Two-Level Adaptive Training 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

***Abstrac**t. High-performance microarchitectures use, among other structures, deep pipelines to help speed up execution. The importance of a good branch predictor to the effectiveness of a deep pipeline in the presence of conditional branches is well-known. In fact, the literature contains proposals for a number of branch prediction schemes. 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.*

This paper proposes a new dynamic branch predictor, the Two-Level Adaptive Paining scheme, which alters the branch prediction algorithm on the basis of information collected at run-time.

Several configurations of the Two-Level Adaptive Training Branch Predictor are introduced, simulated, and compared to simulations of other known static and dynamic branch prediction schemes. Two-Level Adaptive Training Branch Prediction achieves 97 percent accuracy on nine of the ten SPEC benchmarks, compared to less than 93 percent for other schemes. Since a prediction miss requires flushing of the speculative execution already in progress, the relevant metric is the miss rate. The miss rate is 3 percent for the Two-Level Adaptive Training scheme vs. 7 percent (best case) for the other schemes. This represents more than a 100 percent improvement in reducing the number of pipeline hushes required.

1. Introduction

Pipelining, at least as early as [18] and continuing to the present time [6], has been one of the most effective ways to improve performance on a single processor. On the other hand, branches impede machine performance due to pipeline stalls for unresolved branches. As pipelines get deeper or issuing bandwidth becomes greater, the negative effect of branches on performance increases.

Among different types of branches, conditional branches have to wait for the condition to be resolved and the target address to be calculated before the target instruction can be fetched. Unconditional branches have to wait for the target address to be calculated. In conventional computers, instruction issuing stalls until the target address is determined, resulting in pipeline bubbles. When the number of cycles taken to resolve a branch is large, the performance loss due to the pipeline stalls is considerable. There are two ways to reduce the loss: the first is to resolve the branch as early as possible to reduce the instruction fetch pipeline bubbles. The second is to provide fast fetching and decoding of the target instruction to reduce the execution pipeline bubbles. Branch prediction is a way to reduce the execution penalty due to branches by predicting, prefetching and initiating execution of the branch target before the branch is resolved.

Branch prediction schemes can be classified into static schemes and dynamic schemes depending on the information used to make predictions. Static branch prediction schemes can be as simple as predicting that all branches are not taken or predicting that all branches are taken. Predicting that all branches are taken can achieve approximately 68 percent prediction accuracy as reported by Lee and Smith [13]. In the dynamic instructions of the benchmarks used in this study, about 60 percent of conditional branches are taken. Static predictions can also be based on the opcode. Certain classes of branch instructions tend to branch more in one direction than the other. The branch direc-

Originally published in *Proc. 24th Ann. Int'l Symp. Microarchitecture*, 1991, pp. 51–61. Copyright © 1991 ACM. All rights reserved.

tion can also be taken into consideration such as the Backward Taken and Forward Not Taken scheme [16] which is fairly effective in loop-bound programs, because it misses only once over all iterations of a loop. However, this scheme does not work well on programs with irregular branches. Profiling [12, 5] can also be used to predict the branch path by measuring the tendencies of the branches and presetting a static prediction bit in the opcode. However, program profiling has to be performed in advance with certain sample data sets which may have different branch tendencies than the data sets that occur at run-time.

Dynamic branch prediction takes advantage of the knowledge of branches' run-time behavior to make predictions. Lee and Smith proposed a structure they called a Branch Target Buffer [13] which uses 2-bit saturating up-down counters to collect history information which is then used to make predictions. The execution history dynamically changes the state of the branch's entry in the buffer. In their scheme, branch prediction is based on the state of the entry. The Branch Target Buffer design can also be simplified to record only the result of the last execution of the branch. Another dynamic scheme also proposed by Lee and Smith is the Static Training scheme [13] which uses the statistics collected from a pre-run of the program and a history pattern consisting of the last *n* run-time execution results of the branch to make a prediction. The major disadvantage of the Static Training scheme is that the program has to be run first to accumulate the statistics and the same statistics may not be applicable to different data sets.

There is serious performance degradation in deep-pipelined and/or superscalar machines caused by prediction misses due to the large amount of speculative work that has to be discarded [1, 8]. This is the motivation for proposing a new, higher-accuracy dynamic branch prediction scheme. The new scheme uses two levels of branch history information to make predictions. The first level is the history of the last *n* branches. The second is the branch behavior for the last *s* occurrences of that unique pattern of the last *n* branches. The history information is collected on the fly without executing the program beforehand, eliminating the major disadvantage of Static Training Prediction. The scheme proposed here is called Two-Level Adaptive Training Branch Prediction, because predictions are based not only on the record of the last *n* branches, but moreover on the record of the last *s* occurrences of the particular record of the last *n* branches.

Trace-driven simulations were used in this study. The Two-Level Adaptive Training branch prediction scheme as well as the other dynamic and static branch prediction schemes were simulated on the SPEC benchmark suite. By using Two-Level Adaptive Training Branch Prediction, the average prediction accuracy for the benchmarks reaches 97 percent, while most of the other schemes achieve under 93 percent. This represents more than 100 percent reduction in mispredictions by using the Two-Level Adaptive Training scheme. This reduction can lead directly to a large performance gain on a high-performance processor.

Section two gives an introduction to the proposed Two-Level Adaptive Training Branch Prediction scheme. Section three discusses the methodology used in this study and the simulated prediction models. Section four reports the simulation results of a wide selection of schemes including both the dynamic and the static branch predictors. Section five contains some concluding remarks.

2. Two-Level Adaptive Training Branch Prediction

The Two-Level Adaptive Training Branch Prediction scheme has the following characteristics:

- Branch prediction is based on the history of branches executed during the current execution of the program.
- Execution history pattern information is collected on the fly of the program execution by updating the pattern history information in the branch history pattern table of the predictor. Therefore, no pre-runs of the program are necessary.

2.1 Concept of Two-Level Adaptive Training Branch Prediction

The Two-Level Adaptive Training scheme has two major data structures, the branch history register (HR) and the branch history pattern table (PT), similar to those used in the Static Training scheme of Lee and Smith [13]. In Two-Level Adaptive Training, instead of accumulating statistics by profiling the programs, the execution history information on which branch predictions are based is collected by updating the contents of the history registers and the pattern history bits in the entries of the pattern table depending on the outcomes of the branches. The history register is a shift register which shifts in bits representing the branch results of the most recent history information. All the history registers are contained in a history register table (HRT). The pattern history bits represent the most recent branch results for the particular contents of the history register. Branch predictions are made by checking the pattern history bits in the pattern table entry indexed by the content of the history register for the particular branch that is being predicted.

Originally published in *Proc. 24th Ann. Int'l Symp. Microarchitecture*, 1991, pp. 51–61. Copyright © 1991 ACM. All rights reserved.

Since the history register table is indexed by branch instruction addresses, the history register table is called a per-address history register table (PHRT). The pattern table is called a global pattern table, because all the history registers access the same pattern table.

The structure of Two-Level Adaptive Training Branch Prediction is shown in Figure 1. The prediction of a branch B_i is based on the history pattern of the last outcomes of executing the branch; therefore, k bits are needed in the history register for each branch to keep track of the history. 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. In order to keep track of the history of the patterns, there are 2^k entries in the pattern table; each entry is indexed by one distinct history pattern.

When a conditional branch B_i is being predicted, the contents of its history register, HR_i, whose content is denoted as $R_{i,\ c-k}R_{i,\ c-k+1}......R_{i,\ c-1}$ for the last k outcomes of executing the branch, is used to address the pattern table. The pattern history bits S_c in the addressed entry $PTR_{i,c-k}R_{ic-k+1},......R_{ic-1}$ in the pattern table are then used for predicting the branch. The prediction of the branch is

$$z_c = \lambda(S_c), \tag{1}$$

where λ is the prediction decision function.

After the conditional branch is resolved, the outcome $R_{i,c}$ is shifted left into the history register HR_i in the least significant bit position and is also used to update the pattern history bits in the pattern table entry $PTR_{i,c-k}R_{ic-k+1},......R_{ic-1}$. After being updated, the content of the history register becomes $R_{ic-k+1}R_{i,c-k+2}......R_{i,c}$ and the state represented by the pattern history bits becomes S_{c+1}. The transition of the pattern history bits in the pattern 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

$$S_{c+1} = \delta(S_c, R_{ic}) \tag{2}$$

A straightforward combinational logic circuit is used to implement the function δ to update the pattern history bits in the entries of the pattern table. The transition function δ, pattern history bits S and the outcome R of the branch comprise a finite-state machine, which can be characterized by equations 1 and 2. Since the prediction is based on the pattern history bits, the finite-state machine is a Moore machine with the output z characterized by equation 1.

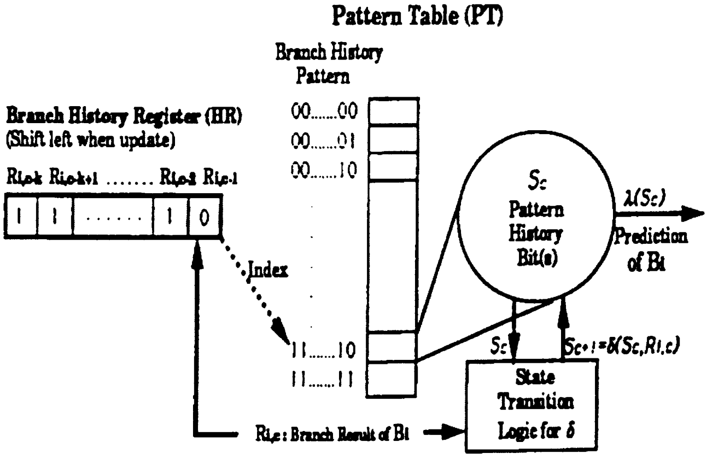

Figure 1: The structure of the Two-Level Adaptive Training scheme.

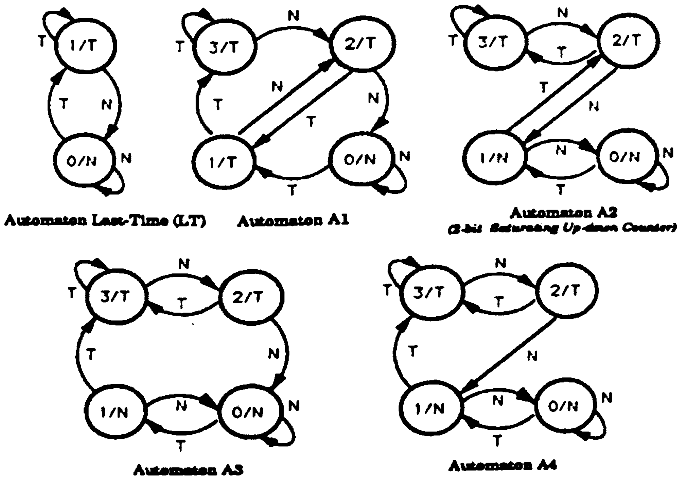

Figure 2: The state transition diagrams of the finite-state machines used for updating the pattern history in the pattern table entry.

The state transition diagrams of the finite-state machines used in this study for updating the pattern history in the pattern table entry are shown in Figure 2. The automaton *Last-Time* stores in the pattern history bit 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 the pattern history information. The automaton *A*l 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 *A*2 is a saturating up-down counter, which is also used, but differently, in Lee and Smith's Branch Target Buffer design [13]. The counter is incremented when the branch is taken and is decremented when the branch is not taken. 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. Automata *A*3and *A*4 are both similar to *A*2.

Both Static Training and Two-Level Adaptive Training 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 table changes dynamically in Two-Level Adaptive Training 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 determined 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 Training, on the other hand, updates the appropriate pattern history information with the actual result of each branch. As a result, given the same branch history pattern, different pattern history information can be found in the pattern table; therefore, there can be different inputs to the prediction decision function for Two-Level Adaptive Training. Predictions of Two-Level Adaptive Training change adaptively in accordance with the program execution behavior.

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

3. Implementation Methods

3.1 Implementations of the Per-address History Register Table

It is not feasible to have a big enough history register table for each static branch to have its own history register in real implementations. Therefore, two approaches are proposed for implementing the Per-address History Register Table.

The first approach is to implement the per-address register table as a set-associative cache. A fixed number of entries in the table are grouped together as a set. Within a set, the 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 used as a tag which is recorded in the entry allocated for the branch. The per-address history register table implemented in this way is called the Associative History Register Table (AHRT). When a conditional branch is to be predicted, the branch's entry in the AHRT is located first. If the branch has an entry in the AHRT, the contents of the corresponding history register is used to address the pattern table. If the branch does not have an entry in the AHRT, a new entry is allocated for the branch. There is an extra cost for implementing the tag store in this approach.

The second approach is to implement the history register table as a hash table. The address of a conditional branch is used for hashing into the table. The per-address history table using this approach is called the Hash History Register Table (HHRT). Since collisions can occur when accessing a hash table, this implementation results in more interference in the execution history. As one would expect, the prediction accuracy for this approach is lower than what would be obtained with an AHRT, but the cost of the tag store is saved.

In this study, the above two practical approaches and the Ideal History Register Table (IHRT), in which there is a history register for each static conditional branch, were simulated for the Two-Level Adaptive Training Branch Predictor. The AHRT was simulated with two configurations: 512-entry 4-way set-associative and 256 entry 4-way set-associative. The HHRT was also simulated with 512 entries and 256 entries The IHRT simulation data is provided to show how much accuracy is lost due to the history interference in the practical history register table designs.

3.2 Prediction Latency

The Two-Level Adaptive Training Branch Predictor needs two sequential table lookups to make a prediction. It is hard to squeeze the two lookups into one cycle, which is usually the requirement for a high-performance processor in determining the next instruction address. The solution to this problem is to perform the pattern table lookup with the updated history pattern of a branch at the time the history register is updated, produce a prediction from the pattern table, and store the prediction as a prediction bit in the history register table with the history register for the branch. Therefore, the next time the branch must be predicted, the prediction is available in the history register table, and the pattern table does not have to be accessed that cycle.

Another problem occurs when the prediction of a branch is required before the result of the previous execution of the branch has been confirmed. This case appears very often when a tight loop is being executed by a deep-pipelined superscalar machine, but not usually otherwise. Since this kind of branch has a high tendency to be taken, the branch is predicted taken and the machine does not have to stall until the previous branch result is confirmed.

4. Methodology and Simulation Model

Trace-driven simulations were used in this study. A Motorola 88100 instruction level simulator (ISIM) 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.

The branch instructions in the M88100 instruction set [4] are classified into four classes: conditional branches, subroutine return branches, immediate unconditional branches, and unconditional branches on registers. Instructions other than the branches are classified into the non-branch instruction class.

Conditional branches have to wait for condition codes in order to decide the branch targets. Subroutine return branches can be predicted by using a return address stack. A return address is pushed onto the stack when a subroutine is called and is popped as the prediction for the branch target address when a return instruction is detected. The return address prediction may miss when the return address stack overflows. For instruction sets without special instructions for returns from sub-routines, the double stacks scheme proposed by Kaeli and Emma in [2] is able to perform the return address prediction. An immediate unconditional branch's target address is calculated by adding the offset in the instruction to the program counter; therefore, the target address can be generated immediately. Unconditional branches on registers have to wait for the register value which is the target address to become ready.

4.1 Description of Traces

Nine benchmarks from the SPEC benchmark suite are used in this branch prediction study. Five are float-ing 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 in-

Originally published in *Proc. 24th Ann. Int'l Symp. Microarchitecture*, 1991, pp. 51–61. Copyright © 1991 ACM. All rights reserved.

cluded because it takes too long to capture the branch behavior of all seven kernels. Among the five floating point benchmarks, *matrix*300 and *tomcatv* have repetitive loop execution; thus, a very high prediction accuracy is attainable. The integer benchmarks tend to have many conditional branches and irregular branch behavior. Therefore, it is on the integer benchmarks where the mettle of the branch predictor is tested.

Since this study focuses on the prediction for conditional branches, all benchmarks except *fpppp* and *gcc* were simulated for twenty million conditional branch instructions. The benchmarks *fpppp* and *gcc* finish execution before twenty millions conditional branches are executed. The number of dynamic instructions simulated for the benchmarks range from fifty million to 1.8 billion.

The dynamic instruction distribution is shown in Figure 3. 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.

The distribution of the dynamic branch instructions is shown in Figure 4. As can be seen from the distribution, about 80 percent of the dynamic branch instructions are conditional branches. The conditional branch is the branch class that should be studied to improve the prediction accuracy. The number of static conditional branches in the trace tapes of the benchmarks are listed in Table 1.

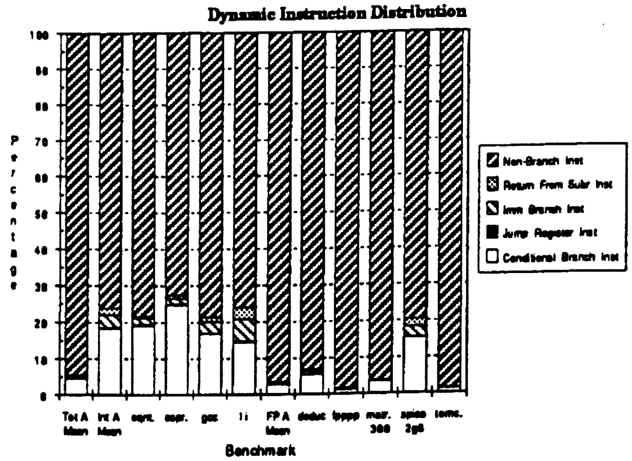

Figure 3: Distribution of dynamic instructions.

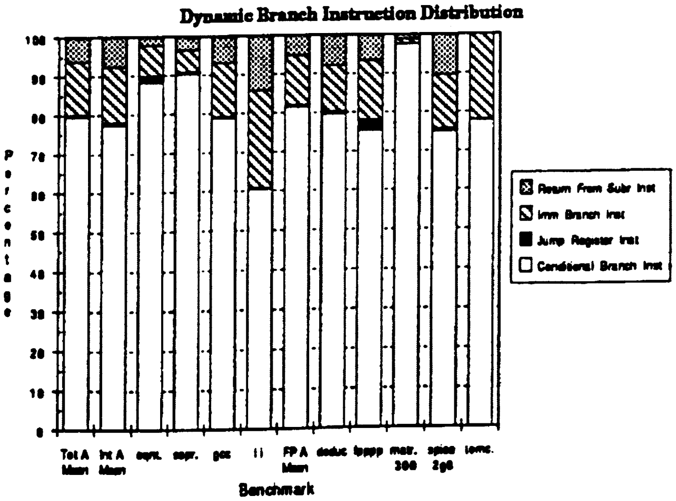

Figure 4: Distribution of dynamic branch instructions.

Originally published in *Proc. 24th Ann. Int'l Symp. Microarchitecture*, 1991, pp. 51–61. Copyright © 1991 ACM. All rights reserved.

Table 1: The number of static conditional branches in each benchmark.

Benchmark name	Number of Static Conditional Branches
eqntott	277
gcc	6922
doduc	1149
Matrix300	213
tomcatv	370
espresso	556
li	489
fpppp	653
Spice2g6	606

4.2 Simulation Model

Several configurations were simulated for the Two-Level Adaptive Training scheme. For the per-address history register table (PHRT), two practical implementations, the associative HRT (AHRT) and the hash HRT (HHRT), along with the ideal HRT (IHRT) were simulated. In order to distinguish the different schemes, the naming convention for the branch prediction schemes is *Scheme*(*History*(*Size*, *Entry_Content*), *Pattern*(*Size*, *Entry_Content*), *Data*). *Scheme* specifies the scheme, for example, Two-Level Adaptive Training (AT), Static Training (ST), or Lee and Smith’s Branch Target Buffer design (LS). In *History*(*Size*, *Entry_Content*), *History* is the implementation for keeping history information of branches, for example, IHRT, AHRT, or HHRT. 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 history register table can be any automaton shown in Figure 2 or a history register. In *Pattern*(*Size*, *Entry_Content*), *Pattern* is the implementation for keeping history information for history patterns, *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 Lee and Smith’s Branch Target Buffer designs, the Pattern part is not included, because there is no pattern history information kept in their designs. *Data* specifies how the data sets are used. When *Data* is specified as *Same*, the same data set is used for both training and testing. When *Data* is specified as *Diff*, different data sets are used for training and testing. If *Data* is not specified, no training data set is needed for the schemes, as in Two-Level Adaptive Training schemes or Lee and Smith’s Branch Target Buffer designs. The configuration and scheme of each simulation model in this study are listed in Table 2.

Since about 60 percent of branches are taken according to our simulation results, the contents of the history register usually should contain more 1’s than 0’s. Accordingly, all the bits in the history register of each entry in the HRT are initialized to l’s at the beginning of program execution. During execution, when an entry is re-allocated to a different static branch, the history register is not re-initialized.

The pattern history bits in the pattern table entries are also initialized at the beginning of execution. Since taken branches are more likely, for those pattern tables using automata, *A*l, *A*2, *A*3, and *A*4, 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.

In addition to the Two-Level Adaptive Training schemes, Lee and Smith’s Static Training schemes and Branch Target Buffer designs, and some dynamic and static branch prediction schemes were simulated for comparison purposes. Lee and Smith’s Static Training scheme is similar to the Two-Level Adaptive Training scheme with an IHRT but with the important difference that the prediction for a given pattern is pre-determined by profiling. The two practical approaches for the HRT were also simulated for Static Training with the same accessing method introduced above.

Table 2: Configurations of simulated branch predictors.

Model Name	**Number of Entries**	**Entry Content**	**Number of Entries**	**Entry Content**
AT (AHRT(256, 12SR), PT(2^{12}, A2),)	256	12-bit SR	2^{12}	ATM A2
AT (AHRT(512, 12SR), PT(2^{12}, A2),)	512	12-bit SR	2^{12}	ATM A2
AT (AHRT(512, 12SR), PT(2^{12}, A3),)	512	12-bit SR	2^{12}	ATM A3
AT (AHRT(512, 12SR), PT(2^{12}, A4),)	512	12-bit SR	2^{12}	ATM A4
AT (AHRT(512, 12SR), PT(2^{12}, LT),)	512	12-bit SR	2^{12}	ATM LT
AT (AHRT(512, 10SR), PT(2^{10}, A2),)	512	10-bit SR	2^{10}	ATM A2
AT (AHRT(512, 8SR), PT(2^{8}, A2),)	512	8-bit SR	2^{8}	ATM A2
AT (AHRT(512, 6SR), PT(2^{6}, A2),)	512	6-bit SR	2^{6}	ATM A2
AT (HHRT(256, 12SR), PT(2^{12}, A2),)	256	12-bit SR	2^{12}	ATM A2
AT (HHRT(512, 12SR), PT(2^{12}, A2),)	512	12-bit SR	2^{12}	ATM A2
AT (IHRT(256, 12SR), PT(2^{12}, A2),)	∞	12-bit SR	2^{12}	ATM A2
ST (AHRT(512, 12SR), PT(2^{12}, PB),Same)	512	12-bit SR	2^{12}	PB
ST (HHRT(512, 12SR), PT(2^{12}, PB),Same)	512	12-bit SR	2^{12}	PB
ST (IHRT(12SR), PT(2^{12}, PB),Same)	∞	12-bit SR	2^{12}	PB
ST (AHRT(512, 12SR), PT(2^{12}, PB),Diff)	512	12-bit SR	2^{12}	PB
ST (HHRT(512, 12SR), PT(2^{12}, 512),Diff)	512	12-bit SR	2^{12}	PB
ST (IHRT(12SR), PT(2^{12}, PB),Diff)	∞	12-bit SR	2^{12}	PB
LS(AHRT(512, A2),,)	512	ATM A2		
LS(AHRT(512, LT),,)	512	ATM LT		
LS(HHRT(512, A2),,)	512	ATM A2		
LS(HHRT(512, LT),,)	512	ATM LT		
LS(IHRT(,A2),,)	∞	ATM A2		
LS(IHRT(,LT),,)	∞	ATM LT		

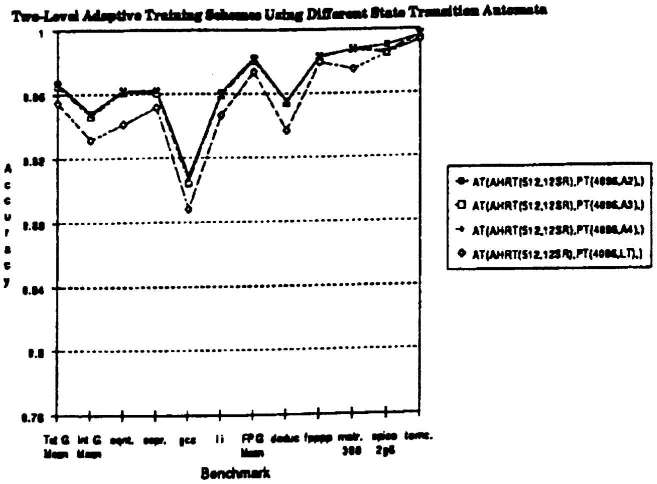

Figure 5: Two-Level Adaptive Training schemes using different state transition automata.

Lee and Smith's Branch Target Buffer designs were simulated with automata *A*2, *A*3, *A*4, and *Last-Time*. The static branch prediction schemes simulated include the Always Taken, Backward Taken and Forward Not taken, and a simple profiling scheme. The profiling scheme is done by counting 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. Since the same data set was used for profiling and execution in this study, the prediction accuracy was calculated by taking the ratio of the sum of the larger number in the two numbers for two possible directions of every static branch over the total number of the dynamic conditional branch instructions.

5. Simulation Results

The simulation results presented in this section were run with the Two-Level Adaptive Raining schemes, the Static Training Schemes, the Branch Target Buffer designs, and some static branch prediction schemes. Figures 5 through 10 show the prediction accuracy across the nine benchmarks. On the horizontal axis, the category labeled as "Tot G Mean" shows the geometric mean across all the benchmarks, "Int G Mean" shows the geometric mean across all integer benchmarks, and "FP G Mean" shows the geometric mean across all floating point benchmarks. The vertical axis shows the prediction accuracy scaled from 76 percent to 100 percent. This section concludes with a comparison between different branch prediction schemes.

5.1 Two-Level Adaptive Training

The Two-Level Adaptive Training schemes were simulated with different state transition automata, different HRT implementations, and different history register lengths to show their effects on prediction accuracy. The simulations of the Two-Level Adaptive Training scheme using an IHRT demonstrate the accuracy the scheme can achieve without history table miss effect and is used as a comparison to Lee and Smith's Static Training scheme which also uses the ideal history register table.

5.1.1 Effect of State Transition Automata

Figure 5 shows the efficiency of different state transition automata. Four state transition automata, *A*2, *A*3, *A*4, and *Last-Time* were simulated. *A*1 is not included, because early experiments indicated it was inferior to the other four-state automata, *A*2, *A*3, and *A*4. The scheme using *Last-Time* performs about l percent worse than the ones using the other automata which achieve similar accuracy around 97 percent. The four-state finite-state machines maintain more history information than the *Last Time* which only records what happened last time; *A*2, *A*3, and *A*4 are therefore more tolerant to noise in the execution history.

In order to show the curves clearly in the following figures, each scheme is shown with the state transition automata *A*2 which usually performs the best among the state transition automata used in this study.

5.1.2 Effect of History Register Table Implementation

Figure 6 shows the effects of the HRT implementations on the prediction accuracy of the Two-level Adaptive Training schemes. Every scheme in the graph was simulated with the same history register length. With the equivalent history register length, the IHRT scheme performs the best, the 512-entry AHRT scheme the second, the 512-entry HHRT scheme the third, the 256-entry AHRT scheme the fourth and the 256-entry HHRT scheme the worst, in the decreasing order of the HRT hit ratio. This is due to the increasing interference in the branch history as the hit ratio decreases.

5.1.3 Effect of History Register Length

Figure 7 shows the effect of history register length on the prediction accuracy of Two-Level Adaptive Training schemes. The Two-Level Adaptive Training schemes using four different history register lengths were simulated. The accuracy increases for about 0.5 percent by lengthening the history registers for 2 bits. According to the simulation results, increasing the history register length often improves the prediction accuracy until the accuracy asymptote is reached.

5.2 Static Training

Static Training Branch Prediction examines the history pattern of the last *n* executions of a branch and the statistics gathered from profiling the program with a training data set to calculate the probabilities the branch will be taken or not-taken with the given history pattern to predict the branch path.

Although the accounting required to gather the training statistics can be done in software, the Static Training scheme needs to keep track of the execution history of every static branch in the program, which requires hardware support. History registers must be used to keep track of the branch execution history of each static branch during run-time. When a branch is being predicted, its recorded history pattern is used to index the branch pattern table which contains preset branch prediction information. The preset prediction bit is then used for predicting the branch. Because the number of static branches varies from one program to another, the number of history registers required changes, which requires the hardware to offer a big enough table like IHRT to hold all the static branches in the programs. In order to consider the effects of practical implementations, in addition to the IHRT, the two practical HRT implementations used in this study were simulated with the Static Training schemes. The cost to implement Static Training is not any less expensive than for Two-Level Adaptive Training, because the history register table and pattern table required by both schemes are similar. However, the state transition logic in the pattern table is simpler for the Static Training scheme.

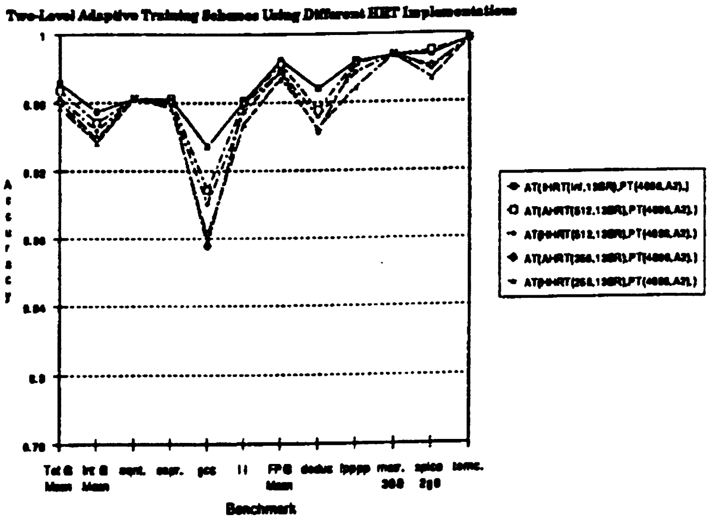

Figure 6: Two-Level Adaptive Training schemes using different history register table implementations.

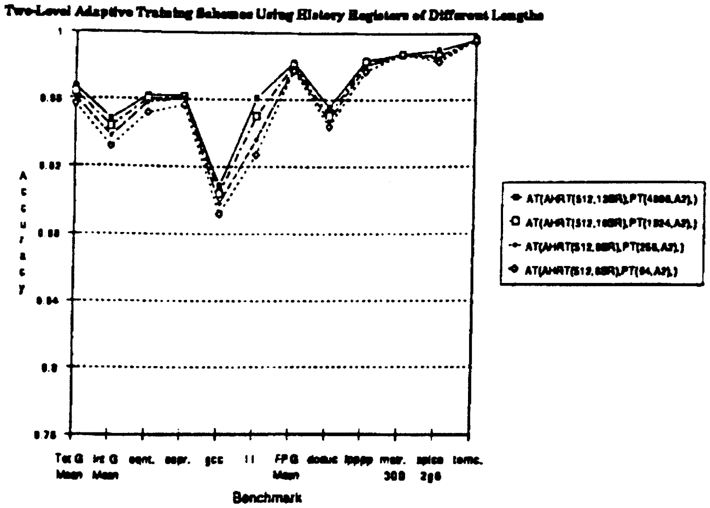

Figure 7: Two-Level Adaptive Training schemes using history registers of different lengths.

In order to show the effects of the training data sets, the simulation results for the schemes (with Same in their names) which were trained and tested on the same data set and those for the schemes (with Diff in their names) which were trained and tested on different data sets are both presented. All the testing data sets are the same as those used by other schemes in order for a fair comparison. In the schemes which were trained and executed on the same data set, the results are the best the Static Training schemes can achieve with that data set, because the best predictions for branches are known beforehand.

Five of nine benchmarks were trained with other applicable data sets. The other four benchmarks, *eqntott*, *matrix*300, *fpppp*, and *tomcatv*, are excluded because there are no other applicable data sets or the applicable data sets are too similar to each other. The data sets used in training and testing are shown in Table 3.

Table 3: Training and testing data sets of each benchmark.

Benchmark Name	**Training Data Set**	**Testing Data Set**
Eqntott	NA	Int_pri_3.eqn
Espresso	Cps	Bca
Gcc	Cexp.i	Dbxout.i
Li	Tower of hanoi	Eight queens
Doduc	Tiny doducin	Doducin
Fpppp	NA	Natoms
Matrix300	NA	NA
Spice2g6	Short greycode.in	Greycode.in
tomcatv	NA	NA

The Static Training schemes with similar configurations to the Two-Level Adaptive Training schemes in Figure 6 are shown in Figure 8. The highest prediction accuracy of the schemes using the same data set for training and execution is about 97 percent. This is achieved by the Static Training scheme using 12 bit history registers and an IHRT. The accuracy is about the same as that achieved by the Two-Level Adaptive Training scheme using 12 bit history registers and a 512-entry 4-way AHRT. However, when different data sets are used for training and execution, the prediction accuracy for *gcc* and *espresso* is about 1 percent lower respectively. The drop in the accuracy for *li* is more significant. It is about 5 percent lower. For the floating point benchmarks, the degradations are not so apparent due to the regular branch behavior of the programs. The degradations are within 0.5 percent. Since the data for the Static Training Schemes using different data sets for training and testing is not complete, the average accuracy for the schemes is not graphed.

5.3 Other Schemes

Figure 9 shows the simulation results of Lee and Smith's Branch Target Buffer designs, Backward Taken and Forward Not taken (BTFN), Always Taken, and the profiling scheme. The Branch Target Buffer designs were simulated with automata, *A*1, *A*2, *A*3, *A*4, and *Last-Time*. Only the results of the designs using *A*2 and *Last-Time* are shown in the figure, because the results of the designs using *A*3 and *A*4 are similar to those of the designs using *A*2. The designs using *A*1 predict about 2 to 3 percent lower than those using *A*2. Three buffer configurations, similar to IHRT, AHRT, and HHRT, were simulated. Using an IHRT in those schemes sets the upper bound at 93 percent for the same schemes with practical HRT implementations. Using *Last-Time* is about 4 percent lower than using *A*2.

BTFN and Always Taken predict poorly compared to the other schemes. Some of the data points fall below 76 percent.

The Backward Taken and Forward Not taken scheme (BTFN) is effective for the loop-bound benchmarks like *matrix*300 and *tomcatv* but not for other benchmarks. For the loop-bound benchmarks, the prediction accuracy is as high as 98 percent. However, for the other benchmarks, its accuracy is often lower than 70 percent. The average accuracy is approximately 69 percent.

The accuracy of the Always Taken scheme changes quite markedly from one benchmark to another. Its average is about 60 percent.

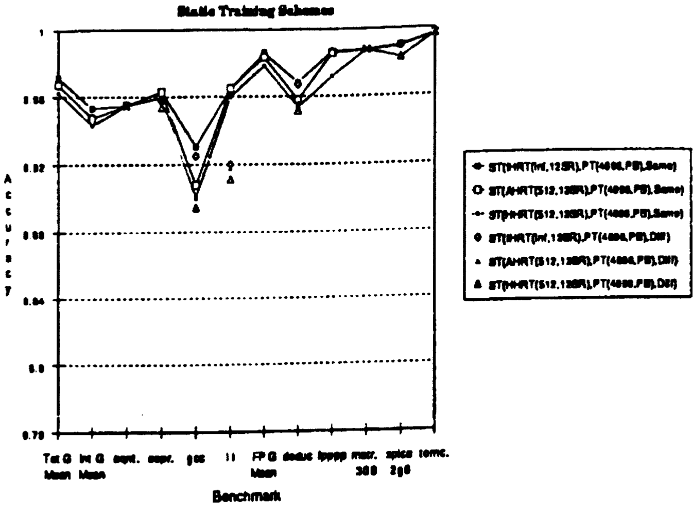

Figure 8: Prediction accuracy of Static Training schemes.

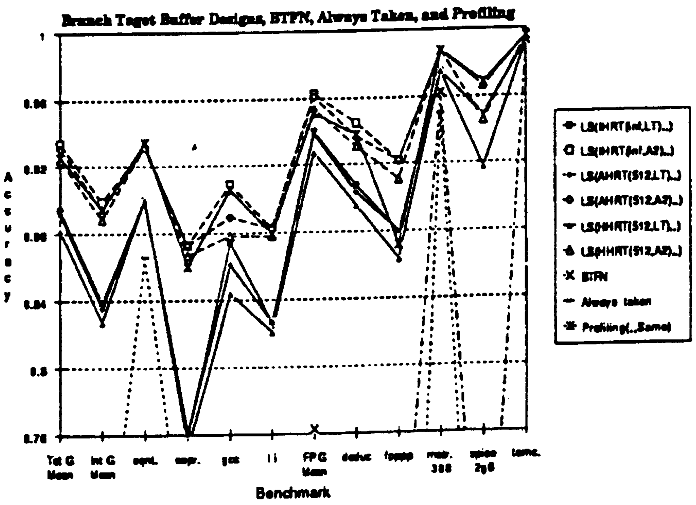

Figure 9: Prediction accuracy of Branch Target Buffer designs, BTFN, Always Taken, and the Profiling scheme.

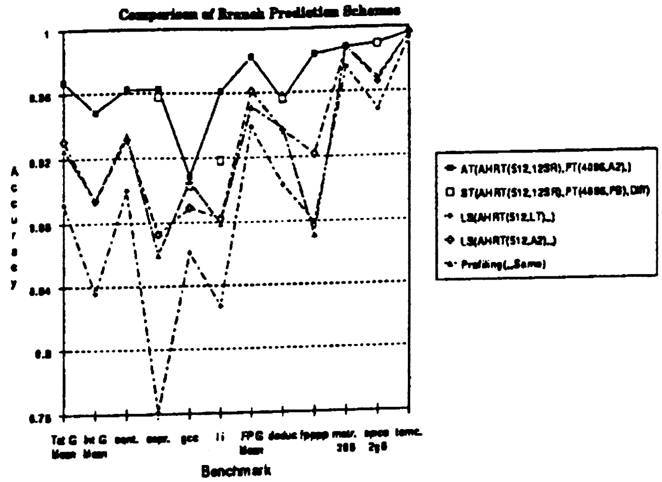

Figure 10: Comparison of branch prediction schemes

The simple profiling scheme simulated here is to run the program once to accumulate the statistics of how many times the branch is taken and how many times the branch is not taken for each branch. The prediction bit in the opcode of the branch is set or cleared depending on whether the taken branch count is larger than the not-taken branch count or not. The run-time prediction of the branch is made according to the prediction bit. The average of this scheme is about 92.5 percent. This scheme is fairly simple but at the cost of profiling and low prediction accuracy.

5.4 Comparison of Schemes

Figure 10 illustrates the comparison between the schemes mentioned above. The 512-entry 4-way AHRT was chosen for all the uses of HRT, because it is simple enough to be implemented. Two-Level Adaptive and Static training schemes are chosen on the basis of similar costs. At the top is the Two-Level Adaptive Training scheme whose average prediction accuracy is about 97 percent. As can be seen from the graph, the Static Training scheme predicts about 1 to 5 percent lower than the top curve. The profiling scheme predicts almost as well as Lee and Smith's Branch Target Buffer design with accuracy around 92.5 percent. The scheme which predicts a branch with the last result of the execution of the branch achieves about 89 percent accuracy.

6. Concluding Remarks

This paper proposes a new branch predictor, Two-Level Adaptive Training. The scheme predicts a branch by examining the history of the last *n* branches and the branch behavior for the last *s* occurrences of that unique pattern of the last *n* branches.

The Two-Level Adaptive Training schemes were simulated with three HRT configurations: the IHRT which is an ideal history register table large enough to hold all static branches, the AHRT which is a set-associative cache, and the HHRT which is a hash table. The IHRT data was included to obtain upper bounds for each of the other schemes. A scheme using an AHRT usually has higher prediction accuracy than the same scheme using an HHRT of the same size, because the AHRT has lower miss rate than the HHRT.

Each Two-Level Adaptive Training scheme was simulated with various history register lengths. As seen from the simulation results, prediction accuracy is usually improved by lengthening the history register.

In addition to the Two-Level Adaptive Training scheme, several other dynamic or static branch prediction schemes such as Lee and Smith's Static Training schemes, Branch Target Buffer designs, Always Taken, Backward Taken and Forward Not taken, and a simple profiling scheme were simulated.

The Two-Level Adaptive Training scheme has been shown to have an average prediction accuracy of 97 percent on nine benchmarks from the SPEC benchmark suite. The prediction accuracy is about 4 percent better than most of the other static or dynamic branch prediction schemes, which means more than a 100 percent reduction in the number of pipeline flushes required. Since a prediction miss causes flushing of the speculative execution already in progress, the performance improvement on a high-performance processor can be considerable by using the Two-Level Adaptive Training scheme.

Deep-pipelining and superscalar execution are effective methods for exploiting instruction level parallelism to improve single processor performance. This effectiveness, however, depends critically on the accuracy of a good branch predictor. Two-Level Adaptive Training Branch Prediction is proposed as a way to support high performance processors by minimizing the penalty associated with mispredicted branches.

References

[1] M. Butler, T-Y Yeh, Y.N. Patt, M. Alsup, H. Scales, and M. Shebanow, "Instruction Level Parallelism is Greater Than Two," *Proc. 18th Int'l Symp. Computer Architecture*, 1991, pp. 276-286.

[2] D.R. Kaeli and P.G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proc. 18th Int'l Symp. Computer Architecture*, 1991, pp. 34-42.

[3] Tse-Yu Yeh, "Two-Level Adaptive Training Branch Prediction," Tech. Report, University of Michigan, 1991.

[4] Motorola Inc., "M88100 User's Manual," Phoenix, Arizona, Mar. 13, 1989.

[5] W.W. Hwu, T.M.Conte, and P.P.Chang, "Comparing Software and Hardware Schemes for Reducing the Cost of Branches," *Proc. 16th Int'l Symp. Computer Architecture*, May 1989.

[6] N.P. Jouppi and D. Wall, "Available Instruction-Level Parallelism for Superscalar and Super-pipelined Machines," *Proc. 3rd Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 1989, pp. 272-282.

[7] D.J. Lila, "Reducing the Branch Penalty in Pipelined Processors," *Computer*, July 1988, pp. 47-55.

[8] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-order Execution Machines," *IEEE Trans. Computers,* Dec. 1987, pp. 1496-1514.

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

[10] J.A. DeRosa and H.M. Levy, "An Evaluation of Branch Architectures," *Proc. 11th Int'l Symp. Computer Architecture,* 1987, pp.1-16.

[11] D.R. Ditzel and H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," *Proc. 13th Int'l Symp. Computer Architecture,* 1987, pp. 2-9.

[12] S. McFarling and J. Hennessy, "Reducing Cost of Branches," *Proc. 13th Int'l Symp. Computer Architecture,* 1986, pp. 396-403.

[13] J. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Jan. 1984, pp. 6-22.

[14] T.R. Gross and J. Hennessy, "Optimizing Delayed Branches," *Proc. 15th Ann. Workshop on Microprogramming,* 1982, pp. 114-120.

[15] D.A. Patterson and C.H. Sequin, "RISC-I: A Reduced Instruction Set VLSI Computer," *Proc. 8th Int'l Symp. Computer Architecture,* 1981, pp.443-458.

[16] J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. 8th Int'l Symp. Computer Architecture,* 1981, pp. 443-458.

[17] L.E. Shar and E.S. Davidson, "A Multiminiprocessor System Implemented Through Pipelining," *Computer*, Feb. 1974, pp.42-51.

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