Classification-Based Hybrid Branch Prediction

A Thesis
In TCC 402

Presented to

The Faculty of the
School of Engineering And Applied Science
University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Electrical Engineering

By

Philo Juang

March 24, 2000

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

_____
Philo Juang

Approved _____
(Technical Advisor)
Professor Kevin Skadron

Approved _____
(TCC Advisor)
Professor Rosalyn Berne

Glossary of Terms

Aliasing – the situation in which two branches, through hashing, happen to index into the same entry in the branch predictor history tables

Branch – a programming construct in which the next instruction to be executed depends on some condition

Direction – whether the condition evaluates true (taken) or false (not taken)

Dynamic References – the number of times a branch is executed during the course of a running program

Path – the sequence of instructions following the execution of a branch

Pipelining – an implementation technique in whereby multiple instructions are overlapped in execution

Pollution – a situation in which some trivial branches mask or inhibit the overall program behavior conveyed by important, non-trivial branches

Profile – a statistical analysis of the behavior of a program

Speculative Execution – the processing of a sequence of instructions before it is known whether they are needed

Static Footprint – the number of branches appearing in the code of a program

Superscalar – processors which can issue more than one instruction per clock cycle

Abstract

One of the largest limiting factors in microprocessor instruction execution throughput is the existence of conditional branches in the instruction stream. Conditional branches disrupt the normal control flow of a program by introducing irregularities; instead of a natural, steady progression, conditional branches order the processor to "jump" to different places in the instruction stream. These irregularities force the processor to stall while the target to jump to is resolved.

The most common solution utilizes a branch predictor in conjunction with speculative execution to accelerate processing. First, the branch predictor attempts to forecast where the target of the branch is; it then immediately begins processing the instruction stream at that target. When the real target is finally resolved, if the processor was correct, then the effects of the disruption have been mitigated. Unfortunately, branch mis-predictions do occur, and the penalty for mis-predicting may be very high.

Current branch predictors invest quite a bit of logic in sophisticated, general purpose constructs designed to handle all branches in the instruction stream. While effective, a more efficient method of attack is to split the branches into distinct sets and direct them to a specialized branch predictor most suited to that particular type. This paper proposes to divide branches into sets based on complexity. Instead of one general purpose predictor, it is expected that this hybrid branch predictor will produce higher accuracy with the same amount of resources.

Furthermore, this thesis will show the impact of several classes of branches on the overall accuracy of the branch predictor. These branches fall into several categories : constant and almost constant, infrequent, repeating pattern, and iterative. These branches can be accurately predicted with a smaller branch predictor than typical general purpose predictors.

Each branch class will be removed from the instruction stream to examine its impact on overall accuracy. It is anticipated that removing one or more of these classes will improve overall accuracy. These types of branches will then be assigned a specialized predictor. The combination of multiple specialized predictors should contribute not only to improving the accuracy of their own branches but also in reducing the pollution of the "normal" branch predictor.

1. Introduction

"Divide-and-conquer" is an oft-used strategy for solving a problem. By splitting up the work, multiple entities can specialize in solving smaller parts of the problem. Often times, this division of labor may prove more efficient than vesting increasing amounts of effort into a general-purpose construct. An example of this lies in the field of branch prediction. Modern branch predictors typically are general-purpose: one unit handles all branches, from exceedingly simple to extraordinarily complex. As a result, newer designs often incorporate more and more sophisticated branch predictors only to see diminishing returns.

1.1 Branch effects

Branch effects occur when the path of an instruction stream forks, depending on some value. Since the processor must wait until the branch instruction tests that value before deciding which instruction to execute next, a "bubble" of lost time occurs before any further instructions can be fetched. Fortunately, the processor need not wait for the branch to complete execution : it may simply "guess" and begin executing one of the paths speculatively. If it chooses correctly, then it has eliminated the bubble. If not, then it simply discards the mis-speculated work and executes the other path, and is no worse off. A better solution is not to simply guess but, rather, to *predict* which path the program will follow. The more accurate the prediction, the fewer clock cycles' worth of mis-speculated work need to be discarded. The gains from predicting branch outcomes are so great that virtually every modern processor includes branch prediction.

1.2 Path and Direction

When the instruction stream forks, two events may occur. If the value to be tested evaluates true, the *direction* the processor takes is said to be "taken." If the value evaluates

false, then the direction the processor takes is said to be “not taken.” The following sequence of instructions that occurs is referred to as the *path*.

In the event that the branch predictor is correct, it has attained a “hit.” If it is incorrect, it has produced a “miss.” There are two types of hits and misses – address and direction. An address hit occurs when the branch predictor correctly determines the target address. When the branch predictor correctly predicts the direction, it has produced a “direction hit.” For the remainder of the paper, prediction accuracy will refer to direction accuracy; likewise, hits refer to direction hits.

1.3 Current state of research

Using a variety of sophisticated techniques, current branch predictors now achieve 90-95% prediction accuracies. Yet current branch predictors are general-purpose structures, treating every branch the same. Branches that are simple and consistent are lumped in with more complex ones, even though predicting simple branches consumes resources which may be more profitably used to predict complex branches. The same is true of branches that are inherently unpredictable. A general-purpose branch predictor therefore wastes much of its sophisticated hardware on branches that cannot take advantage of it.

Rather than explore yet another, more sophisticated branch predictor, this work explores whether the same quantity of hardware can be used more efficiently by classifying branches according to their complexity. If the processor believes that it can predict a certain branch easily, it is routed to a much simpler branch predictor. Harder branches are routed to a more sophisticated branch predictor. Because the different predictors now only deal with a specialized subset of branches, it is expected that their hardware will be used more efficiently. Taken together, these predictors comprise a *complexity-based hybrid branch predictor.*

2. Rationale and Objectives

Augustus Uht et al. speculate that "if branch effects could be eliminated, performance [of microprocessors] could improve 25 to 158 times over that with sequential execution [1]." While realization of such an improvement may be difficult, the amount of potential simply cannot be ignored. Branch effects occur when the path of an instruction stream forks, introducing irregularities and unsteadiness into the instruction sequence.

2.1 Rationale

As branch effects are such a powerful lever on performance, much research has already been done in improving performance. However, modern branch predictors typically are general-purpose: one unit handles all branches, from simple to complex. As a result, newer designs often incorporate more and more sophisticated branch predictors only to see diminishing returns. Under the guidance of Dr. Kevin Skadron, this thesis explores a divide-and-conquer approach to branch prediction, evaluating the benefits of complexity-based hybrid branch prediction.

2.2 Objectives

Branches may be classified in four ways. *Static branches* are particularly easy to identify. These can be predicted by the compiler and need no hardware support. When the compiler's prediction is wrong, the program has encountered an exceptional condition for which performance is not necessarily a concern. *Easy-to-predict branches*, the second category, include branches such as loops that have very consistent behavior and hence require only simple prediction hardware. *Normal branches* are less uniform, but are not erratic. More sophisticated prediction hardware is successful with these branches. The last

category consists of the *hard branches*, which are so difficult to predict that they may simply pollute the branch predictor's state and reduce overall prediction accuracy.

The processor can perform this categorization in several different ways. The first is to use compiler hints. As the compiler must analyze the code anyway, it may be able to flag a hint to the processor that a branch is very predictable. Some sophisticated architectures already permit hints to be embedded in the program code [5]. However, these hints typically are generated assuming the presence of a single, general-purpose branch predictor, and may not be helpful when branches can be steered to one of several different predictors.

It may not always be the case that the compiler can categorize branches based on analysis of the code alone. A second technique adds information from profile data – statistical data gathered from prior executions of the program that conveys patterns of branch behavior. This statistical data is then fed back into the compiler, which modifies the executable accordingly. Profiles must typically be gathered for a range of program inputs; a single run may be a rogue, unrepresentative of typical program behavior.

Uht et al. [2] discuss the merits of using a different type of hybrid prediction, consisting of component predictors that use different strategies but remain general-purpose. In this organization, each component still predicts every branch, but the overall structure learns which component performs best. Chang et al. [6] investigate a hybrid predictor more in line with the approach proposed here. Their work identifies and filters out statically-predictable branches reducing the number of branches that use the main, dynamic branch predictor.

This project investigates the reliability of the proposed complexity-based branch categories, as well as the feasibility of implementing the delicate hardware-software interaction required to build an effective complexity-based hybrid predictor. The project begins by characterizing, for a range of real programs, the frequency of easy and hard branches and their effects on performance. A loose definition of "hard" can be established by analyzing the performance of current branch predictors on branches. Next, simulation of a

hybrid branch predictor attempts to discover the best combination of hardware- and software-based techniques. Further work will investigate the possible overhead associated with this organization, and explore whether the processor can further improve performance by identifying a branch's complexity using dynamic, run-time hardware.

3. Review of relevant literature

One of the core techniques in superscalar processing lies in exploiting the parallelism in the instruction stream. Unfortunately, achieving this parallelism through analysis of the instruction stream may be difficult to do. In particular, general purpose code typically exhibits irregular control flow and substantial conditional branching. The effects of such unsteady behavior, termed "branch effects," is one of the largest impediments to the performance of superscalar machines.

The most common technique used to combat branch effects is speculative execution. In speculative execution, code is executed before the result of the branch is known. In order to decide which path to speculatively execute, the processor must "predict" whether the branch will be taken or not taken. If the processor has predicted incorrectly, the speculative work is discarded. Each misprediction causes a loss in performance, as the processor must recover and try again. As a result, most work has gone into units specializing in determining the path of a branch, called branch predictors [5].

Branch predictors operate under stringent time and accuracy constraints. In pipelined machines, the processor will fetch an instruction every clock cycle. Given this time constraint, it is paramount that the branch be predicted in one clock cycle. If not, the processor will stall. Worse yet, if the branch predictor is inaccurate, this forces the processor to not only discard speculative work, but to restore the machine to the last known correct (non-speculative) state. Naturally, the consequences of a misprediction can potentially be disastrous. Having a 5% misprediction rate (95% accuracy), for example, can penalize performance by up to 32% [6]. As a result, a myriad of techniques have arisen in the implementation of branch predictors. Uht et al. details these techniques in their paper; for sake of brevity, these techniques will be briefly summarized [1].

The simplest form of branch prediction assumes that branches will either be always taken (Intel i486) or always not taken (Sun SuperSparc). These techniques only have an

accuracy of 40 to 60 percent, marginally better than pure guessing. A combination of these forms produces the backward-taken forward-not-taken (BTFN) technique seen in the HP PA-7x00 series. In this technique, backward branches are predicted always taken and forward branches always not taken. This combination improves branch prediction to 65 percent [1]. Along with semistatic prediction and eager execution, these techniques make up the common "static" prediction techniques.

While static prediction techniques are relatively inexpensive, they also render the predictor inflexible. If the predictor is consistently predicting incorrectly, it will continue to predict incorrectly. Instead, if the processor detects that it is consistently incorrect, allowing it to change its prediction will boost accuracy. These techniques are deemed "dynamic" prediction techniques.

The simplest dynamic prediction technique uses a one bit counter. If the counter holds a 1, the branch is predicted taken. Each time the result of the branch returns, however, the counter is reevaluated. If the branch was taken, the counter becomes a 1; if not, the counter becomes a 0. This technique, used in the DEC (Compaq) Alpha 21064, has an accuracy of 77 to 79 percent. Improving upon the one bit counter is the two bit counter. In the one bit counter, one rogue result would result in the next prediction being incorrect. Instead, the two bit counter relies on the most significant bit – a single spurious result will not throw off the prediction. This improves accuracy to between 78 and 89 percent. Two bit counters are fairly popular – processors from the Mips R10000 to the Intel Pentium to the IBM/Motorola PowerPC 604 use this technique [1].

The next step in branch predictor design is in the combination of the above techniques. Recent research has looked into having a two-level adaptive branch predictor. The first level keeps a history of recently executed branches; the second, based on the pattern in the first level, determines the likely direction of the branch [9]. This combination, used in the Intel Pentium Pro and AMD K6, boosts accuracy to 93% [1]. While this is an effective technique, it is very complex. Worse, it is expensive in terms of logic and silicon.

A different technique uses a combination of predictors instead of techniques. The advantage is that the overall complexity is only determined by the most complex predictor. The approach used by the DEC (Compaq) 21264 takes a local branch predictor and mates it with a global one [8]. While the processor is running, the processor keeps a table on success rates of each predictor on each branch. When a branch is encountered, the processor probes both predictors. If both predictors come to the same prediction, that prediction is used. If each predictor supplies a different prediction, the processor selects based on which predictor has been more successful with this branch in the past [12].

The major disadvantage of DEC's selector technique is redundancy. Since both predictors are general-purpose predictors, both will achieve high success rates with normal branches. Only unique and/or difficult branches separate the predictors. This translates into wasted logic and higher silicon costs.

Instead, the next step in improving branch predictor design lies in multiple, specialized branch predictors – "hybrid predictors." Hybrid predictors are, theoretically, more efficient in their use of silicon. In developing hybrid predictors, however, one must decide on how to divide the branches between predictors such that 1) the predictor with the best chance gets the branch, and 2) predictors do not overlap and interfere with each other.

The first constraint is obvious. The second, however, is more subtle. On the surface, it would seem beneficial that correctly predicted branches be recorded in the pattern history table. However, easily predicted branches may actually pollute the pattern history tables. Chang et al. detail the effects of such interference in a two-level adaptive branch predictor. By inhibiting easily predicted branches from polluting the pattern history table, the misprediction rate for the gcc benchmark was reduced by 38% [11].

Fortunately, if multiple predictors have their own history tables, this problem can be minimized at the cost of silicon and logic. The next issue to resolve how to divide the available branches such that the predictor with the best chance gets the branch. Having

lopsided or unequal branch predictors is not an issue; rather, the problematic aspect in the decision lies in determining the best place to draw the line.

One proposed division of labor is by branch classification. In branch classification, branches are divided based on their behavior. A good scheme will group branches exhibiting similar behavior together [10]. Using a profile-guided predictor to handle mostly one-direction branches and two two-level adaptive predictors to handle mixed direction branches, Chang et al. were able to achieve an accuracy of 96.91% on the gcc benchmark, besting the previous high mark of 96.47% – reducing the miss rate by 12.5% in the process [10].

The above technique only filters out simpler, one-direction branches, reducing the number of branches using the dynamic branch predictor. Another method, to be explored during the life cycle of this project, is to classify branches by complexity rather than behavior. By dividing branches based on relative difficulty, one can pair a simple branch predictor with a complex branch predictor. In doing so, one gains more flexibility over a profile guided with dynamic branch predictor setup, as long as both predictors remain general.

In addition, as both predictors are dynamic, it is expected that the accuracy achieved by this configuration will be better than one that combines static and dynamic branch predictors. Profile-guided prediction has been used to some success – IBM/Motorola's earlier PowerPCs used this technique to attain 75% accuracy [1]. However, it must be noted that IBM and Motorola's figures are for all branches – in the split profile-guided and dynamic predictor method, the profile-guided predictor only handles static branches. This ought to increase the accuracy considerably.

Profile guided and compiler assisted branch prediction is not a dead end technology, however. Research into Very Long Instruction Word (VLIW) machines rely heavily on compilers to uncover parallelism [7]. Naturally, branch prediction is part of the package. The most visible research effort is in the Intel/HP Itanium (formerly Merced) processor. Interestingly enough, the Itanium will combine compiler assisted branch prediction with

dynamic branch prediction [7]. The Itanium compiler evaluates branches during the compilation process and attempts to provide a hint. At run-time, the hardware uses the hint to decide whether to evaluate the branch with the compiler assist or to send the branch to the hardware predictor [7]. The effectiveness of this technique is yet to be known.

References

[1] Augustus Uht, Vijay Sindagi, and Sajee Somanathan, "Branch Effect Reduction Techniques," 1997

[2] Doug Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," 1997

[3] Eitan Federovsky, Meir Feder, Shlomo Weiss, “Branch Prediction Based on Universal Data Compression Algorithms,” 1998

[4] I-Cheng K. Chen, John T. Coffey, Trevor N. Mudge, “Analysis of Branch Prediction via Data Compression,” 1996

[5] John L. Hennessy, David A. Patterson, "Computer Architecture : A Quantitative Approach,” Morgan Kaufmann Publishers, Inc., 1996

[6] Kevin Skadron, “Characterizing and Removing Branch Mispredictions.” Ph.D. thesis, Princeton University, June 1999.

[7] Linley Gwennap, “Intel Discloses New IA-64 Features.” *Microprocessor Report*, March 8, 1999, pp. 16-19.

[8] Linley Gwennap, “Digital 21264 Sets New Standard,” *Microprocessor Report*, October 28, 1996

[9] Marius Evers, Sanjay J. Patel, Robert S. Chappell, Yale N. Patt, “An Analysis of Correlation and Predictability : What Makes Two-Level Branch Predictors Work,” 1998

[10] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt, "Branch Classification : a New Mechanism for Improving Branch Prediction Performance," 1998

[11] Po-Yung Chang, Marius Evers, and Yale N. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," 1997

[12] Scott McFarling, “Combining Branch Predictors,” *WRL Technical Note TN-36*, Digital Corporation, 1993

4. Simulation Setup

The benchmarks are run against the sim-bpred.c program provided by the SimpleScalar test suite. The program has been modified to support the identification and storage of constant and almost constant branches as well as the ability to fast-forward through instructions. The initial code to support identification of constant branches was contributed by Dr. Kevin Skadron and modified with help from Adrian Lanning. The fast-forward code was contributed by Michele Co, and is used with minor modification.

Testing typically is performed in two stages. First, the modified sim-bpred (simb) steps through the code normally and gathers statistics on all branches discovered in the program. Each class of branch has its own "generator" program – this increases speed and modularity. Rather than have one large, all-encompassing binary, each branch class has a customized generator.

Branch Class	Binary	Additional Comments
baseline	simb	produces "standard" case accuracy
constant/almost constant	simb2	
infrequent	simbinfq	
"large"	simbla	for comparing 2-level and bimodal prediction
repeating	simbrep	Requires itrepcombine.pl and repeatpat.pl; feeds simblp
iterative	simbitrep	Requires itrepcombine.pl and biaspatterns.pl; feeds simblp

Table 1.1 : Progams Used to Generate Profiles

During the second stage, a very similar sister program, simbwc, reads the data file created by simb and stores the addresses of marked branches in memory. It then begins execution, and upon discovering a branch, checks if the branch was marked. These marked branches are then treated differently in the update stage of the branch predictor. With the exception of repeating and iterative branches, the data file created by simb is a list of

address-direction pairs; the address is the address of the branch in decimal, while the direction is the direction the branch usually takes.

The direction is present for completeness – in the actual execution of the program, the direction element is ignored. Instead, for the marked branches, the prediction is assumed to be perfect. Since querying the branch predictor is non-volatile and does not affect branch update, upon branch update the predicted direction fed to the branch predictor is the *actual* direction, not the predicted direction. In this way, marked branches, to the branch update mechanism, are seen as perfectly predicted.

Repeating branches and iterative branches are a separate case. Since the important element is the number of taken branches or not taken branches before the pattern repeats again, the second element is replaced by the count. Note that because the simbwc-derived programs ignore the direction element, this data file will actually execute and correctly simulate repeating branches and iterative branches being isolated or prevented from updating the global history; however, if the desired result of these tests is to determine the effect and benefits of using a loop predictor, another simulator, simblp, must be used.

The loop predictor is implemented by adding a loop count and a bias bit to the branch table. The looping branches are then marked in the branch table, with the bias bit set to indicate what form it takes. Normal branches ignore the loop count. In cases of emergencies, loop prediction with fixed loop branches exactly correspond to the behavior of preventing marked branches from updating the pattern history tables.

Desired Behavior	Binary	Additional Comments
prevent updates to global history	simbwc_pht	allows updates to pattern history
prevent updates to pattern history	simbwc_nopht	allows updates to global history
isolate branches from predictor	simbwc_i	
loop prediction	simblp	uses standard branch update mechanism

Table 1.2 : New Branch Predictor Schemes

4.1 gcc

The *gcc* benchmark was the most extensively tested of the four selected out of the spec95 suite. *Gcc* is a commonly used compiler in use in many different operating systems. A source file – typically in the C programming language – is taken as input and compiled into binary machine code for execution on the target system. *Gcc* complements the benchmark suite by providing a program in used for a very common purpose in developing software – that of compiling to a binary. It is typically very large, and stresses the branch predictor by having the largest static footprint of all the selected programs.

The test runs are not fast-forwarded, and are run to completion. The program cc1.ss and the file cccp.i are required to duplicate the test runs.

The *gcc* test runs were invoked with these parameters : `-quiet, -funroll-loops -fforce-mem, -fcse-follow-jumps, -fcse-skip-blocks, -fexpensive-optimizations, -fstrength-reduce, -fpeephole, -fschedule-insns, -finline-functions, and -fschedule-insns2`.

4.2 compress

The second benchmark tested was the *compress* benchmark. *Compress* is a file used to reduce the size of files by applying some sort of packing algorithm to the data. When required, another program can apply the same algorithm in reverse to “uncompress,” or expand, the packed data. *Compress* adds to the benchmark suite by providing a program that spends the majority of its time in tight loops, applying the compression algorithm sequentially to sections of the source data. While its static footprint is the smallest of the group, each of the vital branches is executed quite a bit more than branches in other programs, and is extremely data dependent – meaning for more variability and unpredictability in their overall behavior .

Unlike the *gcc* benchmark, however, *compress* needs to be fast-forwarded to skip over the generation of the random data to be compressed. The particular routine used was compress.ss, on the file bigtest.in.

The compress test runs were invoked with these parameters : `-fastfwd 1600000000, -max:inst 100000000.`

4.3 perl

Perl was the next benchmark tested. *Perl* is an interpreter for the perl programming language. A script, or in this case, the scrabbl.pl file – in plain text – is read and executed by the interpreter. The scrabbl.pl script attempts to determine anagrams for a given set of letters by matching the possible outputs to an English dictionary. Perl is a valuable addition to the benchmark suite because unlike *gcc* and compress, *perl*'s strength lies in text parsing and pattern matching. As this is a common use for computers, it is important to measure the branch predictor's performance on such applications.

Like compress, *perl* was fast-forwarded to skip over the initialization period. It was also the shortest benchmark run, at 50,000,000 instructions. However, *perl* needs to be fast-forwarded 1,950,000,000 instructions. To give *perl* a script to work with, scrabbl.pl was paired with dictionary.

The *perl* test runs were invoked with these parameters : `-fastfwd 1950000000, -max:inst 50000000.`

4.4 go

The setup for the *go* benchmark closely resembles the perl benchmark. *Go* is an ancient board game, of which the computational complexity bests even that required by chess. The object of the game is to capture as much "territory" as possible by surrounding it

with game pieces. *Go* is a good complement to the selection of benchmarks because it simulates a highly computational game playing algorithm. Its static footprint is the second largest in the group, and of all of the selected benchmarks, is the least predictable.

Like *perl*, *go* must be fast-forwarded. The simulation calls for fast-forwarding 3,900,000,000 instructions. The input file used was 9stone21.in.

The *go* test runs were invoked with these parameters : `-fastfwd 3900000000 -max:inst 100000000`

4.5 ijpeg

The *ijpeg* benchmarks were fast-forwarded 823,000,000 instructions and ran for 50,000,000 instructions. *Ijpeg* is an image processing program, another valuable commonplace use of computation. Its behavior is somewhat like compress – the application of some processing algorithm to pixels in an image.

The *ijpeg* program, ijpeg.ss, was run against the input file vigo.ppm.

The *ijpeg* benchmarks were run with the following options : `-fastfwd 823000000, -max:inst 50000000, -image_file, -compression.quality 90, -compression.optimize_coding 0, -compression.smoothing_factor 90, -difference.image 1, -difference.x_stride 10, -difference.y_stride 10, -verbose 1, -GO.findoptcomp`.

4.6 xlisp

The *xlisp* benchmark, li.ss, was run against the reference input files 9queens.lsp and xit.lsp. *Xlisp* is an interpreter for the lisp programming language. The interesting set of capabilities that *xlisp* adds to the selection of programs is that lisp is a programming language of which its structure is relatively easy for computers to parse and execute.

The resulting setup was fast-forwarded for 900,000,000 instructions and ran for 100,000,000 instructions.

The *xlisp* benchmarks were run with the following : `-fastfwd 900000000 -max:inst 100000000`

4.7 Branch Predictors

Two branch predictors were used in each branch prediction scheme. The first, a smaller two level adaptive branch predictor, had the following characteristics : a shift register width of five bits coupled with an L2 size of 4096 counters. Naturally, this would require that seven address bits would be required to be concatenated to the history to index into the counters.

The second branch predictor was a larger two level adaptive branch predictor. It had the following characteristics : a shift register width of eight bits and 32,768 counters in its second level. This would require seven address bits as well to be concatenated to the history in order to index to the correct counter.

5. Identify and classify branch behavior

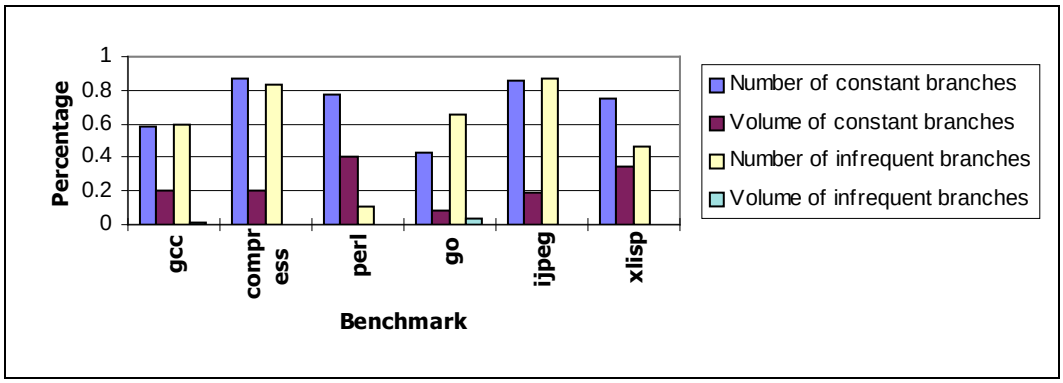

Fig 1.1 Number of Constant and Almost Constant Branches

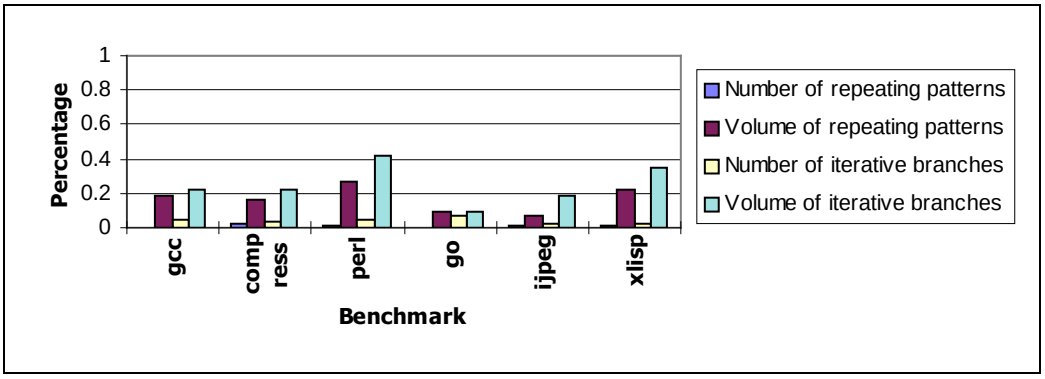

Fig 1.2 Number of Repeating Patterns and Iterative Branches

5.1 Constant/almost-constant branches

Constant branches are defined as always taken or always not taken. In addition to locating and identifying always taken or always not taken branches, the time allotted will provide programming experience with using the SimpleScalar simulator.

Almost-constant branches only have one or two switches. These may be branches that are always taken except for the last path, which is not taken, or branches that have an exception in the middle. Furthermore, almost-constant branches may also be branches which are always taken halfway, and then switches to always not taken.

The *gcc* benchmark shows that 58.91% of unique branches in the program are constant. However, this number is misleading. While a great majority of unique branches are constant, a more detailed analysis shows that constant branches only make up 20.59% of executed branches – that is, non-constant branches are executed a great deal more times than constant branches.

The second benchmark, *compress*, shows even a greater divide – constant branches make up 86.42% of the unique branches, but are only 20.36% of the executed branches. However, compress is more data dependent than *gcc*, and this may change from test run to test run.

Other benchmarks, namely, *perl* and especially *go* show different, more interesting results. 77.94% of *perl's* branches are unique, and they represent 40.18% of the number of branches executed. *Go's* constant branches, however, are only 42.33% of the unique branches; they make up even less of the total number of branches executed, or 8.83%.

Ijpeg and *xlisp* contribute some useful comparison data. In the case of *ijpeg*, 85.45% of the branches are constant – similar to *compress*. In addition, 18.95% of *ijpeg*'s executed branches are considered constant – again mirroring the makeup of *compress*. As a second basis of comparison, 80.95% of the branches in *xlisp* are constant. Unlike *ijpeg* and *compress*, however, 35.08% of *xlisp*'s executed branches are constant.

The first two results warrant some discussion. *Gcc's* constant branches averaged around 2.6% misprediction rate; better for the bimodal predictors, but worse for the two level adaptive predictors. A 2.6% misprediction rate may seem so insignificant as to not be worth investigating, but these constant/almost-constant branches make up 20.59% of the total number of branches. Even a percentage point improvement in the misprediction would make a good impact on the overall branch prediction.

Compress exhibits a stranger result. The prediction rates for constant branches hovered around 99.99999%; overall prediction rate, however, was 92.05% (89.15% for the

bimodal predictor). Further tests will be required to determine what, exactly, this may mean. With 20.36% of the executed branches being executed at 99.99999% in both two level adaptive predictors, this may mean that the non-constant branches are being predicted rather poorly.

Go is the most interesting result. While constant branches make up 8.83% of the total quantity of branches executed, prediction rates for go are much worse than all the others – a 4.65% misprediction rate. Overall, *go* averaged 87.44% prediction accuracy, much less than the 93.86% (*gcc*) to 97.14% (*perl*) of the other three benchmarks. Worse yet, in the smaller branch predictor – with overall accuracy of 83.75% – almost constant branches are mispredicted at a staggering 47.67%.

Constant and almost constant branches are somewhat easier than the others to identify in developing a program. Some can be uncovered even in compiling – as in the case of many error conditions – whereas the others rely on profiling. It can be argued, then, that the effort required in profiling reduces the incentive to implement the following three branch prediction schemes, especially if the performance gain is small.

The counter argument, then, is that the majority of developers who value execution speed already perform profiling on their programs; gathering additional information about branch behavior would not add much of an additional burden to the developer. In addition, by not profiling, the branch predictor is really not hurt any – rather than fully utilize the newer, better branch prediction scheme, the constant and almost constant branches simply fall back on the older, traditional all-in-one branch prediction scheme. Better yet, the contents of this project reveal that the performance boost can be considerable, especially in the case of constant and almost constant branches. As these may be some of the easiest to determine in the process of compiling and profiling, the amount of work for performance gained is well worth it.

5.2 Infrequent branches

Infrequent branches are defined as branches that occur less than one thousand times. Infrequent branches contribute little to the overall patterns, and even may be detrimental to accuracy. If infrequent branches do nothing except pollute the pattern history table, then removing them will show further increases in accuracy. The two common configurations described above will again be used to test the effectiveness of isolating these branches.

Execution of the *gcc* benchmarks shows that 59.48% of all branches in the program are "infrequent" – that is, they are infrequently executed. Remember, however, that number of unique branches in the program is not necessarily a good estimation of the impact of a certain class of branch. Upon examining the total volume of branches executed, infrequently executed branches make up a mere 0.93% of the instruction stream.

The *ijpeg* benchmark shows a somewhat similar makeup. The percentage of branches in *ijpeg* that are classified as infrequent is significant – 86.73%. However, upon examining the statistics of the overall instruction stream, only 0.84% of executed branches in *ijpeg* are infrequent – an almost marginal amount.

Compress shows an even larger divide. While 83.54% of the branches are infrequently accessed, infrequent branches are a virtually negligible 0.00352% of the number of branches executed. Even if the misprediction rate was considerable – in *compress'* case, 23.23% – even mispredicting every infrequent branch would make hardly an impact on the overall prediction rate.

Perl and *xlisp* show results that are better than *compress*, but still almost insignificant when taken with the overall volume of branches executed. In *perl*'s case, 11.17% of unique branches are infrequently executed, but overall only 0.3% of all branches executed are classified as infrequent. In *xlisp*, though 46.03% of branches in the program are infrequent, these branches represent an almost trivial 0.074% of the executed branches in the instruction stream.

Go is the only benchmark that has a significant amount of infrequently executed branches. In *go*'s case, 65.3% of branches existing in the program are classified as infrequent. The volume of executed branches that are infrequent, however, is 3.24% – somewhat small, but still significant in comparison to the other benchmarks in the group.

Infrequently executed branches are somewhat more difficult to determine in the profiling process. For the case of loops with fixed iteration counts, the compiler can quickly determine and flag the branch as such. However, for a program like *compress*, which is dependent on a data file – which may be very long or very short – the determination becomes much more difficult. If the same loop is applied to the length of the file, for example, profiling a very small file would not be at all representative of the normal usage of the program. As a counter argument, profiling a large file would eliminate some of the benefits in performance, but for smaller files, the slight improvement in execution speed may not be significant at all.

5.3 Repeating Patterns

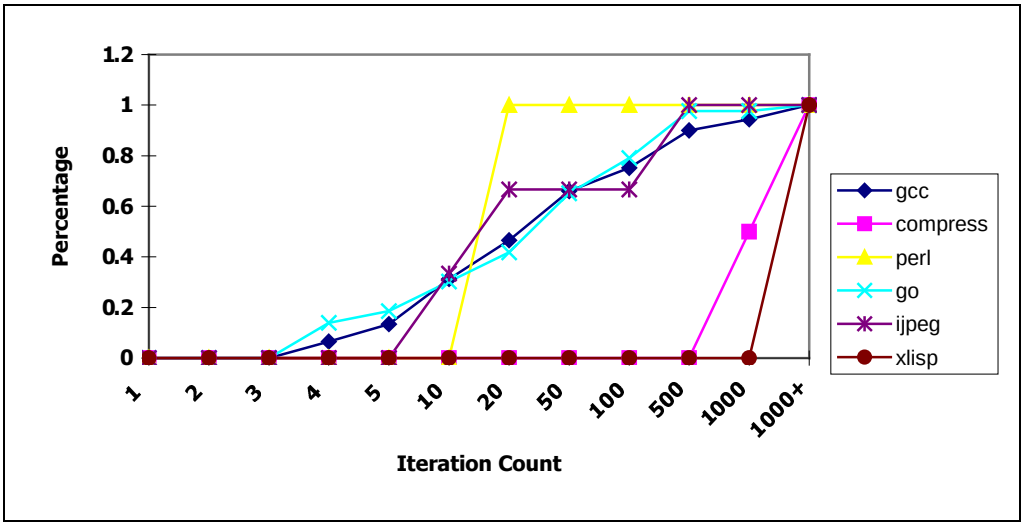

Fig 1.4 Distribution of Iteration Counts for Repeating Patterns

Branches that exhibit repeating patterns are defined as branches that alternate between taken and not taken in regular, predictable patterns, i.e. a series of *n* taken branches followed by *m* not taken branches, or vice versa, where *n* and *m* are greater than one, and exceptions are made for the first and last iterations of the pattern. This pattern then repeats for the duration of the program. Repeating pattern branches have a particularly beneficial property in that branch predictors require less bits for predicting the direction of these branches; rather than keep the entire pattern in a table, counters can represent the same information. One possibility in devising a hybrid predictor may be to isolate repeating patterns into a smaller, specialized repeating pattern branch predictor.

Repeating patterns are divided into two groups – fixed patterns and variable patterns. In fixed repeating patterns, *n* and *m* never change – they are constant for the duration of the program. Variable patterns have either *n*, *m*, or both *n* and *m* varying as the program is executing. Naturally, the fixed repeating patterns are easier to predict, and will be the focus in this project.

Repeating patterns have a significant representation in the volume of executed branches in most of the benchmarks; however, due to their highly regular nature, they are also fairly predictable. In *compress*, for example, repeating patterns make up 21.7% of executed branches – a high representation for a program with only 1.64% repeating pattern branches, of which half are fixed repeating patterns. The prediction rate for the repeating patterns, however, is an excellent 99.991% for the larger branch predictor, and 99.9953% for the smaller branch predictor.

Ijpeg displays similar characteristics. Of all executed branches in *ijpeg*, 19.03% fall in the repeating pattern category, yet they are only 0.4% of all branches in *ijpeg*. All happen to be fixed length repeating pattern, and are predicted with already high accuracy. The larger branch predictor predicts the repeating patterns with 99.77% accuracy, whereas the smaller branch predictor achieves a 99.61% accuracy.

Xlisp is the most extreme – 35.14% of all executed branches in *xlisp* are contained in the repeating pattern branches, but they only make up 0.32% of the branches in *xlisp*'s code. Interestingly enough, *xlisp* contains no fixed length repeating pattern branches. The remaining variable patterns, though, are predicted with 99.912% accuracy in the larger predictor, followed by 99.72% accuracy in the smaller branch predictor.

Perl's makeup resembles *xlisp*. Of the static branches in *perl*, 0.57% are repeating pattern, half of which have fixed repeating patterns. Those few repeating patterns, however, consist of 41.44% of all executed branches. The prediction rate remains high – 99.35% for the larger branch predictor and 97.56% for the smaller branch predictor.

Gcc and *go* are the ones to watch. Unlike the others, the branch predictors have more difficulty predicting the repeating patterns in *gcc* and *go*. The larger branch predictor in *gcc*, for example, achieves only a 97.67% accuracy, whereas the smaller branch predictor is slightly worse, at 94.65%. This is significant, in that repeating patterns comprise 21.97% of all executed branches in *gcc*, and 1.93% of all branches in the code. Included in the statistics are the 1.11% of branches in *gcc* that have fixed repeating patterns.

Go is somewhat similar. Unlike the other five benchmarks, repeating patterns make up a great deal less in *go*. Only 9.86% of its executed branches are repeating patterns, and are predicted at a worse rate than *gcc* – 95.69% for the larger predictor, 92.77% for the smaller one. 2.4% of branches in *go*'s code are repeating patterns, including the 1.25% that have fixed repeating patterns.

5.4 Iterative branches

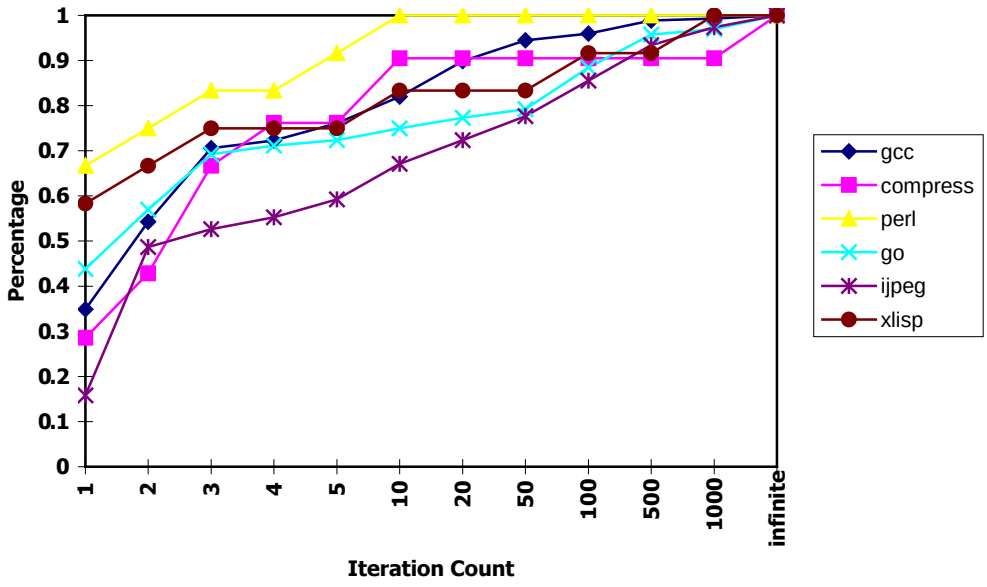

Fig 1.4 Cumulative Distribution of Fixed Length Iteration Counts

Iterative behavior is similar to repeating pattern behavior except that iterative allows for only one exception in the pattern. For example, an iterative pattern may be a sequence of four taken branches followed by a single not taken branch, followed by a second sequence of five taken branches and a not taken branch. Similar to repeating pattern branches, an iterative branch predictor history can be represented with counters. Iterative pattern branches, however, require only one counter – repeating pattern branches usually require more. A bias bit is also required to indicate whether it is of the form not taken, not taken, taken, or taken, taken, not taken.

Furthermore, iterative branches are defined into two categories – fixed length patterns and variable length patterns. Fixed length iterative patterns are branches that repeat the same pattern for the duration of the program execution – an example would be a loop with a fixed number of iterations. Variable length iterative patterns do not have a fixed constraint; instead, they have patterns of varying lengths, however, they still follow the basic behavior

of an iterative sequence. The statistics generated show that for all iterative branches, variable length iterative branches dominate over fixed length iterative branches in terms representation in the benchmark programs as well as representation over all executed branches.

As iterative branches are highly regular in nature and have comparatively low unpredictability, the baseline two-level adaptive branch predictor should perform remarkably well. The initial tests confirmed this behavior, and therefore it would appear that the benefits are to be more geared toward area savings rather than branch predictor accuracy.

In general, iterative branches make up a small portion of the benchmark programs. In terms of execution volume, however, these branches comprise a surprising percentage of the overall execution stream. In the benchmark *gcc*, for example, only 4.15% of the branches in the program have an iterative behavior. Included in that count are fixed iteration count branches, which make up 0.46% of *gcc*. In the overall instruction stream, however, 18.88% of the branches are iterative.

Compress exhibits a similar makeup. While only 3.3% of branches existing in *compress* are iterative, 16.16% of the executed branches are iterative in nature. Fixed iteration count branches are 2.47% of branches in *compress*' code. In part due to their highly regular nature, though, iterative branches are already predicted extremely well. The iterative branches in *compress* – executed many thousands of times – are predicted at 99.9955% accuracy by the baseline two-level adaptive branch predictor.

Perl and *xlisp* produce similar statistics, though not as extreme as *compress*. In *perl*, 4.3% of unique static branches behave in an iterative fashion – including the 0.86% which have a fixed iteration count; in execution volume, however, these iterative branches comprise 26.93% of the total execution stream. Like *compress*, these branches are predicted extremely well – 99.04% accuracy. In *xlisp*, a mere 2.54% of unique branches are iterative in nature. Included are the 0.64% which have a fixed iteration count. Out of all executed branches in *xlisp*, however, 21.67% are iterative, and are predicted at 99.44% accuracy.

The benchmark *ijpeg* shows a slightly different picture. Only 2.42% of the branches in *ijpeg* are iterative – unlike the above four benchmarks, however, execution volume is not quite as significant. Only 6.9% of executed branches in *ijpeg* behave in an iterative manner, counting the 1.71% which are fixed iteration count branches. These branches are already predicted at 99.54% accuracy.

The last and most interesting benchmark in this section is *go*. *Go*, like *ijpeg*, is made up of branches that are primarily not iterative – only 7.12% of branches in the program are iterative – including the 1.72% which have fixed iteration counts – and correspond to merely 9.46% of executed branches. The striking statistic in this benchmark, however, is that iterative branches are predicted at a comparatively poor 94.97% accuracy. Unlike the others, improving the accuracy of iterative branches may in fact impact the overall accuracy a great deal.

Iterative branches are also a bit more difficult than constant/almost constant branches for the compiler to realize. Obviously, there are some branches which are not so subtle in their behavior – loops with fixed iteration counts and no early exit, for example. The others – branches which just happen to exhibit iterative behavior – are more difficult to spot. Even profiling may not necessarily produce the best results. One set of inputs may cause a certain branch to behave in an iterative fashion, but does not when every other set of inputs is applied. In the same logic, profiling a small, simple test run may completely miss whole sets of branches that behave in an iterative manner.

6. Measure impact of not updating the global history register

The rationale for excluding certain classes of branches from updating the global history register in a two-level adaptive branch predictor is that while vital to the execution of the program, certain classes of branches may not be productively contributing to the branch predictor state. At best, the branches are benign. At worst, the branches are detrimental to the prediction accuracy of the branch predictor. It is important to note that while certain branches are prevented from updating the global history register, they are allowed to update the pattern history tables.

6.1 Constant/almost constant branches

The class of branch that benefited the branch predictor most were those of constant and almost constant branches. Better yet, some constant and almost constant branches can be identified at compilation – those of error codes, for example, where one taken branch stops execution of the program. By preventing these branches from updating the global history register and encoding their behavior in the opcode, branch predictor accuracy improves.

In general, the number of constant branches in the program dwarfed the number of almost constant branches – in fact, in the *perl* and *xlisp* benchmarks, there were no branches that could be classified as "almost constant." Furthermore, the smaller branch predictor showed the larger improvement, on average, than the larger branch predictor.

	gcc	compress	perl	go	ijpeg	xlisp
base, 4096	**90.99%**	**90.77%**	**95.00%**	**83.75%**	**89.77%**	**95.49%**
simbwc_pht, 4096	**92.21%**	**91.85%**	**96.49%**	**84.67%**	**90.10%**	**95.48%**
base, 32768	**93.86%**	**91.81%**	**97.14%**	**87.44%**	**90.78%**	**96.95%**
simbwc_pht, 32768	**94.41%**	**92.62%**	**98.04%**	**87.70%**	**91.17%**	**96.97%**

Table 2.1 : Branch Prediction Accuracy of Selected Spec95 Benchmarks

Gcc, for example, has a large static footprint – 20,687 branches in the program. In the smaller branch predictor, *gcc* showed a tremendous improvement – from 90.99% to 92.21%, a boost of 1.22%. *Go* showed a similar improvement – up from 83.75% to 84.67%, or an improvement of 0.92%.

An explanation of this can be found in the fact that the above two programs, with relatively large static footprints, tend to have more branches aliasing and creating destructive interference. With more hardware, the likelihood of interference is reduced, as seen in the statistics for the larger branch predictor – g*cc* only improved 0.55%, while *go* only showed a 0.26% boost. Furthermore, it may be the case that constant and almost constant branches do not contribute very worthwhile information to the branch predictor state. If they did, the expectation is that branch prediction accuracy would decrease. With constant branches making up some 20.59% of *gcc*'s executed branches, a difference either way would make a noticeable impact on overall branch prediction.

It may be argued, however, that since the method assumed all marked branches – in this case, constant and almost constant – as correct, the performance increase is due solely to that factor. Looking slightly ahead, if removing certain branches from updating parts of the branch predictor was not helpful at all, then all three methods examined – preventing the update of the global history, preventing the update of the pattern history, and isolating branches completely – would return similar results. A quick glance ahead shows this to not be true.

In addition, the dramatic rise in prediction accuracy of a benchmark such as *go* cannot simply be explained away by assuming all the gain was due to counting previous mispredictions as correct. *Go* contains only 8.83% constant branches, which are predicted with 90.86% accuracy. An increase of 0.92% accuracy cannot be explained away by the above. Furthermore, one would expect that if the above argument were the case, a similar, albeit smaller, increase would be seen when the branch predictor was increased in size. When using a considerably larger branch predictor, *go*'s constant branches are predicted with 94.7% accuracy, yet the boost was only 0.26%. Since the accuracy was not changed a considerable amount yet the improvement dropped dramatically, it can only be concluded that the benefit came from increasing the size of the branch predictor – a theory advanced a few paragraphs above this one.

The other benchmarks showed similar results. For example, *compress*'s constant branches make up 20.36% of all executed branches, and are predicted at virtually 100% accuracy. If the theory were to uphold, then *compress* would see an increase as well, despite the fact that it would not see a gain by counting previously mispredicted branches as correct. For the smaller branch predictor, *compress* showed a 1.08% increase, from 90.77 to 91.85% – a considerable improvement. Likewise, the larger branch predictor responded with an increase of 0.81%, from 91.81% to 92.62%.

The improvement, however, is not due solely to the fact that more worthwhile static branches can fit in the branch predictor. In fact, *compress* only has 243 static branches in its code – significantly less than *gcc*'s 20,687 and *go*'s 3,450. Instead, the increase is likely due to the fact that some of these constant branches – already predicted at close to 100% – were interfering with the more difficult to predict, non-constant branches. By preventing these from updating the global history register, the branch predictor was better able to concentrate on the remaining non-constant branches.

Perl showed similar and even more astounding results. The smaller branch predictor, when executing *perl*, responded the most favorably – an increase of 1.49%, from an already

good 95% to 96.49%. Couple this with the fact that 40.18% of executed branches in *perl* are constant and are predicted with 98.13% accuracy, and one can easily see that the boost in improvement comes from a source other than simply counting those few mispredicted constant branches as correct. Better yet, when the larger branch predictor was tested, branch prediction accuracy improved 0.9% – astounding in light of the fact that constant branches were already predicted with 99.72% accuracy.

The remaining benchmarks, *ijpeg* and *xlisp*, showed somewhat interesting and unexpected results. In *ijpeg*, the improvement in prediction accuracy was actually better in the larger branch predictor than the smaller branch predictor. The volume of executed branches that are constant in *ijpeg* is sizable – 18.95%, and are well predicted already – 99.47% in the smaller predictor and 99.65% in the larger predictor. When preventing the constant branches from updating the global history register, though, prediction accuracy went up 0.33% in the smaller predictor, but 0.39% in the larger predictor. The small fluctuations may be attributed to simply the way branches hash into the predictor, but the fact remains that a noticeable improvement in branch prediction accuracy was achieved by preventing constant branches from updating the global history register.

Xlisp provided the only negative results. In the smaller branch predictor, accuracy actually decreased by 0.01%, whereas the larger branch predictor only improved 0.02%. While 35.08% of *xlisp*'s branches are constant – *xlisp* contains no almost constant branches – prediction accuracy is already very good. The smaller branch predictor performs at 99.94% accuracy, while the larger predictor shows 99.99% accuracy. It would seem that branch prediction would increase at least a small amount; however, this proved not to be the case. One conclusion that may be drawn is that preventing the branches from updating the global history register does not actually decrease branch predictor performance – as 0.01% is negligible – but in fact is beneficial for the majority of benchmarks.

6.2 Infrequent branches

As mentioned above, some types of branches may not contribute worthwhile information to the branch predictor state. Indeed, for infrequent branches, when taken with the program of the whole, do not represent a large portion of the instruction stream. From preliminary results, the impact, if any, of preventing infrequent branches from updating the global history register was minimal.

	gcc	compress	perl	go	ijpeg	xlisp
base, 4096	90.99%	90.77%	95.00%	83.75%	89.77%	95.49%
simbwc_pht, 4096	91.23%	90.77%	95.05%	84.78%	89.92%	95.49%
base, 32768	93.86%	91.81%	97.14%	87.44%	90.78%	96.95%
simbwc_pht, 32768	94.01%	91.81%	97.17%	88.10%	90.85%	96.96%

Table 2.2 : Branch Prediction Accuracy of Selected Spec95 Benchmarks

For rather large branch predictors, at least, the impact of reducing contention by singling out infrequently executed branches is minimal. As infrequently accessed branches are less likely to alias and destructively interfere with more important, often executed branches in a comparatively large predictor, the impact is almost negligible.

For smaller branch predictors, however, where resources are more scarce, infrequent branches can easily alias into another branch's resources and interfere with the branch predictor's attempt to predict either branch. An example of this can be found in the benchmark programs *gcc* and *go*. Compared to the other four benchmark programs, *gcc* and *go* have a relatively large static footprint – that is, they have many more static branches in their code than do other benchmark programs in the suite. *Gcc* and *go* have static branch counts in the thousands (20,687 and 3,450, respectively), whereas the next closest benchmark is *ijpeg*, with a mere 701 static branches.

With large branch predictors, more of the branches can fit into the branch predictor storage tables. Unfortunately, not all processors can afford to devote large amounts of silicon to the branch predictor. It is easy to see how programs such as *gcc* and *go* can quickly fill up a smaller branch predictor, causing heavy aliasing and interference in the branch predictor.

For the smaller two level adaptive branch predictor, the normal prediction accuracy for *gcc* is 90.99%. When infrequently executed branches were blocked from updating the global history register, prediction accuracy rose to 91.23% – a sizeable improvement of 0.24%. *Go* shows an even larger improvement – from 83.75% in the baseline configuration to 84.78% in the modified configuration, or an improvement of 1.03%.

Part of the improvement can be attributed to the fact that infrequent branches, for the sake of calculations, are assumed to be perfectly predicted. In the smaller branch predictor, g*cc* and *go,* in the baseline configuration, have a relatively poor prediction accuracy – 74.87% and 66.48%, respectively. However, it is important to note that while the accuracies are poor, the volume of branches is small as well. As mentioned previously, infrequent branches in *gcc* make up a mere 0.93% of the executed branches, while infrequent branches in *go* are 3.23% of executed branches.

It may be argued again, then, that the performance improvement is not a function of reducing the interference in the global history register, but rather a function of simply counting all the mispredicted branches as correct. While the counter-arguments here are not as strong as the ones in the constant branch section, take the statistics gathered from *perl*. In *perl*, 0.3% of all branches executed are classified as infrequent. These branches are predicted with 90.05% accuracy with the smaller branch predictor – significantly higher than that of *gcc*. Furthermore, *perl*'s static footprint is significantly smaller than *gcc*'s – *perl* has only 349 static branches, compared to the 20,687 found in *gcc*. *Perl*, however, with the smaller branch predictor, only shows a 0.05% prediction improvement, from 95% to 95.05%. Even assuming a linear scaling, the estimated improvement comes nowhere near that shown by *gcc* and *go*.

Ijpeg shows similar results. Infrequent branches comprise only 0.8% of *ijpeg*'s executed branches, which are predicted with 90.23% accuracy. In addition, *ijpeg*'s static footprint is somewhat larger than *perl*'s – 701 static branches. The improvement in accuracy totals 0.1%, from 89.77% to 89.87%.

Even the larger branch predictor exhibits a noticeable gain in prediction accuracy. In *go*, prediction accuracy improves from 87.44% to to 88.10%, or a 0.66% gain in accuracy. *Gcc* shows a smaller gain – from 93.86% to 94.01%, or a gain of 0.15%. In the larger branch predictor, *gcc* predicts the infrequently executed branches with a 82.95% accuracy, while *go* is slightly worse at 80.65%.

In the larger branch predictor, *perl* shows a small improvement. What is the more illustrative point in this is that while *perl* improves from 97.14% to 97.17% accuracy, it must be noted that the 0.3% of infrequent branches are already predicted at 99.03%. Simply improving the prediction rate does not necessarily account for the 0.03% boost in improvement, especially since preventing the branches from updating the pattern history tables did not show the same improvement.

Compress and *xlisp* show no improvement. This can be attributed to the fact that infrequently executed branches make up 0.00325% of branches executed in *compress*, and 0.074% of branches in *xlisp*. Furthermore, *compress* and *xlisp* have small static footprints – 243 and 315 branches, respectively – and therefore quite easily fit into even the smaller branch predictor.

6.3 Repeating patterns

Since repeating patterns offer valuable state to the global history register, preventing them from updating would only serve to impede branch prediction instead of help it. For this

reason, tests involving the prevention of repeating patterns from updating the global history were not performed.

6.4 Iterative branches

Iterative branches also contribute useful state to the global history register. Using the same reasoning as with the repeating patterns, iterative branches were also not tested.

7. Measure impact of not updating the pattern history table

In certain cases, especially with programs with large static footprints, a situation that may potentially decrease branch predictor performance occurs when two branches happen to hash into the same second level bimodal counter. As the bimodal counter, used to record pattern history, is useful only for one branch, three situations may occur. The first, and most typical, is that the second branch to hash in destroys the bimodal counter state and decreases branch predictor performance for both branches. The second is that the second branch happens to do nothing – this may occur if the second branch just happens to mirror the behavior of the first. The last situation, and usually least likely, is that the second branch happens to increase in some way, branch predictor performance for both branches.

This project focuses on the first situation – the situation in which a second branch hashes into the same bimodal counter and trashes the branch prediction for both branches. Previous work done indicates that by inhibiting easily predicted branches from updating the pattern history tables – reducing pollution – branch prediction accuracy is increased. In this project, the "marked" branches are diverted to a smaller, specialized predictor, which is assumed to be perfect in its prediction.

7.1 Constant/almost constant branches

As with the previous section, and in line with work done before, constant and almost constant branches benefited the most from the modified branch prediction scheme. As constant and almost constant branches are easily predicted, and may not offer much to the branch predictor, preventing them from polluting the pattern history tables increases the performance of most of the selected benchmarks.

	gcc	compress	perl	go	ijpeg	xlisp
base, 4096	90.99%	90.77%	95.00%	83.75%	89.77%	95.49%
simbwc_nopht, 4096	92.70%	90.77%	96.05%	84.88%	89.87%	95.51%
base, 32768	93.86%	91.81%	97.14%	87.44%	90.78%	96.95%
simbwc_nopht, 32768	94.63%	91.81%	97.32%	88.19%	90.89%	96.95%

Table 3.1 : Branch Prediction Accuracy of Selected Spec95 Benchmarks

Take, for example, *gcc*. With a large static footprint, *gcc* stands to suffer from a lot of collisions in the branch predictor. In its smaller configuration, the branch predictor performs fairly well, achieving a prediction accuracy of 90.99%. When the constant and almost constant branches are prevented from updating the pattern history tables, though, the prediction accuracy increases to 92.7% – an improvement even larger than that seen when protecting the global history register. The larger branch predictor sees a similar gain – from 93.86% to 94.63%, again besting the previous experiments.

In *go*, the smaller branch predictor sees the prediction accuracy increase from 83.75% to 84.88%, an increase of 1.03%, greater than the 0.82% increase seen when filtering the global history register. The larger branch predictor sees a similar gain, from 87.44% to 88.19%, or 0.75% – greater than the 0.26% seen by the previous experiment.

Again, the benefits of isolating constant and almost constant branches decrease as branch predictors get larger – however, notice that the smaller branch predictor almost comes within a percentage point of the large branch predictor in the modified branch prediction scheme, with significantly less hardware. In addition, notice that the branch prediction rates are all beneficial, yet different from the ones achieved when preventing branches from updating the global history register – better, in fact. This would indicate that something other than just counting marked branches which are mispredicted as correct is happening. Moreover, the results are in line with implications from previous work done in this area.

Compress, *ijpeg*, and *xlisp* shardly gain any prediction accuracy at all. *Compress*, in fact, stays the same – no gain whatsoever in both configurations. *Xlisp*, in the smaller branch predictor configuration, only improves 0.02%, whereas in the larger branch predictor it stays constant at 96.95%. It is quite likely that the branch prediction is slightly improved by virtue of counting those mispredicted branches as correct, but what may be happening is that the benefit is seen at thousandths or hundred-thousandths of a percentage point, and are gone unseen by rounding off to the hundredths decimal place.

Ijpeg gains more than the *compress* or *xlisp*, but not by much. In the smaller branch predictor configuration, *ijpeg* only improves 0.1%, from 89.77% to 89.87%. In the larger branch predictor configuration, *ijpeg* gains slightly less – 0.08%, from 90.78% to 90.86%. What should be kept in mind, though, is that *ijpeg*, like *compress* and *xlisp*, have significantly smaller footprints than the *gcc* and *go* and theoretically ought not to benefit as much. Because there are less branches attempting to use the branch predictor, the possibility for aliasing and branches hashing into the same counters is reduced.

Perl offers a slightly differing viewpoint. Unlike *compress*, *ijpeg*, and *xlisp*, *perl* sees some benefit in preventing branches from updating the pattern history tables. The increase, however, differs from *gcc* and *go* in that the improvement is less than that seen in the previous experiment of preventing those branches from updating the global history register. In the smaller branch predictor, *perl* showed a 1.05% increase, from 95% to 96.05% – significant, but considerably less than the 96.49% accuracy seen by previous experiments. In the larger branch predictor, branch prediction increased only 0.18%, from 97.14% to 97.32% – again, less than the 98.04% increase found in inhibiting branches from updating the global history register.

7.2 Infrequent Branches

The second class of branches tested were infrequently executed branches. Note that unlike constant and almost constant branches, infrequent branches actually make up very

little of the executed instruction stream. However, even reducing the pollution in the pattern history tables only slightly can potentially benefit the branch predictor a great deal.

	gcc	compress	perl	go	ijpeg	xlisp
base, 4096	90.99%	90.77%	95.00%	83.75%	89.77%	95.49%
simbwc_nopht, 4096	91.27%	90.77%	95.03%	84.88%	89.87%	95.51%
base, 32768	93.86%	91.81%	97.14%	87.44%	90.78%	96.95%
simbwc_nopht, 32768	94.06%	91.81%	97.14%	88.29%	90.85%	96.96%

Table 3.2 : Branch Prediction Accuracy of Selected Spec95 Benchmarks

Go responded the most favorably to this modified scheme. With the smaller branch predictor, *go* improved 1.29%, while the larger predictor improved 0.85%. In both cases the results bested the results obtained when the branches were being prevented from updating the global history register.

Gcc is another example of this, in the smaller branch predictor configuration. The base prediction rate for *gcc* is 90.99%, but when infrequent branches are prevented from updating the pattern history tables, the accuracy increases to 91.27% – an improvement of 0.28%, or slightly better than preventing branches from updating the global history register. The larger predictor improves as well, from 93.86% to 94.06%, for a gain of 0.2%.

As before, the other four benchmarks showed little, if any, improvement at all. *Compress*, in both configurations, showed no improvement in prediction accuracy. *Ijpeg* and *perl* showed marginal improvement. In the smaller predictor, *ijpeg* improved 0.1% while *perl* improved a mere 0.03%. In the larger predictor, *ijpeg* improved 0.08% while *perl* showed no improvement. *Xlisp* showed similar results – in the smaller predictor, there was no improvement, whereas in the larger predictor, there was only a 0.01% increase.

Similar to the reasoning behind the constant and almost constant branches, *gcc* and *go* stand to gain the most because of their larger static footprints. In addition, infrequently executed branches make up a very small amount of the executed branches to begin with, so,

with the exception of *go*, large gains were not expected. *Compress*, for example, has only 0.00325% of executed branches classified as infrequent.

7.3 Repeating Patterns

The next set of branches to be tested were repeating patterns. As mentioned before, only repeating patterns with fixed *n* and *m* iteration counts were tested. As a side effect, though, to the strict adherence to only fixed *n* and *m* counts is the result that these branches are very regular and thus, somewhat easy to predict. The benefits due to filtering out repeating patterns are therefore slightly smaller. One of the advantages, however, is that repeating pattern branch predictors are much smaller – on the order of the logarithm of base two – than similar local branch predictors. Instead of holding the entire history, only *n* and *m* counters are needed, with perhaps a bias bit to indicate which pattern occurs first.

	gcc	compress	perl	go	ijpeg	xlisp
base, 4096	90.99%	90.77%	95.00%	83.75%	89.77%	95.49%
simbwc_nopht, 4096	91.03%	90.77%	95.07%	83.78%	89.83%	95.49%
base, 32768	93.86%	91.81%	97.14%	87.44%	90.78%	96.95%
simbwc_nopht, 32768	93.88%	91.81%	97.18%	87.45%	90.79%	96.95%

Table 3.3 : Branch Prediction Accuracy of Selected Spec95 Benchmarks

As can be seen from the chart above, gains were minimal across the board. This is due in part to the high regularity of the repeating patterns – *compress*, for example, averages over 99.99% for both of its branch predictors. *Xlisp* is much the same. Moreover, *compress* and *xlisp* have small static footprints, and do not stand to gain very much to begin with.

Gcc, however, shows a marginal gain, as does *perl*. While the larger branch predictor naturally benefits less than the smaller branch predictor, the overall gain is noticeably less

than the results from the other classes of branches. The benefits, then, appear to be more located in the area savings.

Ijpeg shows a moderate improvement in the smaller branch predictor, but this gain is diminished to the point of almost nothing in the larger branch predictor. Like *gcc*, this may be due to the fact that *ijpeg*'s static footprint is somewhat larger than *compress*, *xlisp*, or *perl*. *Perl* is an exception in that it has been responding quite well to the newer branch prediction schemes almost across the board.

7.4 Iterative Branches

The final branches to be tested with this branch prediction scheme were the iterative ones. As iterative branches should contain a good deal of worthwhile state information for the branch predictor, the only benefit would be in reducing the pollution in the pattern history tables. Iterative branches are very regular to begin with; keeping the pattern history, is not needed, especially with a loop counter branch predictor. Even with fixed length iterative branches, bimodal predictors are guaranteed at least one miss every time the pattern repeats. Local history predictor are better, eliminating the miss for most short iteration counts. Loop counter branch predictors, however, are even better than both in that they eliminate the miss, making them more accurate than bimodal predictors, and can predict iterative branches with much longer iteration counts than local history predictors.

It is important to note that in this project, the focus was on branches with fixed iteration counts – branches that have variable iteration counts, while more numerous than those with fixed iteration counts, are much more difficult to predict, even for loop counter branch predictors. Since there is a significant deal of unpredictability in the length of the iteration counts between patterns, only fixed iteration count branches were looked at. Remember that iterative branches perform fairly well already; attempting to predict the

wildly variable iteration count branches with a less sophisticated loop counter predictor may, in fact, be detrimental to performance. Preliminary results confirmed this suspicion.

	gcc	compress	perl	go	ijpeg	xlisp
base, 4096	90.99%	90.77%	95.00%	83.75%	89.77%	95.49%
simbwc_lp, 4096	91.10%	90.77%	95.20%	84.07%	90.73%	95.49%
base, 32768	93.86%	91.81%	97.14%	87.44%	90.78%	96.95%
simbwc_lp, 32768	93.95%	92.62%	97.29%	87.67%	91.61%	96.96%

Table 3.4 : Branch Prediction Accuracy of Selected Spec95 Benchmarks

Ijpeg saw the largest increase in branch prediction accuracy. The larger branch predictor saw an increase from 90.78% to 91.61%, while the smaller branch predictor improved even more, from 89.77% to 90.73%. Both of these increases came despite the fact that iterative branches were already being predicted at near 99% to begin with. Also posting a moderate gain was *perl*, which saw its prediction accuracy rise, but not as much as in *ijpeg*. In the smaller branch predictor, *perl* improved 0.2%, while in the larger branch predictor, the improvement was only 0.15%.

Go produced more supporting evidence as well. The smaller branch predictor responded quite well, improving prediction accuracy from 83.75% to 84.07% – an increase of 0.32%. The larger predictor saw a somewhat smaller increase – from 87.44% to 87.67%, or an improvement of 0.23%

Gcc saw a similar, but less pronounced gain. *Gcc*'s smaller predictor saw an increase of 0.11%, while the larger predictor improved 0.09%. One would expect, though, that a benchmark with such as *gcc*, with a large static footprint, would benefit more than the results would show. Fixed iteration count branches only made up 0.4% of *gcc*'s code, however, and much less than the 18.18% of executed branches, which included both variable iteration counts as well as fixed iteration counts.

Compress and *xlisp* showed almost no gain at all. Iterative branches in *compress* were already performing well, and preventing them from updating the pattern history table appears to have no effect. The data shows somewhat that the iterative branches in *compress*

contributed more to the state of the machine than constant and almost constant branches did. *Xlisp* was similar, except the larger branch predictor gained 0.01%.

It is also true one of the bigger benefits of applying the loop counter prediction to iterative branches is the savings in area – the increase in branch prediction accuracy for iterative branches may, in fact, be only canceling out the decrease in the general branch predictor. However, with the results shown above, it can easily be seen that moving iterative branches out, at worst, does nothing to the existing branch prediction, and at best, such as in *ijpeg*, improves branch prediction a great deal.

8. Measure impact of isolating branches from the branch predictor

The final modified branch prediction scheme removes marked branches from the branch predictor update phase completely. By isolating marked branches, the branch predictor now only has to deal with a smaller set of branches. If the marked branches are not contributing worthwhile state to the branch predictor, than isolating them should have little detrimental effect. On the contrary, by allowing the branch predictor to store more of the important branches in its history, prediction accuracy should improve. Rather than add more hardware to hold more history, this prediction scheme intelligently reduces the amount of branches that need to be stored.

8.1 Constant/almost constant branches

The first class of branches tested were the constant and almost constant branches. The previous two prediction schemes have shown that constant and almost constant branches do not offer much worthwhile state to the branch predictor and can thus be isolated from the branch predictor update phase. The prediction accuracy should rise; whether or not this accuracy is greater than preventing branches from updating the global history or the pattern history is not necessarily guaranteed.

	gcc	compress	perl	go	ijpeg	xlisp
base, 4096	90.99%	90.77%	95.00%	83.75%	89.77%	95.49%
simbwc_i, 4096	92.78%	91.85%	97.00%	84.97%	89.93%	95.49%
base, 32768	93.86%	91.81%	97.14%	87.44%	90.78%	96.95%
simbwc_i, 32768	94.75%	92.62%	98.13%	88.26%	90.86%	96.96%

Table 4.1 : Branch Prediction Accuracy of Selected Spec95 Benchmarks

Perl showed the most dramatic rise in branch prediction accuracy. In the smaller branch predictor configuration, *perl* performs at 95% accuracy – when constant branches are isolated from the branch predictor, though, prediction accuracy rises to 97%. What is

remarkable about this is that constant branches – *perl* has no almost constant branches – are already predicted at 98.23%. Better yet, with this modified prediction scheme, the smaller branch predictor achieves almost the same prediction accuracy as the larger branch predictor. The larger branch predictor benefits as well – an improvement of 0.99%, from 97.14% to 98.13%.

Mirroring this behavior is the *gcc* benchmark. *Gcc*, with the smaller branch predictor, improves 1.79% – more than the previous two schemes achieved. In fact, with considerably less resources, the smaller branch predictor, at 92.7%, by isolating constant and almost constant branches, almost comes within 1% of the larger branch predictor, which normally achieves 93.86% accuracy. For reference, the larger branch predictor improves to 94.75%, a 0.89% increase.

To better correlate the improvement in branch prediction accuracy, one must examine the other benchmark with a large static footprint – *go*. With the smaller branch predictor, *go* showed an increase of 1.24% in prediction accuracy, while the larger branch predictor only improved 0.72%. What is interesting to note in this case is that isolating the branches completely did not show as much improvement as only preventing branches from updating the pattern history table.

It is easy to see, then, that larger branch predictors gain much less than the smaller branch predictor. As mentioned before, this is due primarily to the fact that larger branch predictors have an easier time dealing with programs with a large static footprint – for reference, 20,687 static branches in *gcc*, and 3,450 static branches in *go*. It would be interesting, then, to examine the results from programs that have a significantly smaller footprint – p*erl* is a special case; in all three of the schemes it responded quite positively to reducing the amount of branches accessing parts of the branch predictor.

Compress, for example, has the smallest footprint of all the six selected benchmarks – only 243 branches. Unlike *gcc* and *go*, however, the smaller branch predictor did not gain a large amount more than the larger branch predictor. In the case of *compress*, the smaller

branch predictor improved 1.08%, whereas the larger branch predictor gained 0.81% – a stark contrast to *gcc* and *go*, where the larger branch predictor gained approximately half of what the smaller branch predictor did.

Xlisp produced similar results. The smaller branch predictor improved 0.12%, while the larger branch predictor followed closely with a 0.11% improvement. Seeing as *compress* and *xlisp* has similar static branch counts in their code – 243 for *compress* and 315 for *xlisp* – the result is not especially surprising.

The final benchmark, *ijpeg*, had the most unusual results. Unlike the other five benchmarks, the larger branch predictor actually improved more than the smaller benchmark. The overall improvement, however, was still fairly close – 0.4% for the larger predictor and 0.33% for the smaller predictor.

8.2 Infrequent branches

Following the constant and almost constant branches are the infrequent branches. As before, if a branch is infrequent, it may not be contributing worthwhile information to the overall state of the machine. In this case, the branch would not impact the branch predictor accuracy if isolated from the branch predictor itself, but even better, the removal of the branches may in fact serve to lessen pollution in the pattern history tables and enhance the history contained in the global history register.

	gcc	compress	perl	go	ijpeg	xlisp
base, 4096	90.99%	90.77%	95.00%	83.75%	89.77%	95.49%
simbwc_i, 4096	91.27%	90.77%	95.05%	84.97%	89.87%	95.61%
base, 32768	93.86%	91.81%	97.14%	87.44%	90.78%	96.95%
simbwc_i, 32768	94.05%	91.81%	97.17%	88.20%	90.89%	97.06%

Table 4.2 : Branch Prediction Accuracy of Selected Spec95 Benchmarks

An interesting situation occurred with the *gcc* and *go* benchmarks. While they achieved the highest performance improvement, both shared an interesting fact – when tested with the larger branch predictor, the highest accuracy was not in isolating the branches completely, but actually in preventing the branches from updating the pattern history table. *Gcc*, for example, predicted branches with 94.05% accuracy, a mere 0.01% behind its best mark. *Go* showed similar results, posting at 88.20%, or 0.09% behind. In the smaller branch predictor, *gcc* performed equally as well when completely isolated from the branch predictor, at 92.07%, while *go* achieved an accuracy of 84.97% – slightly less than the 85.04% seen when reducing pollution in the pattern history table alone.

From the above results, it appears that infrequent branches – which are based on their frequency of execution rather than their overall behavior – actually do add some worthwhile state information to the branch predictor. The effects are more pronounced as the branch predictor gets larger; remember that the larger branch predictor has a shift register considerably wider than that of the smaller branch predictor, at eight bits as opposed to five. Being that the larger branch predictor holds a longer history to begin with, the effects of reducing pollution in the pattern history tables is offset somewhat by the negative impact of removing branches with at least some important information about the overall state of the machine.

It is worthwhile to note that the infrequent branches may span the whole range of behavior – from constant, easy to predict branches to wildly varying, unpredictable branches. Furthermore, being classified as infrequent does not assume that these branches are sprinkled sparsely throughout the program. It is quite possible that some of the infrequent branches may be concentrated in a loop, where they are not executed enough to escape the infrequent classification, but are important enough to have a large effect on the branches executed in the near time frame. Were the branches spread somewhat widely throughout the program, it is likely that removing them from the global history register would impact the branch prediction accuracy a great deal less.

Ijpeg and *perl* show a much smaller improvement – still greater than *compress* and *xlisp*, which showed virtually no improvement at all. This is due in part to the fact that *ijpeg* and *perl* are, in terms of volume of infrequent branches, right in the middle between *gcc* and *go* – which have a larger volume of infrequently executed branches – and *compress* and *xlisp*, which have almost none at all. It is also important to note that *gcc* and *go* have considerably larger static footprints than the remaining four benchmarks. Consequently, the *gcc* and *go* should benefit a good deal more than the rest.

Ijpeg, for example, showed a marginal improvement over the previous two branch prediction schemes. In the smaller branch predictor, *ijpeg* improved only 0.04%, and only 0.01% over its previous best. The larger branch predictor produced similar results – an increase of 0.08%, but only 0.01% over the other two branch prediction schemes.

Perl produced much of the same results. Like *ijpeg*, the improvement over the normal branch prediction scheme was minimal – 0.05% in the smaller branch predictor, and 0.03% in the larger one. Worse yet, isolating branches shows no noticeable improvement over simply preventing branches from updating the pattern history tables – both achieved the same branch prediction accuracy, 95.05% in the smaller branch predictor and 97.17% in the larger branch predictor.

Compress and *xlisp* saw no improvement at all. As *compress*, for example, has only 0.00325% of its executed branches as infrequent, the impact either way – beneficial or detrimental – will not influence overall branch prediction accuracy. For all three prediction schemes, *compress* did not respond whatsoever. *Xlisp* was much the same way. As opposed to the previous two branch prediction schemes – which showed no improvement – isolating infrequent branches actually produced a 0.01% improvement. This improvement, though, is too minimal in comparison to the effort required.

8.3 Repeating patterns

Isolating branches from the entire branch predictor involves preventing the marked branch from updating the global history register; since repeating patterns offer valuable state to the global history register, this would only serve to impede branch prediction instead of help it. For this reason, tests involving the prevention of repeating patterns from updating the global history were not performed.

8.4 Iterative branches

In the same reasoning as in repeating patterns, isolating branches would hurt the branch predictor instead of help. Therefore, iterative branches were not tested in this portion of the experiment.

9. Summary

Based on the available data, hybrid branch prediction seems to be a very real approach to designing new branch predictors. Given the data drawn from constant and almost constant branches, coupled with the research previously done in this field, a convincing argument can be made for designing the branch predictors around branch classification. For the most part, this project focused on the impact of several classes of branches, and the resulting increase or decrease in prediction accuracy.

In this project, two branch predictors were tested – both of which were two level adaptive branch predictors, and one of which was considerably larger than the other. In some cases, the smaller branch predictor, by simply filtering out certain classes of branches to a "perfect" predictor, was able to almost achieve the performance of the much larger predictor. Assuming the "perfect" predictor does not consume very large amounts of hardware, a hybrid design would require less hardware yet perform at almost the same rate. It may be implied, then, that a hybrid design with equal amounts of hardware would outperform the standard generalized design.

10. Conclusions

Hybrid branch prediction, with the same amount of hardware, improves branch prediction beyond that of a traditional all-in-one branch predictor. Even in the simple form shown in this paper, implementing a hybrid in which one predictor handles easy to predict, constant branches while the other is a traditional branch predictor shows a noticeable increase in improvement. With considerably less hardware, the smaller hybrid almost achieves the same prediction accuracy as the larger generalized two level branch predictor.

The major improvement comes from taking branches which may not offer a great deal to the branch predictor state but may be polluting the pattern history tables or interfering

in the global history register. By taking these branches and diverting them to their own specialized predictor, not only does their branch prediction accuracy increase, but in cleaning up the pattern history tables and global history register, so does that of the general branch predictor. Furthermore, not just any class of branch may be diverted to its own specialized predictor.

Four major classes of branches were examined in this project – constant and almost constant, infrequently executed, repeating patterns, and iterative branches. The largest gain was seen in specially treating the constant and almost constant branches. As these branches are extremely regular and often times infrequently executed, as in the case of one-time error code checks, these branches do not offer much to the branch predictor. Instead, especially with limited hardware, they tend to jam valuable resources better used for more difficult to predict branches.

Infrequently executed branches saw a much smaller gain. In part, this may be due to the fact that infrequently executed branches are not behavior based, but simply frequency based. Furthermore, branches classified as infrequent were of such small amounts of the total number of branches that their impact was likely to be small, and moreover mostly seen only with the smaller branch predictors.

Repeating patterns and iterative branches saw a smaller gain as well. Since repeating patterns and iterative branches are contributing worthwhile state to the branch predictor, they were only tested in the scheme that prevents branches from updating the pattern history table. As such, minute gains were seen in reducing the pollution in the pattern history tables somewhat.

Of the two branch predictors, the smaller branch predictor saw the largest benefit. This is to be expected, the smaller branch predictor has less resources available to it, and by filtering out branches with no worthwhile state to contribute, the branch predictor can concentrate on a smaller set of branches. In some cases, the branch predictor was able to achieve within approximately 1% of a much larger branch predictor.

The filtering, however, must be carefully examined. It is not the case that any random classification of branches can be chosen. In this paper, the branches that responded the most favorably were those based on an easy to identify behavior – for example, constant and almost constant branches. Those based on other criteria, such as frequency of execution – infrequent branches – did not show as much improvement when removed from updating certain parts of the instruction stream. Furthermore, the idea that marked branches should be diverted to small, simple predictors places an upper limit on the complexity – a loop counter, for example, is much easier to implement than a second, albeit small, two level adaptive predictor.

Finally, branch classification can serve as a very real possibility to implementing a hybrid branch predictor. Through judicious use of profiling and additional "hints" provided by the instruction set architecture, there already exists at least one method of static selection. While this may not be the most effective way, the feasibility of implementing a hybrid branch predictor is already not very far off.

11. Recommendations for future research

In this paper, the results discussed involved a simplistic "perfect" predictor as well as a more generalized two level adaptive predictor. In addition, the selection of whether branches were to enter the perfect predictor or the general predictor was already determined – in real cases, this does not always happen. Future research may look at how branches can be routed to the appropriate predictor, and the impact on performance this may have.

Furthermore, "perfect" predictors do not exist. Another avenue of research may involve looking at how already existing predictors can be combined, such as bimodal and two level adaptive predictors, or combinations of more than two branch predictors. This project was limited in that if the branch class was not marked, it was thrown into the general two level adaptive predictor. In a more complex and higher performing hybrid, this may

mean four or five branch predictors, each handling a special class of branch, with one default branch predictor handling the more difficult branches.

Finally, the hybrid branch prediction scheme does not allow for branches to “float” between predictors. If a branch predictor, originally thought to perform well with a certain branch, is performing quite badly, it will continue to perform quite badly. By allowing branches to float between branch predictors, if a branch was originally marked incorrectly, some performance can still be salvaged, especially if another branch predictor quickly shows its ability to correctly predict that branch.

Appendix A : Annotated Bibliography

Augustus Uht, Vijay Sindagi, and Sajee Somanathan

Augustus Uht, Vijay Sindagi, and Sajee Somanathan, "Branch Effect Reduction Techniques," Computer, 1997, pp. 71-81

Uht, Sindagi, and Somanathan give a solid overview of the current state of branch prediction techniques. In particular, they concentrate on techniques used in commercial processors. They have also included possible avenues of future research – techniques such as hybrids and multiscalar branch prediction as well as announced techniques such as VLIW.

Doug Burger, Todd M. Austin

Doug Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," 1997

The SimpleScalar Tool Set is the basic manual for using the SimpleScalar microprocessor execution simulator. The manual includes installation and basic operation. In addition to the processing core, the SimpleScalar software packages includes many default, basic simulators such as branch prediction (sim-bpred) and out-of-order execution (sim-outorder).

Eitan Federovsky, Meir Feder, Shlomo Weiss

Eitan Federovsky, Meir Feder, Shlomo Weiss, “Branch Prediction Based on Universal Data Compression Algorithms,” Tel Aviv University, 1998

In this paper, Federovsky, Feder, and Weiss suggest applying data compression algorithms to the field of branch prediction. They focus on two compression algorithms – prediction by partial matching and context tree weighting. Finally, they produce as evidence simulation results of their work.

I-Cheng K. Chen, John T. Coffey, Trevor N. Mudge

I-Cheng K. Chen, John T. Coffey, Trevor N. Mudge, “Analysis of Branch Prediction via Data Compression,” University of Michigan, Ann Arbor, 1996

Another paper relating branch prediction and data compression, Chen, Coffey, and Mudge address the theoretical basis behind branch prediction. Specifically, they concentrate on proving that two-level branch predictors have theoretical grounding in the prediction by partial matching algorithm used in data compression.

John L. Hennessy, David A. Patterson

John L. Hennessy, David A. Patterson, "Computer Architecture : A Quantitative Approach", Morgan Kaufmann Publishers, Inc., 1996

Possibly the premier book in computer architecture, Hennessy and Patterson provide an excellent overview of the field of computer architecture, with a focus on microprocessors. Their section on branch prediction provides fundamentals for the beginning of study in branch predictors and speculative execution. Their updated second edition was published in 1996.

Kevin Skadron

Kevin Skadron, "Characterizing and Removing Branch Mispredictions." Ph.D. thesis, Princeton University, June 1999.

In this thesis, Skadron shows that branch prediction is the most important factor in determining processor performance. Using the HydraScalar software suite, Skadron demonstrates the effects of various branch predictor configurations. In addition, conflicts in the state table is not the only contributor to branch predictions, but other less-studied sources. He then proposes solutions for reducing three misprediction types : alloying, speculative update with fixup, and multipath execution.

Linley Gwennap

Linley Gwennap, "Intel Discloses New IA-64 Features." Microprocessor Report, March 8, 1999, pp. 16-19.

Providing a high-level overview of Intel's IA-64 architecture, Gwennap describes the techniques used in Intel's Itanium (formerly Merced) architecture. Of particular interest to this proposal is the section on compiler-assisted branch prediction techniques. In addition, Gwennap details how the Itanium incorporates both hardware and software branch prediction techniques.

Linley Gwennap, "Digital 21264 Sets New Standard," Microprocessor Report, October 28, 1996

Gwennap's article on the Compaq (formerly Digital) 21264 microprocessor is of interest to this proposal mainly for his section on Digital's selector branch predictor. The selector branch predictor is comprised of two branch predictors – one based on local history and another based on global history. Gwennap describes the interaction and efficiency of the two branch predictors working together.

Marius Evers, Sanjay J. Patel, Robert S. Chappell, Yale N. Patt

Marius Evers, Sanjay J. Patel, Robert S. Chappell, Yale N. Patt, "An Analysis of Correlation and Predictability : What Makes Two-Level Branch Predictors Work," University of Michigan, Ann Arbor, 1998

Evers, Patel, Chappel, and Patt investigate the behavior of branches – why branches are predictable. They attempt to provide understanding on why branches are predictable, and why current branch predictors cannot take advantage of this predictability. Specifically, they concentrate on the two-level adaptive branch predictor.

Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt

Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt, "Branch Classification : a New Mechanism for Improving Branch Prediction Performance," University of Michigan, Ann Arbor, 1998

In this paper, Chang, Hao, Yeh, and Patt explore a new technique in improving branch prediction – branch classification. By associating a type of branch with a specialized branch predictor, they show that their hybrid branch predictor surpasses any previous designed. They implement and explore several different types of designs and investigate each design's performance. Finally, they show that their best branch predictor involved statically and dynamic predictor selection.

Po-Yung Chang, Marius Evers, and Yale N. Patt

Po-Yung Chang, Marius Evers, and Yale N. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," University of Michigan, Ann Arbor, 1997

In this paper, Chang, Evers, and Patt describe the improvements in branch prediction accuracy by removing easy to predict branches from entering the pattern history table. Because easily predicted branches may pollute the table, Chang, Evers, and Patt show the performance improvement that results from preventing easy to predict branches from updating the pattern history table, thus creating interference.

Scott McFarling

Scott McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, Digital Corporation, 1993

McFarling, in his technical note for Digital Corporation, explains in detail the effects of combining branch predictors in Digital's 21264 microprocessor. The 21264, which incorporates both a local history and a global history branch predictor, uses the combined approach to great success. Here, McFarling describes the implementation and effectiveness of the 21264 technique.