The 19th Annual International Symposium on Computer Architecture
pp. 124 - 134, May 19 - 21, 1992, Gold Coast, Australia.

Alternative Implementations of Two-Level Adaptive Branch Prediction

Tse–Yu Yeh and Yale N. Patt
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

Abstract

As the issue rate and depth of pipelining of high performance Superscalar processors increase, the importance of an excellent branch predictor becomes more vital to delivering the potential performance of a wide-issue, deep pipelined microarchitecture. We propose a new dynamic branch predictor (Two-Level Adaptive Branch Prediction) that achieves substantially higher accuracy than any other scheme reported in the literature. The mechanism uses two levels of branch history information to make predictions, the history of the last k branches encountered, and the branch behavior for the last s occurrences of the specific pattern of these k branches. We have identified three variations of the Two-Level Adaptive Branch Prediction, depending on how finely we resolve the history information gathered. We compute the hardware costs of implementing each of the three variations, and use these costs in evaluating their relative effectiveness. We measure the branch prediction accuracy of the three variations of Two-Level Adaptive Branch Prediction, along with several other popular proposed dynamic and static prediction schemes, on the SPEC benchmarks. We show that the average prediction accuracy for Two-Level Adaptive Branch Prediction is 97 percent, while the other known schemes achieve at most 94.4 percent average prediction accuracy. We measure the effectiveness of different prediction algorithms and different amounts of history and pattern information. We measure the costs of each variation to obtain the same prediction accuracy.

1 Introduction

As the issue rate and depth of pipelining of high performance Superscalar processors increase, the amount of speculative work due to branch prediction becomes much larger. Since all such work must be thrown away if the prediction is incorrect, an excellent branch predictor is vital to delivering the potential performance of a wide-issue, deep pipelined microarchitecture. Even a prediction miss rate of 5 percent results in a substantial loss in performance due to the number of instructions fetched each cycle and the number of cycles these instructions are in the pipeline before an incorrect branch prediction becomes known.

The literature is full of suggested branch prediction schemes [6, 13, 14, 17]. Some are static in that they use opcode information and profiling statistics to make predictions. Others are dynamic in that they use run-time execution history to make predictions. Static schemes can be as simple as always predicting that the branch will be taken, or can be based on the opcode, or on the direction of the branch, as in "if the branch is backward, predict taken, if forward, predict not taken" [17]. This latter scheme is effective for loop intensive code, but does not work well for programs where the branch behavior is irregular. Also, profiling [6, 13] can be used to predict branches by measuring the tendency of a branch on sample data sets and presetting a static prediction bit in the opcode according to that tendency. Unfortunately, branch behavior for the sample data may be very different from the data that appears at run-time.

Dynamic branch prediction also can be as simple as in keeping track only of the last execution of that branch instruction and predicting the branch will behave the same way, or it can be elaborate as in maintaining very large amounts of history information. In all cases, the fact that the dynamic prediction is being made on the basis of run-time history information implies that substantial additional hardware is required. J. Smith [17] proposed utilizing a branch target buffer to store, for each branch, a two-bit saturating up-down counter which collects and subsequently bases its prediction on branch history information about that branch. Lee and A. Smith proposed [14] a Static Training method which uses statistics gathered prior to execution time coupled with the history pattern of the last k run-time executions of the branch to make the next prediction as to which way that branch will go. The major disadvantage of Static Training methods has been mentioned above with respect to profiling; the pattern history statistics gathered for the sample data set may not be applicable to the data that appears at run-time.

In this paper we propose a new dynamic branch predictor that achieves substantially higher accuracy than any other scheme reported in the literature. The mechanism uses two levels of branch history information to make predictions. The first level is the history of the last k branches encountered. (Variations of our scheme reflect whether this means the actual last k branches encountered, or the last k occurrences of the same branch instruction.) The second level is the branch behavior for the last s occurrences of the specific pattern of these k branches. Prediction is based on the branch behavior for the last s occurrences of the pattern in question.

For example, suppose, for $k = 8$, the last k branches had the behavior 11100101 (where 1 represents that the branch was taken, 0 that the branch was not taken). Suppose further that $s = 6$, and that in each of the last six times the previous eight branches had the pattern 11100101, the branch alternated between taken and not taken. Then the second level would contain the history 101010. Our branch predictor would predict "taken."

The history information for level 1 and the pattern information for level 2 are collected at run time, eliminating the above mentioned disadvantages of the Static Training method. We call our method Two-Level Adaptive Branch Prediction. We have identified three variations of Two-Level Adaptive Branch Prediction, depending on how finely we resolve the history information gathered. We compute the hardware costs of implementing each of the three variations, and use these costs in evaluating their relative effectiveness.

Using trace-driven simulation of nine of the ten SPEC benchmarks [1], we measure the branch prediction accuracy of the three variations of Two-Level Adaptive Branch Prediction, along with several other popular proposed dynamic and static prediction schemes. We measure the effectiveness of different prediction algorithms and different amounts of history and pattern information. We measure the costs of each variation to obtain the same prediction accuracy. Finally we compare the Two-Level Adaptive branch predictors to the several popular schemes available in the literature. We show that the average prediction accuracy for Two-Level Adaptive Branch Prediction is about 97 percent, while the other schemes achieve at most 94.4 percent average prediction accuracy.

This paper is organized in six sections. Section two introduces our Two-Level Adaptive Branch Prediction and its three variations. Section three describes the corresponding implementations and computes the associated hardware costs. Section four discusses the Simulation model and traces used in this study. Section five reports the simulation results and our analysis. Section six contains some concluding remarks.

2 Definition of Two-Level Adaptive Branch Prediction

2.1 Overview

Two-Level Adaptive Branch Prediction uses two levels of branch history information to make predictions. The first level is the history of the last k branches encountered. (Variations of our scheme reflect whether this means the actual last k branches encountered, or the last k occurrences of the same branch instruction.) The second level is the branch behavior for the last s occurrences of the specific pattern of these k branches. Prediction is based on the branch behavior for the last s occurrences of the pattern in question.

To maintain the two levels of information, Two-Level Adaptive Branch Prediction uses two major data structures, the branch history register (HR) and the pattern history table (PHT), see Figure 1. Instead of accumulating statistics by profiling programs, the information on which branch predictions are based is collected at run-time by updating the contents of the history registers and the pattern history bits in the entries of the pattern history table depending on the outcomes of the branches. The history register is a k-bit shift register which shifts in bits representing the branch results of the most recent k branches.

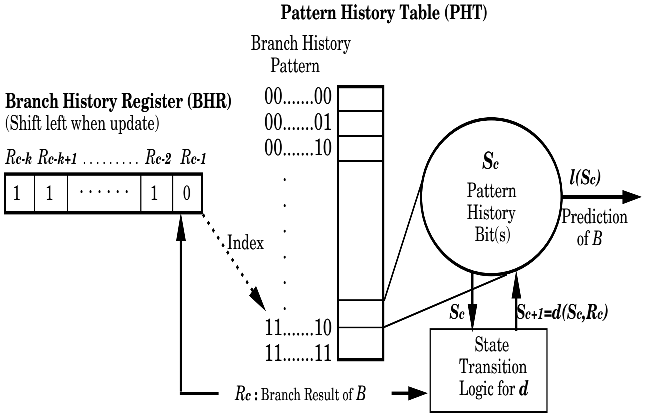


Figure 1: Structure of Two-Level Adaptive Branch Prediction.

If the branch was taken, then a "1" is recorded; if not, a "0" is recorded. Since there are k bits in the history register, at most 2^k different patterns appear in the history register. For each of these 2^k patterns, there is a corresponding entry in the pattern history table which contains branch results for the last s times the preceding k branches were represented by that specific content of the history register.

When a conditional branch B is being predicted, the content of its history register, HR, denoted as $R_{c-k}R_{c-k+1}......R_{c-1}$, is used to address the pattern history table. The pattern history bits S_c in the addressed entry $PHT_{R_{c-k}R_{c-k+1}......R_{c-1}}$ in the pattern history table are then used for predicting the branch. The prediction of the branch is

$$z_c = \lambda(S_c), \qquad (1)$$

where λ is the prediction decision function.

After the conditional branch is resolved, the outcome R_c is shifted left into the history register HR in the least significant bit position and is also used to update the pattern history bits in the pattern history table entry $PHT_{R_{c-k}R_{c-k+1}......R_{c-1}}$. After being

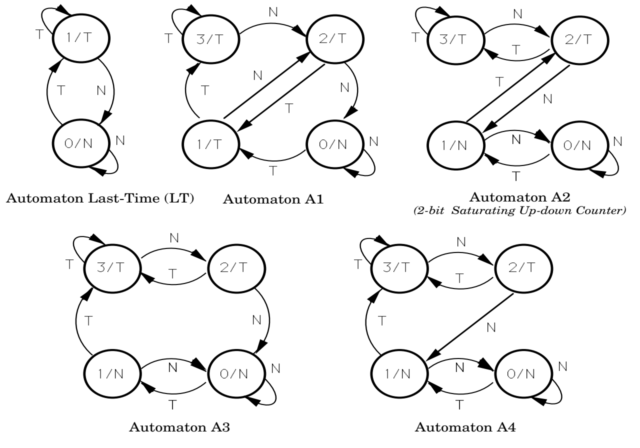
[1] The Nasa7 benchmark was not simulated because this benchmark consists of seven independent loops. It takes too long to simulate the branch behavior of these seven kernels, so we omitted these loops.

updated, the content of the history register becomes $R_{c-k+1}R_{c-k+2}......R_c$ and the state represented by the pattern history bits becomes S_{c+1}. The transition of the pattern history bits in the pattern history table entry is done by the state transition function δ which takes in the old pattern history bits and the outcome of the branch as inputs to generate the new pattern history bits. Therefore, the new pattern history bits S_{c+1} become

$$S_{c+1} = \delta(S_c, R_c). \qquad (2)$$

A straightforward combinational logic circuit is used to implement the function δ to update the pattern history bits in the entries of the pattern history table. The transition function δ, predicting function λ, pattern history bits S and the outcome R of the branch comprise a finite-state Moore machine, characterized by equations 1 and 2.

State diagrams of the finite-state Moore machines used in this study for updating the pattern history in the pattern history table entry and for predicting which path the branch will take are shown in Figure 2. The automaton *Last-Time* stores in the pattern history only the outcome of the last execution of the branch when the history pattern appeared. The next time the same history pattern appears the prediction will be what happened last time. Only one bit is needed to store that pattern history information. The automaton $A1$ records the results of the last two times the same history pattern appeared. Only when there is no taken branch recorded, the next execution of the branch when the history register has the same history pattern will be predicted as not taken; otherwise, the branch will be predicted as taken. The automaton $A2$ is a saturating up-down counter, similar to the automaton used in J. Smith's branch target buffer design for keeping branch history [17].



Automaton Last-Time (LT)

Automaton A1

Automaton A2
(2-bit Saturating Up-down Counter)

Automaton A3

Automaton A4

Figure 2: State diagrams of the finite-state Moore machines used for making prediction and updating the pattern history table entry.

In J. Smith's design the 2-bit saturating up-down counter keeps track of the branch history of a certain branch. The counter is incremented when the branch is taken and is decremented when the branch is not taken. The branch path of the next execution of the branch will be predicted as taken when the counter value is greater than or equal to two; otherwise, the branch will be predicted as not taken. In Two-Level Adaptive Branch Prediction, the 2-bit saturating up-down counter keeps track of the history of a certain history pattern. The counter is incremented when the result of a branch, whose history register content is the same as the pattern history table entry index, is taken; otherwise, the counter is decremented. The next time the branch has the same history register content which accesses the same pattern history table entry, the branch is predicted taken if the counter value is greater or equal to two; otherwise, the branch is predicted not taken. Automata $A3$ and $A4$ are variations of $A2$.

Both Static Training [14] and Two-Level Adaptive Branch Prediction are dynamic branch predictors, because their predictions are based on run-time information, i.e. the dynamic branch history. The major difference between these two schemes is that the pattern history information in the pattern history table changes dynamically in Two-Level Adaptive Branch Prediction but is preset in Static Training from profiling. In Static Training, the input to the prediction decision function, λ, for a given branch history pattern is known before execution. Therefore, the output of λ is determined before execution for a given branch history pattern. That is, the same branch predictions are made if the same history pattern appears at different times during execution. Two-Level Adaptive Branch Prediction, on the other hand, updates the pattern history information kept in the pattern history table with the actual results of branches. As a result, given the same branch history pattern, different pattern history information can be found in the pattern history table; therefore, there can be different inputs to the prediction decision function for Two-Level Adaptive Branch Prediction. Predictions of Two-Level Adaptive Branch Prediction change adaptively as the program executes.

Since the pattern history bits change in Two-Level Adaptive Branch Prediction, the predictor can adjust to the current branch execution behavior of the program to make proper predictions. With these run-time updates, Two-Level Adaptive Branch Prediction can be highly accurate over many different programs and data sets. Static Training, on the contrary, may not predict well if changing data sets brings about different execution behavior.

2.2 Alternative Implementations of Two-Level Adaptive Branch Prediction

There are three alternative implementations of the Two-Level Adaptive Branch Prediction, as shown in Figure 3. They are differentiated as follows:

Two-Level Adaptive Branch Prediction Using a Global History Register and a Global Pattern History Table (GAg)

In GAg, there is only a single global history register (GHR) and a single global pattern history table (GPHT) used by the Two-Level Adaptive Branch Pre-

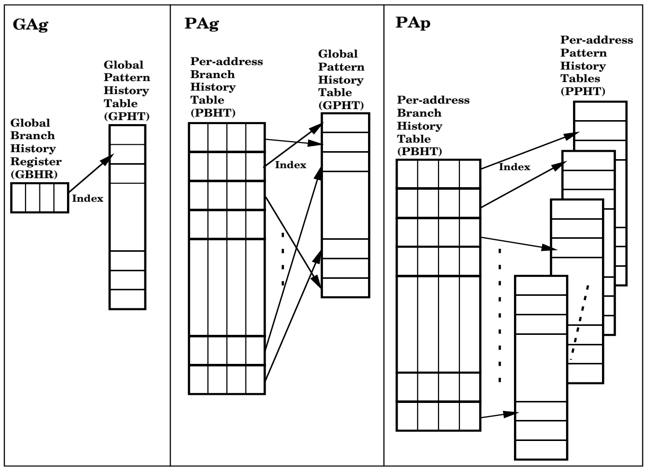

GAg
Global Pattern History Table (GPHT)
Global Branch History Register (GBHR)
Index
PAg
Per-address Branch History Table (PBHT)
Global Pattern History Table (GPHT)
Index
PAp
Per-address Pattern History Tables (PPHT)
Per-address Branch History Table (PBHT)
Index

Figure 3: Global view of three variations of Two-Level Adaptive Branch Prediction.

diction. All branch predictions are based on the same global history register and global pattern history table which are updated after each branch is resolved. This variation therefore is called Global Two-Level Adaptive Branch Prediction using a global pattern history table (GAg).

Since the outcomes of different branches update the same history register and the same pattern history table, the information of both branch history and pattern history is influenced by results of different branches. The prediction for a conditional branch in this scheme is actually dependent on the outcomes of other branches.

Two-Level Adaptive Branch Prediction Using a Per-address Branch History Table and a Global Pattern History Table (PAg)
In order the reduce the interference in the first level branch history information, one history register is associated with each distinct static conditional branch to collect branch history information individually. The history registers are contained in a per-address branch history table (PBHT) in which each entry is accessible by one specific static branch instruction and is accessed by branch instruction addresses. Since the branch history is kept for each distinct static conditional branch individually and all history registers access the same global pattern history table, this variation is called Per-address Two-Level Adaptive Branch Prediction using a global pattern history table (PAg).

The execution results of a static conditional branch update the branch's own history register and the global pattern history table. The prediction for a conditional branch is based on the branch's own history and the pattern history bits in the global pattern history table entry indexed by the content of the branch's history register. Since all branches update the same pattern history table, the pattern history interference still exists.

Two-Level Adaptive Branch Prediction Using Per-address Branch History Table and Per-address Pattern History Tables (PAp)

In order to completely remove the interference in both levels, each static branch has its own pattern history table a set of which is called a per-address pattern history table (PPHT). Therefore, a per-address history register and a per-address pattern history table are associated with each static conditional branch. All history registers are grouped in a per-address branch history table. Since this variation of Two-Level Adaptive Branch Prediction keeps separate history and pattern information for each distinct static conditional branch, it is called Per-address Two-Level Adaptive Branch Prediction using Per-address pattern history tables (PAp).

3 Implementation Considerations

3.1 Pipeline Timing of Branch Prediction and Information Update

Two-Level Adaptive Branch Prediction requires two sequential table accesses to make a prediction. It is difficult to squeeze the two accesses into one cycle. High performance requires that prediction be made within one cycle from the time the branch address is known. To satisfy this requirement, the two sequential accesses are performed in two different cycles as follows: When a branch result becomes known, the branch's history register is updated. In the same cycle, the pattern history table can be accessed for the next prediction with the updated history register contents derived by appending the result to the old history. The prediction fetched from the pattern history table is then stored along with the branch's history in the branch history table. The pattern history can also be updated at that time. The next time that branch is encountered, the prediction is available as soon as the branch history table is accessed. Therefore, only one cycle latency is incurred from the time the branch address is known to the time the prediction is available.

Sometimes the previous branch results may not be ready before the prediction of a subsequent branch takes place. If the obsolete branch history is used for making the prediction, the accuracy is degraded. In such a case, the predictions of the previous branches can be used to update the branch history. Since the prediction accuracy of Two-Level Adaptive Branch Prediction is very high, prediction is enhanced by updating the branch history speculatively. The update timing for the pattern history table, on the other hand, is not as critical as that of the branch history; therefore, its update can be delayed until the branch result is known. With speculative updating, when a misprediction occurs, the branch history can either be reinitialized or repaired depending on the hardware budget available to the branch predictor. Also, if two instances of the same static branch occur in consecutive cycles, the latency of prediction can be reduced for the second branch by using the prediction fetched from the pattern history table directly.

3.2 Target Address Caching

After the direction of a branch is predicted, there is still the possibility of a pipeline bubble due to the time it takes to generate the target address. To eliminate this bubble, we cache the target addresses of branches. One extra field is required in each entry of the branch history table for doing this. When a branch is predicted taken, the target address is used to fetch the following instructions; otherwise, the fall-through address is used.

Caching the target addresses makes prediction in consecutive cycles possible without any delay. This also requires the branch history table to be accessed by the fetching address of the instruction block rather than by the address of the branch in the instruction block being fetched because the branch address is not known until the instruction block is decoded. If the address hits in the branch history table, the prediction of the branch in the instruction block can be made before the instructions are decoded. If the address misses in the branch history table, either there is no branch in the instruction block fetched in that cycle or the branch history information is not present in the branch history table. In this case, the next sequential address is used to fetch new instructions. After the instructions are decoded, if there is a branch in the instruction block and if the instruction block address missed in the branch history table, static branch prediction is used to determine whether or not the new instructions fetched from the next sequential address should be squashed.

3.3 Per-address Branch History Table Implementation

PAg and PAp branch predictors all use per-address branch history tables in their structure. It is not feasible to have a branch history table large enough to hold all branches' execution history in real implementations. Therefore, a practical approach for the per-address branch history table is proposed here.

The per-address branch history table can be implemented as a set-associative or direct-mapped cache. A fixed number of entries in the table are grouped together as a set. Within a set, a Least-Recently-Used (LRU) algorithm is used for replacement. The lower part of a branch address is used to index into the table and the higher part is stored as a tag in the entry associated with that branch. When a conditional branch is to be predicted, the branch's entry in the branch history table is located first. If the tag in the entry matches the accessing address, the branch information in the entry is used to predict the branch. If the tag does not match the address, a new entry is allocated for the branch.

In this study, both the above practical approach and an Ideal Branch History Table (IBHT), in which there is a history register for each static conditional branch, were simulated for Two-Level Adaptive Branch Prediction. The branch history table was simulated with four configurations: 4-way set-associative 512-entry, 4-way set-associative 256-entry, direct-mapped 512-entry and direct-mapped 256-entry caches. The IBHT simulation data is provided to show the accuracy loss due to the history interference in a practical branch history table implementations.

3.4 Hardware Cost Estimates

The chip area required for a run-time branch prediction mechanism is not inconsequential. The following hardware cost estimates are proposed to characterize the relative costs of the three variations. The branch history table and the pattern history table are the two major parts. Detailed items include storage space for keeping history information, prediction bits, tags, and LRU bits and the accessing and updating logic of the tables. The accessing and updating logic consists of comparators, MUXes, LRU bits incrementors, and address decoders for the branch history table, and address decoders and pattern history bit update circuits for the pattern history table. The storage space for caching target addresses is not included in the following equations because it is not required for the branch predictor.

Assumptions of these estimates are:

- There are a address bits, a subset of which is used to index the branch history table and the rest are stored as a tag in the indexed branch history table entry.
- In an entry of the branch history table, there are fields for branch history, an address tag, a prediction bit, and LRU bits.
- The branch history table size is h.
- The branch history table is 2^j-way set-associative.
- Each history register contains k bits.
- Each pattern history table entry contains s bits.
- Pattern history table set size is p. (In PAp, p is equal to the size of the branch history table, h, while in GAg and PAg, p is always equal to one.)
- C_s, C_d, C_c, C_m, C_{sh}, C_i, and C_a are the constant base costs for the storage, the decoder, the comparator, the multiplexer, the shifter, the incrementor, and the finite-state machine.

Furthermore, i is equal to $log_2 h$ and is a non-negative integer. When there are k bits in a history register, a pattern history table always has 2^k entries.

The hardware cost of Two-Level Adaptive Branch Prediction is as follows:

$$
\begin{aligned}
&Cost_{Scheme}(BHT(h,j,k), p \times PHT(2^k, s)) \\
&\quad = \quad Cost_{BHT}(h,j,k) + p \times Cost_{PHT}(2^k, s) \\
&\quad = \quad \{BHT_{Storage\_Space} + BHT_{Accessing\_Logic} + \\
&\qquad\quad BHT_{Updating\_Logic}\} + p \times \{PHT_{Storage\_Space} + \\
&\qquad\quad PHT_{Accessing\_Logic} + PHT_{Updating\_Logic}\} \\
&\quad = \quad \{[h \times (Tag_{(a-i+j)\_bit} + HR_{k\_bit} + Prediction\_Bit_{1\_bit} \\
&\qquad\quad + LRU\_Bits_{j\_bit})] + \\
&\qquad\quad [1 \times Address\_Decoder_{i\_bit} + 2^j \times \\
&\qquad\quad Comparators_{(a-i+j)\_bit} + 1 \times 2^j X1\_MUX_{k\_bit}] + \\
&\qquad\quad [h \times Shifter_{k\_bit} + 2^j \times LRU\_Incrementors_{j\_bit}]\} + \\
&\qquad\quad p \times \{[2^k \times History\_Bits_{s\_bit}] + \\
&\qquad\quad [1 \times Address\_Decoder_{k\_bit}] + [State\_Updater_{s\_bit}]\}
\end{aligned}
$$

$$\begin{aligned} = \quad & \{h \times [(a-i+j)+k+1+j] \times C_s + \\ & [h \times C_d + 2^j \times (a-i+j) \times C_c + 2^j \times k \times C_m] + \\ & [h \times k \times C_{sh} + 2^j \times j \times C_i]\} + p \times \{[2^k \times s \times C_s] + \\ & [2^k \times C_d] + [s \times 2^{s+1} \times C_a]\}, \quad a+j \geq i. \end{aligned} \tag{3}$$

In GAg, only one history register and one global pattern history table are used, so h and p are both equal to one. No tag and no branch history table accessing logic are necessary for the single history register. Besides, pattern history state updating logic is small compared to the other two terms in the pattern history table cost. Therefore, cost estimation function for GAg can be simplified from Function 3 to the following Function:

$$\begin{aligned} & Cost_{GAg}(BHT(1, \ , k), 1 \times PHT(2^k, s)) \\ = \quad & Cost_{BHT}(1, \ , k) + 1 \times Cost_{PHT}(2^k, s) \\ \simeq \quad & \{[k+1] \times C_s + k \times C_{sh}\} + \\ & \{2^k \times (s \times C_s + C_d)\} \end{aligned} \tag{4}$$

It is clear to see that the cost of GAg grows exponentially with respect to the history register length.

In PAg, only one pattern history table is used, so p is equal to one. Since j and s are usually small compared to the other variables, by using Function 3, the estimated cost for PAg using a branch history table is as follows:

$$\begin{aligned} & Cost_{PAg}(BHT(h, j, k), 1 \times PHT(2^k, s)) \\ = \quad & Cost_{BHT}(h, j, k) + 1 \times Cost_{PHT}(2^k, s) \\ \simeq \quad & \{h \times [(a + 2 \times j + k + 1 - i) \times C_s + C_d + \\ & k \times C_{sh}]\} + \\ & \{2^k \times (s \times C_s + C_d)\}, \quad a+j \geq i. \end{aligned} \tag{5}$$

The cost of a PAg scheme grows exponentially with respect to the history register length and linearly with respect to the branch history table size.

In a PAp scheme using a branch history table as defined above, h pattern history tables are used, so p is equal to h. By using Function 3, the estimated cost for PAp is as follows:

$$\begin{aligned} & Cost_{PAp}(BHT(h, j, k), h \times PHT(2^k, s)) \\ = \quad & Cost_{BHT}(h, j, k) + h \times Cost_{PHT}(2^k, s) \\ \simeq \quad & \{h \times [(a + 2 \times j + k + 1 - i) \times C_s + C_d + \\ & k \times C_{sh}]\} + \\ & h \times \{2^k \times (s \times C_s + C_d)\}, \quad a+j \geq i. \end{aligned} \tag{6}$$

When the history register is sufficiently large, the cost of a PAp scheme grows exponentially with respect to the history register length and linearly with respect to the branch history table size. However, the branch history table size becomes a more dominant factor than it is in a PAg scheme.

4 Simulation Model

Trace-driven simulations were used in this study. A Motorola 88100 instruction level simulator is used for generating instruction traces. The instruction and address traces are fed into the branch prediction simulator which decodes instructions, predicts branches, and verifies the predictions with the branch results to collect statistics for branch prediction accuracy.

4.1 Description of Traces

Nine benchmarks from the SPEC benchmark suite are used in this branch prediction study. Five are floating point benchmarks and four are integer benchmarks. The floating point benchmarks include *doduc*, *fpppp*, *matrix*300, *spice*2*g*6 and *tomcatv* and the integer ones include *eqntott*, *espresso*, *gcc*, and *li*. *Nasa*7 is not included because it takes too long to capture the branch behavior of all seven kernels.

Among the five floating point benchmarks, *fpppp*, *matrix*300 and *tomcatv* have repetitive loop execution; thus, a very high prediction accuracy is attainable, independent of the predictors used. *Doduc*, *spice*2*g*6 and the integer benchmarks are more interesting. They have many conditional branches and irregular branch behavior. Therefore, it is on the integer benchmarks where a branch predictor's mettle is tested.

Since this study of branch prediction focuses on the prediction for conditional branches, all benchmarks were simulated for twenty million conditional branch instructions except *gcc* which finished before twenty million conditional branch instructions are executed. *Fpppp*, *matrix*300, and *tomcatv* were simulated for 100 million instruction because of their regular branch behavior through out the programs. The number of static conditional branches in the instruction traces of the benchmarks are listed in Table 1. History register hit rate usually depends on the number of static branches in the benchmarks. The testing and training data sets for each benchmark used in this study are listed in Table 2.

Benchmark	Number of	Benchmark	Number of
	Static		Static
Name	Cnd. Br.	Name	Cnd. Br.
eqntott	277	espresso	556
gcc	6922	li	489
doduc	1149	fpppp	653
matrix300	213	spice2g6	606
tomcatv	370		

Table 1: Number of static conditional branches in each benchmark.

Benchmark	Training	Testing
Name	Data Set	Data Set
eqntott	NA	int_pri_3.eqn
espresso	cps	bca
gcc	cexp.i	dbxout.i
xlisp	tower of hanoi	eight queens
doduc	tiny doducin	doducin
fpppp	NA	natoms
matrix300	NA	Built-in
spice2g6	short greycode.in	greycode.in
tomcatv	NA	Built-in

Table 2: Training and testing data sets of benchmarks.

In the traces generated with the testing data sets, about 24 percent of the dynamic instructions for the integer benchmarks and about 5 percent of the dynamic instructions for the floating point benchmarks are branch instructions. Figure 4 shows about 80 percent of the dynamic branch instructions are conditional branches; therefore, the prediction mechanism for conditional branches is the most important among the prediction mechanisms for different classes of branches.

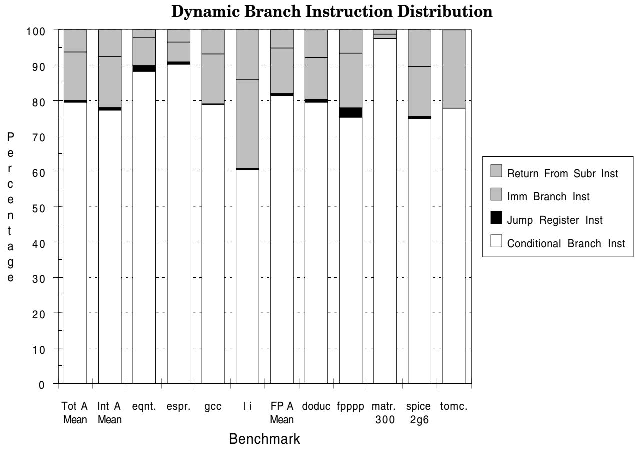


Figure 4: Distribution of dynamic branch instructions.

4.2 Characterization of Branch Predictors

The three variations of Two-Level Adaptive Branch Prediction were simulated with several configurations. Other known dynamic and static branch predictors were also simulated. The configurations of the dynamic branch predictors are shown in Table 3. In order to distinguish the different schemes we analyzed, the following naming convention is used: *Scheme*(*History*(*Size*, *Associativity*, *Entry_Content*), *Pattern_Table_Set_Size* × *Pattern*(*Size*, *Entry_Content*), *Context_Switch*). If a predictor does not have a certain feature in the naming convention, the corresponding field is left blank.

Scheme specifies the scheme, for example, GAg, PAg, PAp or Branch Target Buffer design (BTB) [17]. In *History*(*Size*, *Associativity*, *Entry_Content*), *History* is the entity used to keep history information of branches, for example, HR (A single history register), IBHT, or BHT. *Size* specifies the number of entries in that entity, *Associativity* is the associativity of the table, and *Entry_Content* specifies the content in each branch history table entry. When *Associativity* is set to 1, the branch history table is direct-mapped. The content of an entry in the branch history table can be any automaton shown in Figure 2 or simply a history register.

In *Pattern_Table_Set_Size* × *Pattern*(*Size*, *Entry_Content*), *Pattern_Table_Set_Size* is the number of pattern history tables used in the scheme, *Pattern* is the implementation for keeping pattern history information, *Size* specifies the number of entries in the implementation, and *Entry_Content* specifies the content in each entry. The content of an entry in the pattern history table can be any automaton shown in Figure 2. For Branch Target Buffer designs, the *Pattern* part is not included, because there is no pattern history information kept in their designs. *Context_Switch* is a flag for context switches. When *Context_Switch* is specified as c, context switches are simulated. If it is not specified, no context switches are simulated.

Since there are more taken branches than not taken branches according to our simulation results, a history register in the branch history table is initialized to all 1's when a miss on the branch history table occurs. After the result of the branch which causes the branch history table miss is known, the result bit is extended throughout the history register. A context switch results in flushing and reinitialization of the branch history table.

Model	BHT Config.			PHT	PHT Config.	
Name	# of Entr.	Asc	Entry Cont.	Set Size	# of Entr.	Entry Cont.
GAg(HR(1, ,r-sr), 1×PHT(2^r,A2),[c])	1		r-bit sr	1	2^r	Atm A2
PAg(BHT(256,1,r-sr), 1×PHT(2^r,A2),[c])	256	1	r-bit sr	1	2^r	Atm A2
PAg(BHT(256,4,r-sr), 1×PHT(2^r,A2),[c])	256	4	r-bit sr	1	2^r	Atm A2
PAg(BHT(512,1,r-sr), 1×PHT(2^r,A2),[c])	512	1	r-bit sr	1	2^r	Atm A2
PAg(BHT(512,4,r-sr), 1×PHT(2^r,A1),[c])	512	4	r-bit sr	1	2^r	Atm A1
PAg(BHT(512,4,r-sr), 1×PHT(2^r,A2),[c])	512	4	r-bit sr	1	2^r	Atm A2
PAg(BHT(512,4,r-sr), 1×PHT(2^r,A3),[c])	512	4	r-bit sr	1	2^r	Atm A3
PAg(BHT(512,4,r-sr), 1×PHT(2^r,A4),[c])	512	4	r-bit sr	1	2^r	Atm A4
PAg(BHT(512,4,r-sr), 1×PHT(2^r,LT),[c])	512	4	r-bit sr	1	2^r	Atm LT
PAg(IBHT(inf, ,r-sr), 1×PHT(2^r,A2),[c])	∞		r-bit sr	1	2^r	Atm A2
PAp(BHT(512,4,r-sr), 512×PHT(2^r,A2),[c])	512	4	r-bit sr	512	2^r	Atm A2
GSg(HR(1, ,r-sr), 1×PHT(2^r,PB),[c])	1		r-bit sr	1	2^r	PB
PSg(BHT(512,4,r-sr), 1×PHT(2^r,PB),[c])	512	4	r-bit sr	1	2^r	PB
BTB(BHT(512,4,A2), ,[c])	512	4	Atm A2			
BTB(BHT(512,4,LT), ,[c])	512	4	Atm LT			

Asc *– Table Set-Associativity,* ***Atm*** *– Automaton,* ***BHT*** *– Branch History Table,* ***BTB*** *– Branch Target Buffer Design,* ***Config.*** *– Configuration,* ***Entr.*** *– Entries,* ***GAg*** *– Global Two-Level Adaptive Branch Prediction Using a Global Pattern History Table,* ***GSg*** *– Global Static Training Using a Preset Global Pattern History Table,* ***IBHT*** *– Ideal Branch History Table,* ***inf*** *– Infinite,* ***LT*** *– Last-Time,* ***PAg*** *– Per-address Two-Level Adaptive Branch Prediction Using a Global Pattern History Table,* ***PAp*** *– Per-address Two-Level Adaptive Branch Prediction Using Per-address Pattern History Tables,* ***PB*** *– Preset Prediction Bit,* ***PSg*** *– Per-address Static Training Using a Preset Global Pattern History Table,* ***PHT*** *– Pattern History Table,* ***sr*** *– Shift Register.*

Table 3: Configurations of simulated branch predictors.

The pattern history bits in the pattern history table entries are also initialized at the beginning of execution. Since taken branches are more likely for those pattern history tables using automata $A1$, $A2$, $A3$, and $A4$, all entries are initialized to state 3. For *Last-Time*, all entries are initialized to state 1 such that the branches at the beginning of execution will be more likely to be predicted taken. It is not necessary to reinitialize pattern history tables during execution.

In addition to the Two-Level Adaptive schemes, Lee and A. Smith's Static Training schemes, Branch Target Buffer designs, and some dynamic and static branch prediction schemes were simulated for comparison purposes. Lee and A. Smith's Static Training scheme is similar in structure to the Per-address Two-Level Adaptive scheme with an IBHT but with the important difference that the prediction for a given pattern is pre-determined by profiling. In this study, Lee and A. Smith's Static Training is identified as PSg, meaning per-address Static Training using a global preset pattern history table. Similarly, the scheme which has a similar structure to GAg but with the difference that the second-level pattern history information is collected from profiling is abbreviated PSg, meaning Global Static Training using a preset global pattern history table. Per-address Static Training using per-address pattern history tables (PSp) is another application of Static Training to a different structure; however, this scheme requires a lot of storage to keep track of pattern behavior of all branches statically. Therefore, no PSp schemes were simulated in this study. Lee and A. Smith's Static Training schemes were simulated with the same branch history table configurations as used by the Two-Level Adaptive schemes for a fair comparison. The cost to implement Static Training is not less expensive than the cost to implement the Two-Level Adaptive Scheme because the branch history table and the pattern history table required by both schemes are similar. In Static Training, before program execution starts, extra time is needed to load the preset pattern prediction bits into the pattern history table.

Branch Target Buffer designs were simulated with automata $A2$ and *Last-Time*. The static branch prediction schemes simulated include the Always Taken, Backward Taken and Forward Not Taken, and a profiling scheme. Always Taken scheme predicts taken for all branches. Backward Taken and Forward Not Taken (BTFN) scheme predicts taken if a branch branches backward and not taken if the branch branches forward. The BTFN scheme is effective for loop-bound programs, because it mispredicts only once in the execution of a loop. The profiling scheme counts the frequency of taken and not-taken for each static branch in the profiling execution. The predicted direction of a branch is the one the branch takes most frequently. The profiling information of a program executed with a training data set is used for branch predictions for the program executed with testing data sets, thus calculating the prediction accuracy.

5 Branch Prediction Simulation Results

Figures 5 through 11 show the prediction accuracy of the branch predictors described in the previous session on the nine SPEC benchmarks. "Tot GMean" is the geometric mean across all the benchmarks, "Int GMean" is the geometric mean across all the integer benchmarks, and "FP GMean" is the geometric mean across all the floating point benchmarks. The vertical axis shows the prediction accuracy scaled from 76 percent to 100 percent.

5.1 Evaluation of the Parameters of the Two-Level Adaptive Branch Prediction Branch Prediction

The three variations of Two-Level Adaptive Branch Prediction were simulated with different history register lengths to assess the effectiveness of increasing the recorded history length. The PAg and PAp schemes were each simulated with an ideal branch history table (IBHT) and with practical branch history tables to show the effect of the branch history table hit ratio.

5.1.1 Effect of Pattern History Table Automaton

Figure 5 shows the efficiency of using different finite-state automata. Five automata $A1$, $A2$, $A3$, $A4$, and *Last-Time* were simulated with a PAg branch predictor, having 12-bit history registers in a four-way set-associative 512-entry BHT. $A1$, $A2$, $A3$, and $A4$ all perform better than *Last-Time*. The four-state automata $A1$, $A2$, $A3$, and $A4$ maintain more history information than *Last-Time* which only records what happened the last time; they are therefore more tolerant to the deviations in the execution history. Among the four-state automata, $A1$ performs worse than the others. The performance of $A2$, $A3$, and $A4$ are very close to each other; however, $A2$ usually performs best. In order to show the following figures clearly, each Two-Level Adaptive Scheme is shown with automaton $A2$.

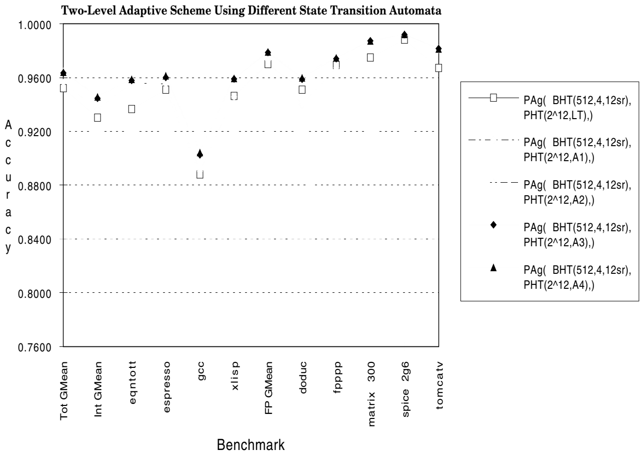


Figure 5: Comparison of Two-Level Adaptive Branch Predictors using different finite-state automata.

5.1.2 Effect of History Register Length

Three variations using history registers of the same length

Figure 6 shows the effects of history register length on the prediction accuracy of Two-Level Adaptive schemes. Every scheme in the graph was simulated with the same history register length. Among the variations, PAp performs the best, PAg the second, and GAg the worst.

GAg is not effective with 6-bit history registers, because every branch updates the same history register, causing excessive interference. PAg performs better than GAg, because it has a branch history table which reduces the interference in branch history. PAp predicts the best, because the interference in the pattern history is removed.

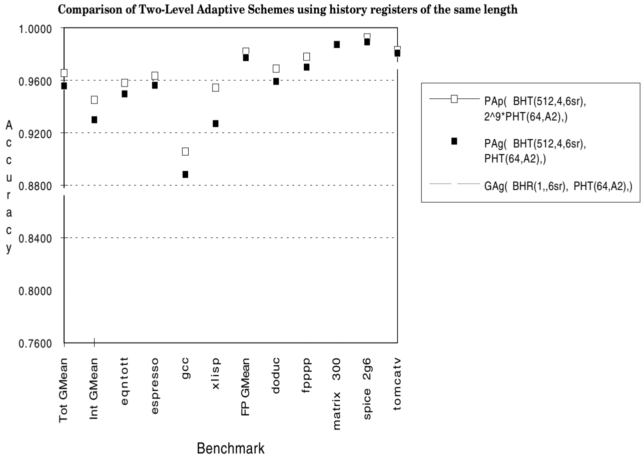


Figure 6: Comparison of the Two-Level Adaptive schemes using history registers of the same length.

Effects of various history register lengths

To further investigate the effect of history register length, Figure 7 shows the accuracy of GAg with various history register lengths. There is an increase of 9 percent in accuracy by lengthening the history register from 6 bits to 18 bits. The effect of history register length is obvious on GAg schemes. The history register length has smaller effect on PAg schemes and even smaller effect on PAp schemes because of the less interference in the branch history and pattern history and their effectiveness with short history registers.

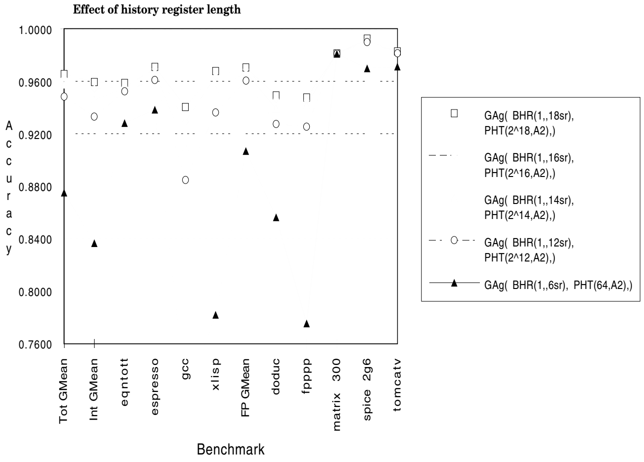


Figure 7: Effect of various history register lengths on GAg schemes.

5.1.3 Hardware Cost Efficiency of Three Variations

In Figure 6, prediction accuracy for the schemes with the same history register length were compared. However, the various Two-Level Adaptive schemes have different costs. PAp is the most expensive, PAg the second, and GAg the least, as you would expect. When evaluating the three variations of Two-Level Adaptive Branch Prediction, it is useful to know which variation is the least expensive when they predict with approximately the same accuracy.

Figure 8 illustrates three schemes which achieve about 97 percent prediction accuracy. One scheme is chosen for each variation to show the variation's configuration requirements to obtain that prediction accuracy. To achieve 97 percent prediction accuracy, GAg requires an 18-bit history register, PAg requires 12-bit history registers, and PAp requires 6-bit history registers. According to our cost estimates, PAg is the cheapest among these three. GAg's pattern history table is expensive when a long history register is used. PAp is expensive due to the required multiple pattern history tables.

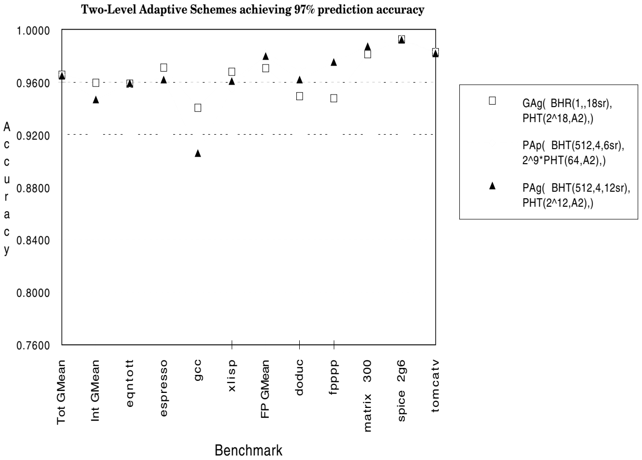


Figure 8: The Two-Level Adaptive schemes achieve about 97 percent prediction accuracy.

5.1.4 Effect of Context Switch

Since Two-Level Adaptive Branch Prediction uses the branch history table to keep track of branch history, the table needs to be flushed during a context switch. Figure 9 shows the difference in the prediction accuracy for three schemes simulated with and without context switches. During the simulation, whenever a trap occurs in the instruction trace or every 500,000 instructions if no trap occurs, a context switch is simulated. After a context switch, the pattern history table is not re-initialized, because the pattern history table of the saved process is more likely to be similar to the current process's pattern history table than to a re-initialized pattern history table. The value 500,000 is derived by assuming that a 50 MHz clock is used and context switches occur every 10 ms in a 1 IPC machine. The average accuracy degradations for the three schemes are all less than 1 percent. The accuracy degradations for *gcc* when PAg and PAp are used are much greater than those of the other programs because of the large number of traps in *gcc*. However, the excessive number of traps do not degrade the prediction accuracy of the GAg scheme, because an initialized global history register can be refilled quickly. The prediction accuracy of *fpppp* using GAg actually increases when context switches are simulated. There are very few conditional branches in *fpppp* and all the conditional branches have regular behavior; therefore, initializing the global history register helps clear out the noise.

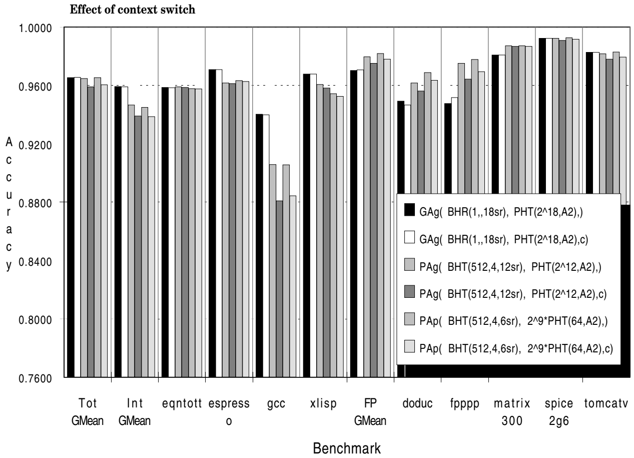


Figure 9: Effect of context switch on prediction accuracy.

5.1.5 Effect of Branch History Table Implementation

Figure 10 illustrates the effects of the size and associativity of the branch history table in the presence of context switches. Four practical branch history table implementations and an ideal branch history table were simulated. The four-way set-associative 512-entry branch history table's performance is very close to that of the ideal branch history table, because most branches in the programs can fit in the table. Prediction accuracy decreases as table miss rate increases, which is also seen in the PAp schemes.

5.2 Comparison of Two-Level Adaptive Branch Prediction and Other Prediction schemes

Figure 11 compares the branch prediction schemes. The PAg scheme which achieves 97 percent prediction accuracy is chosen for comparison with other well-known schemes, because it costs the least among the three variations of Two-Level Adaptive Branch Prediction.

The 4-way set-associative 512-entry BHT is selected to be used by all schemes which keep the first-level branch history information, because it is simple enough to be implemented. The Two-Level Adaptive scheme and the Static Training scheme were chosen on the basis of similar costs.

The top curve is achieved by the Two-Level Adaptive scheme whose prediction accuracy is about 97 percent.

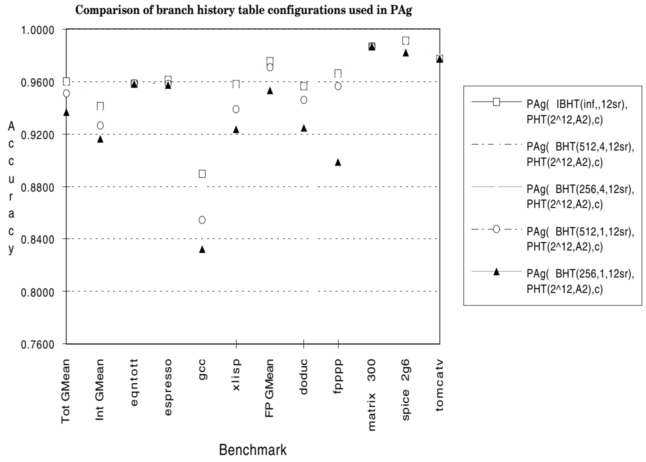


Figure 10: Effect of branch history table implementation on PAg schemes.

Since the data for the Static Training schemes are not complete due to the unavailability of appropriate data sets, the data points for *eqntott*, *fpppp*, *matrix*300 , and *tomcatv* are not graphed. PSg is about 1 to 4 percent lower than the top curve for the benchmarks that are available and GSg is about 4 to 19 percent lower with average prediction accuracy of 94.4 percent and 89 percent individually. Note that their accuracy depends greatly on the similarities between the data sets used for training and testing. The prediction accuracy for the branch target buffer using 2-bit saturating up-down counters [17] is around 93 percent. The Profiling scheme achieves about 91 percent prediction accuracy. The branch target buffer using *Last-Time* achieves about 89 percent prediction accuracy. Most of the prediction accuracy curves of BTFN and Always Taken are below the base line (76 percent). BTFN's average prediction accuracy is about 68.5 percent and Always Taken's is about 62.5 percent. In this figure, the Two-Level Adaptive scheme is superior to the other schemes by at least 2.6 percent.

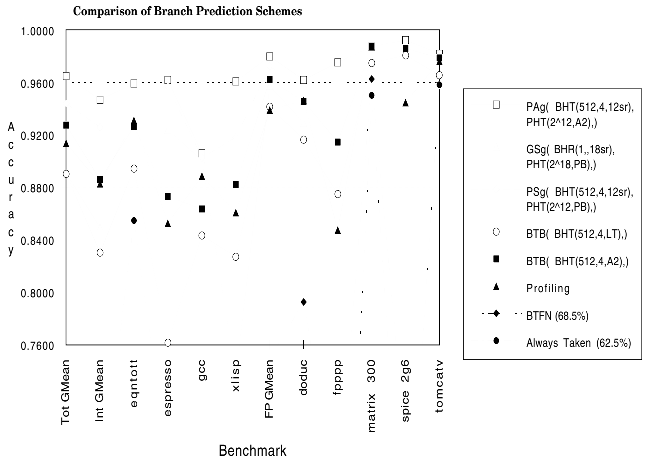


Figure 11: Comparison of branch prediction schemes.

6 Concluding Remarks

In this paper we have proposed a new dynamic branch predictor (Two-Level Adaptive Branch Prediction) that achieves substantially higher accuracy than any other scheme that we are aware of. We computed the hardware costs of implementing three variations of this scheme and determined that the most effective implementation of Two-Level Adaptive Branch Prediction utilizes a per-address branch history table and a global pattern history table.

We have measured the prediction accuracy of the three variations of Two-Level Adaptive Branch Prediction and several other popular proposed dynamic and static prediction schemes using trace-driven simulation of nine of the ten SPEC benchmarks. We have shown that the average prediction accuracy for Two-Level Adaptive Branch Prediction is about 97 percent, while the other known schemes achieve at most 94.4 percent average prediction accuracy.

We have measured the effects of varying the parameters of the Two-Level Adaptive predictors. We noted the sensitivity to k, the length of the history register, and s, the size of each entry in the pattern history table. We reported on the effectiveness of the various prediction algorithms that use the pattern history table information. We showed the effects of context switching.

Finally, we should point out that we feel our 97 percent prediction accuracy figures are not good enough and that future research in branch prediction is still needed. High performance computing engines in the future will increase the issue rate and the depth of the pipeline, which will combine to increase further the amount of speculative work that will have to be thrown out due to a branch prediction miss. Thus, the 3 percent prediction miss rate needs improvement. We are examining that 3 percent to try to characterize it and hopefully reduce it.

Acknowledgments The authors wish to acknowledge with gratitude the other members of the HPS research group at Michigan for the stimulating environment they provide, and in particular, for their comments and suggestions on this work. We are also grateful to Motorola Corporation for technical and financial support, and to NCR Corporation for the gift of an NCR Tower, Model No. 32, which was very useful in our work.

References

[1] T-Y Yeh and Y.N. Patt, "Two-Level Adaptive Branch Prediction", *Technical Report CSE-TR-117-91, Computer Science and Engineering Division, Department of EECS, The University of Michigan*, (Nov. 1991).

[2] T-Y Yeh and Y.N. Patt, "Two-Level Adaptive Branch Prediction", *The 24th ACM/IEEE International Symposium and Workshop on Microarchitecture* , (Nov. 1991), pp. 51-61.

[3] M. Butler, T-Y Yeh, Y.N. Patt, M. Alsup, H. Scales, and M. Shebanow, "Instruction Level Parallelism is Greater Than Two", *Proceedings of the 18th International Symposium on Computer Architecture*, (May. 1991), pp. 276-286.

[4] D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns" , *Proceedings of the 18th International Symposium on Computer Architecture*, (May 1991), pp. 34-42.

[5] Motorola Inc., "M88100 User's Manual", *Phoenix, Arizona*, (March 13, 1989).

[6] W.W. Hwu, T.M.Conte, and P.P.Chang, "Comparing Software and Hardware Schemes for Reducing the Cost of Branches", *Proceedings of the 16th International Symposium on Computer Architecture*, (May 1989).

[7] N.P. Jouppi and D. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (April 1989), pp. 272-282.

[8] D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processors ", *IEEE Computer*, (July 1988), pp.47-55.

[9] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-order Execution Machines", *IEEE Transactions on Computers*, (December 1987), pp.1496-1514.

[10] P. G. Emma and E. S. Davidson, "Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance" , *IEEE Transactions on Computers*, (July 1987), pp.859-876.

[11] J. A. DeRosa and H. M. Levy, "An Evaluation of Branch Architectures ", *Proceedings of the 14th International Symposium on Computer Architecture*, (June 1987), pp.10-16.

[12] D.R. Ditzel and H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", *Proceedings of the 14th International Symposium on Computer Architecture*, (June 1987), pp.2-9.

[13] S. McFarling and J. Hennessy, "Reducing the Cost of Branches", *Proceedings of the 13th International Symposium on Computer Architecture*, (1986), pp.396-403.

[14] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *IEEE Computer*, (January 1984), pp.6-22.

[15] T.R. Gross and J. Hennessy, "Optimizing Delayed Branches", *Proceedings of the 15th Annual Workshop on Microprogramming*, (Oct. 1982), pp.114-120.

[16] D.A. Patterson and C.H. Sequin, "RISC-I: A Reduced Instruction Set VLSI Computer", *Proceedings of the 8th International Symposium on Computer Architecture*, (May. 1981), pp.443-458.

[17] J.E. Smith, "A Study of Branch Prediction Strategies", *Proceedings of the 8th International Symposium on Computer Architecture*, (May. 1981), pp.135-148.

[18] T. C. Chen, "Parallelism, Pipelining and Computer Efficiency", *Computer Design*, Vol. 10, No. 1, (Jan. 1971), pp.69-74.