PAPER

Multiple Branch Prediction for Wide-Issue Superscalar*

Shu-Lin HWANG†, Che-Chun CHEN††, *Nonmembers, and* Feipei LAI††, *Member*

SUMMARY Modern micro-architectures employ superscalar techniques to enhance system performance. Since the superscalar microprocessors must fetch at least one instruction cache line at a time to support high issue rate and large amount speculative executions. There are cases that multiple branches are often encountered in one cycle. And in practical implementation this would cause serious problem while there are variable number of instruction addresses that look up the Branch Target Buffer simultaneously. In this paper, we propose a Range Associative Branch Target Buffer (RABTB) that can recognize and predict multiple branches in the same instruction cache line for a wide-issue micro-architecture. Several configurations of the RABTB are simulated and compared using the SPECint95 benchmarks. We show that with a reasonable size of prediction scope, branch prediction can be improved by supporting multiple / up to 8 branch predictions in one cache line in one cycle. Our simulation results show that the optimal RABTB should be 2048 entry, 8-column range-associate and 8-entry modified ring buffer architecture using PAs prediction algorithm. It has an average 5.2 IPC_f and branch penalty per branch of 0.54 cycles. This is almost two times better than a mechanism that makes prediction only on the first encountered branch.

key words: *branch prediction, wide-issue superscalar, branch target buffer, branch penalty*

1. Introduction

A deeply pipelined micro-architecture can potentially achieve extremely high performance since the delay of each pipeline stage is very short. But several factors may break the pipeline flow: one is branch misprediction. So the branch prediction plays an important role in high performance micro-architectures. In order to reduce the pipeline bubbles produced by branch instructions, various mechanisms to reduce branch misprediction penalty have been proposed: delayed branches, predicated execution [1], static and dynamic branch predictions combined with the BTB. Many techniques for increasing branch prediction accuracy have been proposed, such as the BTB [2], [3], Two-Level Adaptive Branch Prediction [4], [5], Branch Classification [6], etc. To support high issue rate and large amount of speculative execution, superscalar microprocessors must fetch more than one instruction cache line in order to fetch multiple basic blocks per cycle [9], [13], [14]. Because multiple branches may be encountered in a cycle, the branch prediction architecture must be able to recognize and predict multiple branches per cycle [8], [13], [16]. The trace cache has been proposed as a mechanism for providing increased bandwidth by allowing the processor to fetch across multiple branches in a single cycle [18]. The trace cache works in concert with a multiple branch predictor and trace cache lines are constructed by the fill logic.

To simplify fetch unit design for implementation, practical superscalar processors employ one fetch unit and only fetch one instruction cache line per cycle. So, we propose a Range Associative Branch Target Buffer (RABTB) that can recognize and predict multiple branches in a single cache line. Only one incoming fetch address is needed to look up the RABTB, but our mechanism can make multiple predictions.

Due to the identification mechanism, most early solutions only predict the first branch instruction in each instruction cache line. This will cause the degradation of performance when there are taken branches after the first one. Therefore, it is necessary to evaluate the benefits and effects of predicting multiple branches in a single cache line.

This paper is organized in 7 sections. Section 2 presents the primitives of multiple branch predictions for these branches in one instruction cache. Section 3 summarizes some related work. Section 4 provides an overview of the multiple branch identifications by the RABTB. Section 5 describes the RABTB architecture in detail. The simulation model and results are showed in Sect. 6. Finally, Sect. 7 concludes the paper.

2. Primitives

The early prediction mechanism consists of an array of two-bit saturation up-down counters associated with each branch instruction. Whenever the outcome of a branch is taken, its corresponding counter is increased. Otherwise, decreased. In the Fetch stage of pipeline, the prediction unit looks up the BTB with instruction address. If identical tag exists and the value of its counter is greater than 1, we predict this branch as

Manuscript received July 31, 1998.
Manuscript revised December 18, 1998.

†The author is with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan.

††The authors are with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.

*This work was partially supported by the National Science Council, Taiwan, ROC, under contract No.NSC-2622-E-009-009.

taken and feed the target address to the fetch unit for the next fetch.

Intuitively, we can easily send the fetch addresses to the BTB as the original BTB [2], [3] architecture does. But there may be many instructions fetched in a superscalar microprocessor per cycle, the fetch unit then will send multiple instruction addresses to the BTB for instruction identification. For the sake of the variable instruction length and number under x86 architecture, it is difficult to implement.

Firstly, we assess the potential benefit of predicting multiple branches in a cache line. Assume branch instructions are randomly distributed in a cache line and the probability of taken branch is p. Then Fig. 1 shows the distribution of taken branch when there are four branches in a single line. We will derive the probability $T(n)$ of finding a taken branch when there are n branches in a line.

$$\begin{aligned}
&T(1) = p,\\
&T(2) = T(1) + (1-p) * p,\\
&T(3) = T(2) + (1-p) * (1-p) * p,\\
&\dots\\
&T(n) = T(n-1) + (1-p)^{n-1} * p
\end{aligned}$$

If there are n branches in a cache line, then the probability, Taken_inc(n), of encountering a taken branch after the first branch is:

$$\begin{aligned}
\text{Taken\_inc}(n) &= T(n) - T(1)\\
&= (1-p) * p + (1-p)^2 * p + \dots\\
&\quad + (1-p)^{n-1} * p\\
&= p * \left(\sum_{k=1}^{n-1} (1-p)^k \right)
\end{aligned}$$

In the SPECint95 suite, for some input patterns, conditional branches represent nearly 56.8% of the total branch instructions (refer to Table 2) and the taken probability of a conditional branch approximates 0.59 in our simulation. So,

$$p = 0.59 * 0.568 + 1.0 * 0.432 = 0.767.$$

For $p = 0.767$, if $n = 2$ then Taken_inc(2) $= 0.767 * 0.233 = 17.87\%$, and if $n = 4$ then Taken_inc(4) $= 0.767 * (0.233 + 0.0543 + 0.01264) = 23\%$.

If we can predict the branches after the first branch correctly using the RABTB, then we expect that prediction accuracy may be improved better than a conventional BTB scheme that only makes one prediction and identification for the first branch in a single cache line.

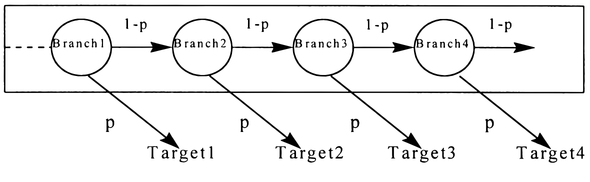


Fig. 1 Four branches at one cache line.

Our simulation results indicate that the average occurrence of branch per cache line (assume 32 bytes) is two (see, Table 2). This strongly suggests that making predictions on multiple branches in each clock cycle is worthwhile.

3. Related Work

A superscalar processor fetches, issues, and executes multiple instructions in one cycle to exploit instruction level parallelism. It must predict the fetch address prior to the first pipeline stage to avoid pipeline bubbles whenever a branch is taken.

The BTB [2], [3] is a cache which stores data on recently executed branches. Each BTB entry generally has the following fields: branch target address, branch prediction information, and branch instruction address. If there is a hit in the cache, then we know that it is a branch instruction and make prediction by the information kept in the entry. If it is predicted taken, then the target address of this branch will be sent back to the Fetch Unit as the next instruction address. When the branch is actually resolved in the execution stage, the BTB entry will be updated with the correct prediction path and target address. Surely, the pipeline will be flushed if the actual target address is different from the one we predicted.

Due to the identification mechanism, this BTB architecture only predicts the first branch instruction in each instruction cache line. It cannot make multiple identifications if several branches are fetched in one clock cycle.

Yeh & Patt proposed two index schemes for solving the branch identification problem: Basic Block and Fetch Address Based schemes [8].

A basic block contains at most one branch instruction, so the address of the first instruction can be used to identify a specific branch. The first approach identifies a branch using its basic block starting address and predicts the branch when the processor starts fetching the basic block. However, a branch may jump into the middle of a piece of straight-line code that ends with a branch and thus break a basic block into two. This will lead to multiple identifications of a branch. This means that two starting addresses may be used to identify one branch. Another problem is that a basic block may be very large, so the newly-predicted fetch address should not be used until the entire basic block has been fetched. The second scheme uses the address of an instruction cache line to identify the first branch in the cache line but uses the starting addresses of a basic block to identify subsequent branches in the same line.

When one cache line is being fetched, its address will hit in the BTB if the data associated with the first branch in this line was also in the BTB. A problem is that a basic block may span several cache lines.

There is a complicated mechanism proposed in [9] for predicting multiple branches and fetching multiple non-consecutive basic blocks in each cycle. It extended the Branch Address Cache (like the BTB) to support two or three branch predictions by combining their branch information in the same entry. A BAC supporting two branch predictions per cycle would have a total of 212 bits per entry. Its hardware cost is too high; its entry is 3 times the size of a conventional BTB entry storing data for one branch. There is also another problem: the fetch unit must be able to fetch multiple non-consecutive instruction cache lines every cycle.

Conte et al. [13] also proposed an interleaved branch target buffer to predict multiple branch targets and detect short forward branches that stay within the same cache line. He introduced a mechanism called the *Collapsing*. The scheme features an interleaved-coupled BTB/BHT providing one entry to each instruction of a cache line. The major drawback is the requisite use of an address-only based prediction scheme. Moreover, as the I-cache line size keeps growing in current processors, the interleaving factor of the BTB grows as well and the collapsing logic become more complex.

In addition to proposing a caching structure, Seznec et al. [15] also presented a multiple branch predictor capable of predicting two branches per cycle. The originality of their mechanism is to use information associated with the current instruction block to predict the block following the next instruction block. They have introduced the Two-Block Ahead Branch Predictor and can be extended to a multiple-block ahead branch predictor fetching multiple basic blocks in a single cycle.

Franklin and Dutta [14] proposed subgraph oriented branch prediction mechanisms that use local history to form a prediction that encodes multiple branches. All parameters required to describe a subgraph are stored in a Subgraph History Table (SHT). Their scheme mostly relies on compiler work to partition the CFG (Control Flow Graph) into tree like subgraph of depth 3. Sine each entry in the SHT holds a rigid subgraph structure, there might be many underutilized or wasted fields. In [19], the authors improve the prediction accuracy of control flow prediction by using different extents of correlation. They also attempt to increase the tree depth by including three conditional branches to increase the fetch size.

The trace cache has been proposed as a mechanism for providing increased bandwidth by allowing the processor to fetch across multiple branches in a single cycle [18]. Since the heart of the trace cache is its ability to fetch multiple basic blocks each cycle, effective multiple block branch predictor is critical to its performance. Trace cache lines are constructed by the fill unit. The fill unit will attempt to maximize the size of the segment by combining newly arriving instructions with instructions latched from previous cycles. In [17], they take a different approach to next trace prediction — they treat the traces as basic units and explicitly predict sequence of traces. They propose and study next trace predictors that collect histories of trace sequences and make predictions based on these histories.

In above, we have introduced some mechanisms that can fetch multiple basic blocks in a single cycle. These mechanisms need complex fetch mechanisms, branch target buffer, or branch predictors, so the budget is too expensive to implement. To simplify the fetch unit design for implementation, our design only employs one fetch unit and can only fetch one instruction cache line per cycle. This paper presents a Range Associative Branch Target Buffer (RABTB) that can recognize and predict multiple branches at the same cache line for a wide-issue micro-architecture. The RABTB is of a simple architecture and easy to implement practically.

4. Multiple Branch Prediction

4.1 Multiple Branch Identification

Our design is one part of the x86 compatible NSC98 processor [20], so we focus on the x86 compatible superscalar architecture. Because the x86 has variable instruction lengths, there will be variable number of instructions per cache line (32 bytes assumed). As a practical limit, we assume that maximums of 8 instructions are issued per clock cycle. At first, it seems that we need to provide the maximum address ports, eight, for branch identification and prediction. This enables simultaneous search for these addresses in the BTB, finding associated Branch Histories (level one branch information), then indexing to the Pattern History Table using Branch History for prediction.

As previously described, a serious problem may occur when the implementation was taken into consideration. A large number of BTB read ports are thus needed for multiple branch identifications — 8 in this case. The hardware cost is too expensive, so we designed another indexing scheme (RABTB) based on Yeh and Patt's Fetching Address scheme [8]. It is a modification of the original BTB architecture. The main difference is that the traditional BTB finds exactly one of the associative branches in the BTB, but ours predicts multiple branches' targets within a specific range then returns the one we need for prediction.

4.2 Range Associativity

"Range Associativity" means that the RABTB gathers branch information together and make predictions in a specific range. The detailed functions are: when a fetch address arrives, the RABTB splits the address into three parts: tag, index, and offset. The RABTB first uses the index to locate a row and compares the tag with the incoming tag of fetch address. If it matches, the RABTB filters out those branches whose offsets are smaller than the incoming offset and obtains the data for those branches after the fetch address within the specific range. Predictions are made based on this information. Finally, the RABTB sends the target address of the first predicted-taken branch to the fetch unit for fetching the next instruction cache line. After the execution stage, we update the prediction information in the RABTB. Here two-level adaptive prediction is our basic algorithm for prediction.

Our RABTB is organized into N-column, S-row memory. Each row memory is separated by instruction address. For example, we use 8 bits of the fetch address to select one of the 256 rows; each row can store N sets of branch information. When an address is sent to the RABTB, the lower R bits, the Offset, specify a "range" — they don't select a row. With this range information, we can filter out the branches we do not need.

The varying number of branches in a cache line may cause inefficient use of the dedicated fields of a row. Our simulation results show that the average branch number per cache line ranges from 1.8 to 2.4 with a maximum value of 8, meaning that some columns of a RABTB row may be not used. To utilize all columns of a RABTB row, we can add the tag to each field in the RABTB, as shown in Fig. 2.

The next problem is the tradeoff between the amount of branch information in a row and the number of rows for the same hardware resource. Using results from Figs. 9 and 10, we choose a row with 4 to 8 sets of branches information.

The main consideration of our design is on the range and the number of branches supported in this range. Usually the range is the fetch size of the fetcher in each clock cycle. With a smaller range, we would get less improvement but the comparison and selection circuits is simpler. And, with a wider range, we would need the fetch unit with the ability to fetch larger line size at once and the comparison and selection circuits become very complex.

It is not necessary to speculatively update the history in the BTB for prediction [7]. This implies we can shift the action of prediction from the Fetch Stage to the Execution Stage. Since there is only one Branch Unit in the simulated machine and one branch instruction is executed at one time, so only one write port is needed for updating the RABTB. After making prediction in advance, we store the result of prediction into branch information field of the RABTB entry. Then in the Fetch Stage, we can select target address based on the previously predicted result.

5. The Architecture of the RABTB

5.1 The 8-Column RABTB

Figure 3 shows a block diagram of an 8-column RABTB with two-level branch prediction mechanism and an eight-entry modified ring buffer. The RABTB is organized into 256 rows and 8 columns. If the range is 32 bytes, then each column needs 67 bits. Each column contains a tag (20 bits), offset (5 bits), branch type (2 bits), LRU bit for replacement (1 bit), last prediction result (1 bit), branch history (6 bits), and target address (32 bits). The RABTB has two read ports and one write port. One read port is for branch prediction, the other is for updating branch information. The write port is also for updating.

The RABTB has two operation stages: prediction and update. In the prediction stage, the RABTB identifies multiple branches in a cache line and decides a target address for the Fetch Unit. Information needed for prediction is updated in the Update stage after the actual branch instruction is executed in the Branch Unit.

Prediction Stage

The algorithm for Prediction Stage is:

While the Fetch Unit fetches a cache line with a fetch address

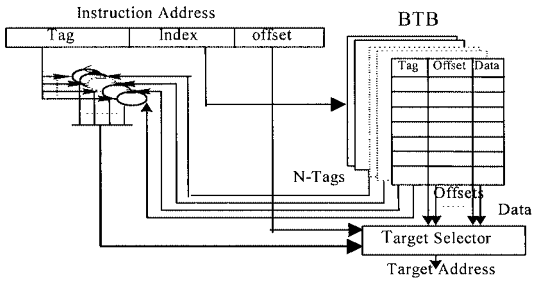


Fig. 2 Block diagram of the RABTB.

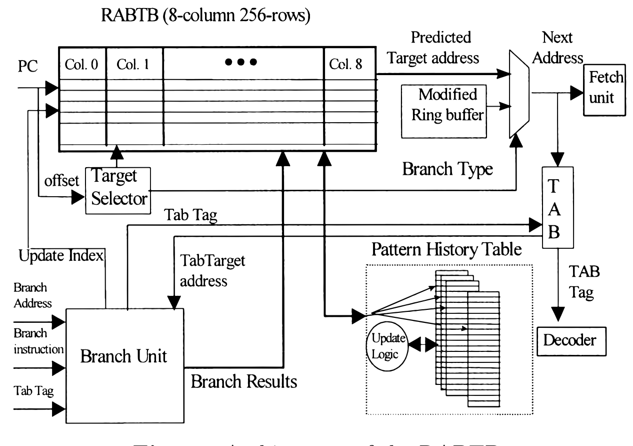


Fig. 3 Architecture of the RABTB.

```
{
The RABTB searches for branches after the fetch address of this line;
The RABTB then selects the target address of the first predicted-taken branch, and sends it to the Fetch Unit for the next fetch;
}
```

The RABTB begins with an empty buffer, and its Pattern History Table is initialized with 2 (10_2). Since the possibility for a branch to be taken is higher than not taken. If we use 1 or 0 as initial value when a new branch is taken and added to the RABTB, then the branch will be predicted as not taken while it hits in the RABTB next time. There are extra mispredictions resulted from the mismatch between branch behavior and initial value.

When the fetch unit starts to fetch a cache line with a fetch address, the RABTB monitors this signal and the prediction mechanism starts working. The RABTB compares the upper 20 (tag) bits to the tags of the 8 indexed RABTB entries. If a match is found, it selects those entries with offset larger or equal to the lower 5 bits (range) of the fetch address. Then it checks their "last prediction" bits and sends the target address of the first entry marked "predicted taken" to the Fetch Unit for next fetch. We complete this task with the help of the Target Selector which checks the validity of the 8 columns of the RABTB row and filters out tag values different from those of the fetch addresses then finds out the first predicted-taken branch.

Update Stage

When a branch instruction is executed in the Branch Unit, it will send its tag to the Target Address Buffer for the prediction of target address of this branch — an invalid tag implies no prediction was made by the RABTB. The Branch Unit then compares the calculated target address with that returned from the Target Address Buffer (TAB), a "Misprediction" signal will be issued if there is a mismatch and the target address is sent to the fetch unit for the next fetch. The RABTB will be updated whenever there is a branch.

On prediction, the RABTB firstly uses the branch address sent from the Branch Unit to index a branch row, then locates the branch information with the same offset — if none is found, the replacement algorithm is applied. The Update algorithm is:

```
While there is a branch instruction issued to the Branch Unit
{
// Because there are three RABTB accesses, these actions must be pipelined.
Three-stage pipelined Update Unit works as follows:
1. Read the RABTB row where the incoming branch resides;
2. Append the new branch result to the branch history and
3. Read the PHT with the branch information;
    3.1 make prediction with the new history;
    3.2 replace or merge within the RABTB row;
    3.3 write the new branch information back to the RABTB;
}
```

5.2 Target Address Buffer

For the Branch Unit to check for misprediction, there must be a mechanism for each branch instruction to pass the predicted target address from the RABTB to the Branch Unit. An intuitive method is to append the predicted target address to the instruction. This will result in a large waste on circuit and buffer in the pipeline. A more efficient way is that the predicted address is stored in a target address buffer called the TAB, and the TAB will append an index to each instruction. The Branch Unit uses this index to obtain the predicted target address from the TAB for comparison.

Because our design focuses on an x86 compatible superscalar architecture. And, the pipeline stages can be larger than 4 between the fetch and execution stage. The number of cycle between the fetch and execution stage is assumed to be 4 in simulated micro-architecture. The simulation results from Table 2 indicate that the average occurrence of branch per cache line is 2. Ideally, the TAB is designed as an 8-entry buffer to store predicted address.

If the size of the TAB is too small, then the TAB full will dramatically increase and prevent RABTB from making prediction. We also make simple simulations to realize the condition. When the sizes of the TAB are 4 and 6, the TAB full is occurred frequently. If we use one entry for the prediction on a single cache line while one predicted address is generated per cycle. Therefore the 6 entry TAB should be enough.

5.3 The Modified Ring Buffer for Return Target Prediction

The target address of the return instruction changed dynamically when there are several callers calling the same subroutine. Its target address must be associated with the caller instruction. In the case, the BTB does have some disadvantage when executing the return instruction. A "Return Stack" is implemented to help the BTB to predict the return target address [11], [12]. What is a return stack? It's a stack where the next instruction address of the caller instruction will be stored as return address when the caller instruction calls subroutine. Whenever there is a CALL instruction, the next effective address of this CALL instruction will be considered as the return address and pushed into the stack. The return address will be popped off during a BTB hit on a RET instruction then it is considered as a target address for this RET instruction. By this way, we can avoid the problem that the return instruction uses other's return address.

If the subroutine calls are nested more than the size of the return stack, the BTB normally will not provide prediction and will flush the stack for providing spaces for the incoming CALL instructions' return address. To overcome the drawback, we present a new scheme, the ring buffer, for the target prediction of return instructions. The ring buffer uses a "Circular Queue" as the basic structure. When there are many consecutive call instructions and without return instructions, the circular queue may be full and will overlap the entry that stores the return address of the previous call. Different to the return stack that flushes the contents, the circular queue still store the most recent several entries (depends on the size) of return address and can provide the prediction for those corresponding return instructions.

When a process is running recursively, some consecutive call instructions that have the same return addresses will be stored in the entry of return target no matter a return stack or a ring buffer is implemented. We employ the "Modified Ring Buffer" that adds a counter for each entry of the circular queue in the RABTB. When the next return address is the same as the last one stored in the circular queue, the modified ring buffer will not store this address as a new entry. Instead, the modified ring buffer will increment the counter of the last entry and keep the head pointer in the same position. In this way, we can predict more target addresses for return instructions with the same size of circular queue. The only cost is to add a counter for each entry. The simulation results of different architectures are shown in Fig. 11.

6. Simulation and Discussion

6.1 Simulation Environment & Process

The simulation process consists of two parts. First, we compile SPECint95 benchmarks into ELF binary code using *gcc* on a Linux system. Then *gdb*, a GNU debugging tool, extracts all the instruction addresses and instruction types during benchmarks' execution. *gdb* is set to single-step tracing mode and will automatically report both addresses and instruction types. The standard output of *gdb* will be piped to *gzip* for compressing.

Finally, we use our RABTB simulator to simulate these benchmark traces and obtain statistics (x86 instruction) including: average instruction length, number of instruction per cache line, dynamic and static branches, average occurrence of each static branch, total prediction accuracy, replacing frequency of the RABTB entry, the RABTB hit rate, etc. Figure 4 shows the complete simulation process.

The simulated micro-architecture is a superscalar, x86 instruction compatible processor. It is similar to the NSC98 microprocessor [20]. The NSC98 microprocessor converts x86 instruction to a sequence of NSC98 primitive operations (or POP). The average conversion rate is approximate 1.6. The POPs form an NSC98 RISC instruction set. The POPs are issued to an 8-way superscalar RISC core. The RISC core supports eight POP issue in a cycle. A reorder associative buffer (RAB) is used to facilitate speculative execution, out-of-order execution, out-of-order completion, and in-order commitment. The I-cache hit rate is assumed to be 100 percent to eliminate the effect of I-cache capacity. We assume only one Fetch Unit in a machine that can only fetch one cache line (32 bytes) per cycle.

6.2 Benchmark Programs and Analysis

The SPECint95 benchmark suite was used for experiments, which includes *compress, gcc, go, ijpeg, li, perl, m88 km* and *votex.* The detailed descriptions are showed below:

129.compress4.0:	A widely used compress program under UNIX system.
126.gcc 2.7.2:	A GNU C compiler (because we can't compile it under LINUX, so we replace it with original gcc 2.7.2).
099.go:	A self-played chess game.
132.ijpeg:	Graphic compression and decompression tools.
130.li:	A LISP interpreter.
134.perl:	Manipulates strings and prime number in perl.
124.m88 ksim:	A simulator for the 88100 microprocessor.
147.vortex:	Single-user object oriented database

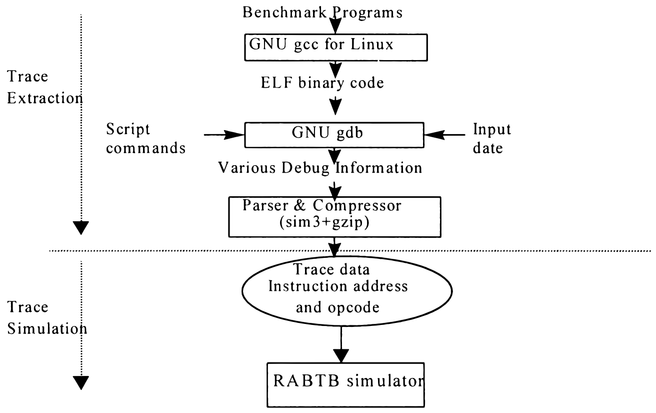


Fig. 4 The flow of simulation methodology.

Table 1 Benchmark statistics.

Benchmark	Input data	Total Ins.	Avg. Ins./Line	Dynamic	Static
compress 4.0 (SPEC95)	train.in	45,925,032	9.090502	8,200,791	624
real_gcc 2.7.2	2dbxout.i	72,203,567	7.363901	15,485,977	22233
go (SPEC 95)	size:10x9	140,507,979	8.485076	28,719,286	9070
ijpeg (SPEC 95)	specmun.ppm	339,886,031	9.618074	42,933,687	1340
li(SPEC95)	test.lsp	57,641,509	9.699835	14,389,274	1627
m88ksim (SPEC 95)	ctl.raw	7,000,435	6.860777	1,363,577	1958
perl(SPEC95)	primes.pl,primes.in	11,680,432	9.243107	2,220,596	3094
vortex (SPEC 95)	vortex.in	128,857,556	12.201655	19,776,338	8360

Table 2 Branch distribution in benchmark.

Benchmark	Branch per Line	Max.	Conditional%	Unconditional%	Return%
compress 4.0 (SPEC95)	1.8074	5	47.4554	41.2739	11.2564
real_gcc 2.7.2	2.3733	8	76.5365	12.8822	7.2295
go (SPEC 95)	2.0964	6	61.8485	32.3202	5.3425
ijpeg (SPEC 95)	1.9619	7	50.1860	42.7598	6.9896
li(SPEC95)	2.2161	7	46.7583	42.1258	9.5073
m88ksim (SPEC 95)	1.9991	7	55.9995	31.8254	9.9794
perl(SPEC95)	1.9219	6	57.9490	27.7514	7.8468
vortex (SPEC 95)	2.0223	6	57.6142	30.5615	10.6655
Average	2.0500	6.5	56.7934	32.6875	8.6021

transaction benchmark.

Table 1 shows input data set, total instruction count, number of instruction per cache line and dynamic and static branches. Here "static" means the number of branch (x86 instructions) in the source program and the "dynamic" means that the number of the branch instructions actually executed.

Table 2 shows the branch distributions of the Benchmark. The number of branches in each cache line mostly ranges from 2 to 3, with a maximum of 8 in *real_gcc*. This implies that less than 2 entries per row would not be suitable for allocating the branches in a cache line. However, 4 to 8 entries would be enough most of the time. We do not waste entries, because of the tag in each entry. With more than 8 entries per row, the aliasing problem would become serious and results in smaller number of rows. An RABTB with more columns may provide more dynamic resource allocation capability suitable for unbalanced branch distribution.

Because the predicted taken entry with the smallest offset in a row would be selected, the sorting must be done each cycle after updating the BTB. That means the complexity of the hardware logic would increase in ratio to the size of the row, and it will surely add much implementation cost if we do not limit the entries per row.

6.3 Performance Metric

Performance of the RABTB depends on two factors: the branch penalty and IPC_f. IPC_f is now used extensively to rate an instruction fetch mechanism, which will be much improved if the branch prediction method yields satisfying results. We count the waste cycles by the following cases:

1. Incorrect branch prediction
2. RABTB misses on taken branches

Since we do not simulate the rest of the machine, the exact mispredicted branch penalty is approximated. Because our design focuses on the x86 compatible superscalar architecture. The number of cycles between the fetch stage and execution stage is assumed to be 4 in the simulated micro-architecture. Another cycles are required to flush pipeline and fetch the correct instruction line when branch prediction is wrong. So, a 6-cycle penalty for case 1 is assumed. The penalty for case 2 is less than the penalty for case 1. In case 2 that a branch instruction is discovered after the instructions are decoded, so the penalty for case 2 is assumed to be 4 cycle.

The branch penalty per branch can be derived from the RABTB utilization and the branch prediction accuracy [10]. The branch penalty is:

$$\begin{aligned}\text{Branch penalty} = {} & \text{Percent RABTB hit rate}\\ & * \text{Percent incorrect prediction}\\ & * \text{misprediction penalty cycles}\\ & + (\text{Percent RABTB miss rate}\\ & * \text{Taken branches}\\ & * \text{miss penalty cycles})\end{aligned}$$

In order to keep some entries free all the time, we do not save data for branches that miss in the RABTB and are not taken. The simulation will only count the

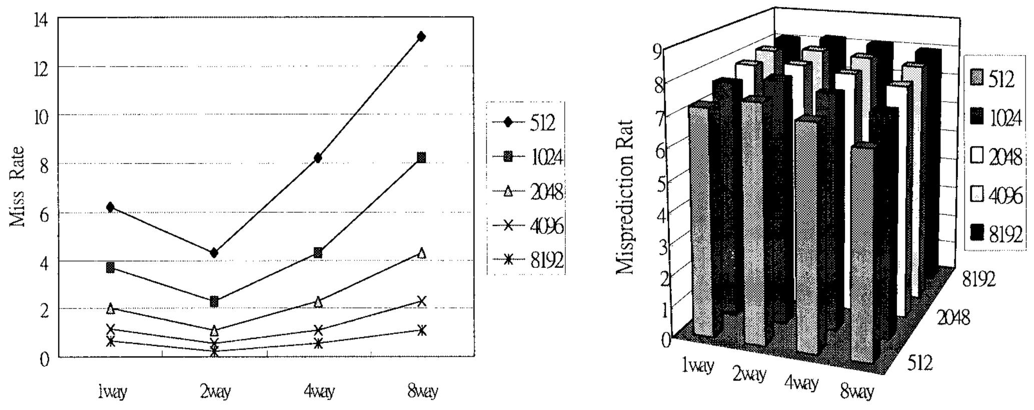


Fig. 5 The Miss Rate and Misprediction Rate in the scalar machine.

miss rate of branches that are taken. Therefore, in the above formula taken branches will be 100%. Because the misprediction rate in our simulations is the percentage of total dynamic branch instructions for which branch paths are predicted incorrectly, the formula can be simplified to:

$$\begin{aligned}&\text{Branch penalty}\\&\quad = \text{Misprediction rate} * 6\\&\qquad + (\text{Percent RABTB miss rate} * 4)\end{aligned}$$

Though our experiments focus on multiple branch instructions in a cache line, and pick the target address of the first taken-branch. The misprediction rate still includes all branches' prediction accuracy, not just the cache line misprediction rate.

6.4 Experimental Results and Analysis

In this section, the results of the trace-driven simulation are presented and plotted. In order to measure the efficiency of our scheme, we first simulate a single prediction scalar machine. Then we analyze those simulations that only predict the first branch instruction in a cache line for a wide-issue micro-architecture. Finally, several configurations of the RABTB were simulated and compared. Before setting out the results, a few of their features must be explained.

1. The RABTB entries mean the total used entries. Every entry includes data for tag, offset, branch type, LRU bit for replacement, last prediction result, branch history, and target address. The replacement algorithm is Least Recently Used (LRU). The RABTB is used for all type of branch instructions.
2. The branch history is assumed to be 6 bits in most cases. The branch history in the RABTB entry is initialized to all 1's (0x3f).
3. We use the PAs [4], [5] — a Per-address History Two-Level Adaptive branch predictor as our basic prediction algorithm. For per-address history scheme, the most cost-effective one is PAs(6,16) with a lower hardware cost [5]. The PHT size will become 1 k entrries ($2^6 * 16$). The PHT is divided into 16 sets and there are 64 (2^6) two-bit up-down saturating counters in each set.
4. An eight-entry modified ring buffer is used for predicting the return instructions.

6.4.1 Performance of a Single Prediction Scalar Machine

To measure the performance of single prediction in a scalar machine, we ran simulations by using PAs(6,16) [5] and various BTB sizes — from 512 to 8192 entries. To show the effect of the multiple way set-associative, the simulations considered 1-, 2-, 4- and 8-way for each size.

Figure 5 shows the Miss Rate and Misprediction Rate in the scalar machine for each configuration. The Miss Rate decreases obviously while total entries increase from 512 to 2048. And the best one is two-way set-associative in all configurations. The aliasing problems in level one branch information dominate the performance of PAs scheme [4]. The two-way set-associative structure can eliminate effectively the aliasing problem in most of benchmarks and obtain better Miss Rate. With more ways, the benefits will be reduced due to less number of rows can be indexed. The Misprediction Rates of all results are less than 8%, the difference between various ways is little while the total entries are more than 2048. And we call these schemes "scalar schemes."

We can decide the best configuration for a scalar system from the results shown in Fig. 6. Because the BTB must be on the CPU chip, tradeoffs between performance and cost are important. Therefore the best selection for scalar machine would be 2048 entry, 2-way set-associative BTB.

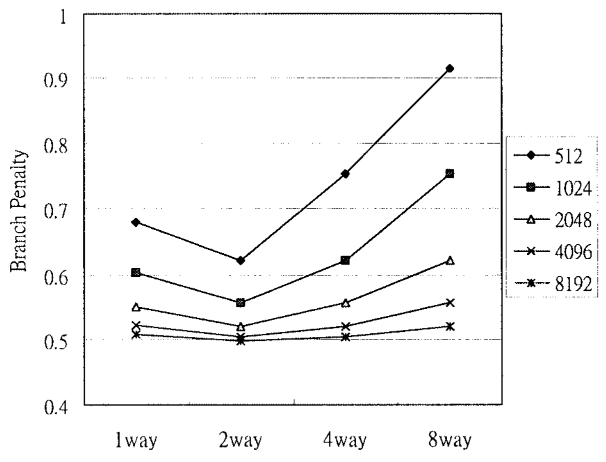


Fig. 6 The total Branch Penalty in the scalar machine.

6.4.2 The Degradation of Only the First Branch Predicted for a Wide-Issue Superscalar

Figure 7 shows the Miss Rate and Misprediction Rate when only the first branch is predicted using the BTB in the superscalar machine. Because only the first branch is predicted, other branches in the same cache line will be predicted as not taken naturally. But the probability of finding a taken branch after the first branch is above 17%. In Fig. 7, the Misprediction Rates of most results are larger than 15% and the degradation of prediction accuracy is 8%–10% compared with the "scalar schemes." Two-way set-associative is still the best in all configurations. We call these schemes the "first branch prediction schemes."

The branch penalties are all above 1 clock cycle as shown in Fig. 8. Therefore the IPC will be too small to support high issue rate. There is more than 80% degradation of the branch penalty compared with the results of the "scalar schemes." This is a bad solution for a wide-issue superscalar. We must modify the scheme to meet the requirement of high performance.

6.4.3 Performance of RABTB

In this subsection, we investigate the performance of the RABTB and determine the best architecture for the RABTB. To evaluate the RABTB performance, we ran simulations using PAs(6,16) [5] on the various RABTB sizes — from 512 to 8192 entries. Because a 32-byte line is fetched and the average cache line has 2 to 3 branches (Table 2), the number of columns must be greater than 2 and the simulations consider 2-, 4-, 8-, 16-, 32- and 64-column for each size.

Figure 9 shows the Miss Rate and Misprediction Rate of the RABTB in a superscalar machine for each configuration. The Miss Rate clearly decreases as the total entries increase from 512 to 2048. When the RABTB size is 512 entries, the 4-column scheme is the best, but the 8-column scheme outperforms the others from 1024 to 4096 entries. The miss rate is very low for 4-column to 32-column schemes with 8192 entries, because these schemes have enough entries to eliminate most of the aliasing cases. The Misprediction Rate is mostly less than 8%. The number of columns has little effect when there are more than 4096 entries. There are many improvements compared with the "first branch prediction scheme."

Figure 10 shows the branch penalty per branch of the RABTB in a superscalar machine. The branch penalty clearly decreases as the total entries increase from 512 to 2048. When the RABTB has 512 entries, the 4-column scheme has the smallest penalty. When the RABTB size is above 2048, there is little difference between 4-, 8- and 16-column schemes. The branch penalty of these schemes is under 0.56 clock cycle and similar to the "scalar scheme." When the RABTB size is larger than 2048, the 8-column scheme outperforms the others.

From our analysis and simulation results presented above, we conclude for the RABTB design:

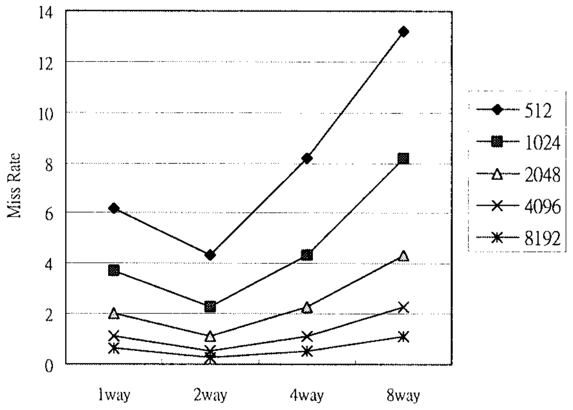
"When the RABTB size is under 2048, the 4-column scheme will be the best selection, but for larger RABTBs the 8-column scheme could be the optimal design."

6.4.4 The Comparisons of Different Architectures for Return Target Prediction

In the SPECint95 suite, for some input patterns, return instructions represent nearly 8.6% of the total branch instructions (refer to Table 2). The miss-predicted probability of its target address approximates 50% using the conventional BTB without return stack. Here, the simulations were performed with the context switch taken into consideration. The simulator will context-switch between different benchmarks. Whenever there are 10,000 instructions (default setting) executed or simulated, the simulator will flush the return stack or ring buffer and continue to simulate the next benchmark. The simulation results of different architectures with various size are shown in the left side of Fig. 11. The Misprediction Rate is mostly less than 18%. The improvement of prediction accuracy is about 64% better than the conventional BTB scheme without the return stack. As we discussed in Sect. 5.3, the ring buffer structure would have higher prediction accuracy than the return stack. The difference between various sizes is little when the size is larger than 10 in the ring buffer and the modified ring buffer. Under the consideration of reasonable hardware cost, the optimal configuration will be the 8-entry "Modified Ring Buffer." Also, it has the better performance than the 30-entry "return stack." The right side of Fig. 11 shows the miss rate

Table 3 Comparisons of branch penalty for three scheme.

Scheme / Size	2-way BTB (scalar)	2-way BTB (superscalar)	4-column RABTB	Degradation	8-column RABTB	degradation
512	0.62311	1.17098	0.74368	19.35%	0.76524	28.27%
1024	0.55769	1.12696	0.63374	13.64%	0.63806	9.02%
2048	0.52214	1.10626	0.55272	5.86%	0.53965	3.35%
4096	0.50553	1.0969	0.52302	3.46%	0.50740	0.37%
8192	0.49834	1.09344	0.50399	1.13%	0.4953	-0.61%



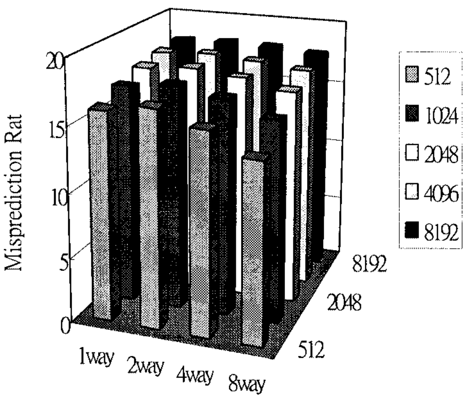


Fig. 7 Miss Rate and Misprediction Rate of first branch predicted.

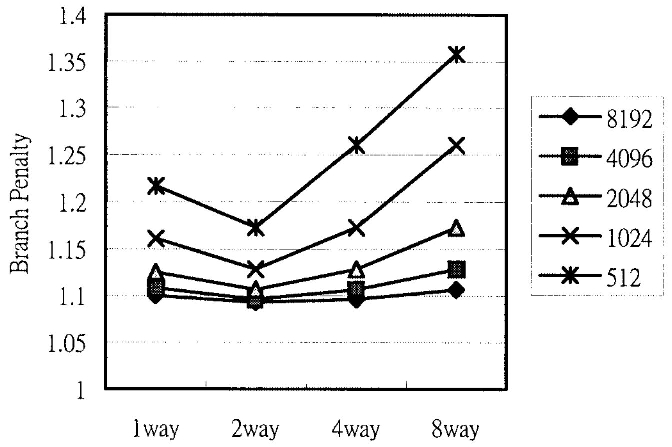


Fig. 8 Branch Penalty: only the first branch predicted.

of the benchmark with 8-entry scheme. Obviously, the modified ring buffer is the best or at least the same as the ring buffer structure. Up to half of the miss prediction rates have been diminished on some benchmarks (e.g. li, vortex) with the ring buffer structure.

6.4.5 Comparison of System Performance

We compared three schemes ("scalar scheme," "first branch prediction scheme" and the RABTB) using the best system performance (branch penalty) for each configuration. These comparisons are set out in Table 3. The degradation column compares the previous column and the 2-way BTB (scalar scheme) column. The branch penalty of an 8-column RABTB with 2048 entries is about 4.31% higher than the "scalar scheme" of the same size. When the size is 8192, the degradation of an 8-column RABTB becomes -0.61%: the negative value means that 8-column RABTB outperforms the "scalar scheme."

The trend in modern processors is toward larger prediction units. Therefore, with reasonable hardware cost, the best configuration of our RABTB will be 2048 entries, 8-column range-associate, and 8-entry modified ring buffer architecture using PAs prediction algorithm.

6.4.6 IPC_f Number Evaluation

In this subsection, we will discuss the relation between branch prediction and instruction fetching by the IPC_f. IPC_f is the average number of instructions fetched and issued per cycle assuming ideal performance for all other parts of the processor and no data dependency between instructions. The following formula will be used:

$$\begin{aligned}\text{IPC\_f} &= \text{Total Instruction Counts}\\ &\quad /(\text{Cache Line Fetch Cycles}\\ &\quad + \text{Branch Stall Cycles}),\end{aligned}$$

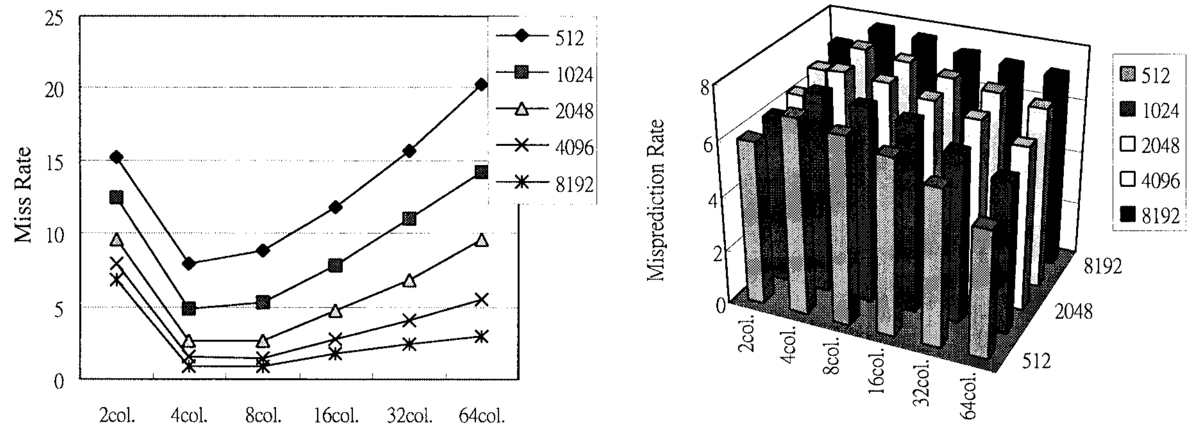


Fig. 9 Miss Rate and Misprediction Rate of the RABTB.

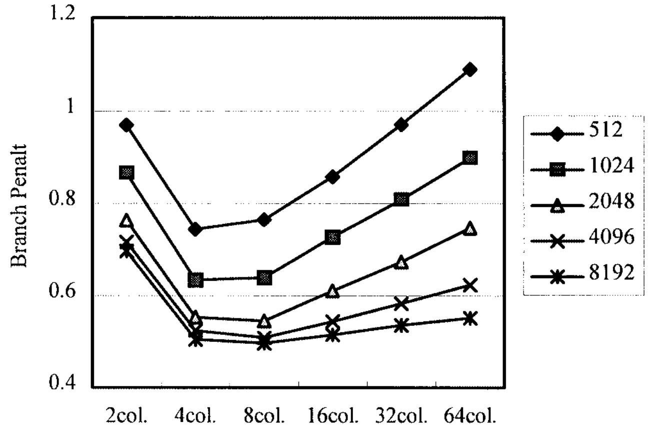
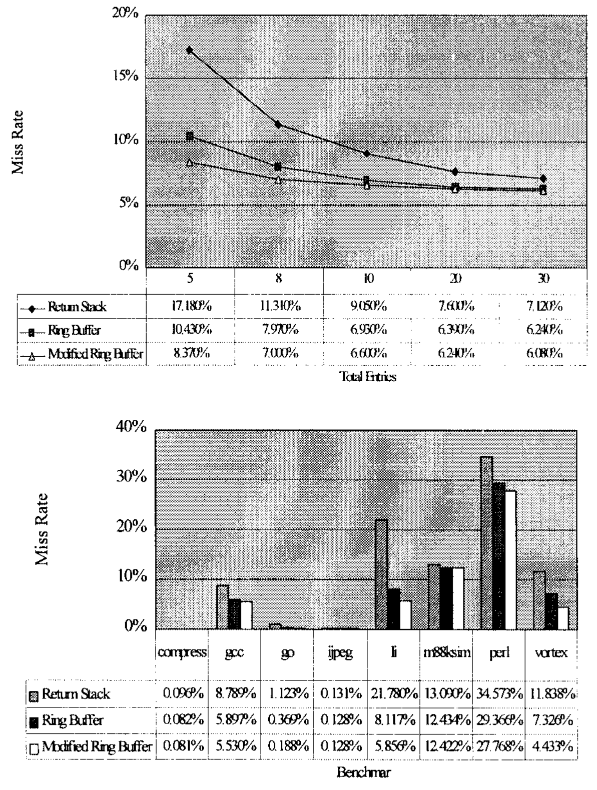


Fig. 10 Branch Penalty of the RABTB.



	5	8	10	20	30
Return Stack	17.180%	11.310%	9.050%	7.600%	7.120%
Ring Buffer	10.430%	7.970%	6.930%	6.390%	6.240%
Modified Ring Buffer	8.370%	7.000%	6.600%	6.240%	6.080%

	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex
▨ Return Stack	0.096%	8.789%	1.123%	0.131%	21.780%	13.090%	34.573%	11.838%
■ Ring Buffer	0.082%	5.897%	0.369%	0.128%	8.117%	12.434%	29.366%	7.326%
□ Modified Ring Buffer	0.081%	5.530%	0.188%	0.128%	5.856%	12.422%	27.768%	4.433%

Fig. 11 The comparisons of three different architectures and the miss rate of the 8-entry scheme.

$$\begin{aligned}&\text{Branch Stall Cycles}\\ &\quad = \text{Total Branch counts}\\ &\qquad * \text{Branch penalty per branch}\end{aligned}$$

The branch stalls include misprediction stalls and the stalls of misses on taken branches. As described in Sect. 6.3, their penalties are assumed to be 6 and 4 clock cycles respectively. We will use this formula to calculate the IPC_f of the optimal configuration proposed in the last subsection.

From the results shown in Fig. 12, we see that the IPC_f (POPs) is above 4 for all the benchmarks except for *real_gcc* and *go*. The IPC_f of *real_gcc* and *go* are 4.2 and 3.27, respectively, which are acceptable for these two benchmarks even though their prediction accuracy is not good enough. The misprediction rates of the two benchmarks are both larger than 10%. The IPC_f of vortex reaches as high as 7.11. The optimal RABTB can produce an average of 5.15 for IPC_f (POPs) and 7.4 for misprediction rate.

6.4.7 The Effect of Branch Misprediction Penalty

To investigate the effect of branch misprediction penalty on IPC_f of the optimal RABTB, we will vary the misprediction penalty from 4 cycles to 10 cycles. Figure 13 shows the effect of branch misprediction penalty on IPC_f. The performance degradation when the misprediction penalty increased from 4 cycles to 10 cycles is less than 22%. The individual degradation is

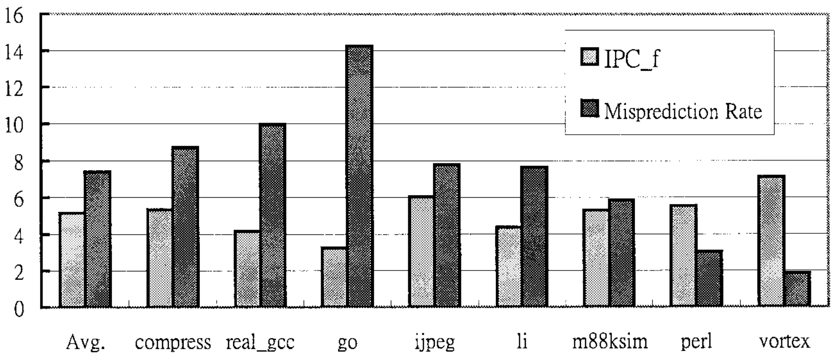


Fig. 12 The IPC_f of the optimal RABTB in a superscalar machine.

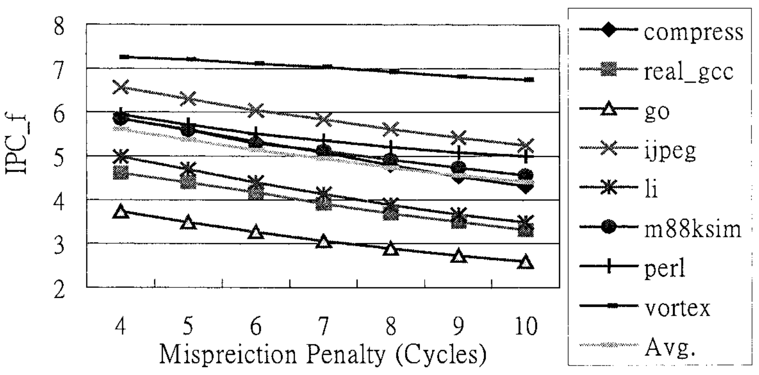


Fig. 13 The effect of Branch Misprediction Penalty on IPC_f.

about 7.1% to 30.6%. The *go* has the maximum degradation and the *vortex* has the minimum degradation.

7. Conclusion

From simulation results, we know that traditional branch prediction scheme does not generate good accuracy for multiple branch prediction. In this paper we have evaluated the influence of multiple branch prediction on a wide-issue superscalar microprocessor and proposed a new mechanism (RABTB) for multiple branch prediction.

Benchmark traces for SPEC95int used in our experiment were run on an x86 architecture. The distribution, behavior and types of branches are extracted from those benchmarks. The number of branches in each cache line mostly ranges from 2 to 3, with a maximum value of 8 in *real_gcc*. So 4 to 8 columns would be considered as our solution for the RABTB.

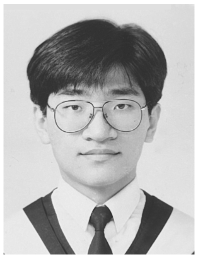
For the benchmarks we have simulated, the optimal RABTB has been verified as a 2048 entry, 8-column range-associate, and 8-entry modified ring buffer architecture using PAs prediction algorithm. It is an acceptable solution for multiple branch prediction with an average miss rate of 2.4% and a misprediction rate of 7.4%. It also achieves a system performance of 5.15 IPC_f and 0.54 cycles of branch penalty per branch, which is almost twice the performance of a mechanism that predicts only on the first encountered branch.

References

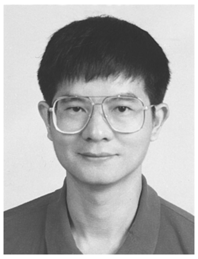
[1] G.S. Tyson, "The effect of predicated execution on branch prediction," 27th Int'l Symp. on Microarchitecture, pp.196–206, Nov. 1994.
[2] J.K.F. Lee and A.J. Smith, "Branch prediction strategies and branch target buffer design," 8th Int'l Symp. on Computer Architecture, pp.135–148, May 1981.
[3] C.H. Perleberg and A.J. Smith, "Branch target buffer design and optimization," IEEE Trans. Comput., vol.42, no.4, pp.396–411, April 1993.
[4] T.-Y. Yeh and Y.N. Patt, "Two-level adaptive training branch prediction," 24th Int'l Symp. on Microarchitecture, 1991.
[5] T.-T. Yeh and Y.N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," 20th Int'l Symp. on Computer Architecture, pp.257–266, May 1993,
[6] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt, "Branch classification: A new mechanism for improving branch predictor performance," 27th Int'l Symp. on Microarchitecture, pp.22–31, Nov. 1994.
[7] E. Hao, P.-Y. Chang, and Y. Patt, "The effect of speculatively updating branch history on branch prediction accuracy, revisited," 27th Int'l Symp. on Microarchitecture, pp.228–232, Nov. 1994.
[8] T.-Y. Yeh and Y.N. Patt, "Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors," 26th Int'l Symp. on Microarchitecture, pp.164–175, 1993.

[9] T.-Y. Yeh, D.T. Marr, and Y.N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," Proc. 7th ACM Int'l Conference on Supercomputing, pp.67–76, July 1993.
[10] J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publisher, San Francisco, Calif., 1996.
[11] D.R. Kaeli and P.G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," ACM, 1991.
[12] C.F. Webb, "Subroutine call/return stack," IBM Tech. Discl. Bull., vol.30, no.11, April 1988.
[13] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," 22nd Int'l Symp. on Computer Architecture, pp.333–344, 1995.
[14] S. Dutta and M. Franklin, "Control flow prediction with tree-like subgraphs for superscalar processors," 28th Int'l Symp. on Microarchitecture, pp.258–263, 1995.
[15] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictions," Int'l Conf. on Architectural Support for Programming Languages and Operating Systems VII, 1996.
[16] S. Wallace and N. Bagherzadeh, "Instruction Fetching Mechanisms for Superscalar Microprocessors," Euro-Par'96, 1996.
[17] Q. Jacobsen, E. Rotenberg, and J.E. Smith, "Path-based next trace prediction," 30th Int'l Symp. on Microarchitecture, pp.258–263, 1997.
[18] S.J. Patel, D.H. Friendly, and Y.N. Patt, "Critical issue regarding the trace cache fetch mechanism," Technical Report CEE-TR-335-97, Dept. of Electronics Engineering and Computer Science, University of Michigan, 1997.
[19] B. Cyril and M. Franklin, "A study of tree-based control flow predictions schemes," Int'l Conf. on High Performance Computing, 1997.
[20] C.L. Wu, "The definition of NSC98 microarchitecture," Document of Tentative NSC98 Microprocessor Design and Manufacture. WWW site http://www.act.nctu.edu.tw, Sept. 1996.

Shu-Lin Hwang received the B.S.E.E. degree from National Taiwan Institute of Technology, Taiwan, in 1988, and M.S. degree from the Electrical Engineering Department of National Taiwan University, Taiwan, in 1993. He is an instructor of the Department of Electrical Engineering at Mingchi Institute of Technology. He was also a Ph.D. candidate in the Electrical Engineering Department of National Taiwan University. He is a member of the Institute of Information & Computing Machinery. He is also the student member of IEEE and ACM. His current research interests are microprocessor architecture design and branch prediction design.



Che-Chun Chen received his B.S. degree from National Chiao Tung University, Taiwan, in 1995 and the M.S. degree from National Taiwan University, Taiwan in 1997. He is a member of the Institute of Information & Computing Machinery. His current research interests are branch prediction design and simulator design.



Feipei Lai received a B.S.E.E. degree from National Taiwan University in 1980, and M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1984 and 1987, respectively. He is a professor in the Department of Electrical Engineering and in the Department of Computer Science and Information Engineering at National Taiwan University. He was a visiting professor in the Department of Computer Science and Engineering at the University of Minnesota, Minneapolis, U.S.A. He was also a guest Professor at University of Dortmund, German and a visiting senior computer system engineer in the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. Dr. Lai holds four Taiwan patents and two USA patents currently. He served as a consultant at ERSO, ITRI during 1988 and at Faraday Technology Corp. from 8/94 to 7/95. His current research interests are high performance microprocessor chip design, computer architecture, optimizing compiler, VLSI design. Prof. Lai is one of the founders of the Institute of Information & Computing Machinery. He is also a member of Phi Kappa Phi, Phi Tau Phi, ACM, and The Chinese Institute of Engineers. He received Acer awards five times in 1989, 1991, 1992, 1993 and 1995 and The Taiwan Fuji Xerox Research award in 1991. Dr. Lai is a Senior member of IEEE and included in "Who's Who in Science and Engineering" and "Who's Who in the World."