BRANCH PREDICTION USING LARGE SELF HISTORY

John D. Johnson

Technical Report No. CSL-TR-92-553

December 1992

The work described herein was supported in part by equipment donation from Hewlett-Packard Company and by facilities supplied under NASA grant NAGW 419

BRANCH PREDICTION USING LARGE SELF HISTORY

by

John D. Johnson

Technical Report No. CSL-TR-92-553

December 1992

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

St anford University

Stanford, California 94305-4055

Abstract

Branch prediction is the main method of providing speculative opportunities for new high performance processors, therefore the accuracy of branch prediction is becoming very important. Motivated by this desire to achieve high levels of branch prediction, this study examines methods of using up to 24 bits branch direction history to determine the probable outcome of the next execution of a conditional branch. Using profiling to train a prediction logic function achieves an average branch prediction accuracy of up to 96.9% for the six benchmarks used in this study.

Key Words and Phrases: Branch prediction, Trained branch prediction, Adaptive branch prediction

Copyright © 1992
by
John D. Johnson

Contents

1 Introduction

High performance processors are increasing the amount of speculative work they perform in order to improve instruction-level parallelism they discover. Branch prediction is the main method of providing speculative opportunities for a processor, therefore the accuracy of branch prediction is becoming very important. This study investigates using moderately large self history of conditional branches to predict the next outcomes of branches.

Previous investigations into branch prediction have been divided into two major categories: static and dynamic branch prediction [Smi81]. Static branch prediction is performed at compile time and uses static code analysis to either insert branch prediction hints into the emitted opcodes, or to reorganize the emit ted code so the branch prediction implemented by the targeted machine's hardware is more effective. Dynamic branch prediction uses run time information when predicting conditional branch outcomes. This information may be the previous history of a branch itself or it may also include the outcomes of other branches near the branch to be predicted [SR92].

A category of branch prediction that falls between static and dynamic is branch profiling. This method uses branch history information gathered during one run of a program to improve the branch prediction accuracy during following runs. Improving the accuracy may be done statically by creating a compiler that uses the collected profile information to modify its static prediction for individual branches. As discussed in section 3, profile results may also be used to modify how the machine's hardware utilizes dynamic run-time history for branch prediction.

Most previous work in branch prediction used only a limited amount of a branch's history to predict its next outcome. A common approach is to predict a given branch will take the same directions as it took last time, through the use of a "Branch Target Buffer" [Los82]. While a Branch Target Buffer has advantages for pipelined implementations in that it can provide the predicted target address very early in the instruction fetch sequence, it always predicts a branch will be to the same address as its previous execution. This is equivalent to only 1 bit of branch direction history.

Lee and Smith [LS84] use profiling to build a distribution table of only five consecutive execution of conditional branches. They also report that through the use of nonuniform history retention, two bits for storing the state of a state machine branch predictor performs about as well as a five bit taken/not taken history sequence. Yeh and Patt report a few results for histories of 6 to 12 bits [YP91] and 6 to 18 bits [YP92] for the case of their "Two-Level Adaptive Branch Prediction".

To achieve high prediction rates for the looping behavior found in some programs requires a moderately large amount of self history. Assume a conditional branch is used to implement a loop that repeats for 9 times then exits on the tenth time. Any scheme that does not keep track of at least ten consecutive executions is limited to a 90% prediction rate.

Motivated by the desire to achieve a high level of branch prediction, this study examines possible methods of using up to 24 bits of branch direction history for determining the probable next outcome of a conditional branch. Figure 1 diagrams the basic structure studied. For each conditional branch in the compiled code, a shift register of n bits is allocated. Executing a branch causes a taken/not taken bit to be shifted into the shift registers associated with the branch. Each shift register thus contains a direction history for the last n executions of the associated conditional branch.

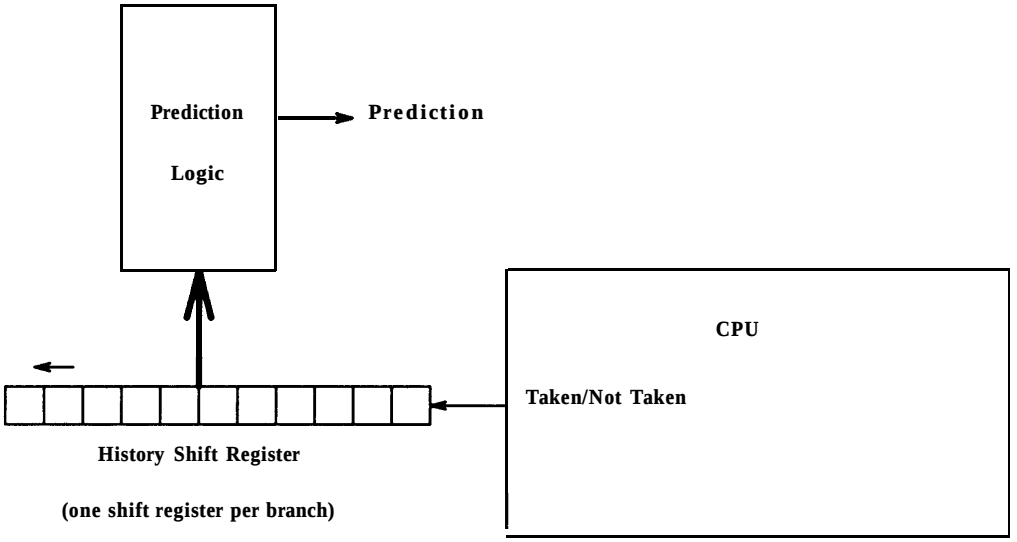

Figure 1: History Shift Register Block Diagram

The parallel output of the each shift register is transmitted to a logic block for predicting branch direction. The function implemented by this prediction logic is the main variant considered by this study. The simplest function studied is a population count over the input bits from the history shift register. This function predicts a branch is taken if the majority of the previous executions of this branch were taken. The next step in complexity for this function is to augment the population count with the patterns generated by fixed size loops. For example, for 14 bits of history, the 3 trip count loop pattern 00100100100100 would predict 1, even though there are fewer ones than zeros in this history.

Next, the idea of training is introduced. By profiling the execution of programs, it is possible to construct the function that best predicts branch outcome based upon the training samples. Training may be unbiased, that is, trained without using the application for which branch prediction is being performed. The training may instead be biased, that is, previous runs of the application itself are used during the training. The training can also be adaptive, as studied by Yeh and Patt. Adaptive training uses the current run to modify the prediction logic function while the application is running.

The reminder of this report is organized as follows. Section 2 presents the methodology, tools and benchmarks used during this analysis. Section 3 presents and discusses the branch prediction accuracy for the various prediction functions. Finally, section 4 offers concluding remarks and future directions.

2 Methodology, Tools and Benchmarks

Trace driven simulation produces the results of this study. Figure 2 presents the overall methodology. A benchmark is first compiled and optimized using the standard DEC Ultrix 4.2 C compiler to emit Ucode intermediate language [Nye82]. This Ucode is then delivered to an instrumentation program that inserts additional Ucode at basic block boundaries. The instrumented benchmark's

Ucode is assembled, linked and executed. Executing the benchmark emits a basic block trace while the benchmark runs, which is written to a file. Although not shown by figure 2, the instrumentation program also generates static code information, such as the type of branch at the end of each basic block, for use by the branch prediction simulator.

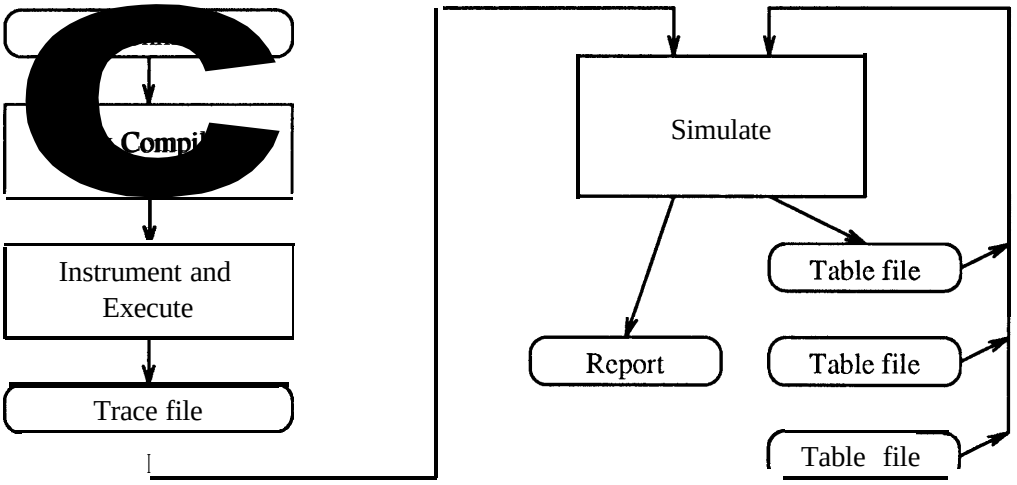

Figure 2: Tools Block Diagram

Once a trace file exists it can be used by the branch prediction simulator. One function performed by this simulator is producing the table files used for training the prediction logic. In this mode, the simulator reads a trace file and uses the history shift registers to index into a table of counters. During this process 24 separate tables are maintained in parallel, one for each history shift register length that will be used during prediction accuracy simulations. Every time a conditional branch is executed, the counter selected by each length of history shift register is incremented if the branch is taken, or decremented if it is not. After the the trace is consumed these tables are written out to a table file. Later, one or more of these tables files can be used as training input to the simulator.

Simulations using more than 24 bits of branch history were not performed because the real memory requirements exceed the memory available on the workstation executing the simulations. For the case of 24 bits of history, roughly one million counters are incremented and decremented during the training run. A hash table is used to store and identify the counters, and each entry required 8 bytes if the count stayed between -127 and 127, or 12 bytes if it exceeded this range. All lengths of direction history are simulated in parallel, so the memory requirements become large, and the simulation becomes very slow when limited to the 64 Mbytes of real memory available on the workstation.

Results presented in this report are generated by the branch prediction simulator. For the cases when the prediction logic is untrained the simulator is instructed to generate the prediction logic function, read a trace file and produce a report. For the cases when the prediction is trained, the simulator reads one or more training tables before starting simulation. Which tables are read by the simulator determines whether the training is biased or unbiased. No table files are used for adaptive branch prediction simulations, however the adaptive logic state is initialized to predict the same as the simple loop logic case.

Benchmark	Description
compress	Reduces the size of a file using adaptive Lempel-Ziv coding – compressing a 150KB tar file.
espresso	Boolean expression minimizer – reducing a 14-bit input, 8-bit output PLA matrix.
fft	Fast Fourier transform – 1024x1024 2-D fft
gcc1	Gnu C compiler version 1.36 – compiling (and optimizing) to assembly code a 1500 line C program.
spice3	Circuit simulator – simulation of a Schottky TTL edge-triggered register.
tex	Document preparation system – formatting of a 14 page technical report.

Table 1: Description of the Benchmarks

Benchmark	Static Instructions	Static Branches	Dynamic Instructions	Dynamic Branches
compress	1690	9.94%	13252875	15.95%
espresso	24183	7.83%	153611785	14.92%
fft	364	4.40%	6692459	5.24%
gcc1	135069	10.90%	42774378	14.72%
spice3	117339	9.03%	154030044	17.48%
tex	40253	7.40%	75865081	11.12%

Table 2: Benchmark Instruction Counts Branch Percentages

Six benchmarks were selected for this study and Table 1 provides a brief description of each one. The benchmarks were selected to mimic an execution profile similar to a typical workstation environment. Five of the benchmark programs are also used in the SPEC benchmark suite, although the version of the programs and input data sets used here are not the same as those used in the SPEC suite. All the benchmarks are written in the C language.

Table 2 presents both the static and dynamic instruction count for the benchmarks, along with the percentage of the instructions that are conditional branches. The tracing methodology does not support tracing run time library code nor system code, so the numbers presented are just for the compiled C code.

The f f t benchmark is somewhat different from the other five in that is has a much lower percentage of conditional branches. This is due to its structure being numerical or matrix code style, that is, almost all control flow is loops over array data structures. Later it is reported this type of control leads to very good branch prediction, and this benchmark is included to demonstrate this behavior.

The other five benchmarks have a higher percentage of conditional branches and much more varied control structures. They execute many IF statements, have variable length loops and chase link lists. Spice3 is considered a floating point program while the other four in this set are considered integer programs.

3 Accuracy of the Prediction Functions

This section presents simulation results for the various prediction logic functions. Results are presented in graphical form, with each benchmark plotted on its own graph. High variance between the benchmarks motivated this arrangement of a separate graph for each benchmark. If averaging was brought into effect it would mask the fact that benchmarks do not behave uniformly to changes in the prediction function.

All the graphs have the same axis and scale. The horizontal axis is the number of branch outcomes recorded by the history shift register, ranging from 1 branch to 24 branches. The vertical axis is the conditional branch prediction accuracy, expressed as the percentage of conditional branches that are correctly predicted. Note the all the graphs start at 75% correct prediction as almost all of the prediction functions achieve better than this accuracy. When branch predictions rates fall below 75% they are not plotted.

To make it easy to compare changes in accuracy between one prediction logic and the next, most of the set of graphs plot both the results for the logic being considered and the accuracy of the previous function considered.

3.1 Optimal Static Branch Prediction

Optimal static branch prediction is used as a baseline for the various dynamic branch prediction methods presented by this report. Static prediction requires a single predicted direction for each conditional branch in a program's code. Optimal static branch prediction is achieved when each predicted branch direction is the direction each branch selects the most frequently during the dynamic execution of a program.

Optimal static branch prediction is easily simulated when a dynamic trace is available. A pre-pass examining the trace file to determine the majority direction of each conditional branch during the entire execution of the program allows the optimal static prediction directions to be determined. Optimal static prediction achieves better accuracy than a compiler can be expected to achieve as examining the trace file provides knowledge that is available only after the input data set is known. This knowledge is not available to a compiler. As the main emphasis of this report is on dynamic branch prediction, this report does not discuss how closely a compiler can match optimal static branch prediction or how the predicted branch direction should be encoded into the instructions.

Table 3 presents the branch prediction accuracies for optimal static branch prediction. The "Average Accuracy" column is the average over all forward and backward conditional branches executed

Benchmark	Average Accuracy	Forward Branch Accuracy	Backwards Branch Accuracy
compress	85.04%	83.57%	97.59%
espresso	85.62%	84.69%	87.32%
fft	93.98%	60.32%	94.02%
gcc1	89.05%	89.51%	87.59%
spice3	86.04%	90.33%	79.83%
tex	86.77%	86.52%	88.09%
Average	87.75%	82.49%	89.07%

Table 3: Optimal Static Branch Prediction Accuracy

by each benchmark. The forward and backwards columns are the averages for just the forward and backwards conditional branches respectively. The ratio of the forwards to backwards determines how much each direction effects the average accuracy. Backwards branches usually have a better prediction accuracy than forwards branches, but spice3 is an exception. Backwards branch prediction if often more accurate than forward branch prediction because backwards branches frequently implement loops that are repeated many times.

3.2 Population Count Prediction Logic

The first prediction logic function studied is a population count function over the history bits in a branch's shift register. This function's accuracy graphs for each benchmark is presented by figure 3. The population count function predicts a given branch will be taken if the majority of the previous executions of this branches were taken. In the case when exactly the same number of taken and not taken branches occur in the history shift register, the tie is resolved by predicting the branch will be in the same direction as its previous execution. For example, the 6 bit history "111000" would predict "0" while the history "000111" would predict "1".

This population count function has the advantage of being easy to understand and it would be relativity easy to implement in hardware. It has the disadvantage of not performing very well. Performance is gauged by comparing the prediction accuracy of this function to the accuracy of optimal static branch prediction, which is also plotted on the graphs in figure 3. Since static branch prediction uses no direction history, the accuracy of static prediction is plotted as horizontal lines in the figure.

Examining the graphs of figure 3 reveals that population count logic over 5 bits of branch direction history achieves about the same accuracy as the optimal static prediction. Averaged over the six benchmarks, 5 bit population count prediction achieves an 87.5% accuracy while the optimal static predictions achieves 87.7% accuracy. Another feature of figure 3 worth noting is that two bits of history does not improve the performance over 1 bit of history. Both 1 and 2 population count prediction have an average accuracy of 84.4%. As shown by Lee and Smith [LS84], a two bit shift register is not the best method of allotting two bits of storage for predicting branches. They show that relevant history for branch prediction can be more effectively captured by a two bit state machine than by a two bit shift register.

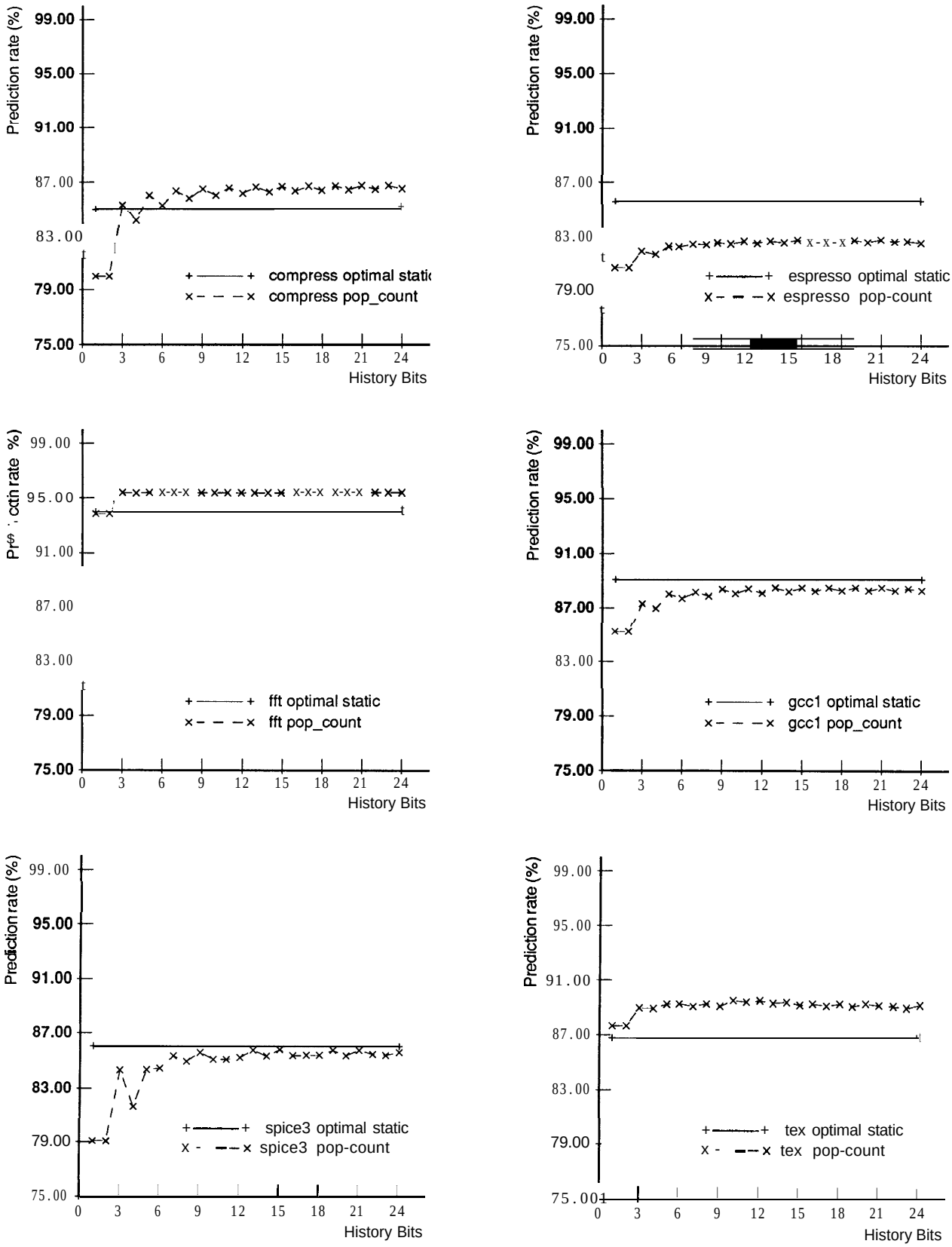

Figure 3: Prediction Accuracy for Population Count Prediction Logic

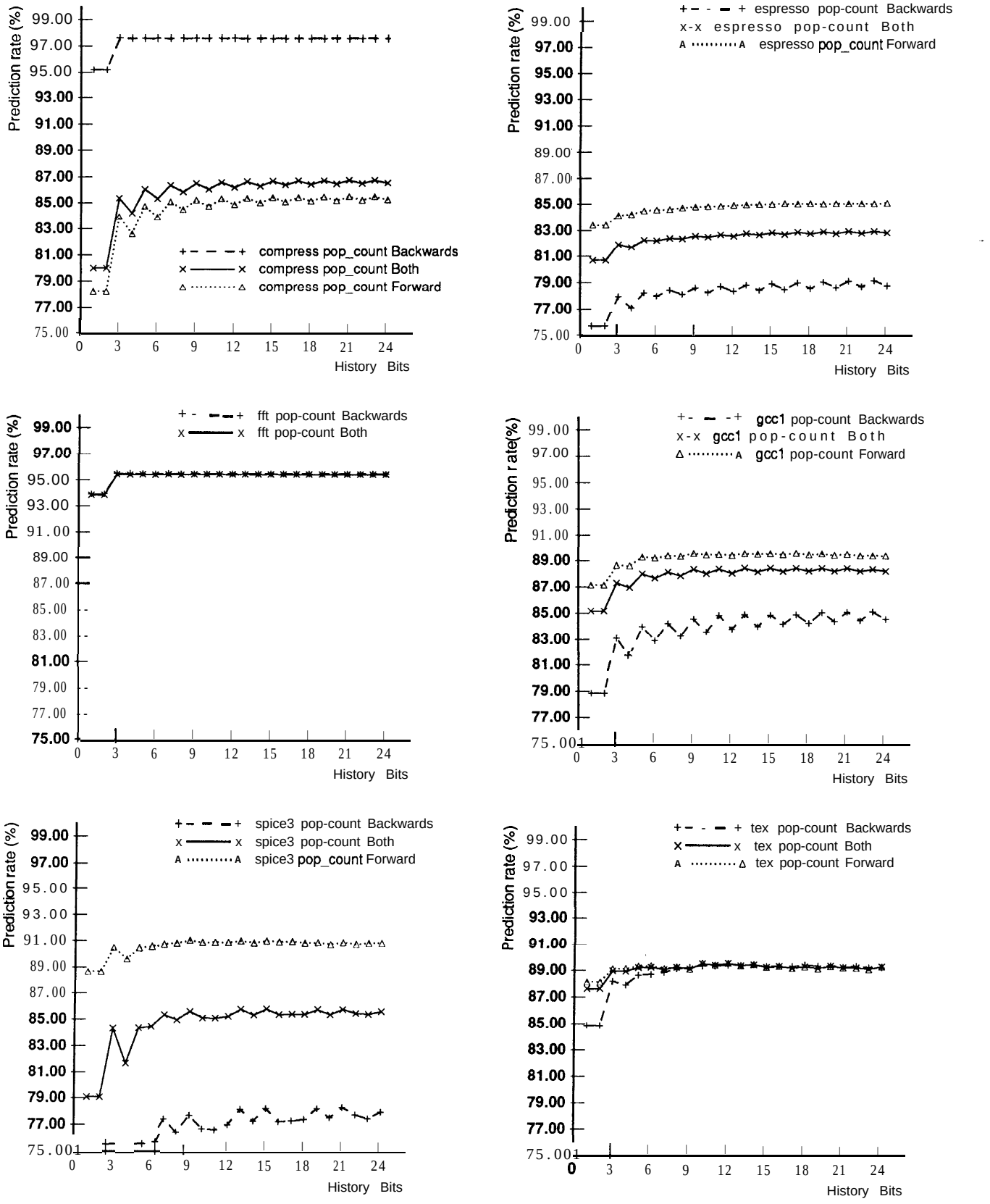

Figure 4: Forwards and Backwards Accuracy for Population Count Prediction Logic

Benchmark	Backward Branches	Forward Branches	Bwd/Fwd Ratio
compress	221354	1892530	0.12
espresso	8122589	14796726	0.55
fft	350152	378	926.33
gcc1	1505854	4790373	0.31
spice3	10993330	15929770	0.69
tex	1320804	7112983	0.19

Table 4: Backward to Forward Branch Ratios

Figure 3 reveals there is little accuracy to be gained by using a large self history with the population count prediction logic. Almost all of the benchmarks have essentially no change in prediction accuracy for 5 to 24 bits of branch history. This is expected as the population count function is not able to identify repeating patterns occurring within the history shift registers.

Additional insight as to how this branch prediction strategy behaves is gained by plotting the forward and backward branches separately. Figure 4 plots the branch prediction accuracy in this manner, along with the average prediction rate that was plotted by figure 3. Examining figure 4 illustrates that for most benchmarks, prediction accuracy between forward and backward branches is quite different. For example, compress achieves a 97.5% accuracy for backwards branches, while it achieves only a 85% accuracy for forward branches. The `f ft` benchmark forward accuracy is always below 75% and is not plotted because of the scale of the graph. However, `fft` has a very small number of forward branches compared to backward branches, so the average prediction rate is almost that of the backwards branch prediction rate. The relative frequency of forward and backwards branches determines whether the average prediction accuracy is closer to the forward or backwards branch prediction accuracy.

Domination by backward or forward branches is not uniform between the benchmarks, and their prediction rate is even more varied. Table 4 provides the dynamic ratio of backward to forward branches and exhibits that backwards branches are dominant only for the `fft` benchmark. As for prediction accuracy, backward branches can be predicted more accurately in compress and `fft`. Forwards branches are predicted more accurately for espresso, `gcc1` and `spice3`. Forward and backward branch prediction accuracy is about equal for tex.

3.3 Simple Static Loop Prediction Logic

Correctly predicting the branches implementing fixed count loops repeating fewer times than the number of maintained history bits can be achieved by replacing some of prediction logic terms generated by the population count function. Figure 5 presents the results for a prediction logic function called *simple-table*. This functions is the same as the population count function, except that patterns for fixed size loops take precedence over the population count function.

Very few of population count function's truth table entries need to be changed when implementing the *simple-table* function. Since the population count function correctly predicts a loop's branch

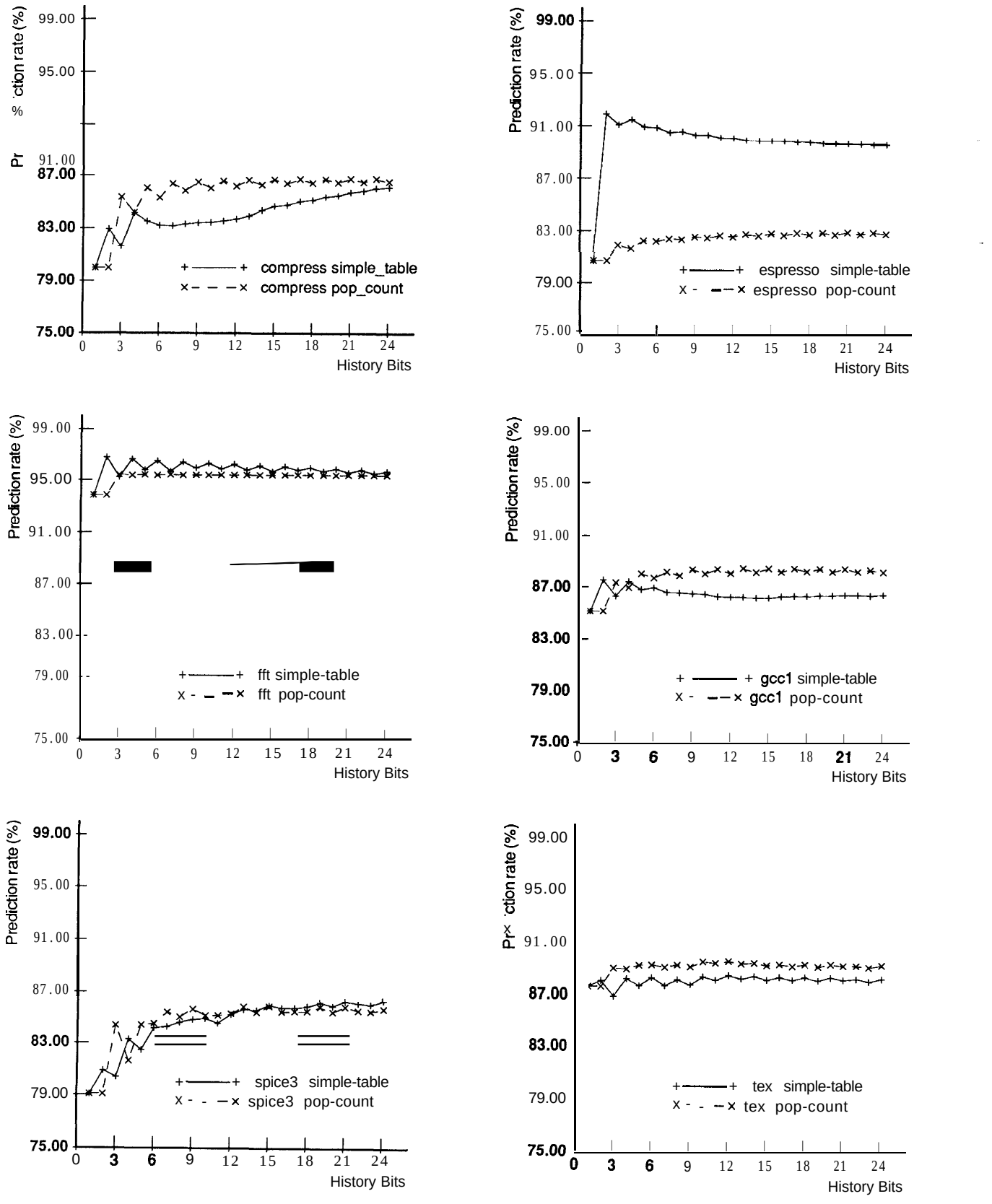

Figure 5: Prediction Accuracy for Simple Static Loop Prediction Logic

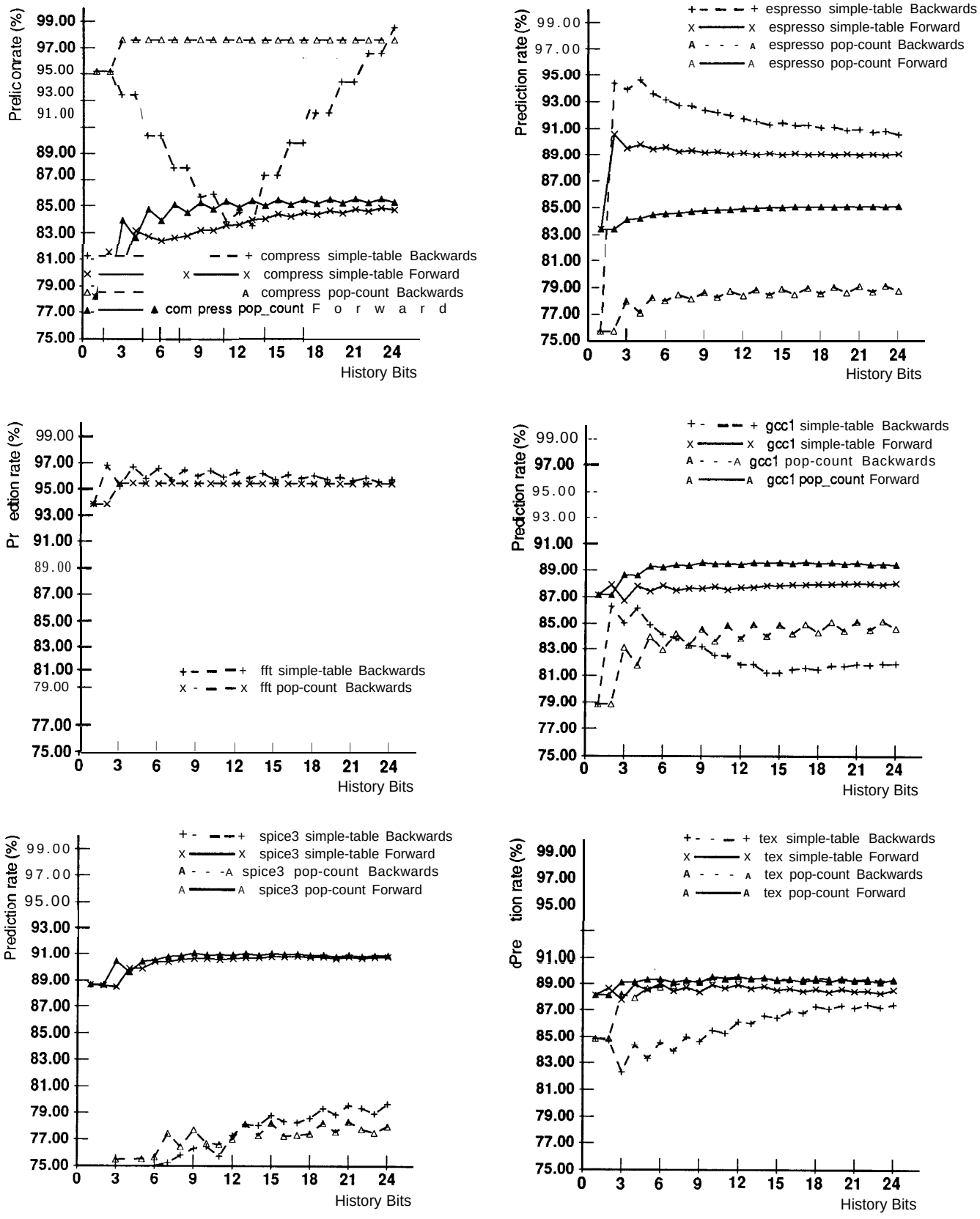

Figure 6: Forwards and Backwards Accuracy for Simple Static Loop Prediction Logic

Loop Length	History Pattern	Prediction
5	01111	0
4	10111	0
3	11011	0
2	10101	0
2	01010	1

Figure 7: History Length 5 Differences for "Simple Table"

for all cases except for the the last transversal, only one truth table entry needs to be changed for each length of loop that fits within the maintained history. For example, figure 7 shows the 5 entries that do not match the population count function for a 5 bit branch direction history. The 5 bit history table has 5 entry changes, the 6 bit table has 6 entry changes, and so forth. The 24 bit history table has only 24 entry changes, which is only 0.0001% of the total entries in the table.

Augmenting the population count function with fixed loop trip count predictions achieved a healthy 9% prediction accuracy improvement for the espresso benchmark, but surprisingly achieves only very small increases or decreases in prediction accuracy for the other benchmarks. Compress has the most notable decrease, a decrease of about 3% in the range of 5 to 15 bits of branch history.

Fixed loop trip count prediction performs well for espresso, improving accuracy from about 83% to about 91%. Furthermore, only 2 bits of history are need to attain this improvement. Scrutinizing espresso reveals that this program has many short loops with fixed loop counts, thereby corresponding well to this prediction logic.

Fixed loop count prediction does not perform well for the other benchmarks. Examining the behavior of forward and backward branches conveys some the reasons for the prediction accuracy decreases. Figure 6 plots the forward and backward branch prediction accuracy for both population count prediction and fixed loop prediction. One of the most prominent features of this figure is the dip in the backward branch prediction accuracy of compress. The reason is compress has only 6 static backward branches and only 2 of them account for more than 98% of the backwards dynamic branches. One of these two branches account for 32% of the backward branches and is taken 65,145 out of 70,472 executions, or is taken 92% percent of the time. This branch implements a while loop and does not have a fixed loop count, so occasionally predicting this branch is not taken reduces the prediction accuracy as compared to predicting it is always taken.

Overall, adding the fixed loop predictions does not achieve a good average improvement in prediction accuracy. It achieves limited success for backwards branches, and attains almost no improvement for forward branches. Forward branches make up the majority of the conditional branches for most of the benchmarks, so there is a need for a prediction function that address their behavior. The next section formulates a method for finding an improved branch prediction function.

3.4 Majority Trained Prediction Logic

This section introduces a training method to construct the prediction logic function. Through the use of profiling it is possible gather samples describing which patterns in the branch prediction logic are utilized and how successful each pattern is at predicting branches. These samples are called training samples.

There are many possible methods of combining the training samples from various benchmark runs into a branch prediction logic function. In this section, the goal is to combine the samples in a manner that does not result in very complex function, and a function that is not obviously biased towards a given execution of a given benchmark. The case of training with a bias towards a given benchmark is discussed in the next section.

The method chosen for combining training samples in this section is, for each branch history pattern, an average over the benchmarks provided at least 5 of the 6 benchmarks has the pattern present in their training samples. For example, assume 6 bits of history. Also assume the branch history pattern 001001 occurs in 5 of the 6 benchmark training samples and that 2 of the benchmark's samples report the next use of this pattern should prediction taken while 3 report that the next used should prediction not taken. This 001001 pattern qualifies for including in the prediction logic as it occurred in at least 5 of the benchmark run. The value of the the prediction logic for this input pattern is "not taken" as more of the benchmarks report not taken for this particular pattern.

This methods of training is called majority training as there must be $5/6$ majority of the benchmark voting on a pattern before the pattern can be enter into the branch prediction logic. If the $5/6$ majority is not met for a given pattern, the population count function is used on this pattern predict the branch outcome. By requiring a $5/6$ majority of the benchmarks to participate in creation of a entry, only the most frequently encountered patterns will be included in the branch prediction logic, thereby limiting the complexity of this function.

Figure 8 shows the number of terms in the branch prediction functions that are different from a population count prediction function. In the figure, the diagonal line is the the total number of entries for a given number of branch history bits. The shaded area is the number of entries that are different from the population count function. As the number of branch history bits becomes large, the number of entries that are different from the population count function drops off. This is because of the $5/6$ majority voting requirement. When a large number of history bits is used there are many more different possible patterns so there are fewer that meet the $5/6$ majority requirement.

Figure 9 plots the branch prediction accuracy with this $5/6$ majority trained prediction logic, along with last section's *simple_table* prediction accuracy for comparison. There is rapid improvement in prediction accuracy for the range of 1 to 6 bits. At 1 bit the average accuracy is 84.4% and by 6 bits it has increased to 91.3%. At 6 bits of history, there are 16 entries in the branch prediction logic that are different from the population count function and these entries change the average branch prediction rate from 88.3% to 91.3%, and increase of 3%.

From 6 to up to about 16 bits of branch history, prediction accuracy usually gradually improves. At 16 bits the average accuracy is 92.8%, up 1.3% from the 6 bit accuracy. For 16 bits, there are

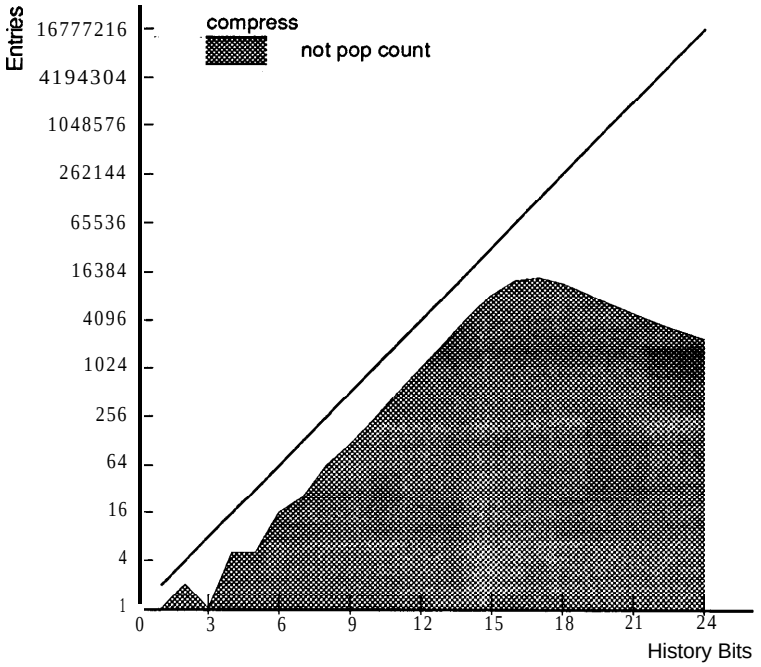

Figure 8: Number of Entries Not Matching Population Count for Majority Training

12,835 entries in the branch prediction logic that are different the population count function, which is 19.6% of the 65,536 total entries. These entries change the average prediction accuracy at 16 bits of history from 88.5% for the population count logic to 92.8% for the majority trained logic, and increase of 4.3%.

Above 16 bits is the range where the 5/6 majority rule starts restricting the number of trained entries in the branch prediction logic function. Since training in this area is weak, the branch prediction accuracy tends to fall off slowly, decreasing down to an average accuracy of 91.8% at 24 bits.

In summation, training is useful in creating a prediction logic function, especially in the range of 6 bits of branch history. At 16 bits of branch history the 12,000 additional terms in the prediction logic function may be somewhat expensive to implement and achieves an average accuracy increase of only 1.3% over 6 bits of branch direction so the worthiness of implementing large fixed prediction function logic is questionable. The next section introduces the idea of a prediction logic function that can be customized for a particular program in order to allow a large branch direction history to be more fully exploited.

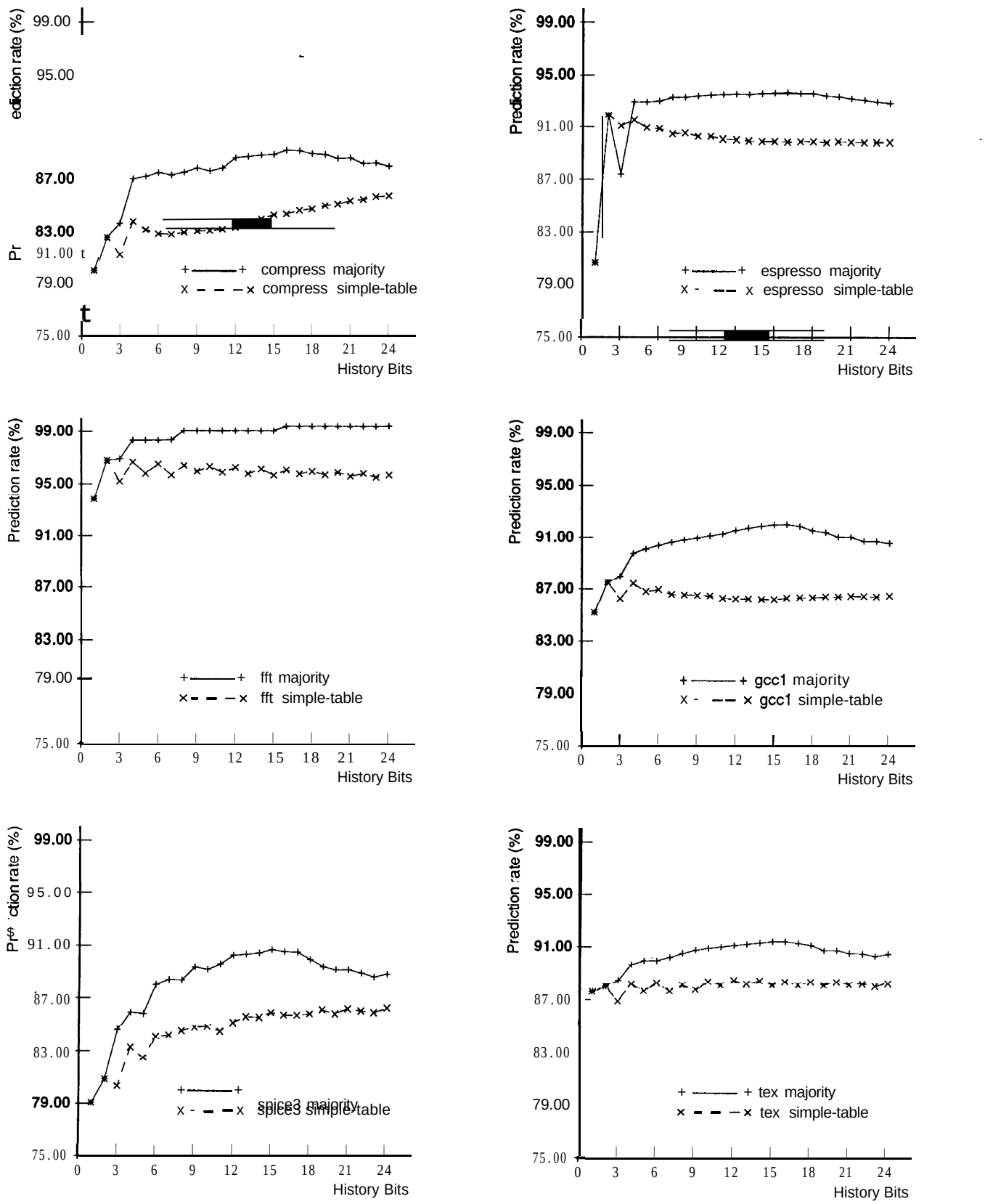

Figure 9: Prediction Accuracy for Majority Trained Prediction Logic

3.5 Self Trained Prediction Logic

All the previous sections assume the prediction logic function is determined at the time the machine is designed and thus can not depend upon the program being executed. This section relaxes this constraint and assumes the prediction logic function can be configured when a program is loaded. However, the prediction logic is still constrained so that the function does not change during the execution of programs.

The configurable prediction logic function could be implemented as a full look up table, with 2^n entries if n bits of branch directions history are maintained. Or it could be implemented as a population count function over the n bits of branch directions history along with an associative table of entries that are different from the population count function. Data is presented later in the section describing how many entries are used in a associative table specifying non-population count function entries.

Training the prediction logic function with the same trace as the one used for the prediction accuracy simulation realizes the best possible results for the shift register feeding predictions logic structure being studied. Figure 10 presents the accuracy results for this self training of the prediction logic, along with the accuracy of the majority trained logic of last section for comparison.

A striking feature of the self trained prediction logic graphs is after rapid improvement for the first few bits of branch history, the accuracy continues to improve almost linearly with the number of maintained history bits. At 5 bits of branch direction history an average accuracy of 91.0% is achieved, almost the same as the 90.4% achieved with majority training. At 24 bit of branch direction history the average accuracy is 96.9%, an improvement of more than 5% over the majority training case.

Part of the explanation for this continued improvement is in some sense the future is trained into the branch prediction logic. Because the same trace is used both for training and for simulation it is possible in effect to "know the future." In the unrealistic environment of knowing the future, it would be possible to store a list describing each direction of all branches taken and achieve 100% branch prediction. In the shift register indexing a table environment studied here, branches having very hard to predict behavior cause the history shift register to act somewhat like a hash function into a branch direction list. For large prediction logic functions there are enough entries in the function that the behavior of some branches is in effect stored in this function without interference from other branches.

Even though it is not usually possible to train prediction logic with the same program and data set used during the re-execution a program, the continued improvement of prediction accuracy seen here is an interesting result. It is likely there are many programs where changes in the input data sets will have only minor changes in the branch pattern, thereby making it possible to achieve very high prediction rates. Using this type of branch prediction logic training requires the prediction logic function be able to implement the large number of entries that occur.

Figure 11 plots the forwards, backwards and average prediction accuracy for the six benchmarks with self trained prediction logic. The behavior of the forward and backward branches is as expected. Both branch direction improve with increasing number of branch history bits and there

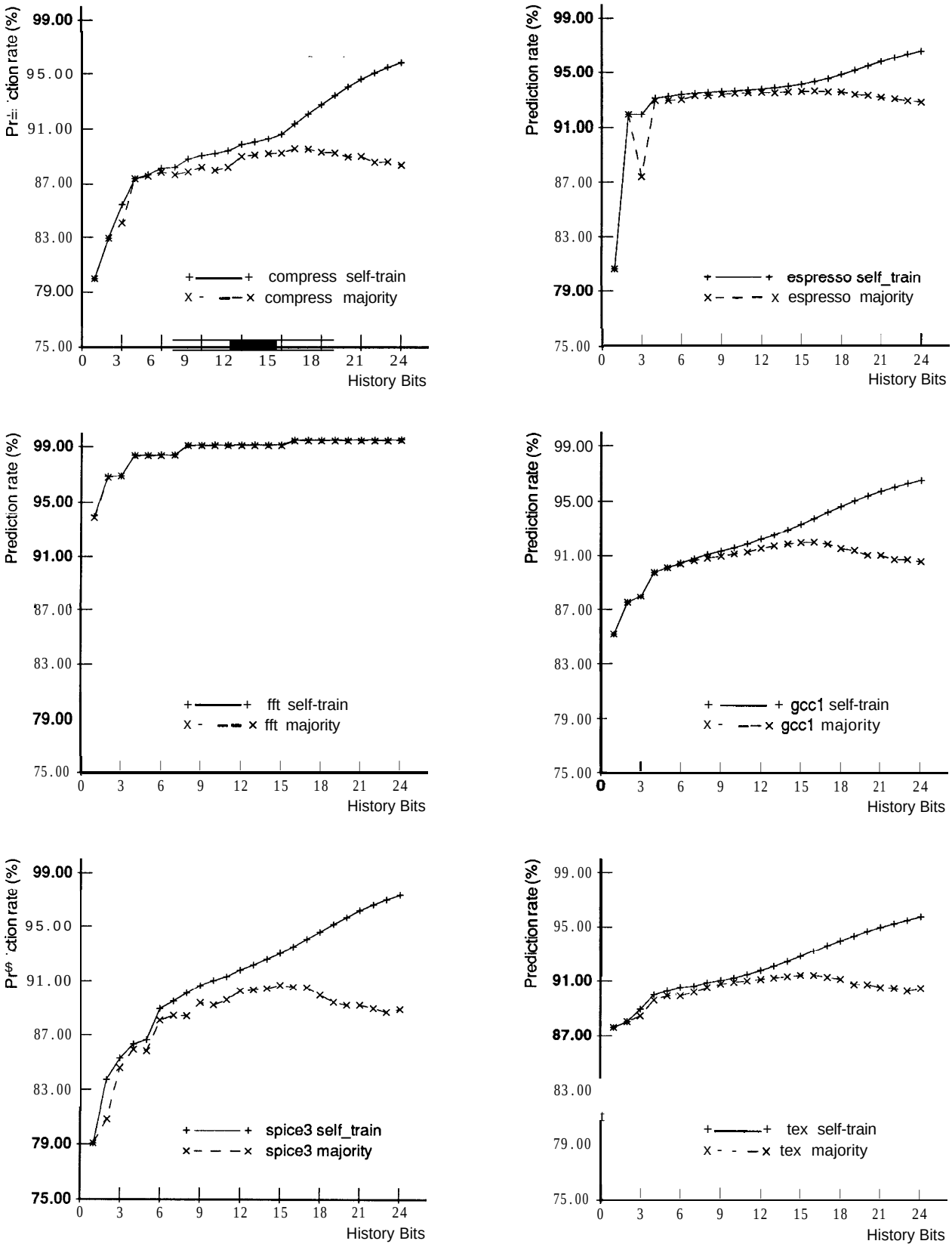

Figure 10: Prediction Accuracy for Self Trained Prediction Logic

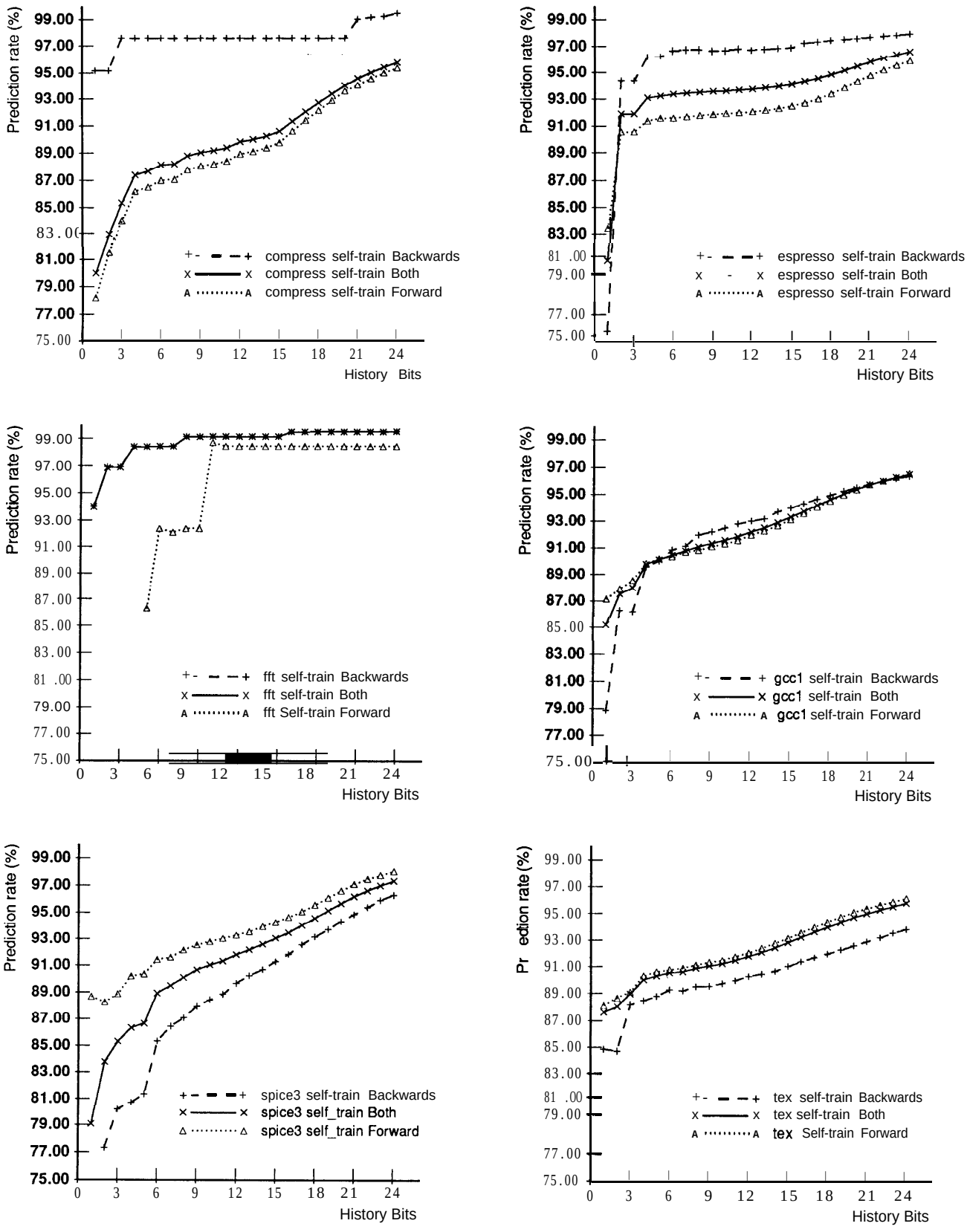

Figure 11: Forward/Backwards Accuracy for Self Trained Prediction Logic

are no peculiarities that are not explained in the previous sections.

Figure **12** presents for each benchmark the number of entries used in the prediction logic function. An entry is considered used if at some time during a run its value is present in the branch history shift register. The darkly shaded area of the graphs represents entries that are both used and differ from the population count, while the lightly shared area represents entries that are used and match the population count function.

For 1 to about 8 branch history bits, all the the possible entries of the prediction logic function are used. The **f ft** benchmark does not have a varied branch behavior so it does not require more than about 300 entries to achieve an accuracy of 99.5% correct prediction. The other benchmarks have a more varied branch history patterns and require many more entries in their prediction logic functions. These benchmarks have 100,000 to 400,000 entries not matching the population count function at 24 bits of branch history.

Note that in figure 12, the log scale obscures the fact that the number of used entries matching population count is almost always more than 2 times the number not matching population count. In contrast, figure 13 plots only the portion of used entries that match population count and the portion of used entries that do not match population count. These graphs present the same information as the graphs of figure 12, but plotted in a manner that demonstrates the amount of change needed to the population count function for achieving the reported prediction accuracies. For more that 8 bits of branch direction history, 20% to 40% of the used entries are different from the population count over the branch history bits.

In summary, these benchmark demonstrate it is possible to achieve very good prediction accuracies using large branch direction history through the use of profiling to train the prediction logic function. At 24 bits of branch direction history, these benchmarks have an average branch prediction accuracy of 96.9%. Achieving this accuracy requires several hundred thousand entries to differ from the population count prediction function and requires the training data set create branch behavior very similar to the behavior occurring during execution. The next section presents a method of removing the profiling requirement by replacing it with an adaptive learning technique.

3.6 Adaptive Prediction

Self training of the prediction logic achieves very good prediction branch prediction accuracy with large branch direction history but has the problematic requirement of profiling the application. In this section we present the branch prediction accuracy when profiling is replace by an adaptive learning technique. This adaptive technique is similar to the two-level adaptive branch prediction of Yeh and Patt [YP91] but this study extend their results to larger branch history shift registers and more state in the adaptive prediction logic.

Figure 14 presents a block diagram of this adaptive prediction method. Instead of a static prediction logic function, adaptive prediction uses an array of saturating up/down counters. A branch direction history shift register is used to index into this array and select one counter. If the value of counter is greater than or equal to zero the branch is predicted taken, else it is predicted not taken. Branch result information is also routed to the counter. After the branch outcome is known, the counter

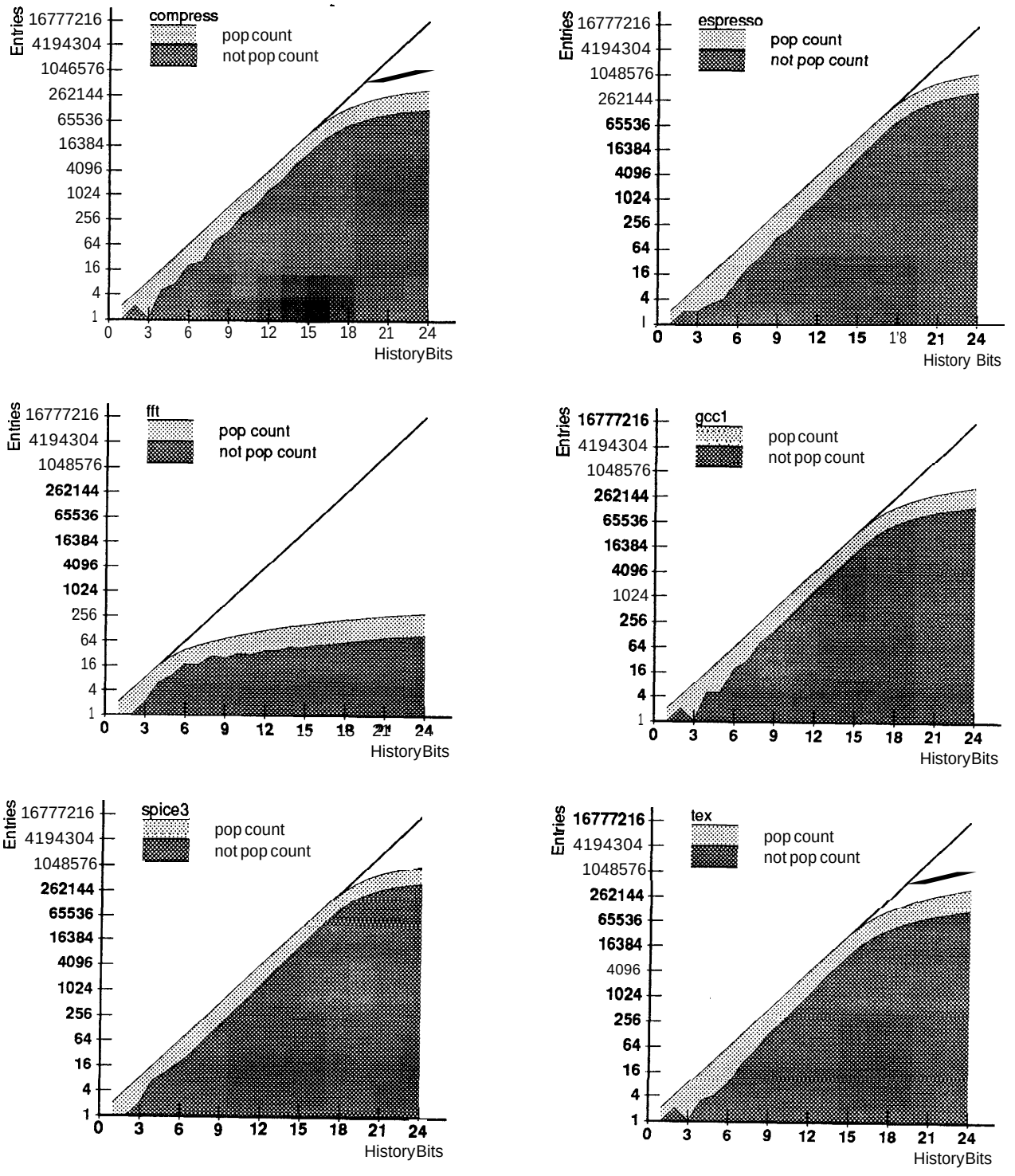

Figure 12: Table Entries Filled

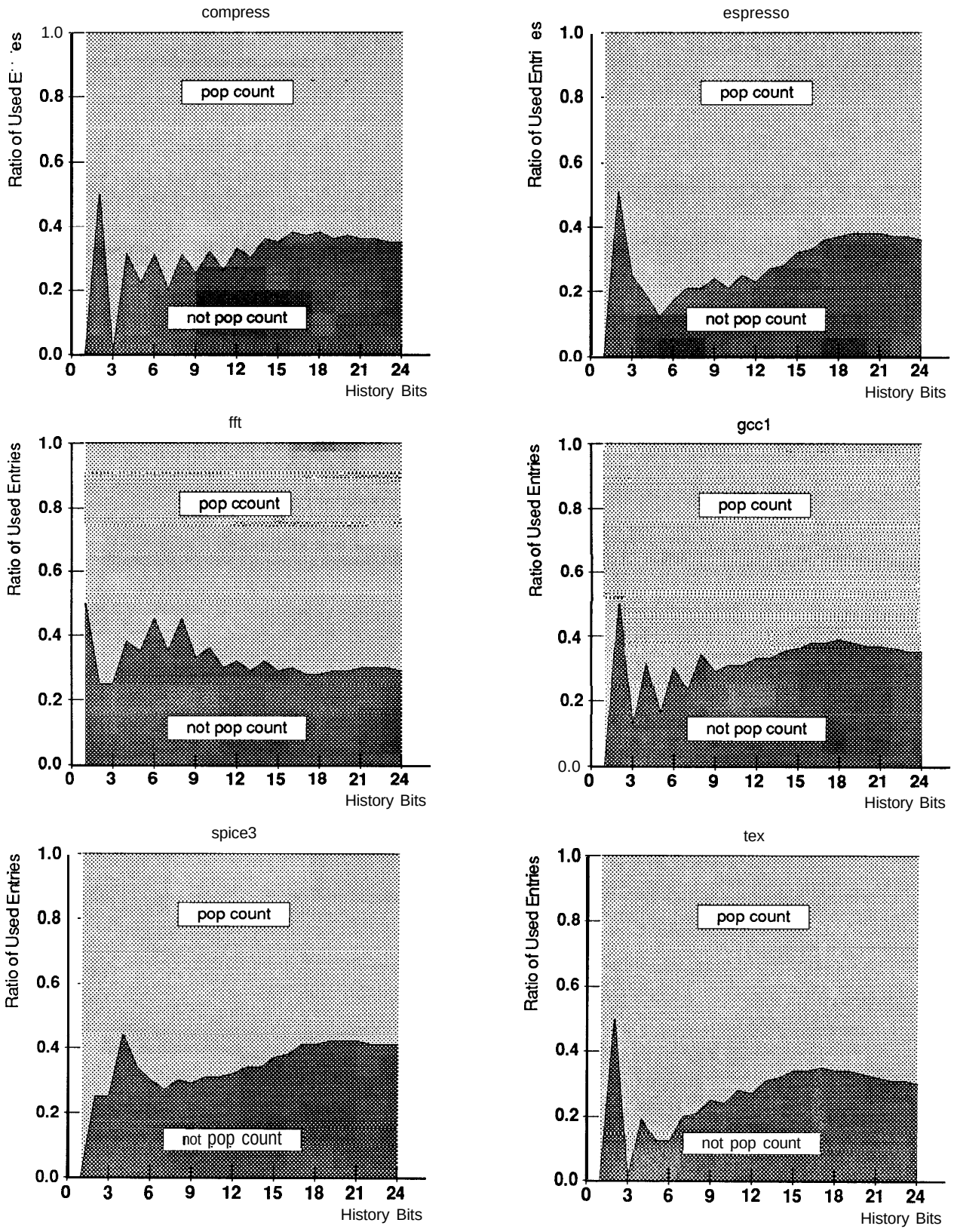

Figure 13: Ratios of Filled Entries

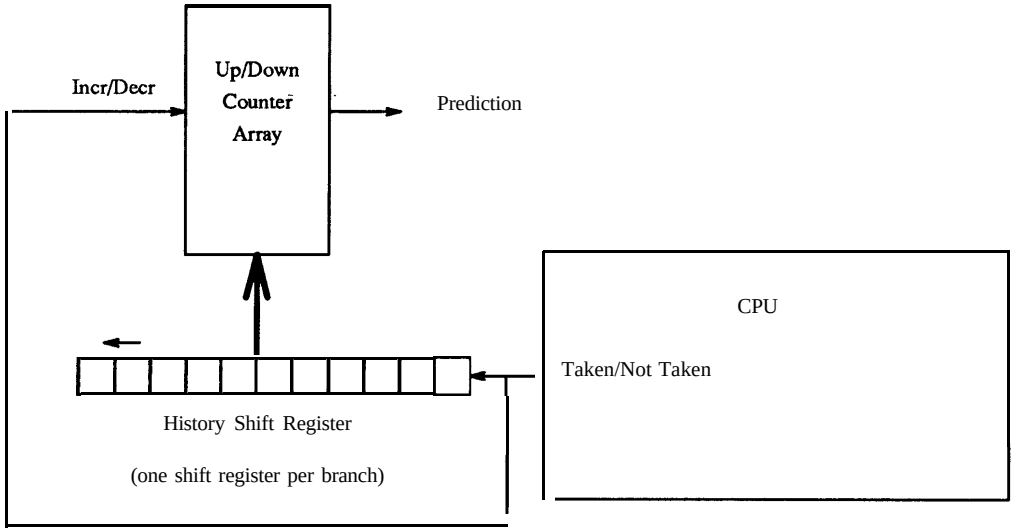

Figure 14: Adaptive Prediction Block Diagram

is incremented if the branch was taken, otherwise it is decremented. Saturating counters are used, so incrementing a counter that is already at its maximum value results in the counter remaining at its maximum value. Decrementing the minimum value also remains at the minimum value. Eight bit counters are used for the data presented in this report.

The motivation for this configuration is to adapt to patterns found in the branch history shift register. If a given history pattern almost always corresponds to a taken branch, then the up/down counter selected by this history pattern will increment up and saturate, solidly predicting a taken branch. If given history pattern correspond a near 50/50 percent chance of be correctly predicted, then the counter will stay near zero and the prediction will track short term variation in the average branch direction.

Figure 15 presents the prediction accuracy for adaptive branch prediction, along with self trained and 5/6 majority trained prediction logic. Averaged over the six benchmarks, the average branch prediction accuracy is 93.5% at 15 bits of branch direction history and 94.1% at 24 bits of history. Adaptive branch prediction accuracy is almost always between self trained and majority trained prediction logic.

The spice3 benchmark shows adaptive branch prediction accuracy improvements with the number of history bits all the way out to the maximum simulated length of 24 bits. For the `gcc1` and tex benchmarks, adaptive branch prediction accuracy improves with increasing branch history up to about 15 bits, but shows little improvement with larger numbers of history bits. For the other three benchmarks, adaptive branch prediction achieves only slightly better accuracy than majority trained prediction logic.

For a few benchmarks and a low number of branch history bits, adaptive branch prediction does slightly better than self trained prediction logic. For example, at 3 bits of branch history the compress benchmark shows a 85.9% accuracy for adaptive verse a 85.3% accuracy for self trained prediction logic, an improvement of 0.6%. This is due to the adaptive branch prediction's ability to track changes occurring while the program is executing. The non-adaptive prediction logic is fixed

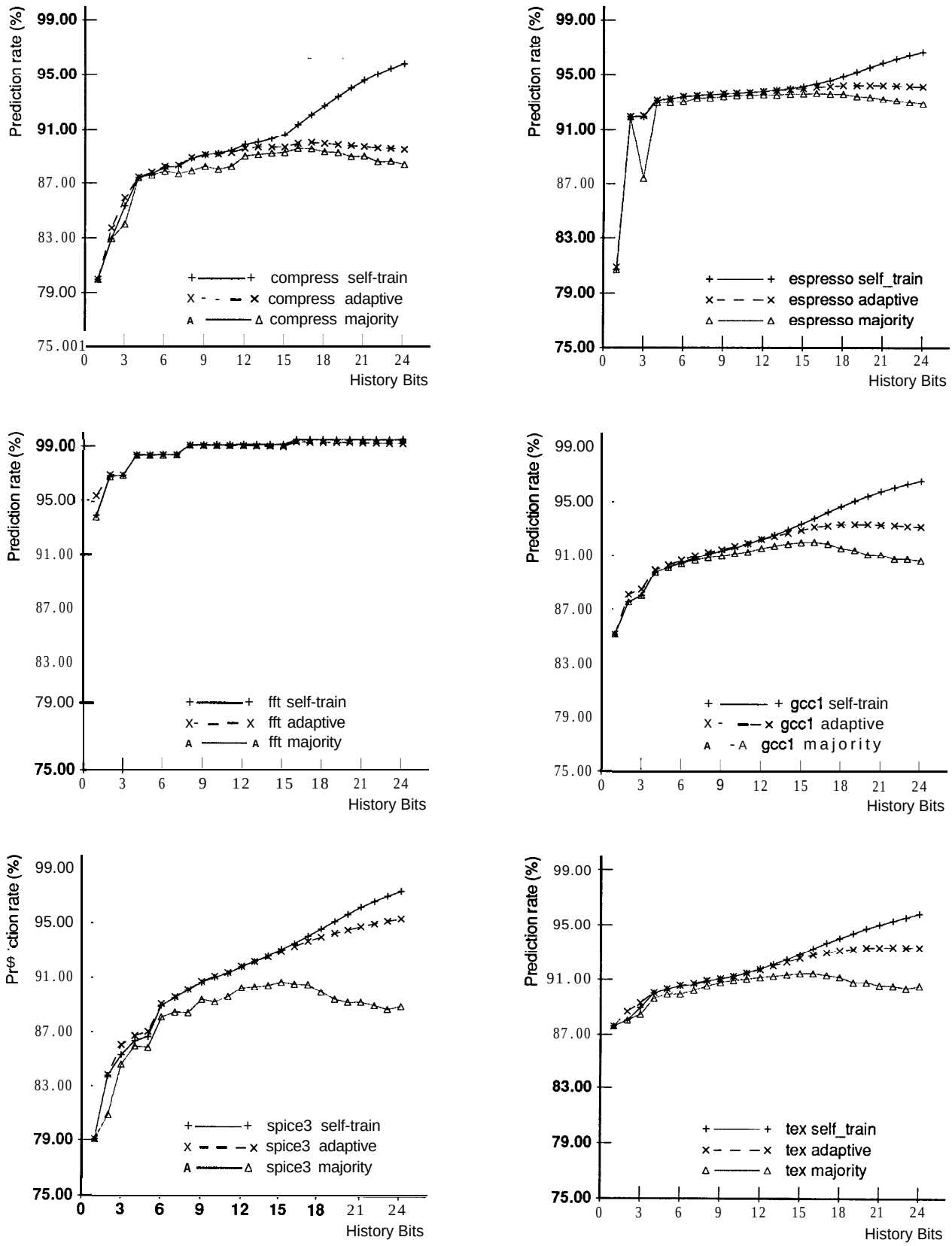

Figure 15: Prediction Accuracy for Adaptive Prediction

for the entire execution of the program. If a certain branch history pattern should predict "taken" during one phase of the program and "not taken" during another, then static training cannot achieve high accuracy for this time varying behavior. However, since adaptive branch prediction is seldom better than self trained prediction logic, it appears this type of time varying behavior of branch direct patterns is rare in these benchmarks.

Overall, the effectiveness of adaptive branch prediction varies widely for large branch direction history. It is very good for the spice benchmark, but offered little improvement over majority trained prediction logic for the compress benchmark. There are implementation complexities of the adaptive mechanism as it probably requires to both reading one counter and writing another during a single cycle, so further study is needed to determine if this is a cost-effective method of branch prediction.

4 Conclusions

This report presents the branch prediction accuracies achieved when maintaining for each conditional branch a direction history of up to 24 bits. Several methods for fabricating predictions based upon patterns found in the branch direction histories are presented. These methods range from a simple population count over the direction history bits to a self trained prediction logic to adaptive training.

Simple static prediction logic benefits very little from branch direction histories over about 3 bits. Complicated static prediction logic, as demonstrated by the 5/6 majority trained case, does not benefit from branch direction histories of more than 12 bits. No regular simple prediction logic function for achieving attractive branch prediction was found, and the trained complicated functions tend to have about 1/3 of their truth table values differing from that of a population count function.

Self training the prediction logic function can achieve very good branch prediction accuracy, provided the data set used for training models very similar branch behavior to that occurring during program execution. When the training and execution data sets are the same, the reported prediction accuracy increases almost linearly with the number of maintained branch direction history bits, up to an average prediction accuracy of 96.9% with 24 bits of branch direction history.

This study shows that large branch direction history can be effective in predicting branch outcome if self trained is used, however, additional study is required before this technique can be cost-effectively implemented in machines. Sensitivity to input data sets is another question that requires additional study.

The arrays used in this study to store the prediction logic function are large and sparse. This inefficiency points towards studying associative or set associative structures for implementing the prediction logic function. Not all terms in the prediction logic function contribute equally to the prediction accuracy, so cost-effective implementations may have methods of detecting the most important patterns found in the branch history shift register and only storing a limited number of these patterns. This and other implementation issues are attractive areas for future work.

Another area for future work in improving branch prediction accuracy is using additional information to a branch's self history. Correlation between nearby branches seems an attractive area to study, as well as correlation- between recently computed data values. Compiler help may be a possible method to identify the branch and data items that may be correlated to conditional branches outcome.

Future deep-pipelined and superscalar machines will depend on effective branch prediction. This study reports that improved branch prediction can be achieved when the application branch behavior is well identified through the use of self training and large branch direction history. This identifies a direction for continued study into methods of achieving the highly accurate branch prediction required by these future machines.

References

[Los82] J. J. Losq. Generalized History Table for Branch Prediction. *IBM Technical Disclosure Bull.,* 25(1):99–101, June 1982.

[LS84] J. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer,* 17(1):6–22, January 1984.

[Nye82] Peter Nye. U-code an intermediate language for pascal* and fortran. S-1 Document PAIL-8, Stanford University, May 1982.

[Smi81] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 10th Annual Symposium on Computer Architecture,* pages 135-148, Minneapolis Minnesota, May 1981. IEEE.

[SR92] Shieb-Tai Pan Kimming So and Joseph T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. *In Conference Proceedings, Architectural Support for Programming Languages and Operating Systems (ASPLOS V),* pages 76-84, October 1992.

[YP91] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture,* pages 51–61, November 1991.

[YP92] Tse-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Conference Proceedings, The 19th Annual Symposium* ***on Computer Architecture,*** pages 124-134, May 1992.