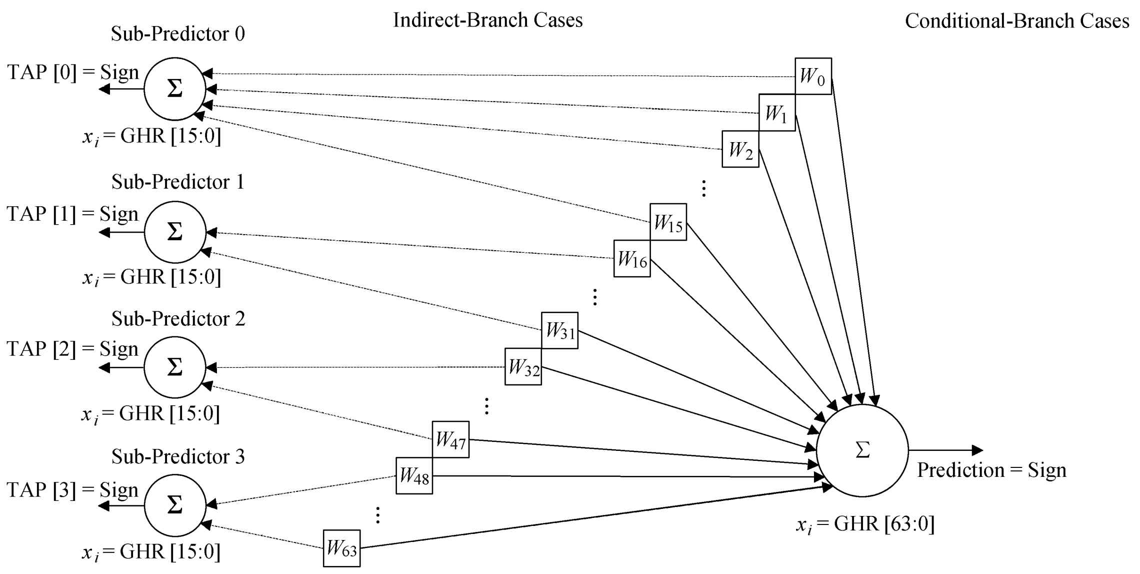


Fig.6. Sub-predictor implementation of TAP-Perceptron scheme. In each access of indirect branch cases, only 4 bits of target address pointers will be obtained. Because of this, a 7-bit target address pointer should be obtained by accessing the branch predictor twice.

Table 1. Sub-Predictor Implementations of Various Branch Predictors

Predictor Scheme	Structural Description
TAP-GShare	Original configuration: 32 K-entry GShare PHT, 1-cycle prediction delay
	Construct sub-predictors: divide the original PHT into 4 small PHTs
	Each sub-predictor has 8 K-entry PHT indexed by XOR result of the PC and low 13-bit of GHR
	The higher bit of each obtained saturating counter represents the bit of the target address pointer
TAP-Perceptron	Original configuration: path-based perceptron, 64 KB table, using 64-bit GHR as inputs. Each entry has 64 weights (w_i). 2-cycle prediction delay
	Construct sub-predictors: divide the 64 weights (w_i) into 4 parts. Each part constructs a sub-predictor, using the low 16-bit of GHR and the segmented weights to calculate the output. The output is redefined as one bit of the target address pointer
TAP-O-GEHL	Original configuration: 64 K-bit, including 8 predictor tables. Tables are indexed with hash functions of branch history, path history, and PC. The set of used global history lengths forms a geometric series, i.e., $L(j) = \alpha^{j-1}L(1)$. Its result is the sum of each tables' output. 2-cycle prediction delay
	Construct sub-predictors: divide 8 predictor tables into 4 sub-predictors. Each sub-predictor has 2 predictor tables and captures the recent 20 global branch histories. All tables are extended with two read ports (or implemented with 2 SRAM cells) to provide 4 signed counters for each sub-predictor. Each sub-predictor performs as well as the O-GEHL predictor to generate one bit of the target address pointer
TAP-TAGE	Original configuration: 64 K-bit, including 8 predictor tables. Tables are indexed with hash functions of branch history, path history, and PC. The set of used global history lengths forms a geometric series, i.e., $L(j) = \alpha^{j-1}L(1)$. The predicted result comes from the table with the longest history, which the access hits 2-cycle prediction delay
	Construct sub-predictors: divide 8 predictor tables into 4 sub-predictors. Each sub-predictor has 2 predictor tables and captures the recent 20 global branch histories. All tables are extended with two read ports (or implemented with 2 SRAM cells) to provide 4 signed counters for each sub-predictor. Each sub-predictor performs as well as the TAGE predictor to generate one bit of the target address pointer

TAP-TAGE. The separation of this kind of predictor performs as similarly as that of O-GEHL predictor, only using PHT selections instead of those computation logics.

The benefits of constructing sub-predictors from original branch predictor are as follows. First, TAP prediction transforms the prediction of target address pointers into the direction predictions of multiple sub-predictors, and thus the improvements in conditional-branch prediction accuracy can also benefit TAP prediction. Second, the implementation is quite simple. It only needs to copy the logic of the existing branch predictor, including both prediction logic and updating logic. Third, it is quick to obtain the prediction results because sub-predictors work simultaneously. The number of accesses in the TAP branch predictor is $(\log_2 N)/k$, where N is the maximum possible targets number of an indirect branch, and k is the number of constructed sub-predictors. It is much faster than that of the VPC's serial prediction which is $O(N)$.

4.2 TAP Prediction Flow

According to previous descriptions, TAP prediction requires two steps to get the predicted indirect-branch target with the help of a target address pointer. Though this is proved to predict target accurately, its prediction is slower than that of the conventional branch prediction flow. In order to accelerate the prediction, TAP prediction uses the last taken target stored in BTB, which corresponds to its PC address, as well.

Fig.7 is a diagram to illustrate the prediction stages of TAP prediction. In the first cycle of fetching an instruction, TAP prediction performs as the same as conventional instruction fetch: both the BTB and the branch predictor are accessed simultaneously. If the BTB hits and it is predicted as taken, the target stored in the BTB entry (last-taken target) is issued to the pipeline. Meanwhile, the corresponding flag is checked to investigate whether it is an indirect branch. If this branch is an indirect branch, TAP prediction forces the processor's fetch unit to access the branch predictor, which is accessed as several sub-predictors, once more (using a transformed branch history such as left-shifting the branch history with zero) to generate the target address pointer. It may take more than one cycle to obtain the whole target address pointer if the defined pointer's length is greater than the number of sub-predictors. After calculating the virtual address through $f_{\rm TAP}(PC)$, in the next cycle, TAP prediction accesses the BTB (or TTC) using the obtained virtual address for the predicted indirect-branch target. If it misses, TAP prediction stalls fetching instructions until the actual target is calculated in the pipeline. If it hits, the stored target is compared with the previously issued target. If they are different, the newly obtained target is issued to the pipeline, and the previously issued target must be canceled. Otherwise, the processor continues using the previously issued target.

4.3 TAP Training Flow

The training flow of TAP prediction is illustrated in Fig.8. For conditional branches, TAP prediction does not change their original training procedures. For indirect branches, though it appears a little complex, it will be finished in a few cycles in most cases (the results are shown in Section 6).

For correct predictions of indirect branches, TAP prediction firstly performs the branch predictor updating and the BTB updating simultaneously in the training flow. The branch predictor updating is to train the sub-predictors to be the correct target address pointer.

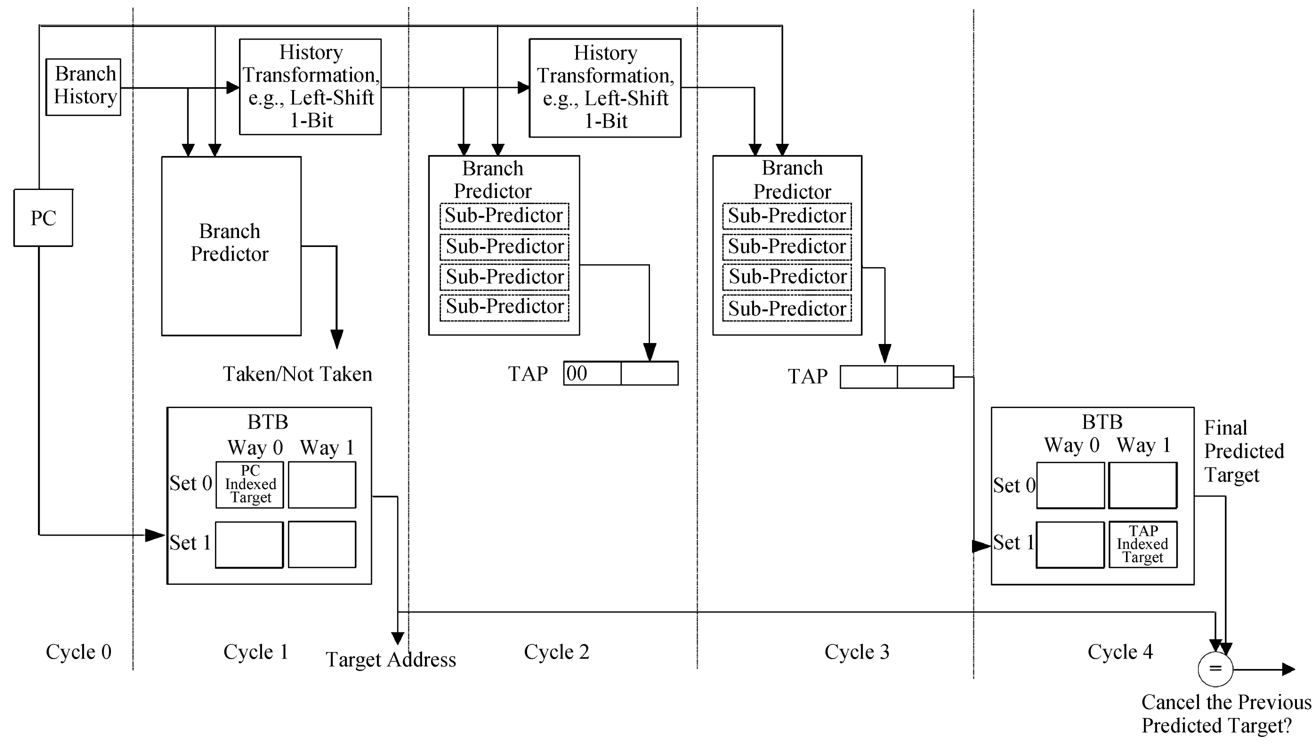


Fig.7. Diagram to illustrate the TAP's prediction stages.

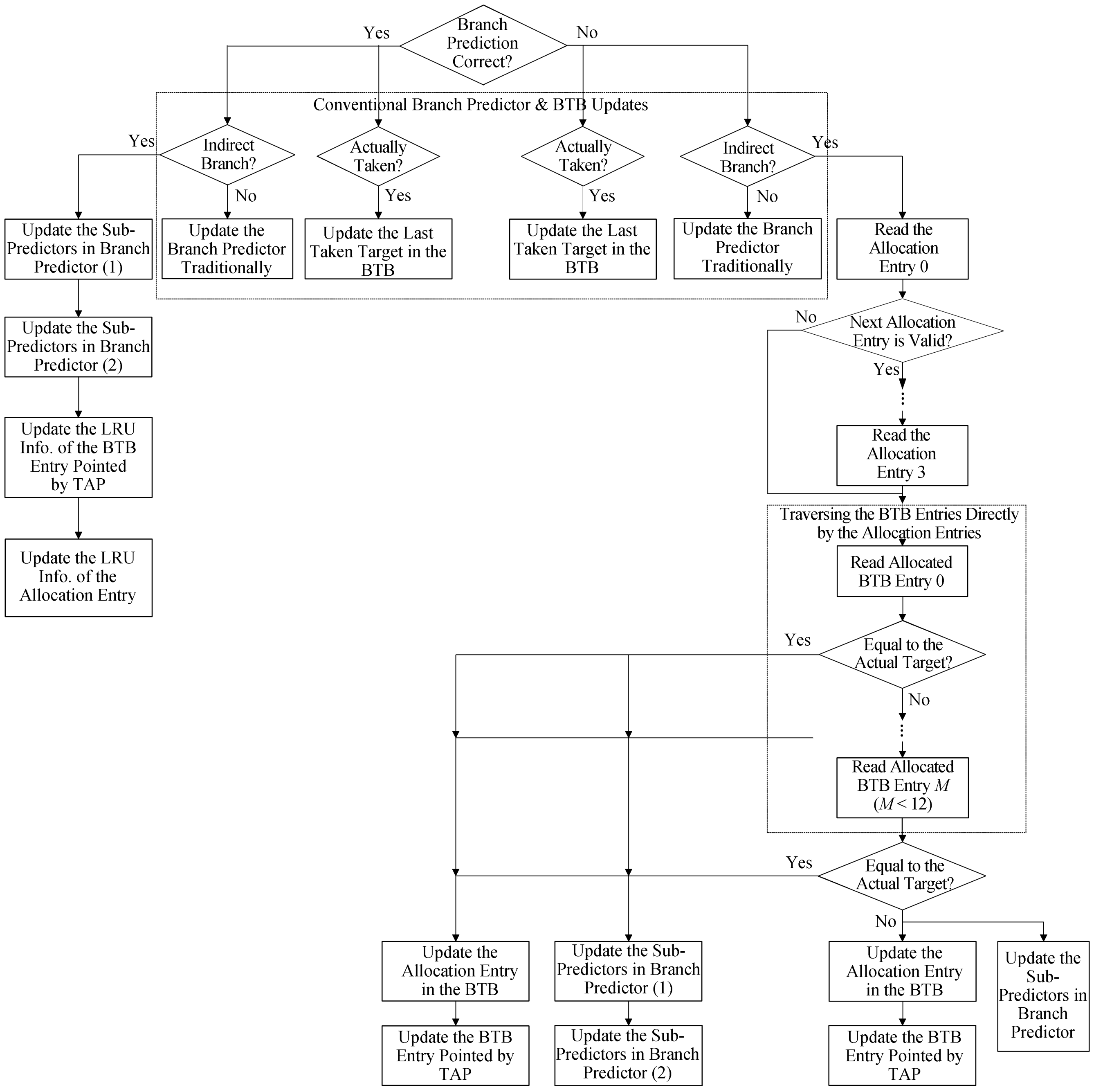

Branch Prediction Correct?
Yes
No
Conventional Branch Predictor & BTB Updates
Indirect Branch?
Actually Taken?
Actually Taken?
Indirect Branch?
Yes
No
Yes
Yes
No
Yes
Update the Sub-Predictors in Branch Predictor (1)
Update the Branch Predictor Traditionally
Update the Last Taken Target in the BTB
Update the Last Taken Target in the BTB
Update the Branch Predictor Traditionally
Read the Allocation Entry 0
Update the Sub-Predictors in Branch Predictor (2)
Next Allocation Entry is Valid?
No
Yes
Update the LRU Info. of the BTB Entry Pointed by TAP
Read the Allocation Entry 3
Update the LRU Info. of the Allocation Entry
Traversing the BTB Entries Directly by the Allocation Entries
Read Allocated BTB Entry 0
Equal to the Actual Target?
Yes
No
Read Allocated BTB Entry M (M < 12)
Equal to the Actual Target?
Yes
No
Update the Allocation Entry in the BTB
Update the Sub-Predictors in Branch Predictor (1)
Update the Allocation Entry in the BTB
Update the Sub-Predictors in Branch Predictor
Update the BTB Entry Pointed by TAP
Update the Sub-Predictors in Branch Predictor (2)
Update the BTB Entry Pointed by TAP

Fig.8. TAP's training flow.

If the bit number of the target address pointer is bigger than the number of constructed sub-predictors, it needs several times of iteration to train the sub-predictors.

For BTB updating, it has possibility that TAP prediction uses the pointed target instead of the last taken target in correct prediction cases, therefore the corresponding BTB entry containing the last taken target of the indirect branch will be replaced by the correct target rather than update only its LRU information. It is also necessary to update the LRU information of the entry pointed by the target address pointer and the corresponding allocation entry[3].

For the misprediction cases, TAP prediction should train the corresponding target address pointer to point

[3]In this paper, we modeled our design using full-synthesizable design. In full-synthesizable designs, the BTB's target and the LRU information are usually implemented separately and are constructed using SRAMs and registers respectively. In these cases, the LRU updates can be performed simultaneously. We used such configuration for the following experiments.

at the BTB entry storing the correct indirect-branch target and update the allocation entry. In the first cycle of training, it is the same as the traditional training flow: updates the BTB with the correct target if the branch is actually taken. Then TAP prediction starts to search whether its correct target is stored in the BTB. First, the allocation entries are read from the BTB, and then TAP prediction traverses the allocated entries according to the allocation information. While traversing, the target stored in each entry is compared with the correct target. If they are identical, each sub-predictor regards the bit of the correct target address pointer as its training goal. These cases are called wrong pointers. If no matched entry is found, called meaningless pointers, it will allocate a new target-entry in sequence and update the allocation-entries. The traversing process also checks whether the information in allocation entries should be updated (if an allocated target-entry is evicted). Finally, sub-predictors are trained to generate the correct target address pointer.

Although the traversing target is a little complex, TAP employs three methods to simplify this procedure. First, if the last taken target is not found in the BTB, TAP prediction starts to allocate a new entry for this indirect branch directly, avoiding the traversing flow. In these cases, the indirect branch is either in its first appearance of the program or not encountered for a long time. To the latter cases, its targets are possibly replaced by other branch targets, thereby it is more efficient to allocate directly rather than to traverse. Second, during reading the contents of allocation entries, if the flag of an allocation entry is invalid, it is unnecessary to access the next allocation entry. This is used to accelerate the cases where an indirect branch does not have too many targets. Third, if the target-entry number of traversing equals the number of pipeline stages[4], TAP prediction should stop the traversing and allocate an entry directly. The maximum penalty of a branch prediction is the number of the pipeline stages; thus keeping traversing is meaningless and causes more penalties than the original design.

5 Experimental Methodology

We employed gem5[22], a cycle-accurate performance simulator, in x86 full-system mode to evaluate the effects of TAP prediction. Table 2 shows the parameters of our baseline processor, which employs only the BTB to predict indirect branches. The latencies of various branch prediction strategies and the prediction/training delays introduced by TAP prediction have been considered in our experiments. Our workload includes the benchmarks of SPEC CPU INT 2006[23] and the DaCapo-9.12 benchmarks[24]. The SPEC CPU INT 2006 evaluation shows the effects of TAP prediction on various kinds of benchmarks. DaCapo benchmarks are written in Java, which consists of a set of open source, client-side, real-world applications with non-trivial memory loads. Their evaluation shows the effects on indirect branch sensitive programs. The DaCapo benchmarks are running on the OpenJDK-1.6.20[5] with -Xint option. Table 3 shows the characteristics of the examined benchmarks on the baseline processor.

Table 2. Baseline Processor Configuration

Item	Configuration
Pipeline depth	16 stages; out-of-order execution
Instruction fetch	4-instruction per cycle; fetch and at first pred taken branch
Regs	Physical integer Regs: 256; physical float regs: 256
Execution engine	4-wide decode/rename/dispatch/issue/writeback; load queue: 64-entry; store queue: 64-entry
Branch predictor	4 K-entry, 4-way BTB (LRU), 1-cycle prediction delay; 32-entry return address stack; structures of the direction predictors are listed in Table 1; 15-cycle branch misprediction penalty
Caches	32 KB, 8-way, 2-cycle L1 DCache & ICache; 1MB, 16-way, 10-cycle unified L2 cache, 64B block size with LRU replacement policy
Memory	200-cycle memory latency

Note: Regs means general purpose registers.

6 Results and Analysis

In this section, we offer not only the performance evaluations of various indirect-branch prediction strategies, but their results of energy consumption as well.

6.1 Timing Estimation

To evaluate the timing impact of TAP prediction implementation, we model the TAP schemes by extending the front-end of a commercial full-synthesizable 64-bit superscalar processor that runs about 1GHz under TSMC 65 nm technology. It can issue and complete three instructions per clock cycle. Instructions complete in order, but execute out of order. Its dynamic branch prediction employs a 2 K-entry, 2-level branch predictor and a 512-entry, 4-way set-associative BTB,

[4]The exact number is the result of subtracting the number of allocation entries from the number of pipeline stages.

[5]http://openjdk.java.net/, Sept. 2014.

Table 3. Characteristics of Evaluated Benchmarks

Configuration	Number of Indirect Branches in Dynamic Execution (K)	Indirect Branches MPKI
astar	1 330	0.01
bzip2	803	0.01
gcc06	123	0.17
gobmk	829	0.47
h264ref	25	0.01
hmmer	1 617	5.76
libquantum	6	0.02
mcf	146	0.17
omnetpp	3 779	0.61
perlbench	4 265	15.67
sjeng	1 751	4.94
avrora	5 904	32.60
batik	8 418	42.63
eclipse	8 307	37.75
fop	6 107	18.50
h2	5 958	34.74
jython	4 168	22.51
luindex	6 127	18.57
lusearch	6 402	35.61
pmd	6 141	18.63
sunflow	6 464	31.72
tomcat	3 106	7.69
tradebean	6 166	18.75
tradesoap	6 105	18.49
xalan	6 577	36.40

which has 1-cycle access latency. To catch up with the technology development, we perform timing evaluations of various implementations using TSMC 28 nm technology instead of 65 nm technology. The implementations and their timing parameters are listed in Table 4. The evaluation results prove that constructing sub-predictors does not increase the latency of the conditional branch prediction. This is because a sub-predictor uses fewer predictor tables (for fewer GHRs), thereby saving some multiplexors or reducing the number of adding operands, compared to the original branch predictor.

6.2 MPKI Impacts and Performance Improvement

Fig.9 shows the indirect-branch MPKI of the baseline and the TAP schemes. TAP prediction reduces indirect-branch MPKI significantly for all indirect-branch sensitive programs. For example, the TAP-TAGE scheme reduces the average MPKI from 16.10 to 9.51. Table 5 shows the conditional-branch MPKI impacts. As TAP prediction reuses existing branch predictor to generate the target address pointers, it is inevitable to have adverse impacts on conditional branches. However, those impacts are relatively small compared to the significant indirect-branch MPKI improvement achieved by TAP prediction. The results show that the impacts on conditional branch prediction vary under those four TAP schemes. The TAP-GShare scheme has the most contentions, whereas the TAP-TAGE scheme has the least contentions. This is because the mechanisms in aggressive high-performance branch predictors, which are inherently used to avoid aliasing problems and contentions for conditional branches, are also effective to avoid the interference between predicting branch directions and target address pointers.

Table 4. Timing Parameters of Branch Components Under TSMC 28 nm Typical Process (0.9 V)

Components		SRAM Cell	Timing (ns)		Estimated Max Latency		
			T_{setup}	T_{ClkToQ}	w/o TAP	w/TAP	Sub-Predictor in TAP
BTB	Data	128 × 148	0.053 27	0.332 3	0.408 7	0.408 7	–
	Tag	128 × 120	0.054 13	0.264 8			
GShare		128 × 32	0.040 60	0.229 0	0.302 2	0.302 2	0.289 7
Perceptron		128 × 64	0.043 04	0.237 6	0.461 2	0.461 2	0.428 9
O-GEHL		128 × 16	0.043 17	0.224 6	0.437 0	0.437 0	0.400 1
TAGE		128 × 16	0.043 17	0.224 6	0.401 1	0.401 1	0.394 5

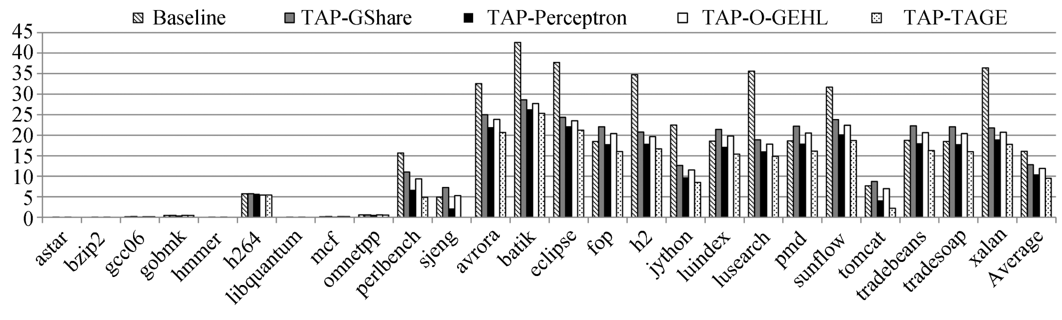


Fig.9. Indirect-branch MPKI impacts of TAP predictors.

Table 5. Conditional-Branch MPKI Comparisons

Configuration	GShare		Perceptron			O-GEHL		TAGE	
	Baseline	TAP	Baseline	TAP	VPC	Baseline	TAP	Baseline	TAP
astar	11.04	11.05	8.31	8.40	8.80	9.13	9.71	8.20	8.22
bzip2	26.29	26.31	20.67	20.68	21.14	21.81	22.02	20.52	20.53
gcc06	2.39	2.40	0.42	0.43	0.93	1.87	2.07	0.23	0.24
gobmk	38.11	38.85	30.32	30.65	30.78	31.65	31.97	30.15	30.16
h264ref	0.63	0.64	0.13	0.14	0.64	0.49	0.50	0.09	0.09
hmmer	5.54	6.04	4.04	4.06	4.53	4.57	4.60	3.97	3.98
libquantum	2.25	2.33	0.46	0.47	0.98	1.77	1.80	0.30	0.30
mcf	28.30	28.36	21.68	21.72	22.15	23.44	23.64	21.45	21.51
omnetpp	10.52	10.59	7.13	7.21	7.63	8.64	8.65	6.93	6.95
perlbench	7.50	11.05	4.88	4.94	5.38	6.14	7.69	4.72	4.74
sjeng	23.86	28.05	18.28	18.48	18.76	19.76	22.97	18.09	18.14
avrora	9.30	11.80	6.79	6.86	7.28	7.67	9.13	6.67	6.69
batik	6.06	7.95	4.32	4.37	4.82	4.99	6.00	4.23	4.24
eclipse	6.32	8.12	4.58	4.63	5.08	5.22	6.23	4.50	4.51
fop	15.74	19.23	11.78	11.92	12.27	13.02	15.31	11.62	11.66
h2	12.10	14.44	9.18	9.29	9.67	10.01	11.69	9.08	9.10
jython	10.16	12.55	7.54	7.63	8.04	8.39	9.90	7.43	7.46
luindex	15.90	19.38	11.91	12.05	12.40	13.15	15.45	11.76	11.79
lusearch	10.82	13.16	8.11	8.20	8.60	8.94	10.51	8.01	8.03
pmd	15.67	19.16	11.72	11.86	12.21	12.96	15.24	11.56	11.60
sunflow	7.13	10.08	4.81	4.87	5.31	5.85	7.22	4.67	4.69
tomcat	13.45	17.24	9.76	9.87	10.25	11.10	13.25	9.58	9.61
tradebean	15.78	19.26	11.81	11.95	12.30	13.05	15.34	11.65	11.69
tradesoap	15.52	19.01	11.59	11.73	12.08	12.83	15.10	11.44	11.47
xalan	12.34	14.65	9.39	9.50	9.88	10.21	11.91	9.29	9.31
Average	12.91	15.96	9.58	9.69	10.08	10.67	12.59	9.45	9.47

The MPKI reduction leads to attractive IPC improvement shown in Fig.10, where GMEAN means the geometric mean for results of each benchmark. As TAP prediction does not increase the latency of the branch prediction, IPC evaluation generally reflects the performance impact. The TAP schemes achieve significant performance improvements for indirect-branch sensitive programs, and have no adverse impacts for other programs. The IPC improvements of TAP-GShare, TAP-Perceptron, TAP-O-GEHL and TAP-TAGE are 3.17%, 9.31%, 6.24% and 9.46%, respectively. As DaCapo benchmarks written in Java are more indirect-branch sensitive compared to the SPEC CPU INT 2006 benchmarks, they have much more significant performance improvements. The effect of the updating mechanism in TAP-Perceptron scheme is listed in Table 6. The result shows that it takes only 2.83 cycles for each indirect-branch update averagely. It can only improve performance by 1.14% averagely as if each update could be finished in one cycle. Therefore, although the traversing to the target-entries makes the training a little more complex, it would not affect performance significantly.

We also evaluate the various breakdowns of indirect branch mispredictions in TAP prediction shown in Fig.11. Fig.11(a) shows that only one allocation entry is used for the majority of training process, while all

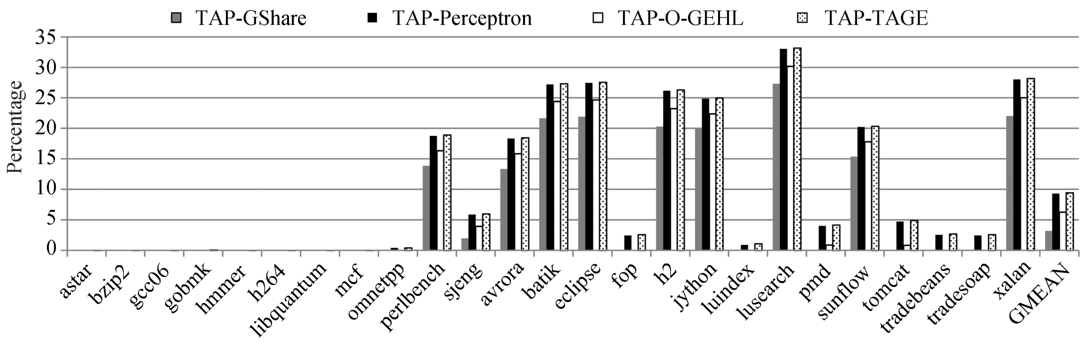


Fig.10. IPC improvement of TAP predictors.

Table 6. Training Penalty Cycles in TAP-Perceptron Scheme

Configuration	Number of IB (K)	Cycles per IB Update	IPC Δ w/Ideal Update (%)
astar	1 330	2.36	1.22
bzip2	803	3.05	1.14
gcc06	123	4.49	1.32
gobmk	829	3.39	1.09
h264ref	25	1.11	1.32
hmmer	1 617	1.56	1.25
libquantum	6	2.03	0.66
mcf	146	3.79	0.92
omnetpp	3 779	4.52	1.28
perlbench	4 265	4.93	1.67
sjeng	1 751	4.16	1.20
avrora	5 904	2.58	1.24
batik	8 418	1.98	1.26
eclipse	8 307	1.86	1.25
fop	6 107	2.55	0.88
h2	5 958	2.35	1.21
jython	4 168	2.45	1.23
luindex	6 127	3.54	1.23
lusearch	6 402	2.38	1.22
pmd	6 141	2.15	0.74
sunflow	6 464	3.13	1.28
tomcat	3 106	3.93	1.25
tradebean	6 166	3.54	1.23
tradesoap	6 105	3.55	1.23
xalan	6 577	2.31	1.20
Average	3 870	2.83	1.14

Note: IB stands for indirect branch.

four allocation entries are accessed in very rare cases. Fig.11(b) shows that the traversing process can be finished in 16 cycles in most cases. Fig.11(c) shows that using the last taken target can quickly identify the misprediction type, and the cases of the meaningless pointers happen less often than those of the wrong pointers.

As not all benchmarks in SPEC CPU INT 2006 are indirect-branch sensitive, we choose only a subset of SPEC CPU INT 2006, each of which gains at least 5 percent performance with a perfect indirect branch predictor, together with the DaCapo benchmarks for the following experiments.

6.3 Comparison with Other Indirect-Branch Predictors

We choose the TAP-Perceptron predictor as a representative example to compare with other hardware-based indirect-branch predictors, the TTC predictor, the Cascaded predictor and the VPC predictor, for the Perceptron predictor has been widely implemented in newly commercial processors. Fig.12 and Fig.13 illustrate the comparisons of the indirect-branch MPKI and the IPC improvement.

1) *Comparison with TTC Predictor and Cascaded Predictor*. The TTC predictor and the Cascaded predictor are representative dedicated-storage-predictors. We simulate various sizes of the TTC predictor from 256-entry to 64 K-entry and the Cascaded predictor from 64-entry to 16 K-entry filter, each entry of which has 5 bytes (40 bits). On average, the TAP-Perceptron predictor achieves the equivalent performance provided by an 8 K-entry TTC predictor, or a Cascaded predictor with 2 K-entry filter. We conclude that if the TTC has fewer than 64 entries, it would affect the performance averagely, for the frequent replacements lead to low prediction accuracy. On the other hand, the performance improvement growth of dedicated-storage predictors would not be increased obviously if they have some more of entries, e.g., TTC has more than 8 K-entry, for the GShare-like algorithm constrains its performance improvement.

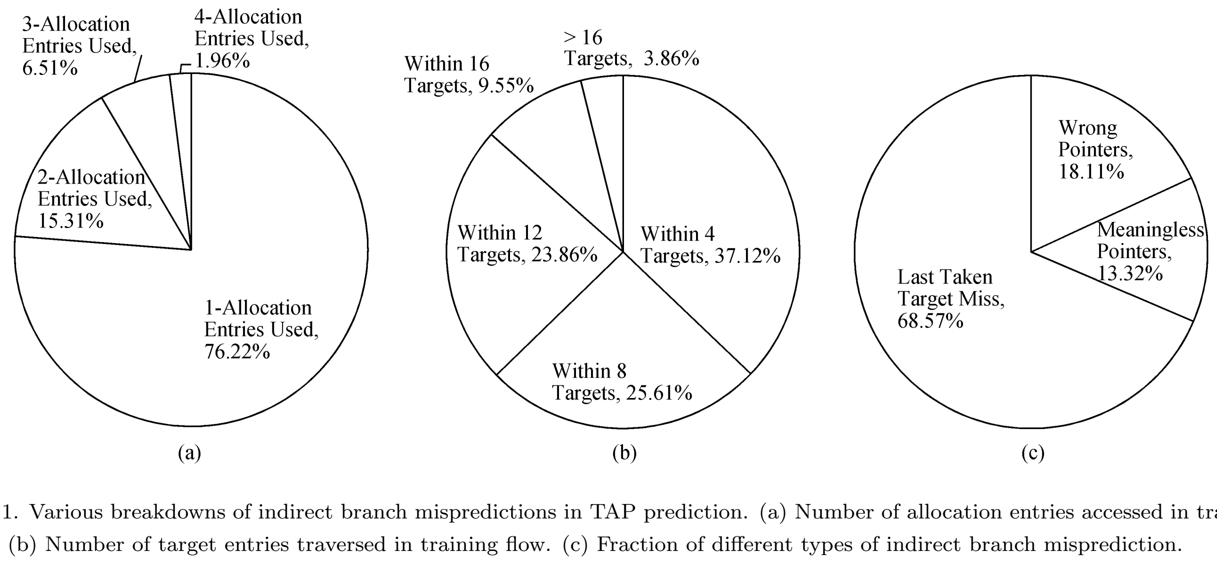


Fig.11. Various breakdowns of indirect branch mispredictions in TAP prediction. (a) Number of allocation entries accessed in training flow. (b) Number of target entries traversed in training flow. (c) Fraction of different types of indirect branch misprediction.

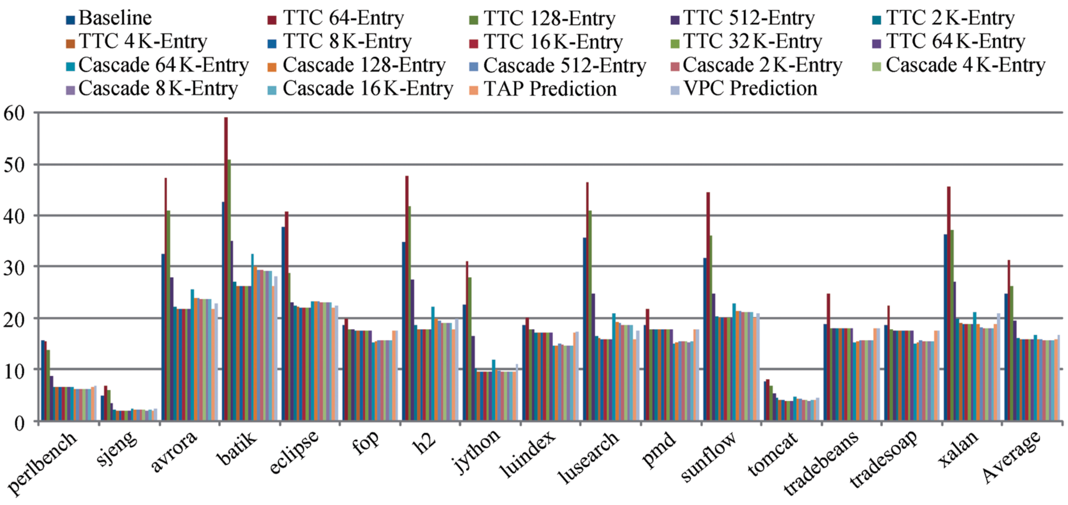


Fig.12. Comparisons of indirect-branch MPKI.

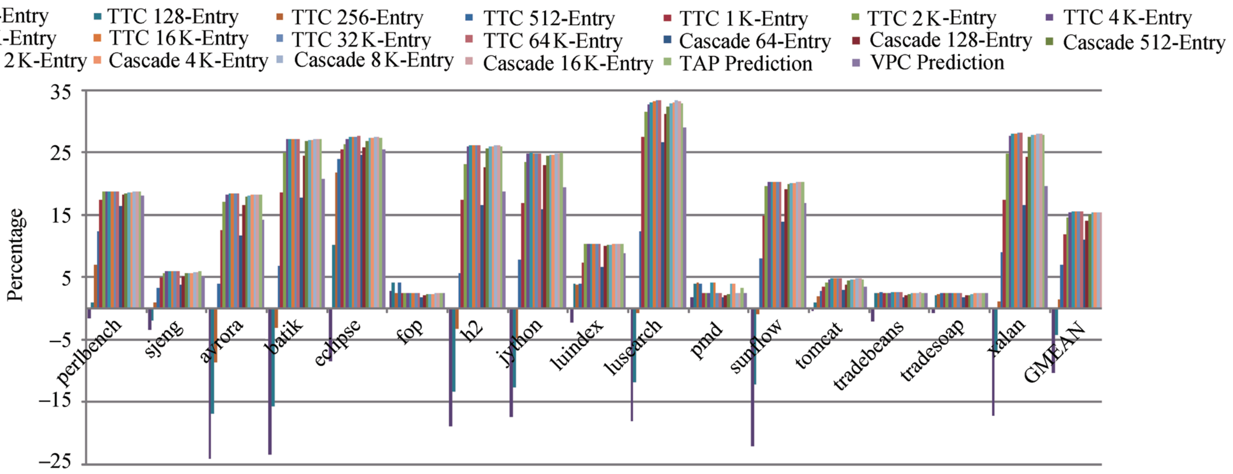


Fig.13. Comparisons of IPC improvements.

2) *Comparison with VPC Predictor.* We also compare the efficiency of the VPC predictor and the TAP predictor. In our experiments, the maximum iteration number of VPC prediction is set to 15, for the branch penalty we define is only 15. In addition, as the LFU training algorithm of VPC prediction is difficult to implement in hardware, we adopt a simple way for updating: insert the newly encountered indirect-branch target sequentially at the tail of the access sequence. We find that VPC prediction in our evaluations usually requires 3 to 6 iterations to finish predictions, which is less effective than the results in [9]. Although VPC prediction has some alternative training algorithms like random replacement and linked list appendance to reduce its complexity, they largely depend on the time when they construct the target linked list. For instance, in the linked list appendance algorithm, if the first element in the list was used infrequently, VPC prediction would not have good performance as we expected. In our selected program segments, as the frequently predicted targets usually appear later than some other targets, they are inserted into the middle or at the tail of the access sequence.

In our experiments, the TAP predictor outperforms a VPC predictor by 2.39%. Besides the total indirect-branch MPKI of VPC prediction, we also evaluate its MPKI if VPC predictions are finished in three cycles (3-cycle-MPKI). Fig.14 and Fig.15 illustrate that although the total indirect-branch MPKI of VPC prediction is similar to that of TAP prediction, the 3-cycle-MPKI increases significantly, especially for the benchmarks with higher number of dynamic targets. This finding is similar to the analysis in research [14]. It indicates that VPC prediction requires more cycles to reach the same accuracy as that of TAP prediction. On the other hand, as shown in Table 5, VPC prediction's adverse impact on conditional branch predictions is also greater than that of TAP prediction. VPC prediction generates and

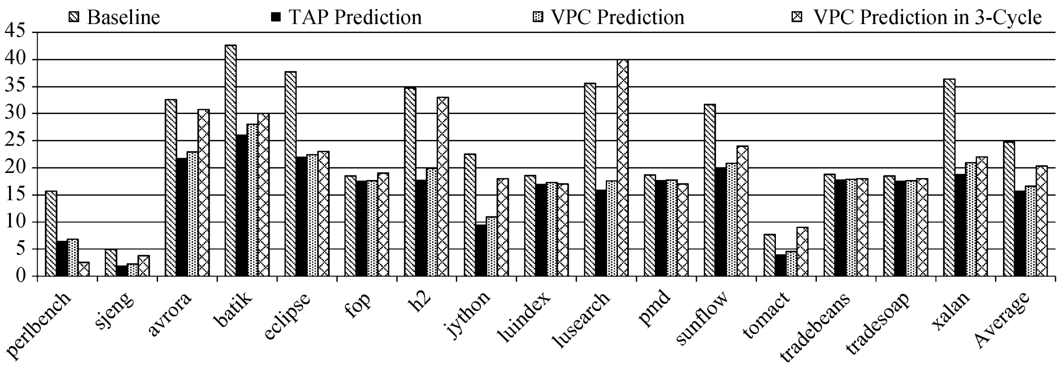


Fig.14. MPKI comparisons among TAP prediction and various types of VPC predictions.

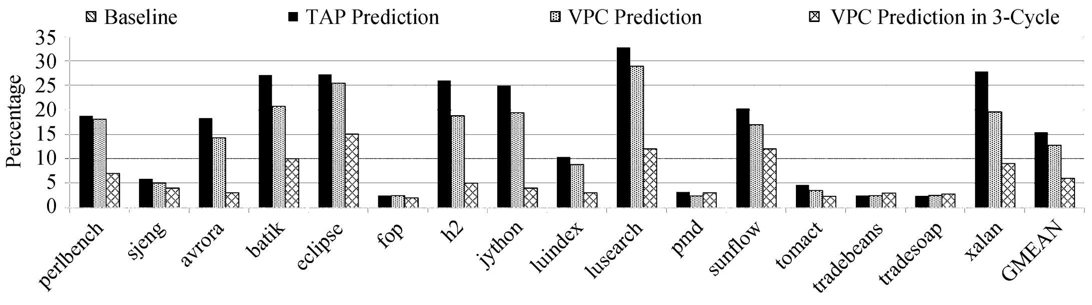


Fig.15. IPC improvement comparisons among TAP prediction and various types of VPC predictions.

predicts a virtual branch in each cycle. Thus, VPC prediction usually occupies more table entries of the branch predictor for one branch occurrence.

6.4 Sensitivity to Different BTB Sizes

We also evaluate the effects of different BTB sizes. We take the TAP-Perceptron scheme as an example. The results in Table 7 show that TAP prediction could provide significantly performance improvement even with small BTB sizes.

Table 7. Effect of Different BTB Sizes in TAP-Perceptron Scheme

BTB	Baseline IB MPKI	TAP IB MPKI	IPC Δ (%)
8 K	22.00	13.89	17.77
4 K	24.69	15.78	15.40
2 K	27.25	16.60	12.20
1 K	28.97	18.14	10.19
512	31.12	20.22	8.22

Note: IB stands for indirect branch.

6.5 With the Help of TTC

The BTB was used to store indirect-branch targets in previous evaluations. As modern processors usually have a specific, independent indirect branch predictor, such as TTC, TAP prediction can also place the indirect-branch targets in the TTC rather than occupy the BTB resources. In this way, TAP prediction changes its mechanism that is based on the simple GShare algorithm to the more aggressive mechanism used in the high-performance branch predictor for the purpose of higher indirect-branch prediction accuracy.

Instead of all types of indirect-branch entries stored in the BTB, placing different types of entries in the TTC and in the BTB respectively is a better choice. The TTC contains both the target-entries and the allocation-entries. The index of TTC is the virtual address generated by the $f_{\mathrm{TAP}}(PC)$ instead of the PC XOR GHR. The BTB is still responsible for storing the last taken branch target and recognizing an indirect branch. When fetching an instruction, the BTB is accessed in the first cycle, which is the same as that of the conventional branch prediction, and in the following cycles, TTC is accessed to provide the pointed target addresses. In the training process, the BTB and the TTC are trained for their own goals respectively. Fig.16 and Fig.17 show the MPKI impacts and the IPC improvement of TAP prediction with the help of the TTC predictor. It can reduce the MPKI from 24.69 to 12.93, resulting in 28.87% performance improvement.

6.6 Area and Energy Consumption Estimation

The additional area of indirect branch prediction is mainly dominated by its SRAM requirement. TAP prediction together with VPC prediction requires only extra 512 bytes attached to the BTB as the indirect-branch flags, while TTC 8 K-entry needs extra 80 KB SRAMs for target storing and Cascaded 2 K-entry

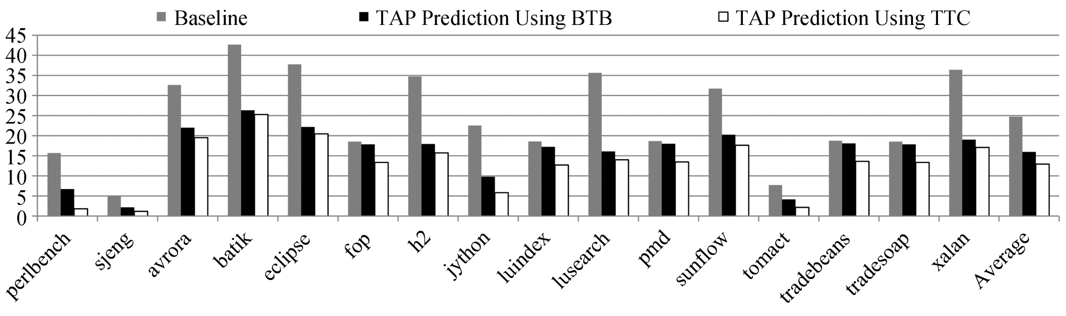


Fig.16. MPKI impacts of TAP predictors with the help of TTC predictor.

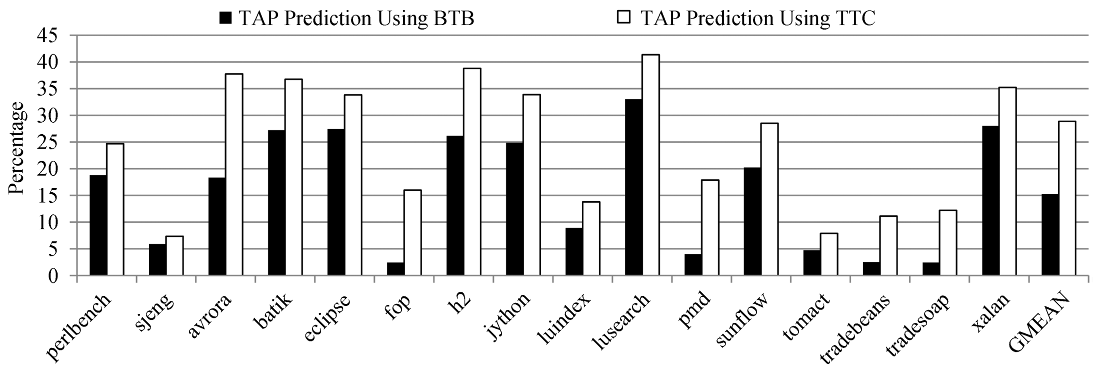


Fig.17. IPC improvement of TAP predictors with the help of TTC predictor.

needs extra 16 KB SRAMs as filters. The energy dissipation of additional hardware is also included in our energy evaluation.

The parameters of this energy evaluation are derived from the Cacti tool[25], which is configured with TSMC 28 nm technology. The power consumption of the predecoding logic has been also included in our simulations. As Fig.18 shows, the energy consumption of TAP-Perceptron prediction is reduced by 8.43% over that of the baseline processor and that of other indirect branch predictors. TAP prediction achieves energy reduction by making fewer pipeline flushes and fewer wrong-path instruction executions due to high indirect-branch prediction accuracy. The energy-delay product (EDP) improvements compared to the baseline processor are shown in Fig.19. TAP prediction achieves the best in EDP by 25.45% over the baseline processor.

7 Conclusions

In this paper, we proposed and evaluated a fast hardware-reusing technique of indirect-branch prediction called TAP prediction. It reuses the existing branch predictor to construct several sub-predictors, which are used to generate the bits of the target address pointer when fetching an indirect branch. This pointer is used to calculate a virtual address indexing the predicted target stored in the BTB or the TTC.

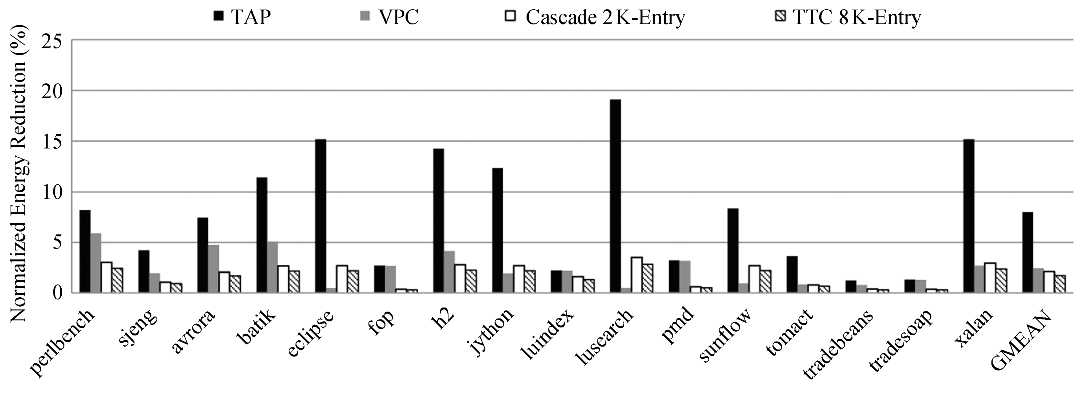


Fig.18. Comparisons of energy reduction using the TAP-Perceptron predictor.

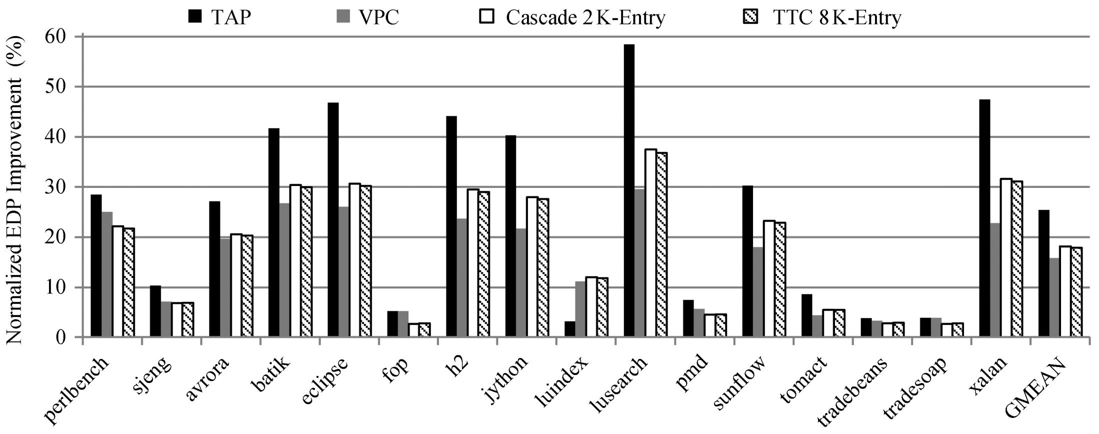


Fig.19. Comparisons of energy-delay product (EDP) improvement using TAP-Perceptron predictor.

With the help of such pointers, TAP prediction achieves similar time to that of dedicated-storage-predictors without additional dedicated storage. TAP prediction is also highly extensible to various branch prediction structures. All examined predictors benefit from this technique and only a minor hardware modification is required for implementation. It reduces the energy consumption by making fewer pipeline flushes and fewer wrong-path instruction executions due to high indirect-branch prediction accuracy as well.

References

[1] Mudge T. Power: A first-class architectural design constraint. *Computer*, 2001, 34(4): 52-58.

[2] Jiménez D, Lin C. Dynamic branch prediction with perceptrons. In *Proc. the 7th Int. Symposium on High-Performance Computer Architecture*, Jan. 2001, pp.197-206.

[3] Seznec A. A 64 Kbytes ISL-TAGE branch predictor. In *JWAC-2: Championship Branch Prediction*, Jun. 2011. http://www.jilp.org/jwac-2/program/cbp3_03_seznec.pdf, Oct. 2014.

[4] Seznec A. Analysis of the O-GEometrichistory length branch predictor. In *Proc. the 32nd ISCA*, Jun. 2005, pp.394-405.

[5] Seznec A, Michaud P. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-Level Parallelism* (*JILP*), 2006, 8: 1-23.

[6] Seznec A. Storage free confidence estimation for the TAGE branch predictor. In *Proc. the 17th IEEE Int. Symposium on HPCA*, Feb. 2011, pp.443-454.

[7] Driesen K, Hölzle U. Multi-stage Cascaded prediction. In *Proc. the 5th Int. Euro-Par Conf. Parallel Processing*, Aug. 1999, pp.1312-1321.

[8] Chang P, Hao E, Patt Y. Target prediction for indirect jumps. In *Proc. the 24th ISCA*, June 1997, pp.274-283.

[9] Kim H, Joao J, Mutlu O, Lee C, Patt Y, Cohn R. Virtual program counter (VPC) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware. *IEEE Trans. Computers*, 2009, 58(9): 1153-1170.

[10] Calder D, Grunwald D, Zorn B. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 1994, 2(4): 313-351.

[11] Lanier T. Exploring the design of the Cortex-A15 processor: Arm's next generation mobile applications processor. 2012, http://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf, Oct. 2014.

[12] Fog A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. Technical Report, Technical University of Denmark, 2012.

[13] Driesen K, Hölzle U. Accurate indirect branch prediction. In *Proc. the 25th ISCA*, April 1998, pp.167-178.

[14] Joao J, Mutlu O, Kim H, Agarwal R, Patt Y. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *Proc. the 13th ASPLOS*, Mar. 2008. pp.80-90.

[15] Yeh T, Patt Y. Two-level adaptive training branch prediction. In *Proc. the 24th MICRO*, Sept. 1991, pp.51-61.

[16] Lee J, Smith A. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 1984, 17(1): 6-22.

[17] Driesen K, Hölzle U. The cascaded predictor: Economical and adaptive branch target prediction. In *Proc the 31st Annual ACM/IEEE Int. Symposium on Microarchitecture*, Nov. 30-Dec. 2, 1998, pp.249-258.

[18] Xie Z C, Tong D, Huang M K, Shi Q Q, Cheng X. Swip prediction: Complexity-effective indirect-branch prediction using pointers. *Journal of Computer Science and Technology*, 2012, 27(4): 754-768.

[19] Farooq M, Chen L, John L. Value based BTB indexing for indirect jump prediction. In *Proc. the 16th HPCA*, Jan. 2010, pp.1-11.

[20] Azizi O, Mahesri A, Lee B C, Patel S, Horowitz M. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. In *Proc the 37th ISCA*, June 2010, pp.26-36.

[21] Xie Z C, Tong D, Huang M K, Wang X Y, Shi Q Q, Cheng X. Tap prediction: Reusing conditional branch predictor for indirect branches with target address pointers. In *Proc. the 29th ICCD*, Oct. 2011, pp.119-126.

[22] Binkert N, Beckmann B, Black G *et al.* The gem5 simulator. *ACM SIGARCH Comput. Archit. News*, 2011, 39(2): 1-7.

[23] Henning J. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006, 34(4): 1-17.

[24] Blackburn S, Garner R, Hoffmann C *et al.* The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. the 21st OOPSLA*, Oct. 2006. pp.169-190.

[25] Thoziyoor S, Muralimanohar N, Ahn J H, Jouppi N P. CACTI 5.1. Technical Report, HP Labs, 2008.

Zi-Chao Xie received his B.E. degree in microelectronics in 2006 and Ph.D. degree in computer science in 2012, both from Peking University. His research interests include processor microarchitecture, multi-core system, and HW/SW co-design.

Dong Tong is an associate professor in the School of Information Science and Technology of Peking University. He is the cofounder of the UNITY system architecture. He also led the design of the UNITY microprocessor, SoC chip and IP cores. His research interests include computer architecture, storage system, interconnection network, and System-on-Chip. He is a member of CCF and ACM.

Ming-Kai Huang is a Ph.D. candidate in Microprocessor Research and Development Center, Peking University. He received his bachelor degree from Peking University in 2008. His research interests include compiler, runtime system and computer architecture.