Xie ZC, Tong D, Huang MK *et al.* SWIP prediction: Complexity-effective indirect-branch prediction using pointers. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(4): 754–768 July 2012. DOI 10.1007/s11390-012-1262-8

SWIP Prediction: Complexity-Effective Indirect-Branch Prediction Using Pointers

Zi-Chao Xie (谢子超), Dong Tong* (佟　冬), *Member*, *CCF*, *ACM*, Ming-Kai Huang (黄明凯), Qin-Qing Shi (史秦青), and Xu Cheng (程　旭), *Senior Member*, *CCF*

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

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

School of Electronic Engineering and Computer Science, Peking University, Beijing 100871, China

E-mail: {xiezichao, tongdong, huangmingkai, shiqinqing, chengxu}@mprc.pku.edu.cn

Received January 6, 2011; revised January 18, 2012.

Abstract Predicting indirect-branch targets has become a performance bottleneck for many applications. Previous high-performance indirect-branch predictors usually require significant hardware storage or additional compiler support, which increases the complexity of the processor front-end or the compilers. This paper proposes a complexity-effective indirect-branch prediction mechanism, called the Set-Way Index Pointing (SWIP) prediction. It stores multiple indirect-branch targets in different branch target buffer (BTB) entries, whose set indices and way locations are treated as set-way index pointers. These pointers are stored in the existing branch-direction predictor. SWIP prediction reuses the branch direction predictor to provide such pointers, and then accesses the pointed BTB entries for the predicted indirect-branch target. Our evaluation shows that SWIP prediction could achieve attractive performance improvement without requiring large dedicated storage or additional compiler support. It improves the indirect-branch prediction accuracy by 36.5% compared to that of a commonly-used BTB, resulting in average performance improvement of 18.56%. Its energy consumption is also reduced by 14.34% over that of the baseline.

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

1 Introduction

Modern high-performance processors employ branch prediction components as an essential part to exploit instruction-level parallelism. Previous branch prediction researches have focused on predicting conditional branches accurately[1-5]. Recently, however, indirect-branch prediction is becoming a performance bottleneck for two reasons. First, indirect branches are more commonly used in object-oriented programs[6]. Second, indirect branches are hard to predict, since they require the prediction of the target address instead of the branch direction. A commonly-used branch target buffer (BTB) could only predict the last taken targets for indirect branches. Fig.1 shows the percentage of indirect-branch mispredictions per 1K instructions (MPKI) for the simulated benchmarks with a 4-way 4 096-entry BTB, illustrating the poor indirect-branch prediction accuracy of the BTB structure.

Previous indirect-branch prediction techniques have been proved to predict targets accurately[7-13]. However, energy consumption will increase significantly if target addresses of different branch occurrences are stored in a large dedicated structure[7]. Multi-stage predictors[8,12] implement two kinds of predictors, increasing the complexity of the processor front-end. Virtual Program Counter (VPC) prediction, a complexity-effective method, accesses the existing branch components iteratively until a stored target is predicted as taken. It is ineffective in cases where an indirect branch has many targets[14]. To be more effective, the compiler-assistant techniques[14-15] need to modify the instruction set architecture (ISA) or transform programs.

Since energy-efficiency has become an important metric in processor and System-on-Chip (SoC) designs[16-18], a designer must consider the cost-benefit tradeoffs, choosing those structures that achieve high performance per unit energy. For that reason, improving performance based on the existing components is a good choice of energy-performance tradeoffs. Our goal is to merge a highly effective indirect-branch prediction

Regular Paper

This work is supported by the "HGJ" National Science and Technology Major Project of China under Grant No. 2009ZX01029-001-002.

*Corresponding Author

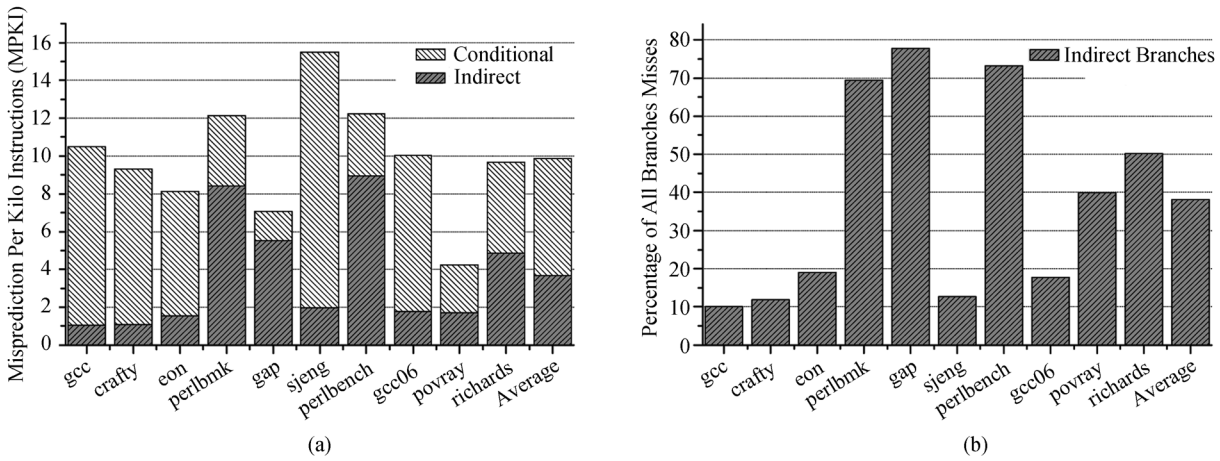

Fig.1. (a) MPKI for conditional and indirect branches. (b) Percentage of mispredictions due to indirect branches.

into the existing components, providing a fast, energy-efficient mechanism.

This paper proposes a new complexity-effective indirect-branch prediction mechanism, called the Set-Way Index Pointing (SWIP) prediction. Its key idea stems from the pointer concept. Pointers have been widely used to construct various data structures such as list, tree, and queue, providing a highly effective way to access data. SWIP prediction reuses the branch direction predictor to provide the hardware pointers to the BTB. These pointers, called set-way index pointers, point to the BTB entries storing the predicted indirect-branch targets. Multiple indirect-branch targets of an indirect branch are stored in different BTB entries, and the corresponding BTB set indices and way locations are treated as the set-way index pointers. These pointers are stored in the table entries of the existing branch-direction predictor, replacing original conditional-branch direction information. By redefining the functions of existing structures, SWIP prediction does not require large dedicated target storage or any additional compiler support.

Our evaluation shows that SWIP prediction improves the indirect-branch prediction accuracy over the commonly-used BTB-based prediction by 36.45% for a more-aggressive 4-wide 15-stage pipeline processor, and by 36.78% for a less-aggressive 2-wide 8-stage pipeline processor. The average instructions per cycle (IPC) is improved by 18.56% for the more-aggressive processor, and by 6.26% for the less-aggressive processor. Compared with the previously proposed hardware-based indirect-branch predictors, the SWIP predictor achieves performance improvement equivalent to that provided by 48 KB tagged target cache (TTC)[7] design, and it also outperforms the VPC predictor[9] by 7.65%.

The contributions of SWIP prediction are as follows:

1) To our knowledge, SWIP prediction is the first mechanism that uses pointers to predict indirect branches. Different pointers corresponding to various indirect-branch occurrences point to the predicted targets. By using pointers, SWIP prediction requires no extra large storage.

2) SWIP prediction utilizes the existing branch-direction predictor, which previously had been ineffective in indirect-branch prediction, to distinguish different indirect-branch occurrences and record set-way index pointers, without requiring any compiler support.

3) SWIP prediction is also applicable to processors with a short pipeline. Since SWIP prediction takes only two or three cycles, the processor with the short pipeline could also have attractive performance improvement.

2 Previous Work

A modern processor employs the BTB to predict indirect branches[19]. The BTB records only the last taken target, which cannot distinguish various indirect-branch occurrences, resulting in poor prediction accuracy. In this case, the branch direction predictor is also ineffective because it always predicts taken for indirect branches.

Some specialized indirect-branch predictors[7-8,12] were proposed to predict the indirect-branch targets. Chang *et al.*[7] first proposed tagged target cache (TTC) to use branch history information to distinguish differences among indirect-branch occurrences. Its concept is similar to that of a two-level branch-direction predictor[20]. Each TTC entry contains a target address and a tag field. When fetching an indirect branch, the TTC predictor is indexed using the XOR of the program counter (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. This technique

requires a large dedicated structure to store targets of different occurrences.

Driesen *et al.*[8,12] proposed a cascaded predictor by combining two target predictors: a simple first-stage predictor for easy-to-predict indirect branches, and a more complex second-stage predictor for hard-to-predict indirect branches. Other techniques that work well are dependence-based pre-computation[21], data-compression technique[11], and IT-TAGE predictor[13]. These techniques, however, require additional hardware resources or significantly increase the complexity of the processor front-end.

Recently, VPC prediction[9], an energy-efficient technique, was proposed to use the existing conditional-branch prediction components to predict indirect-branch targets. 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 conditional-branch predictor iteratively. One of T virtual direct branches is predicted as well as a conditional branch in each iteration. This iterative process stops when a virtual direct branch is predicted to be taken, or a predefined maximum iteration number is reached. The research[14] demonstrates that the performance of VPC prediction degrades significantly for workloads with larger numbers of dynamic targets.

With the help of the compiler and ISA modification, several proposed techniques significantly improve processor performance[14-15]. Joao *et al.*[14] proposed dynamic predication for hard-to-predict indirect branches (DIP). 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.*[15] proposed a value-based BTB indexing (VBBI) technique, the most recent research done with compiler assistant. 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 address and the hint instruction output values.

3 SWIP Prediction

This section proposes SWIP prediction technique, and describes its detailed structure. In order to achieve high prediction accuracy, SWIP prediction uses the global branch history information (GHR) to detect various indirect-branch occurrences. Using GHR for indirect-branch prediction is widely used in previous hardware-based indirect-branch predictors, such as TTC and VPC. The essential idea of SWIP prediction is to predict the set-way index pointer, which points to the stored indirect-branch target in the BTB, instead of predicting indirect-branch target directly.

3.1 Principle

The key idea of SWIP prediction is using the hardware pointers to make similar time cost of the direct target access (i.e., TTC[7]) without extra large storage. Using pointers is an efficient method to access data. In addition, those pointed data could be stored in distributed memory locations. Based on these two advantages, SWIP prediction uses the hardware pointer to separate the direct mapping between indirect-branch occurrences and multiple targets into two steps, shown in Fig.2: first map each indirect-branch occurrence distinguished by the GHR to a pointer, then use this pointer to fetch the predict target. Hence, a pointer is actually treated as an intermediate representation of the predicted target address, corresponding to an indirect-branch occurrence.

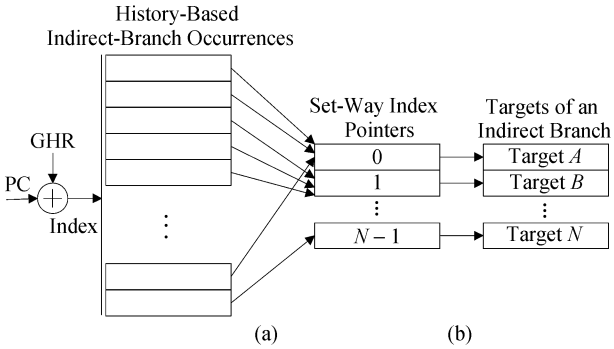

Fig.2. Key idea of SWIP prediction.

3.2 Structure Details

Fig.3 shows the detailed structure of the SWIP predictor.

Unlike the conventional BTB structure, an indirect branch in SWIP prediction occupies multiple BTB entries. Those entries are classified as target-entry or allocation-entry. The tags of these entries are the same as the BTB tag of this indirect branch. As a program executes, SWIP prediction dynamically allocates some BTB entries, called target-entries, to store the encountered targets of this indirect branch. According to the

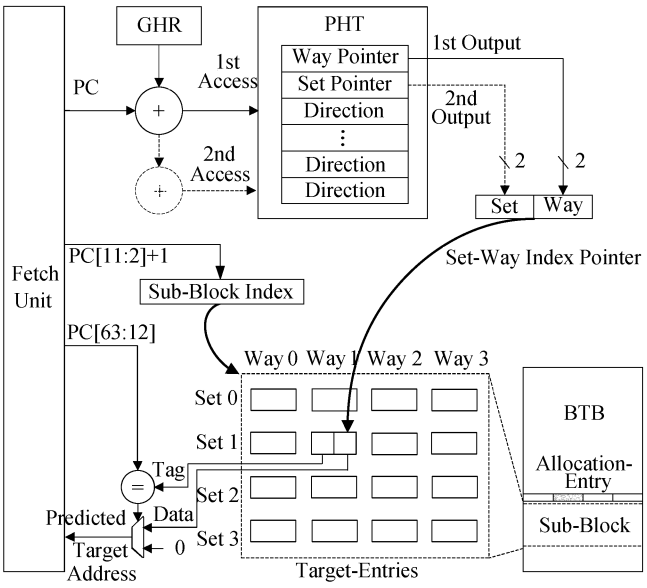

Fig.3. Details of the SWIP prediction structure.

previous analysis[9] and the statistics of our experiments (Fig.4), indirect branches have no more than 16 targets in most programs. Based on this observation, SWIP prediction allocates at most 16 target-entries for an indirect branch. If an indirect branch has more than 16 targets, SWIP prediction will replace the content of a used target-entry by the new encountered target. An allocation-entry, a specific BTB entry, is employed by each indirect branch to record the target-entry allocation information. This information is stored in the low 16 bits of the target address field of this BTB entry. To identify whether the BTB has the target-entries of this indirect branch, SWIP places the allocation-entry in the set indexed by the PC, where is accessed firstly when fetching an indirect branch.

SWIP prediction creates the hardware pointers to the target entries of an indirect branch, shown in Fig.2(b). In order to reach the minimum length of those pointers (4 bits for 16 target-entries), it treats the BTB as a 2-D array, where the BTB set index and way location, called set-way index pointer, points to a BTB entry. Then SWIP prediction places those target-entries of an indirect branch in several consecutive BTB sets to construct a sub-block. In such cases, the internal set index and way location of the sub-block can be used to index an entry in the sub-block. For example, a 4-way set-associative BTB constructs a sub-block with four sets. In order to avoid the confliction with allocation-entry, the sub-block is indexed by the low-order bits of the indirect-branch PC plus 4. The high two bits of the set-way index pointer represent the internal set index, while the low two bits represent the way location. The pointed entry will be accessed directly via the set-way index pointer without any accesses of other entries in the same set of BTB.

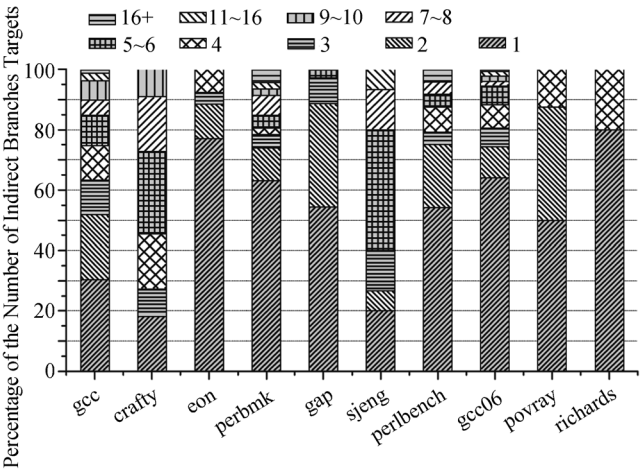

Fig.4. Distribution of the number of indirect-branch targets.

As well as the conditional branch prediction, SWIP prediction uses the history-based branch-direction predictor to distinguish various branch occurrences. State-of-the-art microprocessors[22-23] usually employ an improved hybrid branch-direction predictor which includes one or more pattern history tables (PHTs). Each PHT entry is indexed by XOR (PC, GHR), providing a 2-bit saturated counter used to predict the direction of conditional branches. As for indirect branches, the PHT always predicts taken, wasting those corresponding table entries. SWIP redefines those entries to store the set-way index pointers, implementing the mapping from various indirect-branch occurrences to the pointers, shown in Fig.2(a). Its concept is similar to that of a 2-level direction predictor[2], which combines the branch history information to achieve high prediction accuracy. In other words, the PHT in SWIP can be viewed as a special TTC structure which stores the pointers instead of the targets. Because a set-way index pointer has four bits, it will occupy two PHT entries instead of the original direction information of conditional branches. These two PHT entries are indexed using XOR (PC, GHR) and XOR (PC, GHR $\ll 1$). They should be distributed as widely as possible to avoid interfering with entries recording the saturated counters for conditional branches.

This SWIP structure, however, should take two unusual cases into account. First, it is possible for different branches to have multiple tag matches in the same set. To a 4-way BTB, this happens only when a branch in the next four instructions of an indirect branch accesses the BTB. The BTB will select one of the matched entries as the predicted target, possibly leading to the mispredictions. Fortunately, according to our experimental statistics shown in Subsection 5.1, this case

occurs at a very low rate, no more than 0.3% of total branch prediction for each benchmark. Moreover, accurate indirect-branch prediction has been known to recover the lost performance. Second, if the difference between the indices of two sub-blocks is less than 4 (for indirect-branch PC, is 0x10), these two sub-blocks are spatially overlapped. Some BTB entries may be used by two indirect branches. Also, such cases rarely happen, and their additional penalties can be ignored (its evaluation is shown in Subsection 5.1).

3.3 Prediction Flow

This subsection describes the SWIP prediction flow, based on the BTB and the commonly-used direction predictor which contains the PHT structure. Table 1 demonstrates three possible cases in SWIP prediction. To identify an indirect branch when fetching instructions, SWIP prediction can simply reuse the predecoding bits (e.g., define some reserved combinations of the predecoding bits) to indicate this instruction type. Since the predecoding technique is widely used in state-of-the-art processors, it would not increase the design complexity. Also, such mechanism is required by VPC prediction which needs identifying indirect branches as soon as possible to determine whether to continue the prediction iterations.

In the first cycle, SWIP prediction accesses the BTB and the direction predictor simultaneously. The BTB result is used to identify whether the BTB records the targets of this indirect branch. If there is a BTB hit, the output is the content of the allocation-entry. Otherwise, it indicates that no target of this indirect branch is recorded in the BTB, so SWIP prediction stalls the pipeline until this indirect branch is resolved (i.e., Case 1 in Table 1). The PHT output constructs the low two bits of the set-way index pointer. SWIP prediction implements an auxiliary mechanism to accelerate the case where the indirect branch has no more than four targets. In this case, the high two bits of the set-way index pointer are always zero. Accordingly, SWIP prediction assumes these two bits are zero in this cycle, thereby forming a temporary set-way index pointer. It then uses this pointer to access the BTB in the next cycle.

In the second cycle, SWIP prediction uses the temporary set-way index pointer to access the BTB, and then switches the index to access the PHT. If there is a BTB hit (the entry is valid and the tag matches the indirect-branch address), SWIP uses the output as the predicted target, and then issues it to the pipeline and the instruction cache. Otherwise, SWIP ignores the BTB output, and continues the prediction. Specially, SWIP adopts a simple method to generate the PHT index for the second PHT access, which shifts the original PHT index one bit to the left. The PHT output in this cycle constructs the high two bits of the set-way index pointer. At the end of this cycle, the full set-way index pointer has been obtained. If the high two bits equal zero, we conclude that the BTB entry pointed to by that full set-way index pointer has already been accessed in this cycle. Such being the case, SWIP finishes its indirect-branch prediction (Fast-Pred case).

In the third cycle, SWIP prediction uses the full set-way index pointer to access the BTB. If there is a BTB hit, the BTB output is the predicted target (Full-Pred case). The previous issued target should be canceled. Otherwise, SWIP prediction continues using the last predicted target which has been issued to the pipeline and instruction cache. Altogether, a full-pred SWIP prediction takes three cycles.

3.4 Prediction Example

In this subsection, we use a code example to illustrate how SWIP prediction effectively uses the pointer to distinguish different indirect-branch occurrences based on the previous branch-history information. Fig.5 shows a code example of indirect branch from the Perlbench benchmark in SPEC CPU 2006 suites[24]. The indirect branch, "*jsr*$26, ($27), 0", implements calling the function pointer, "*sortsvp*". As the program executes, different execution results for the conditional branch in line 13 lead to different values for "$27", resulting in different function pointers. If the condition in line 13 is true (CondT), the indirect-branch target is "S_qsortsv" (TargetT). Otherwise (CondF), the target is "S_mergesortsv" (TargetF). We assume the current status of the PHT and the BTB in Fig.6.

The CondT case uses two PHT entries indexed by XOR (PC, $\mathrm{GHR}_{\mathrm{CondT}}$)and XOR (PC, $\mathrm{GHR}_{\mathrm{CondT}} \ll 1$)

Table 1. Possible Cases in SWIP Prediction

Cases		Set-Way Index Pointer (PHT Output)		BTB Accesses Location/Status	Target Prediction
		Way Pointer	Set Pointer		
Case 1	Cycle 1	01	–	PC /Miss	–
Case 2	Cycle 1	01	–	PC /Hit	–
(Fast-Pred)	Cycle 2	–	00	(PC + 4, Way 1) /Hit or Miss	Target A or –
Case 3	Cycle 1	01	–	PC /Hit	–
(Full-Pred)	Cycle 2	–	11	(PC + 4, Way 1) /Hit	Target A
	Cycle 3	–	–	(PC + 4 × 4, Way 1) /Hit	Target B (Cancel Target A)

```
1:  void
2:  Perl_sortsv (pTHX_ SV **array, size_t nmeb, SVCOMPARE_t cmp)
3:  {
4:     void (*sortsvp) (pTHX_ SV **array,
5                       size_t nmemb,
6:                      SVCOMPARE_t cmp,
7:                      U32 flags)
8:        = S_mergesortsv;
9:     SV *hintsv;
10:     I32 hints;
11:
12:     hints = SORTHINTS (hintsv);
13:    if (hints & HINT_SORT_QUICKSORT) {
14:       sortsvp = S_qsortsv;
15:    }
16:    else {
17:       sortsvp = S_mergesortsv;
18:    }
19:
20:    sortsvp (aTHX_ array, nmemb, cmp, 0);
21: }
```

```
ldah $29,0 ($6)
lda $29,0 ($29)
ldah
$1, S_qsortsv ($29)
ldq $3,0 ($0)
lda
$27, S_qsortsv ($1)

$L276:
  jsr $26, ($27), 0

$L273:
ldah
$1, S_mergesortsv ($29)
lda
$27, S_mergesortsv ($1)
br $31, $L276
```

Fig.5. Code example of indirect branch.

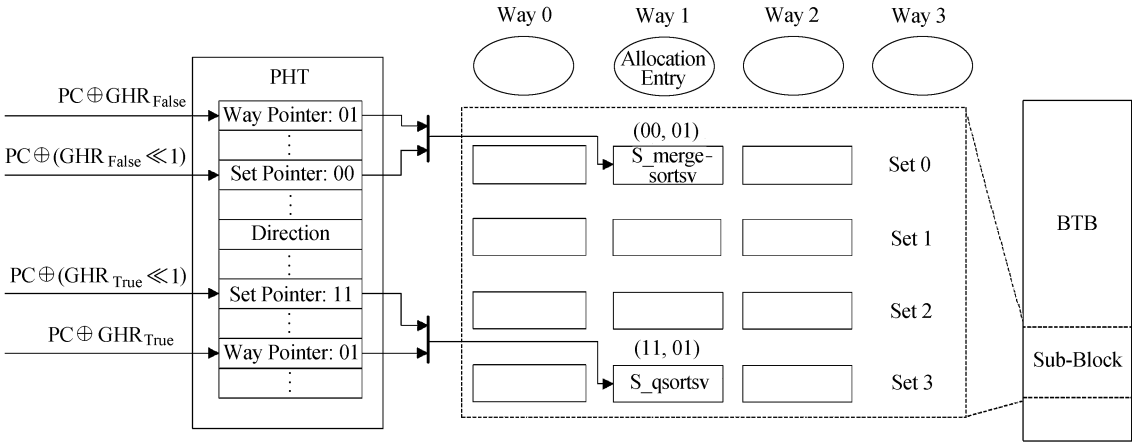

Fig.6. Current status of PHT and BTB in SWIP prediction.

respectively. This PHT usage is the same as the way of conditional branches that uses the branch history to distinguish various branch occurrences. The stored values in those entries form the set-way index pointer, "1101". This pointer is used as the set index and way location of the sub-block indexed by the PC plus 4, pointing to the TargetT entry in the BTB. Likewise, the CondF case also uses two PHT entries. The corresponding set-way index pointer is "0001", pointing to TargetF.

In the prediction flow, when the temporary or the full set-way index pointer is fetched, the pointed target-entry will be accessed in the next cycle. Because the high two bits of the set-way index pointer are zero, the CondF case benefits from the accelerating mechanism, taking only two cycles. For the CondT case, it should take three cycles. Fig.7 illustrates the prediction flow of the CondT case in pipeline view.

3.5 Updating Flow

Since SWIP updating in correct prediction is the same as that of the traditional BTB, which updates the least recently used information of the accessed set, this subsection shows only the misprediction flow. There are two misprediction cases:

1) *Wrong Pointer.* One entry in the sub-block stores the correct indirect-branch target, but the set-way index pointer points to a wrong location.

2) *Meaningless Pointer.* There is no BTB entry storing the correct indirect-branch address, so the set-way index pointer is meaningless.

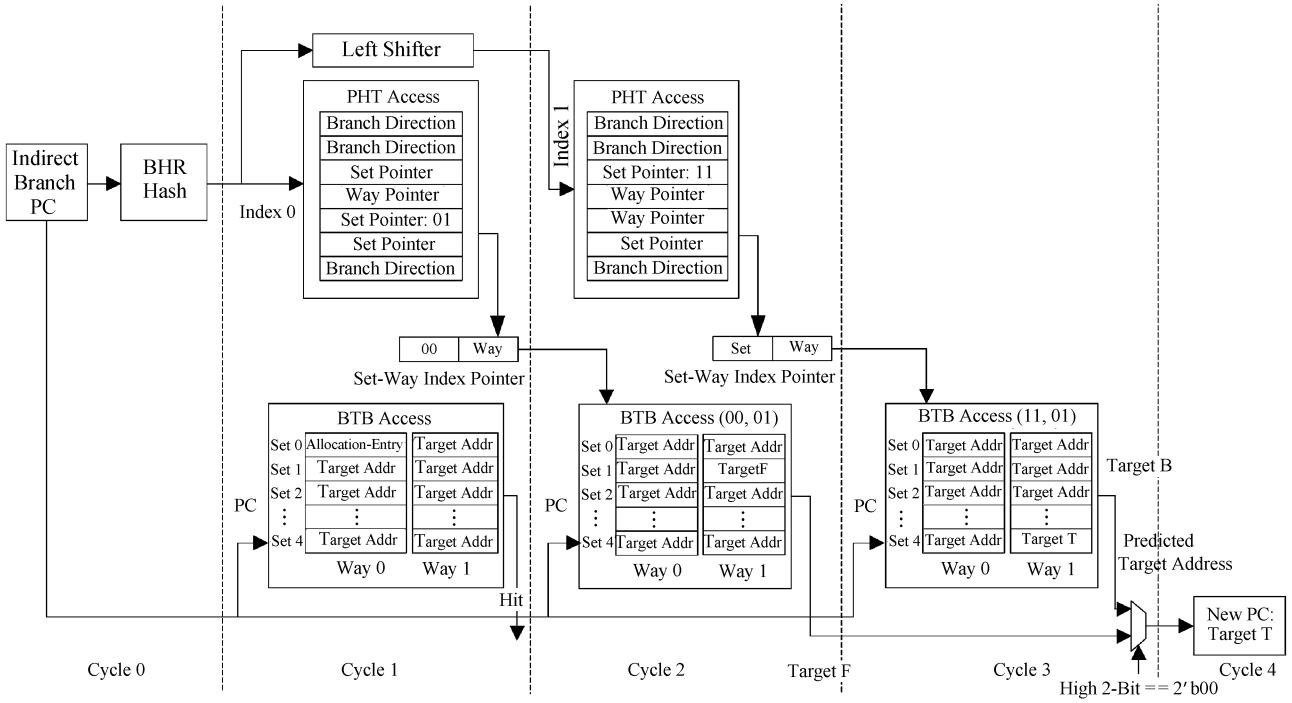

Fig.7. CondT case in SWIP prediction.

The cases of meaningless pointer are caused by either compulsory misses of indirect-branch targets or capacity misses in the case where an indirect branch has more than 16 targets. The cases of wrong pointer are caused by either the limitation of the used branch history length or the interferences with conditional branches. Since SWIP prediction reuses the PHTs, a conditional branch and an indirect branch may share the same PHT entry, resulting in the spatial interference. Such cases is inevitable to methods that reuse the existing branch prediction hardware, such as VPC prediction. The adverse impacts are evaluated in more details in Subsections 5.1 and 5.2.

In order to distinguish these two cases, the updating needs to traverse all the target-entries in the sub-block. During traversing each target-entry, SWIP identifies whether this entry belongs to this indirect branch by checking the tag, and then updates the corresponding bit in the allocation-entry. Meanwhile, if a stored target is the same as the correct target (wrong pointer case), SWIP makes the set-way index pointer point to this BTB entry, and then finishes the updating. If none of the entries matches the correct target (meaningless case), SWIP selects a BTB entry in the sub-block to store the correct target. If there are unallocated entries in the sub-block, it will allocate an new entry to store the new target, and then update the allocation-entry. If all 16 entries have been allocated, it will randomly replace a target-entry for the new target. After the BTB is updated, SWIP also updates the set-way index pointer.

In a traditional BTB, only one BTB entry can be accessed in each cycle, so an updating takes at most 16 cycles, slightly increasing the updating time cost. Although the updating logic needs to traverse the BTB on a misprediction, it occurs relatively infrequently and only traverses those allocated target-entries.

3.6 Hardware Cost and Complexity

As shown in Fig.8, SWIP prediction requires the following modifications to the existing branch prediction hardware:

1) a direct BTB access mode using set-way index pointers, and a register storing the set-way index pointer during prediction and updating,

2) a simple one-bit left-shifter to generate the second PHT access, and a new multiplexer to select the indices,

3) a traversing counter to generate set-way index pointers during updating,

4) registers carrying the content of the allocation-entry throughout the pipeline.

Compared with other previously proposed history-based indirect-branch predictors, SWIP requires no large or complex tables to store the target addresses. Instead, target addresses are naturally stored in the existing BTB, and set-way index pointers are stored in the redefined PHT.

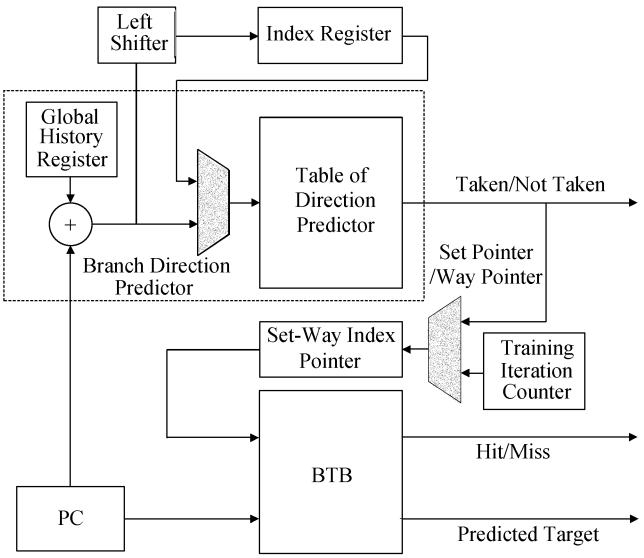

Fig.8. Modifications of SWIP Prediction structure.

The combinational logic required for prediction and updating is also simple. The index of the sub-block is the low-order bit of the branch address plus 4. The index of the second PHT access is generated simply through logical left-shifting the original PHT index. During updating, SWIP uses a simple traversing counter to generate set-way index pointers.

Branch PC and GHR values are used to access the branch prediction structure in the first prediction cycle. While accessing the BTB and the direction predictor, SWIP calculates the indices of these two components for the accesses in the next cycle. Therefore, the logic that generates the set-way index pointer and the index of the second PHT access does not affect the critical path of the branch predictor access.

To evaluate the timing impact of SWIP prediction exactly, we also modeled SWIP scheme by extending the front-end of a 64-bit 2-issue superscalar processor (which has 4-way 512-entry BTB, 2 048-entry PHT). This fully synthesizable processor is designed under TSMC 65 nm technology, and runs at least 800 MHz (worst case). It fetches four instructions per cycle. The estimated timing parameters are listed in Table 2, which prove that the SWIP modifications would not affect the timing of the front-end logics.

Table 2. Timing Parameters of Branch Components under TSMC Dophin Library 65 nm (0.9 V, 125°C, Worst Case): Extended to 4-way 4096-Entry BTB, 32 K-Entry PHT

Units		SRAM Cell	Timing (ns)		Max Latency	
			T_{Setup}	T_{ClkToQ}	w/o SWIP	w/ SWIP
BTB	Data	128x148	0.208	0.878	1.204	1.204
	Tag	128x120	0.212	0.862		
PHT		128x32	0.233	0.772	0.852	0.852

4 Experimental Methodology

We extended SimpleScalar[25] to evaluate SWIP prediction. The Wattch extension[26] is used for the energy estimation. Table 3 shows the parameters of our baseline processor. Because state-of-the-art commercial processors employ predecoding technique, we reuse these bits to identify indirect branches like VPC prediction[9]. Our workload includes fire SPEC CPU 2000 INT benchmarks, four SPEC CPU 2006 benchmarks[24], and a C++ benchmark, Richards[27]. We selected those indirect-branch-sensitive benchmarks from SPEC CPU 2000 and 2006 suites. Richards simulates the task dispatcher in the kernel of an operating system.

We used SimPoint[28] to select a representative simulation region for each benchmark using the reference input set. Each benchmark was run for 100 million Alpha instructions. All binaries were compiled with Compaq C++ V6.5-042 for Compaq Tru64 UNIX V5.1B (Rev. 2650). Table 4 shows the characteristics of the examined benchmarks on the baseline processor.

5 Results

5.1 Performance of SWIP Prediction

Fig.9(a) shows the indirect-branch prediction accuracy for the baseline processor and SWIP prediction. In addition to the typical SWIP structure, we have also evaluated an ideal SWIP structure which ignores the interferences with the other branches in the BTB and the direction predictor. This ideal SWIP structure provides

Table 3. Baseline Processor Configuration

Pipeline Depth	16 Stages
Instruction Fetch	4-instruction per cycle; fetch and at first pred taken branch; using Predecoding technique
Execution Engine	4-wide decode/issue/execution/commit; 512-entry RUU; 128-entry LSQ
Branch Predictor	32 K-entry gshare predictor; 4 K-entry, 4-way BTB with LRU replacement; 32-entry return address stack; 15-cycle min. branch misprediction penalty
Caches	32 KB, 8-way, 1-cycle L1 DCache; 32 KB, 8-way, 1-cycle L1 ICache; 1MB, 16-way, 10-cycle unified L2 Cache. All cache have 64 B block size with LRU replacement policy
Memory	150-cycle memory latency (first chunk), 15-cycle(rest)

Table 4. Characteristics of Evaluated Benchmarks

	gcc	crafty	eon	perlbmk	gap	sjeng	perlbench	gcc06	povray	richards	Avg.
Static IB	79.00	11.00	26.00	46.00	35.00	15.00	24.00	128.00	8.00	5.00	38.00
Dyn. IB (K)	195.00	210.00	557.00	1176.00	1237.00	317.00	1125.00	393.00	696.00	269.00	495.00
IB MPKI	1.05	1.10	1.54	8.43	5.51	1.96	8.95	1.77	1.70	4.85	2.71
IBP Acc. (%)	46.20	48.00	72.40	28.40	55.50	38.20	20.50	55.00	75.60	46.00	45.50
Base IPC	1.26	1.67	1.74	1.38	1.50	1.44	1.09	1.46	1.99	1.58	1.49
PIBP IPC Δ (%)	6.01	10.34	19.30	44.42	46.56	10.68	51.18	10.33	16.49	38.74	23.26

Notice: Static IB: static number of indirect ranches, Dyn. IB: dynamic number of indirect branches, IB MPKI: indirect-branch mispredictions per kilo instructions, IBP Acc.: indirect-branch prediction accuracy, Base IPC: Baseline IPC, PIBP IPC Δ: IPC Δ with Perfect Indirect branch prediction.

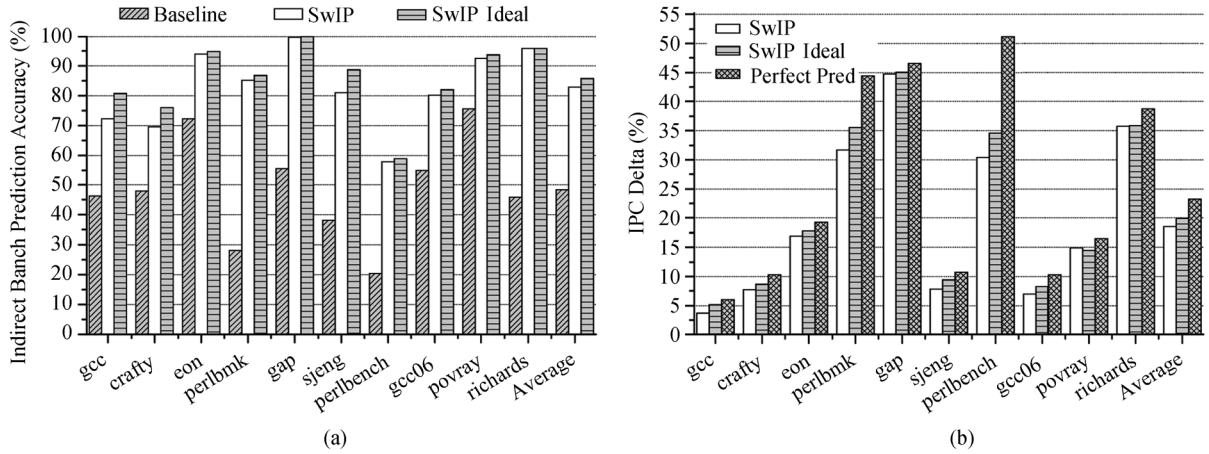

Fig.9. Performance of SWIP prediction. (a) Indirect-branch prediction accuracy, (b) IPC improvement over baseline.

Table 5. MPKI Comparisons

		gcc	crafty	eon	perlbmk	gap	sjeng	perlbench	gcc06	povray	richards	Avg.
Baseline	CB MPKI	9.45	8.23	6.57	3.69	1.56	13.53	3.28	8.25	2.55	4.82	6.19
	IB MPKI	1.05	1.10	1.54	8.43	5.51	1.96	8.95	1.77	1.70	4.85	3.69
SWIP	CB MPKI	9.64	8.39	6.57	5.53	2.35	13.95	4.27	8.65	2.55	5.24	6.71
	IB MPKI	0.54	0.64	0.33	1.74	0.03	0.60	4.86	0.77	0.51	0.36	1.04
VPC	CB MPKI	10.86	10.36	7.55	8.10	3.67	15.37	6.62	10.59	3.82	2.98	7.99
	IB MPKI	0.52	0.66	0.14	2.63	0.14	0.57	5.61	0.70	0.25	0.27	1.15

Note: CB MPKI: conditional-branch MPKI, IB MPKI: indirect-branch MPKI.

the upper bound that SWIP prediction could achieve. SWIP prediction improves the average prediction accuracy by 36.45% over that of the baseline predictor.

Table 5 shows the effect of SWIP prediction on conditional-branch MPKI and indirect-branch MPKI. The average indirect-branch MPKI is reduced significantly, from 3.69 to 1.04. We notice that there are slight adverse impacts on the conditional-branch prediction, which is mainly due to the interferences with conditional branches. Those interferences also affect the indirect-branch prediction. However, because the SWIP scheme already has obvious improvement on prediction accuracy compared to the commonly-used BTB, the reduction has been covered.

Fig.9(b) shows the performance improvement of SWIP prediction over that of the baseline processor. We have also experimented with a perfect indirect-branch predictor, which predicts all the indirect-branch targets with 100% accuracy. It shows an upper bound of performance improvement of 23.26% over that of the baseline processor. On average, SWIP improves performance by 18.56% over the baseline processor performance.

We also evaluated the performance impacts of the two unusual cases in SWIP prediction, as shown in Tables 6 and 7. Table 6 shows that the multiple-tag-matched cases occur at a very low rate, no more than 0.3% of total branch prediction for each benchmark, and subblock-overlapped cases also rarely happen. From the performance evaluation in Table 7, we conclude that most of programs do not suffer performance lost. Moreover, accurate indirect-branch prediction can recover some minor performance reduction.

Besides, we also evaluated the indirect-branch-

Table 6. Statistics of Two Unusual Cases in SWIP Prediction

	Dynamic Number			Rate	
	Multiple Tag Matched	Sub-Block Overlapped	Total Branch Prediction	Multiple Tag Matched	Sub-Block Overlapped
gcc	128 400	380	16 264 824	0.30%	0.00%
crafty	251	374	15 493 273	0.00%	0.00%
eon	0	5	14 880 471	0.00%	0.00%
perlbmk	77 696	1 187	16 392 854	0.18%	0.01%
gap	0	2	14 090 099	0.00%	0.00%
sjeng	28 567	1 420	17 316 820	0.06%	0.01%
perlbench	31 913	262	13 633 032	0.09%	0.00%
gcc06	91 620	1 358	15 787 413	0.22%	0.01%
povray	0	0	15 786 504	0.00%	0.00%
richards	0	0	7 954 221	0.00%	0.00%

Table 7. Performance Impacts of Two Unusual Cases that Cause Penalties

	Penalty Cycles		Total Cycles in Simulated Section (Sim_cycle)	Penalties/Sim_cycle	
	Multiple Tag Matched	Sub-Block Overlapped		Multiple Tag Matched	Sub-Block Overlapped
gcc	183 080	542	77 948 075	0.23%	0.00%
crafty	358	533	56 496 204	0.00%	0.00%
eon	0	0	49 736 484	0.00%	0.00%
perlbmk	110 783	1 692	55 706 758	0.20%	0.00%
gap	0	0	46 557 502	0.00%	0.00%
sjeng	40 732	2 025	65 144 134	0.06%	0.00%
perlbench	45 503	374	71 504 556	0.06%	0.00%
gcc06	130 637	1 936	65 196 146	0.20%	0.00%
povray	0	0	44 134 705	0.00%	0.00%
richards	0	0	14 143 853	0.00%	0.00%

Note: Not all unusual cases cause penalties. Some cases may have correct prediction fortunately.

insensitive benchmarks in SPEC CPU 2000 INT. From the statistics in Table 8, there is hardly any bad impact on those benchmarks. First, only if an indirect branch is fetched, it will perform the SWIP prediction flow. To conditional branches, its original branch prediction flow is not changed. Second, since SWIP prediction dynamically allocates indirect-branch entries according to the encountered indirect branches, fewer indirect branches lead to fewer contentions in the BTB and the PHT, thereby having minor effects on the conditional-branch prediction.

5.2 Comparison with Other Indirect-Branch Predictors

We first compared SWIP prediction with the previously proposed tagged target cache (TTC) predictor[7], one of the best hardware-based indirect-branch predictors for a similar set of benchmarks[14]. We simulated various sizes of the TTC predictor, from 256-entry to 64 K-entry. Each entry of a TTC predictor has a 4-byte target and a 2-byte tag, giving a total size of 1.5 KB to 384 KB respectively. Figs. 10(a) and 10(b) illustrate the comparison results of the MPKI and the performance impact. On average, SWIP prediction for the baseline processor (32 K-entry PHT) achieves the equivalent performance provided by a 48 KB TTC predictor. As shown in Table 9, in six benchmarks, the SWIP predictor performs at least as well as a 96 KB TTC predictor.

Table 5 shows the MPKI comparison between an SWIP predictor and a VPC predictor[9] using the baseline predictor. The maximum iteration number of VPC prediction is set to 12. The total indirect-branch MPKI

Table 8. Evaluations of Indirect-Branch-Insensitive Benchmarks in SPEC CPU 2000 INT

	Dynamic IB Number	Total Branch Prediction	IB Pred Acc w/o SWIP	IB Pred Acc w/ SWIP	CB Pred Acc w/o SWIP	CB Pred Acc w/ SWIP	IPC Δ
gzip	10	18 466 151	90.00%	90.00%	93.56%	93.56%	0.00%
vpr	1	22 365 837	100.00%	100.00%	85.35%	85.35%	0.00%
mcf	2	50 129 512	50.00%	50.00%	94.67%	94.67%	0.00%
parser	0	14 861 471	–	–	92.08%	92.08%	0.00%
vortex	10 259	19 191 571	73.56%	74.10%	97.56%	97.56%	0.00%

Note: IB: indirect branch, IB Pred Acc: indirect-branch prediction accuracy, CB Pred Acc: conditional-branch prediction accuracy.

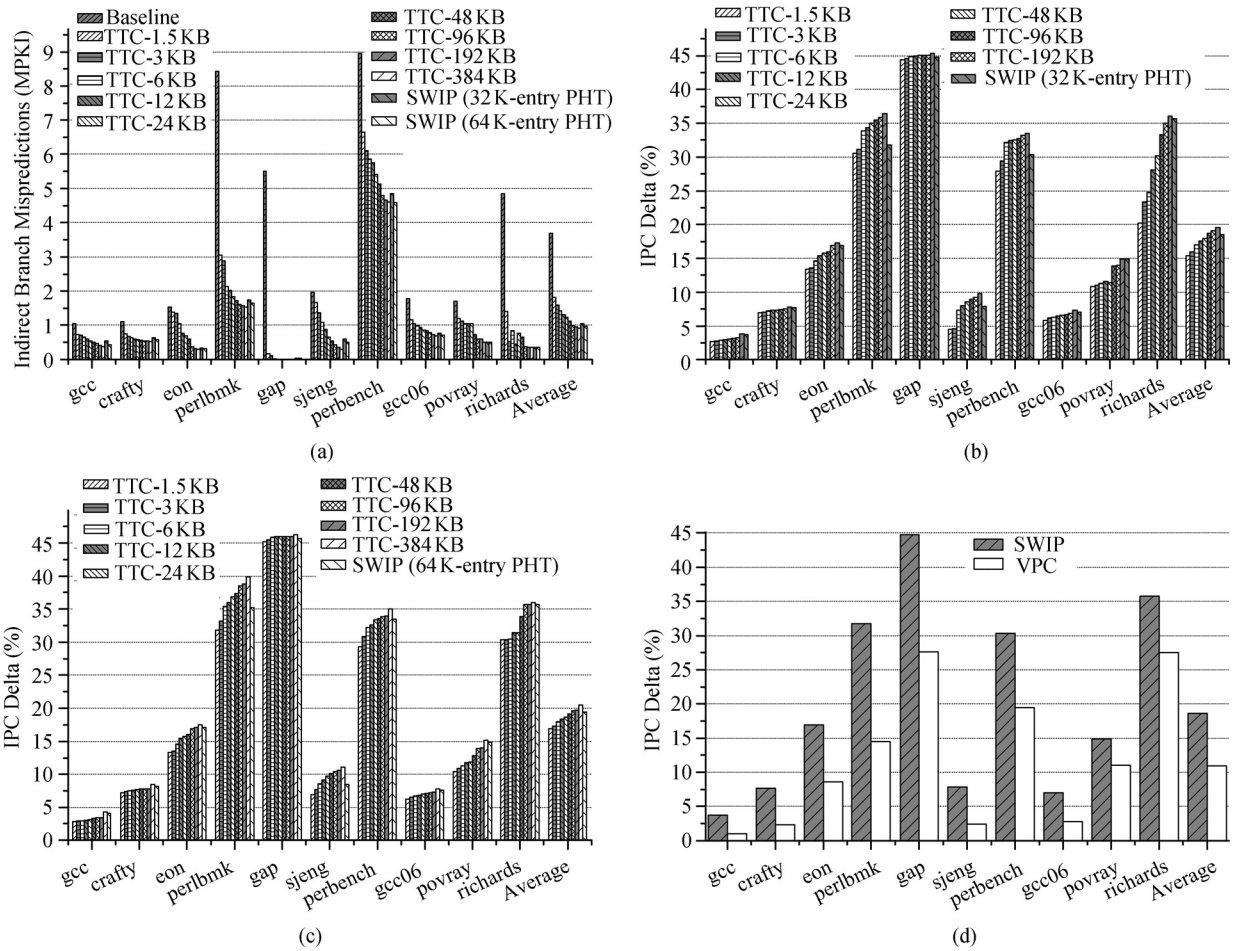

Fig.10. SWIP predictor vs other predictors: 1) TTC comparison: (a) MPKI. (b) IPC improvement with 32 K-entry PHT. (c) IPC improvement with 64 K-entry PHT. 2) VPC comparison: (d) IPC improvement.

Table 9. Size of TTC Predictors that Provide the Equivalent Performance as SWIP Prediction in Terms of IPC

TTC Size (B)	gcc	crafty	eon	perlbmk	gap	sjeng	perlbench	gcc06	povray	richards
SWIP 32 K-entry PHT	192 K	192 K	96 K	3 K	6 K	12 K	3 K	96 K	192 K	192 K
SWIP 64 K-entry PHT	384 K	384 K	192 K	6 K	6 K	6 K	48 K	384 K	384 K	192 K

of VPC prediction is similar to that of the SWIP scheme. However, because VPC requires more cycles to reach the same prediction accuracy of SWIP prediction, its performance improvement is lower than that of SWIP prediction. On the other hand, VPC's adverse impact on conditional branch predictions is also greater than that of SWIP prediction. In VPC prediction, the generated virtual branch in each cycle takes a PHT entry. For one branch occurrence, VPC prediction usually occupies more PHT entries during many prediction iterations, while SWIP prediction takes only two entries. In general, SWIP has better spatial and time costs than the VPC scheme for the baseline processor. Fig.10(d) shows the performance comparison between the SWIP predictor and the VPC predictor. It shows that the SWIP predictor outperforms the VPC predictor by 7.65%.

5.3 Impact of Supporting Multiple PHT Lookups per Cycle

If the processor provides the ability to access two corresponding PHT entries per cycle[29], the PHT access in one cycle can provide the full set-way index pointer, shortening the prediction to two cycles. Fig.11 shows the performance impact of supporting the 2-entry PHT access. From the statistics, SWIP prediction with 2-entry PHT access per cycle achieves performance improvement by 3% over that of the basic SWIP scheme which accesses only one PHT entry per cycle. The

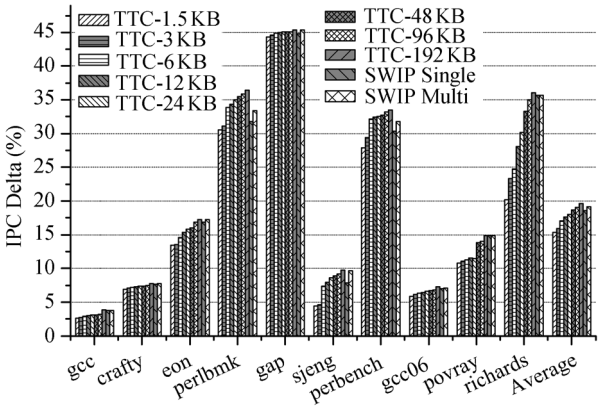

Fig.11. IPC improvement of the single-access and multi-access SWIP prediction.

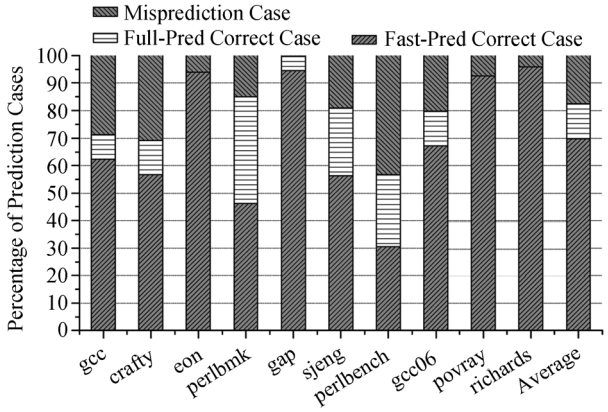

Fig.12. Distribution of the access mode in SWIP prediction.

reasons for the minor accelerating impact are as follows. First, SWIP prediction originally has an accelerating mechanism. The fast-pred mode takes only two cycles to finish the indirect-branch prediction. Fig.12 shows the distribution of the number of cycles needed to obtain the predicted target. On average, 69.67% of indirect-branch predictions are finished correctly only in two cycles (fast-pred case). Only 10% are in full-pred mode. If the PHT supports 2-entry access, it only accelerates those 10% cases, shortening 1-cycle latency for each of those indirect-branch predictions. Second, according to the previous analysis, the main performance constraint of SWIP prediction is the PHT interferences between conditional branches and the stored set-way index pointers. Accelerating the fetch speed of set-way index pointers cannot solve those interferences.

5.4 Sensitivity of SWIP Prediction to Microarchitecture Parameters

5.4.1 Different Branch Predictors

Since SWIP prediction utilizes existing branch components, different branch predictors may influence the efficiency of SWIP prediction.

Fig.13 illustrates the effects of SWIP prediction under various PHT sizes. The conclusion is that the indirect-branch prediction accuracy becomes higher as the PHT size increases. This is because 1) more branch history is used to detect indirect-branch occurrences (for example, 2 K-entry PHT actually uses 11-bit GHR, whereas 64 K-entry PHT uses 16-bit GHR), 2)the entries storing the set-way index pointers can be distributed more widely when the PHT size is greater, resulting in less interference among original conditional branches. Table 9 and Fig.10 show the comparisons between an SWIP predictor with 64 K-entry PHT and the larger TTC predictors. SWIP prediction achieves performance improvement equivalent to that provided by a 192 KB TTC predictor in six benchmarks. On average, it performs as well as a 96 KB TTC predictor.

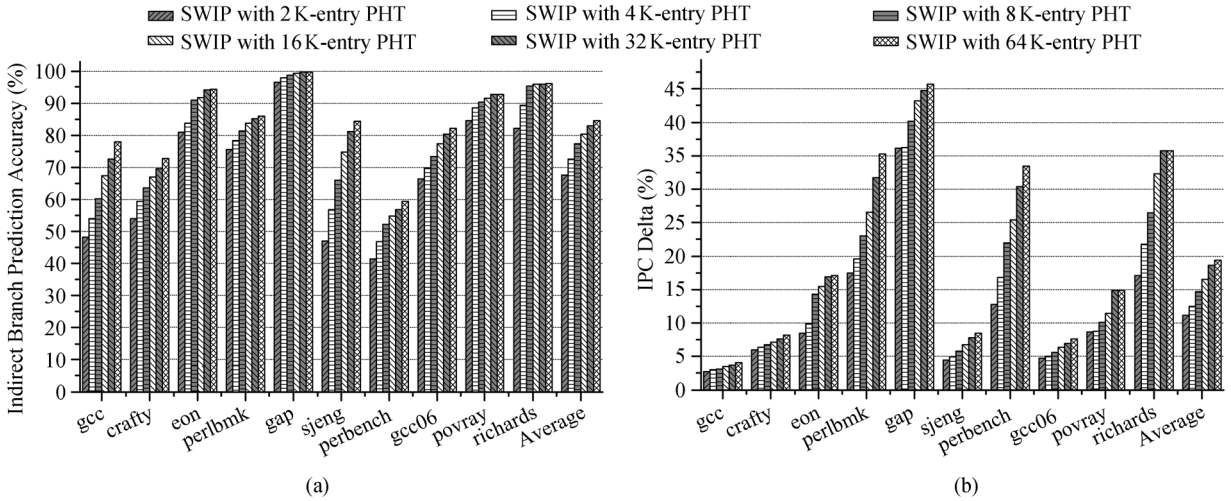

Fig.13. Indirect branch prediction accuracy (a) and performance improvement (b) of SWIP prediction under different PHT sizes.

Table 10. Effect of Different Conditional-Branch Predictors

Cond. BP	Baseline MPKI		SWIP Predictor MPKI		SWIP Predictor IPC Δ (%)
	Cond.	Indir.	Cond.	Indir.	
GS-32 K	6.19	3.69	6.71	1.04	18.56
GS-64 K	5.37	3.69	5.70	0.96	19.43
Hybrid	3.77	3.94	5.93	1.03	18.90

Notes: GS-32 K: gshare with 32 K-entry PHT, GS-64 K: gshare with 64 K-entry PHT, Hybrid 40 K: hybrid predictor.

We next evaluated the efficiency of SWIP prediction using two tables in the hybrid branch predictor to store set-way index pointers. The evaluated hybrid predictor contains a 32 K-entry gshare PHT, a 8 K-entry bimodal, and a selector. Those pointers are stored in either the bimodal or gshare tables in different cases. Table 10 shows, on average, SWIP prediction improves performance by 18.90% over that of the baseline with the hybrid predictor. The space size storing the set-way index pointers determines the IPC improvement.

5.4.2 *Different BTB Sizes*

Since the SWIP predictor occupies only limited number of BTB entries needed to store indirect-branch targets dynamically, SWIP prediction provides significantly performance improvements even with small BTB sizes (Table 11).

Table 11. Performance Improvement of SWIP Prediction under Different BTB Sizes

BTB Entries	Baseline Indir. Pred. Acc. (%)	SWIP Predictor Indir. Pred. Acc. (%)	SWIP Predictor IPC Δ (%)
8 K	45.48	81.90	18.58
4 K	45.47	81.74	18.56
2 K	45.47	81.33	18.12
1K	44.78	80.82	17.45
512	42.79	79.85	17.20

5.4.3 *SWIP Prediction on a Less-Aggressive Processor*

According to the analysis[16], dual-issue processors are optimal for the energy-performance tradeoff space. Table 12 shows the effect of SWIP prediction on this kind of less-aggressive processor, which has 8-stage pipeline, 2-wide issue width, 128 RUU, 32 K-entry PHT, and 512-entry BTB. Because of a smaller branch misprediction penalty and smaller instruction window size, it improves the prediction accuracy by 36.78%, resulting in 6.36% performance improvements over that of the baseline processor.

5.5 Effect of SWIP Prediction on Energy Consumption

The parameters of this energy evaluation are derived from the Cacti tool[30], which is configured with 65 nm technology, 1 GHz.

As Fig.14 shows, the energy of SWIP prediction is reduced by 14.34% over that of the baseline processor, and it also performs better than other indirect-branch predictors. Unlike large TTC predictors, it reuses the existing branch components instead of employing large dedicated storage, so there is no energy consumed by additional storage. SWIP prediction achieves energy reduction by making fewer pipeline flushes and fewer wrong-path instruction executions due to high indirect-branch prediction accuracy. Moreover, its maximum power consumption in one cycle would not increase because it accesses the same components as the conditional branch prediction.

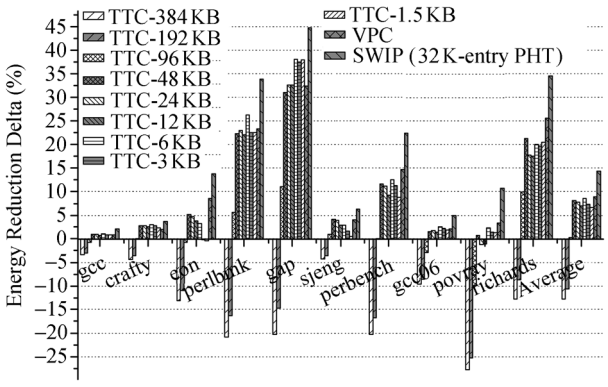

Fig.14. Effect of SWIP prediction on energy reduction.

6 Conclusions

This paper proposed and evaluated a complexity-effective indirect-branch prediction mechanism, SWIP prediction. It exploits the existing BTB to store multiple indirect-branch targets, and redefines the direction predictor to store set-way index pointers. SWIP prediction uses the pointers to implement a fast and accurate indirect-branch prediction, without requiring large dedicated storage or additional compiler assistant.

Table 12. SWIP Prediction on a Less-Aggressive Processor

	gcc	crafty	eon	perlbmk	gap	sjeng	perlbench	gcc06	povray	richards	Average
Pred. Acc. Δ (%)	34.55	22.61	25.80	57.92	44.46	43.57	40.50	31.09	17.22	50.03	36.78
IPC Δ (%)	2.98	4.35	8.76	14.62	16.92	4.59	15.38	4.27	4.99	13.88	6.26

According to the experiments, SWIP prediction achieves attractive performance improvement and reduces the energy used. It can be concluded from experimental statistics that the main performance constraint of SWIP prediction is the interference between the inserted set-way index pointers and the original conditional-branch direction information. Our future work will focus on how to design a better spacial distribution algorithm for set-way index pointers. In addition, we will explore ways in which the concept of SWIP prediction can be effective in other processor-design areas that share set-associative cache structures, such as cache, TLB, and load/store queue.

References

[1] Jiménez D A, Lin C. Dynamic branch prediction with perceptrons. In *Proc. the 7th HPCA*, Jan. 2001, pp.197-206.

[2] McFarling S. Combining branch predictors. Technical Report TN-36, Western Research Laboratory, June 1993.

[3] Kim H, Mutlu O, Stark J, Patt Y. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proc. the 38th MICRO*, Nov. 2005, pp.43-54.

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

[5] Yeh T Y, Patt Y. A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. the 20th ISCA*, May 1993, pp.257-266.

[6] Calder B, Grunwald D, Zorn B. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 1994, 2(4): 323-351.

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

[8] Driesen K, Hölzle U. The cascaded predictor: Economical and adaptive branch target prediction. In *Proc. the 31st MICRO*, Nov. 30-Dec. 2 1998, pp.249-258.

[9] Kim H, Joao J A, Mutlu O, Lee C J, Patt Y, Cohn R. VPC prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *Proc. the 34th ISCA*, June 2007, pp.424-435.

[10] Driesen K, Hözle U. Accurate indirect branch prediction. Technical Report TRCS97-19, University of California, March 1998.

[11] Kalamatianos J, Kaeli D R. Predicting indirect branches via data compression. In *Proc. the 31st MICRO*, Nov. 30-Dec. 2, 1998, pp.272-281.

[12] Driesen K, Hölzle U. Multi-stage cascaded prediction. In *Proc. the 5th Euro-Par Conference on Parallel Processing*, Aug. 31-Sept. 3, 1999, pp.1312-1321.

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

[14] Joao J A, 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*, March 2008, pp.80-90.

[15] Farooq M, Chen L, John L K. Value based BTB indexing for indirect jump prediction. In *Proc. the 16th HPCA*, Jan. 2010.

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

[17] Hameed R, Qadeer W, Wachs M *et al*. Understanding sources of inefficiency in general-purpose chips. In *Proc. the 37th ISCA*, June 2010, pp.37-47.

[18] Lin Y L. Essential Issues in System-On-a-Chip Design, Springer-Verlag, 2006.

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

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

[21] Roth A, Moshovos A, Sohi G S. Improving virtual function call target prediction via dependence-based pre-computation. In *Proc. the 13th ICS*, June 1999, pp.356-364.

[22] Gochman S, Ronen R, Anati I *et al*. The Intel® Pentium® M processor: Microarchitecture and performance. *Intel Technology Journal*, 2003, 7(2): 21-59.

[23] IBM. IBM PowerPC 970FX RISC Microprocessor user's manual. Version 2.3, March 2008.

[24] SPEC. Standard Performance Evaluation Corporation. http://www.spec.org, July 2011.

[25] Burger D, Austin T M. The simpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 1997, 25(3): 13-25.

[26] Brooks D, Tiwari V, Martonosi M. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. the 27th ISCA*, June 2000, pp.83-94.

[27] Wolczko M. Benchmarking Java with the Richards benchmark. http://research.sun.com/people/mario/java_benchmarking/richards/richards.html, July 2011.

[28] Perelman E, Hamerly G, Van Biesbrouck M *et al*. Using SimPoint for accurate and efficient simulation. In *Proc. SIGMETRICS*, June 2003, pp.318-319.

[29] Yeh T Y, Marr D, Patt Y. Increasing the instruction fetch rate via multiple branch prediction and branch address cache. In *Proc. the 7th ICS*, July 1993, pp.67-76.

[30] Thoziyoor S, Muralimanohar N, Ahn J N, Jouppi N P. CACTI 5.1. Technical Report HPL-2008-20, Hp Labs, 2008, http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html.

Zi-Chao Xie is currently a post-doctoral researcher in Peking University. He received his B.E degree in microelectronics from Peking University in 2006, and then received his Ph.D. degree in computer science from Peking University in 2012. His research interests include processor microarchitecture, multicore system, and hardware/software co-design.

Dong Tong received his Ph.D. degree in computer science from Harbin Institute of Technology in 1999. He is now a professor in the School of Electronics Engineering and Computer Science, Peking University. His research interests include processor architecture, reconfigurable computing, interconnection network, and System-on-Chip. He is also 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.

Qin-Qing Shi is a graduate student in the Department of Geography, University of Maryland, USA. She received her B.S. degree in the School of Information Science and Technology of Peking University, China. Currently, her research interest is the application of computer science technologies in the area of remote sensing of geography.

Xu Cheng is a professor and Ph.D. advisor in Peking University. He is the director of Microprocessor Research and Development Center, Peking University and a member of Advisory Committee for State Informatization. His research interests include high performance microprocessor, System-on-Chip, embedded system, instruction-level parallelism, hardware/software co-design and compiler optimization. He is a member of CCF.