Variable Length Path Branch Prediction

Variable Length Path Branch Prediction

Jared Stark Marius Evers Yale N. Patt

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{starkj,olaf,patt}@eecs.umich.edu

Abstract

Accurate branch prediction is required to achieve high performance in deeply pipelined, wide-issue processors. Recent studies have shown that conditional and indirect (or computed) branch targets can be accurately predicted by recording the path, which consists of the target addresses of recent branches, leading up to the branch. In current path based branch predictors, the N *most recent target addresses are hashed together to form an index into a table, where* N *is some fixed integer. The indexed table entry is used to make a prediction for the current branch.*

This paper introduces a new branch predictor in which the value of N *is allowed to vary. By constructing the index into the table using the last* N *target addresses, and using profiling information to select the proper value of* N *for each branch, extremely accurate branch prediction is achieved. For the SPECint95 gcc benchmark, this new predictor has a conditional branch misprediction rate of 4.3% given a 4K byte hardware budget. For comparison, the gshare predictor, a predictor known for its high accuracy, has a conditional branch misprediction rate of 8.8% given the same hardware budget. For the indirect branches in gcc, the new predictor achieves a misprediction rate of 27.7% when given a hardware budget of 512 bytes, whereas the best competing predictor achieves a misprediction rate of 44.2% when given the same hardware budget.*

1 Introduction

Accurate branch prediction is required to achieve high performance in today's deeply pipelined, wide-issue processors. As the pipeline depths and the issue rates increase, the amount of speculative work that must be thrown away in the event of a branch misprediction also increases. Thus, tomorrow's processors will require even more accurate branch prediction to deliver their potential performance.

Recent studies [3, 6, 10, 16] have shown that conditional and indirect (or computed) branch targets can be accurately predicted by recording the path leading up to the branch. The path consists of the target addresses of the branches leading up to the current branch. In current path based branch predictors, the N most recent target addresses are hashed together to form an index into a table, where N is some fixed integer. The indexed table entry is used to predict either the direction (for conditional branches) or target address (for indirect branches) of the current branch.

This paper introduces a new branch predictor in which N indices are formed using N hash functions. The first index, generated by hash function 1, is a function of the most recent target address. The second index, generated by hash function 2, is a function of the two most recent target addresses.... The Nth index, generated by hash function N, is a function of the N most recent target addresses. For each branch, profiling information is used to select the hash function whose result is to be used as the index into the table. By judiciously selecting the appropriate hash function for each branch, much higher branch prediction accuracy can be achieved than with any other known single predictor. For the SPECint95 gcc benchmark, this new predictor has a conditional branch misprediction rate of 4.3% given a 4K byte hardware budget. For comparison, the gshare predictor, a predictor known for its high accuracy, has a conditional branch misprediction rate of 8.8% given the same hardware budget. For the indirect branches in gcc, the new predictor achieves a misprediction rate of 27.7% when given a hardware budget of 512 bytes, whereas the best competing predictor achieves a misprediction rate of 44.2% when given the same hardware budget.

The paper is divided into six sections. Section 2 describes the related work. Section 3 introduces the variable length path prediction algorithm. Section 4 discusses some of the implementation considerations. Experimental results are presented and discussed in Section 5, and some concluding remarks are provided in Section 6.

2 Related Work

Various branch prediction strategies have been studied to improve prediction accuracy. Yeh and Patt proposed the Two-Level Adaptive Branch Predictor [22, 23] (referred to henceforth as the 2-level predictor) which uses two levels of history to make branch predictions. The first level history records the outcomes of the most recently executed branches and the second level history keeps track of the more likely direction of a branch when a particular pattern is encountered in the first level history. To record the first level history, the 2-level predictor uses one or more k-bit shift registers, or branch history registers, to record the outcomes of the most recent k branches. In the GAs version of the 2-level predictor, there is one global branch history register that records the k most recent branch outcomes. In the PAs version of the 2-level predictor, there is one branch history register for each branch recording the k most recent outcomes of that branch. The second level history consists of one or more arrays of 2-bit saturating up-down counters, called Pattern History Tables, to keep track of the more-likely direction for branches. The lower bits of the branch address select the appropriate Pattern History Table (PHT) and the value in the branch history register (BHR) selects the appropriate 2-bit counter to use within that PHT.

McFarling [14] introduced gshare, improving the GAs 2-level predictor by XORing the global branch history with bits of the branch address to generate the index into the PHT. This results in better utilization of the PHT, reducing interference and improving performance. Gshare has since been considered the benchmark of choice for single-scheme branch predictors. It is used as the baseline for comparisons of conditional branch predictors in this paper. McFarling also introduced the concept of hybrid branch predictors. A hybrid branch predictor consists of two or more component branch predictors and a mechanism to select which of these to use for each branch.

Young and Smith [25] introduced the difference between pattern histories, containing branch outcomes, and path histories, containing branch addresses. They gave examples of branches being predictable using a path history but not with a pattern history. They also introduced a transformation using code duplication and profile generated path information to transform a program for better static prediction accuracy.

Chang et. al. [4] proposed a hybrid predictor consisting of two GAs predictors with different history lengths using a shared set of pattern history tables. They showed that this predictor narrowly outperformed gshare for the SPECint92 benchmarks. They also proposed branch classification, a mechanism for letting the compiler select the predictor to use for each branch.

Nair [16] proposed a dynamic path based predictor similar to the global version of the 2-level predictor. Instead of shifting branch outcomes into the branch history register, Nair's predictor would shift q bits of the branch target address into the history register. This has the advantage of being able to represent the path, albeit imperfectly. It has the disadvantage that information from fewer branches could be captured in the history.

Jacobson et. al. [10] proposed a path based mechanism for predicting indirect branches in the multiscalar architecture. Their correlated task target buffer achieved substantial improvements in indirect branch prediction on the 5 SPECint92 benchmarks they used.

Chang, Hao and Patt [3] proposed several mechanisms for applying the 2-level prediction algorithm to the prediction of indirect branches. They showed that a history based predictor dramatically improved prediction accuracy over a branch target buffer based predictor. The study included both pattern and path based predictors, and did not conclude which of the two was better. The path and pattern based predictors of the "tagless" variety from that study are used as the baseline for comparisons of indirect branch predictors in this paper.

Tarlescu et. al. [21] proposed variations of the GAs 2-level predictor and the gshare predictor in which the number of pattern history bits used to generate an index into the PHT(s) was not fixed. For each static branch, the compiler used profiling information to determine the number of history bits that should be used to predict the branch. These variations significantly outperformed the predictors in which the number of history bits was fixed.

Driesen and Hölzle [6] explored a large space of path based indirect branch predictors. In their experiments, a global path history was shown to be better than per-address (i.e., per branch address) path histories. They also introduced a hybrid predictor where both components used global path histories but each component used a different length history.

Juan et. al. [12] proposed variations of global pattern history based prediction schemes in which the number of pattern history bits used to generate an index into the PHT(s) was not fixed. At regular intervals, the hardware selected the number of history bits to be used for making predictions. All predictions made during an interval were made using the number of history bits selected by the hardware at the beginning of the interval. The advantage of these variations is that the number of history bits used dynamically adapts to the code.

3 The Prediction Algorithm

3.1 Overview

Figure 1 shows the basic structure of a variable length path predictor. One of these predictors predicts either conditional or indirect branches, but not both conditional and indirect branches. The predictor uses two levels of history to make predictions.

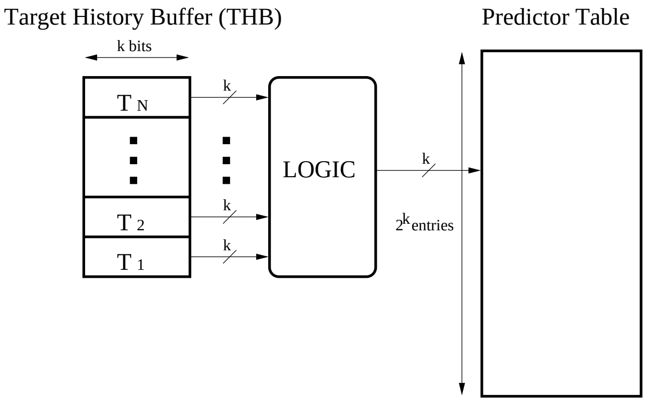

Figure 1: Structure of a Variable Length Path Predictor

The first level of history, stored in the Target History Buffer (THB), consists of N target addresses from the most recently encountered branches. We will use T_X to refer to the Xth most recent target address stored in the THB. Only the target addresses of conditional branches, indirect branches, and (possibly) returns are stored in the THB. Target addresses are compressed into k bits before being stored in the THB. In this study, we compressed the target addresses by simply discarding the higher order bits. The target addresses in the THB are used to generate a k bit index. The index is used to access a table containing the second level of history.

Throughout this paper, the table containing the second level of history will be called the predictor table. Each predictor table entry records the behavior of branches using the given entry. If the predictor predicts conditional branches, the entry records whether or not the branch was taken. If the predictor predicts indirect branches, the entry records the target address of the branch. A prediction is based on the past behavior of the branches that used the same table entry as the entry used by the branch being predicted.

For conditional branches, we used a 2-bit saturating up-down counter for each entry in the predictor table. The counter was incremented when a branch was taken, and decremented when the branch was not taken. A branch was predicted taken if the counter was greater than or equal to two. For indirect branches, we used a register for each entry in the predictor table. The register was large enough to hold one target address. The target address of an indirect branch was written into the register after the target address had been computed. The predicted target address for an indirect branch was simply the value stored in the register at the time of the prediction.

Figure 2 shows how the index into the predictor table is generated. A path of length X ($PATH_X$) consists of the X most recent target addresses in the THB, i.e., $PATH_X$ consists of T_1, T_2, ..., T_X. For a THB that holds N target addresses, N different hash functions (HF_1, HF_2, ..., HF_N) are used to create N predictor table indices (I_1, I_2, ..., I_N). In our experiments, we used a THB that could hold at most 32 target addresses so there were 32 hash functions. Each hash function HF_X uses all target addresses in $PATH_X$ to create the index I_X. In Figure 2, HF_2 uses all the target addresses in $PATH_2$, i.e., the target addresses T_1 and T_2, to create I_2. For each branch, a single hash function is selected. The result of the selected hash function is used as the index into the predictor table. The hash function is selected by either the compiler, the hardware, or some combination of the two.

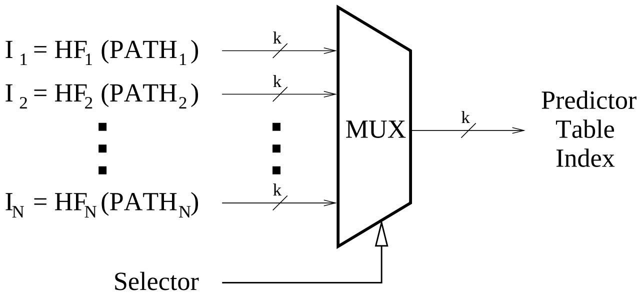

Figure 2: Predictor Table Index Generation

Note that if it is too expensive to implement all N hash functions, a subset of these hash functions may be implemented at reduced benefit. In the discussion above, we assumed that there were N hash functions for a THB that holds N target addresses. However, in a real implementation, the number of hash functions might be less than the number of target addresses that can be stored in the THB. For example, for a THB that can hold at most 32 target addresses, you might implement only the hash functions HF_1, HF_2, HF_4, HF_8, HF_{16}, and HF_{32}.

3.2 Recording the Path

The first level of history, stored in the THB, records the path leading up to the current branch. The path consists of the target addresses from the most recently encountered branches. The goal of constructing the first level history is to accurately record the longest path leading up to the next branch to be predicted. Given that the size of the THB is limited, a target address should not be stored in the THB unless it provides useful information about the path. The target address of an unconditional branch does not provide useful information about the path, so it should not be stored. If the target address of the branch that resulted in a subroutine's call is in the THB, the target address of the corresponding return does not provide any useful information and should not be stored. On the other hand, if the target address of the branch that resulted in a subroutine's call is not in the THB, the target address of the corresponding subroutine return does provide useful information and should be stored. With a THB that holds 32 target addresses, we have found that the branch prediction accuracy does not strongly depend on whether or not the target addresses of returns are stored in the THB. In our experiments, we do not store the target addresses of returns.

3.3 The Hash Functions

Each hash function must combine all the target addresses in a path to create an index. One obvious way to do this is to simply XOR all the target addresses in a path together. However, the order in which the target addresses were encountered will not be encoded in the index if this is done. To preserve the information about order, each target address is rotated before it is XORed. Recall that for a k bit index, k bits are stored for each target address in the THB. Each target address is rotated as a k bit number. The most recent target, T_1, is rotated by 0 bits. T_2 is rotated by 1 bit. T_3 is rotated by 2 bits....

3.4 Selecting the Particular Hash Function to Use

The hash function can be selected by either the hardware, the compiler, or some combination of the two.

If the hardware is involved in the selection, run-time information can be incorporated into the decision. Storage structures are added to the branch predictor that record how accurately the hash functions have predicted each past branch. As with 2-level predictors [23], this information may be recorded either per branch address, per branch set, or globally. For each branch, the hardware uses the information in the storage structures to dynamically select the hash function that has provided the highest accuracy in the past. The advantage of using the hardware to select the hash functions is that higher prediction accuracy can be achieved by using run-time information. The disadvantage is that extra storage structures, which occupy precious die area, are needed to keep track of this run-time information.

If the compiler selects the hash function, there must be some way of communicating this information to the branch prediction hardware. We assume that this information will be communicated via the Instruction Set Architecture (ISA). In this paper, we use profiling information to determine the hash function numbers. For the cases in which profiling is deemed too expensive, the compiler can use a default value. The default value is used for every branch in the program. (Actually, there is one default value for conditional branches, and one default value for indirect branches.) The default value specifies the hash function that provides the highest branch prediction accuracy for the average program. In our experiments, we show that even when this default value is used, the resulting prediction accuracy is still higher than that of competing predictors.

Dean et. al. [5] presented an unobtrusive way to gather profiling data. With this technique, the processor's state is randomly sampled and then squirreled away for later processing. For a variable length path predictor, the contents of the THB could be sampled after some random number of branches. Each sample of the THB would provide information about the behavior of the last N branches that stored their target addresses in the THB. When the processor is idle, the hash function numbers for branches could be calculated (or recalculated) based on the contents of the sampled THBs. With the help of the operating system, the program executables could be modified to reflect the latest profiling information. With this approach, the hash function number assigned to a branch can be changed later. Therefore, the hash function number assigned by the compiler may not be so critical. Data used by programs during this form of profiling is truly reflective of the user's data.

3.5 Our Profiling Heuristic

In our experiments, we used profiling to determine the hash function numbers. The heuristic we used determines the hash function numbers using two steps. In the first step, several candidate hash function numbers are selected for each static branch. The goal of this first step is simply to select, for each static branch, the candidates that will provide the highest prediction accuracy for the branch in question. In the second step, one of the candidates was selected for each branch. The goal of this second step is to select the *set* of candidates, with one candidate for each static branch, that will provide the highest prediction accuracy for the entire program. All static branches not exercised during profiling are assigned the number of the hash function that provides the highest prediction accuracy for the branches that were profiled.

The first step was performed by simulating N fixed length path branch predictors on the program being profiled, with one branch predictor for each of the N hash functions used by the variable length path predictor. Each branch predictor used its corresponding hash function to create an index into a predictor table. This predictor table was only used by the one hash function. That is, there was one hash function per predictor table, and one predictor table per hash function. For each static branch, a record was kept of how many times the branch was correctly predicted by each of the N branch predictors. At the end of the profile run, the candidate hash function numbers were selected. In our experiments, we selected three candidates per static branch. The candidates corresponded to the three branch predictors that provided the greatest number of correct predictions for the branch in question.

In the first step, N hash functions use N predictor tables. In the variable length path predictor, N hash functions use one predictor table. Two branches that did not use the same predictor table entry during the first step may end up using the same predictor table entry in the variable length path predictor. This is called branch interference [20, 24]. The second step is used to reduce this interference.

The second step is performed by simulating one variable length path predictor on the program being profiled. It is iterated several times. On each iteration, a set of candidates is selected and then tested. The result of this test will be used by the next iteration to select the next set of candidates. The set of candidates is selected by choosing one candidate for each branch. The candidate chosen is the candidate that has so far provided the fewest number of mispredictions. In order to do this, a record of how many times the branch was mispredicted by each candidate is kept across the iterations. Note that if the record is initially set up to indicate that the branch was never mispredicted by any of the candidates, you will guarantee any untested candidates will always be chosen first. At the end of the iteration, each candidate writes the number of branches it mispredicted into the record. For all our experiments, the second step was iterated 7 times. (The second step had to be iterated at least 3 times in order to test the 3 possible candidates for each branch.) At the end of the second step, the final candidate that is selected for each branch is the candidate that provided the fewest number of mispredictions.

4 Implementation Considerations

4.1 Evaluating a Hash Function using a Single XOR

The straightforward way to evaluate hash function HF_X is to rotate each of the X target addresses by their proper amounts and then XOR all of them together using several stages of XOR gates. The first stage of XOR gates separates the X rotated target addresses into pairs and XORs the two addresses in each pair together to produce $X/2$ results. The second stage separates the $X/2$ results into pairs and XORs the two results in each pair together to produce $X/4$ results.... Unfortunately, the latency and die area required to evaluate a hash function this way is impractical if X is large.

The solution is to have a k bit register associated with each hash function. This register contains a "partial sum" that is used to quickly evaluate the hash function. When a new target address is inserted into the THB, a hash function is evaluated by XORing its "partial sum", rotated by one bit, with the new target address. The "partial sum" is then updated to account for the new target address being "added" to it, and an old target address being "subtracted" from it. (Each hash function combines a finite number of target addresses. Thus, when a new target address is "added" to a "partial sum", an old target address must be "subtracted" from it.)

Here are the details. The register associated with HF_X contains the last index produced by HF_{X-1}, i.e., I_{X-1}. This register contains the "partial sum". When a new target address is inserted into the THB, the index for HF_X is generated by rotating the value in the register by one bit and then XORing the result with the new target address. (As with a gshare predictor, the value in a register is XORed with an address to generate an index into the predictor table.) There are two techniques for obtaining the new value for the register. The first is to simply obtain it from HF_{X-1}. The second is to rotate T_X by one bit and XOR it with the freshly computed index from HF_X. This second technique works because the freshly computed index from HF_X is the value of the "partial sum" with the new target address "added" to it. Rotating T_X by one bit and XORing it with this index "subtracts" the old target address from the "partial sum". (If the branch predictor does not implement HF_{X-1}, the second technique for obtaining the register value must be used. For the second technique, the computation of the new value and the access of the predictor table occur in parallel.)

4.2 Communicating the Profiling Information to the Hardware

If the compiler selects the hash function, it must be able to communicate the hash function number to the hardware. Some ISAs already set aside a few bits in each branch instruction for profiling information. For example, the Alpha AXP ISA [17] sets aside 14 bits in each indirect branch instruction for profiling information. If an ISA does not provide enough bits to indicate the exact hash function number, the compiler can use the bits to indicate roughly what the hash function number is, and the hardware can refine this number by using run-time information. For example, if only 1 bit has been set aside, and there are 8 hash functions, the compiler could set the bit to 0 to indicate that the hash function number is between 1 and 4, and set it to 1 to indicate that the hash function number is between 5 and 8. The exact number within a range would be determined by the hardware. If the ISA does not have any bits set aside for profiling information, then the ISA could be augmented. (Existing ISAs have been extensively augmented. For example, multimedia instructions were recently added to the venerable x86 ISA [9].)

4.3 Pipelining the Predictor

A variable length path predictor requires two sequential table accesses to make a prediction. Figure 3 illustrates these two table accesses. (The MUX in this figure is the same MUX that is in Figure 2.) The first table that is accessed is the Hash Function Number Table (HFNT). This table is accessed with the low order address bits of the branch being predicted. Each entry in the table contains a hash function number for a branch. The result of the first table access is a *prediction* of the hash function number associated with the branch being predicted. This result is used to select the index that will be used to access the second table. The second table is simply the predictor table.

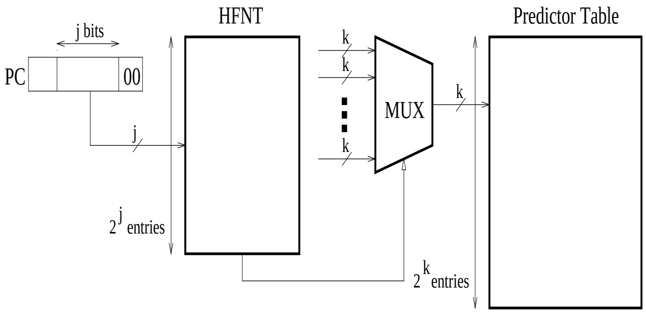

Figure 3: Table Accesses Required to Make a Prediction

When the branch is decoded, the actual hash function number is obtained from the opcode. This number is compared to the predicted hash function number that was obtained from the HFNT. If the two numbers are not equal, the branch is re-predicted using the actual hash function number. The hash function number for the branch is written into the HFNT when the branch retires.

We assume that each table access requires one cycle. As a result, the variable length path predictor requires two cycles to make a prediction. To pipeline the predictions, the organization of the HFNT needs to be modified from the organization described earlier. This new organization is shown at the right of Figure 4. At the left of the figure is a partial control flow graph consisting of the basic blocks A, B, and C. There is a conditional branch at the end of block A. If this branch is not taken, block B follows. If this branch is taken, block C follows. When the HFNT is accessed using the low order address bits of the conditional branch in block A, the result is two predictions of hash function numbers: one prediction of the hash function number for the branch at the end of block B and one prediction of the hash function number for the branch at the end of block C. The branch prediction for the branch in block A is used to select one of these two hash function numbers. The selected hash function number is used to select the index that will be used to access the predictor table. When the next branch is encountered, the predictor table is accessed using the selected index. The result of this access provides the prediction for the branch.

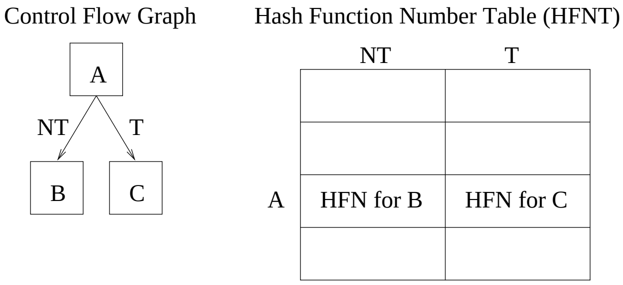

Figure 4: Hash Function Number Table for a Pipelined Variable Length Path Predictor

If the branch in block A is an indirect branch rather than a conditional branch, a single bit from the predicted target address of the indirect branch is used to select from among the two hash function numbers. If the branch is an unconditional branch, the lowest order bit of the branch's address that was not a part of the index into the HFNT is used to select from among the two hash function numbers. If the branch is a subroutine call, the first hash function number is always selected. The second hash function number is pushed on the Return Address Stack along with the return address. (Each entry in the Return Address Stack is expanded a couple of bits so that it can accommodate a hash function number.) The second hash function number will be used to predict the hash function number for the branch following the subroutine's return. If the branch is a subroutine return, the hash function number is popped from the Return Address Stack.

5 The Experiments

The experiments in this paper were done using the basic versions of the variable length path predictor. We compare the performance of these predictors to the performance of basic versions of other predictors (gshare [14] for conditional branches, and pattern and path based predictors for indirect branches [3]). Both the variable length path predictor and the predictors we compared it to can be enhanced. The conditional branch predictors can benefit from interference reduction techniques [2, 13, 15, 18]. The indirect branch predictors can benefit from better replacement algorithms for the target addresses stored in the predictor table [1, 6]. Any of the predictors presented in this paper may be used as a component of a hybrid predictor [14] or a cascaded predictor [7].

5.1 Methodology

The results presented in this paper are for the eight SPECint95 benchmarks. We also present the results for eight other commonly used programs. These non-SPEC benchmarks are: version 4.0 of GNU Chess (chess), version 1.10 of the GNU C++ implementation of the 'troff' document formatting system (groff), version 4.03 of the Aladdin Ghostscript interpreter (gs), version 2.6.2 of the Pretty Good Privacy encryption system (pgp), version 3.5 of the gnuplot plotting program (plot), version 1.4 of the Python object-oriented programming language interpreter (python), version 2.0 of the SimpleScalar superscalar out-of-order execution processor simulator (ss), and version 7.0 of the Web2c port of the TEX document formatting system (tex).

All benchmarks were compiled for the DEC Alpha architecture and instrumented with code to simulate the branch prediction structures using the ATOM instrumentation tool [19]. For those experiments requiring profiling, different profile and test input sets were used. Table 1 lists the benchmarks along with the static and dynamic count of conditional and indirect branches seen when running the test input set. Returns were not included in the indirect branch count as they are not predicted by the indirect branch predictors considered in this paper. Each benchmark was simulated until completion.

	conditional		indirect	
Benchmark	dynamic	static	dynamic	static
099.go	17.6 M	4770	91.4 K	11
124.m88ksim	92.6 M	1095	1.01 M	14
126.gcc	27.6 M	14419	0.99 M	192
129.compress	11.7 M	371	0.16 K	3
130.li	32.4 M	517	1.12 M	11
132.ijpeg	18.2 M	1161	98.2 K	134
134.perl	21.4 M	1536	2.27 M	21
147.vortex	25.8 M	6529	0.11 M	33
chess	52.4 M	1736	0.11 M	7
groff	22.4 M	2322	2.01 M	172
gs	29.4 M	5476	1.63 M	504
pgp	16.5 M	1444	0.18 K	5
plot	25.7 M	1417	0.50 M	43
python	33.8 M	2578	2.02 M	168
ss	22.3 M	1997	0.18 M	29
tex	20.6 M	2970	0.31 M	42

Table 1: Benchmark Summary

5.2 Experimental Results

In this section we compare the path predictor presented in this paper to other predictors. Two versions of the path predictor are used for both indirect and conditional predictions, the fixed length path predictor (without profiling) and the variable length path predictor (with profiling). For conditional branch prediction, these are compared to gshare [14], and for indirect branch prediction, they are compared to the path and pattern based predictors presented in [3].

For the fixed length path predictor, the same path length was used for all benchmarks. The length used was that for which the average misprediction rate for all the benchmarks was the lowest. To avoid unfairly skewing the results in favor of the fixed length predictor, the best path length was determined using the profile input sets, not the test input sets. The path lengths used for the fixed length path predictors are shown in Table 2.

Conditional Branches	
Table Size (KB)	Path Length
1	6
4	9
16	14
64	16
256	23

Indirect Branches	
Table Size (KB)	Path Length
0.5	11
2	21
8	21
32	21

Table 2: Path Length Used for Fixed Length Predictor

For the gcc benchmark, the fixed and variable length path predictors are compared to other predictors over a range of sizes. Due to space limitations only one size (16K bytes for conditional and 2K bytes for indirect) is used for the other benchmarks.

5.2.1 Conditional Branches

Figure 5 and 6 show the misprediction rates of the predictor proposed in this paper both with (variable length path) and without (fixed length path) profiling. These results are compared to the misprediction rate of the gshare predictor. For these experiments, the predictor size is fixed at 16K bytes. The variable length path predictor has 28.6% fewer mispredictions than gshare on average and is significantly better for all 16 benchmarks. The largest reduction in mispredictions is 68.6% for perl, and the smallest is 7.4% for pgp. We achieved a 46.7% reduction in mispredictions for gcc, which will be examined more closely in section 5.2.3. Even the fixed length path predictor, which does not use profiling, is marginally better than gshare.

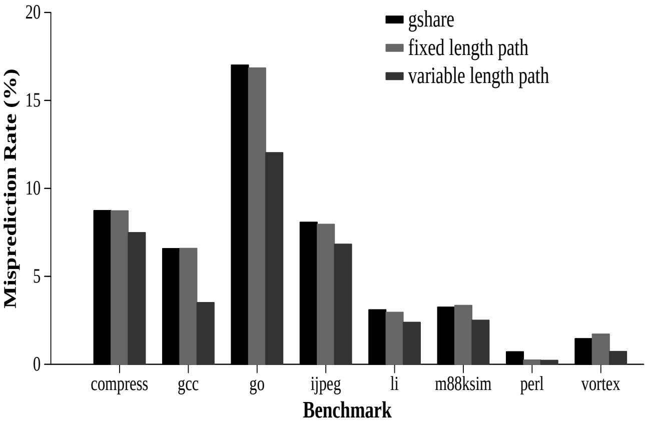

Figure 5: Misprediction Rates for Conditional Branches with a 16K byte Predictor (SPEC)

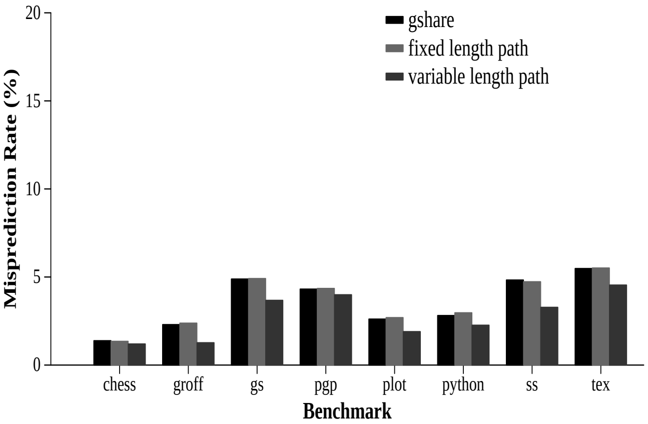

Figure 6: Misprediction Rates for Conditional Branches with a 16K byte Predictor (Non-SPEC)

5.2.2 Indirect Branches

Figure 7 and 8 show the misprediction rates of the predictor proposed in this paper both with (variable length path) and without (fixed length path) profiling. These results are compared to the misprediction rates of both the pattern and path based predictors from [3]. For these figures, the predictor size is fixed at 2K bytes [1]. The 8 benchmarks with the highest indirect branch frequencies are marked in bold on the figures. Indirect branches are more frequent for these benchmarks, so the impact of target mispredictions on the execution time for these benchmarks will be more significant. For this reason, we will focus on these 8 benchmarks.

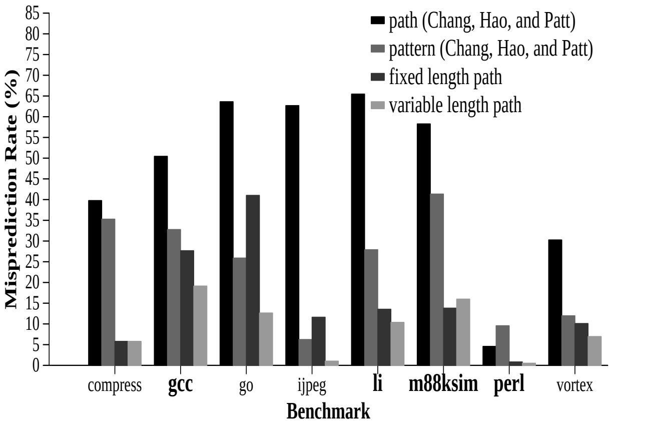

Figure 7: Misprediction Rates for Indirect Branches with a 2K byte Predictor (SPEC)

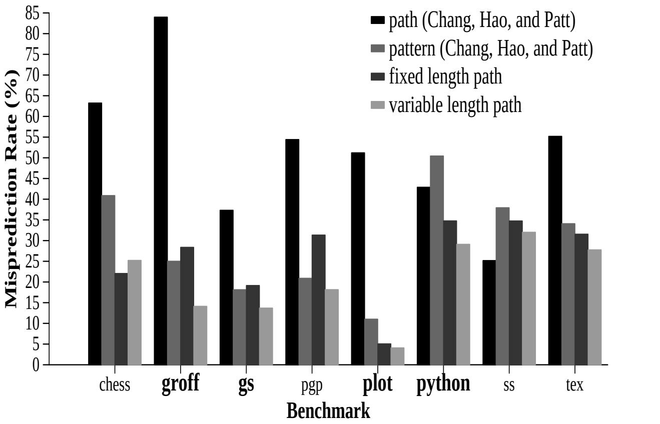

Figure 8: Misprediction Rates for Indirect Branches with a 2K byte Predictor (Non-SPEC)

---

[1] Although indirect branch targets are 64 bits in the Alpha architecture, only the lower 32 bits are stored in the predictor table. The remaining 32 bits are taken from the current fetch address.

The results for the 8 benchmarks with frequent indirect branches are shown in Table 3. The variable length path (VLP) predictor achieves a 24.5% to 94.9% reduction in mispredictions compared to the pattern based variation from [3]. Even the fixed length path (FLP) predictor, which does not require profiling, is significantly better than the pattern based predictor for 6 of the 8 benchmarks. This shows that for indirect branches, the path predictor presented in this paper is superior.

Benchmark	path [3]	pattern [3]	FLP	VLP
124.m88ksim	58.24%	41.31%	13.79%	15.96%
126.gcc	50.42%	32.75%	27.64%	19.12%
130.li	65.44%	27.88%	13.52%	10.36%
134.perl	4.56%	9.54%	0.80%	0.49%
groff	83.97%	25.00%	28.36%	14.10%
gs	37.31%	18.12%	19.13%	13.68%
plot	51.19%	11.00%	5.04%	4.06%
python	42.87%	50.42%	34.75%	29.09%

Table 3: Misprediction Rates for Indirect Branches on Selected Benchmarks

5.2.3 Gcc

Figure 9 shows the conditional branch misprediction rate of the predictor presented in this paper with (variable length path) and without (fixed length path) profiling and the gshare predictor for a range of sizes. In addition, it shows the misprediction rates of a predictor that uses both profiling *and* a fixed length path (fixed length path (tuned)). We chose to show the misprediction rates for the gcc benchmark because it executes a large number of both conditional and indirect branches, and because its characteristics are generally well known.

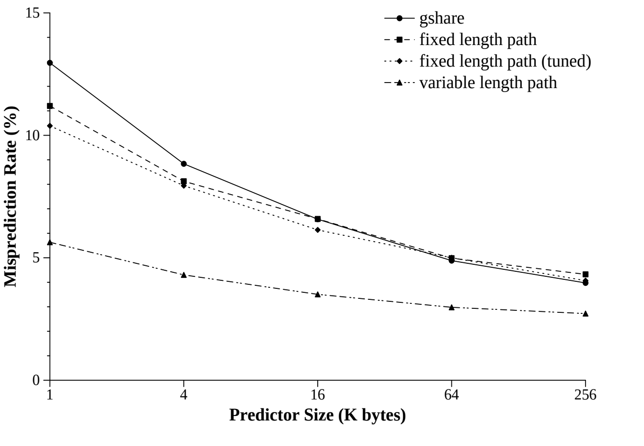

Figure 9: Misprediction Rates for Conditional Branches in Gcc

Recall that the length used for the fixed length path predictor was the length for which the average misprediction rate for all the benchmarks was the lowest. The length used for the "tuned" fixed length path predictor was the length for which the misprediction rate of the gcc benchmark was the lowest. This length was determined using the profile input set, not the test input set.

Note that the compiler only needs to communicate a single hash function number to a "tuned" fixed length path predictor. No ISA changes are necessary to accomplish this. It can be accomplished by loading the hash function number into a processor specific hardware register at the beginning of each program.

The variable length path predictor performs much better than gshare for all sizes. The variable length path predictor also performs much better than the "tuned" fixed length path predictor, illustrating the benefit of varying the path length for each branch. The fixed length path predictor performs better than gshare for sizes under 16K bytes, and slightly worse for sizes over 16K bytes.

Figure 10 shows the misprediction rates for indirect branches. For all sizes, both the variable and the fixed length path predictors perform outrageously better than the competing predictors. At 32K bytes, the fixed length path predictor reduces mispredictions by 29% over the best competing predictor. The variable length path predictor reduces mispredictions by 51%.

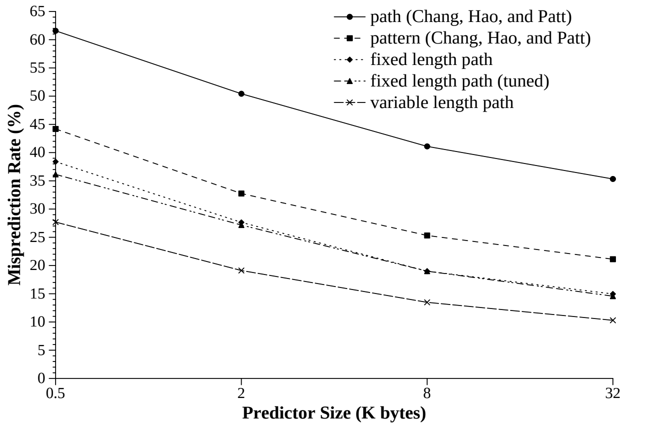

Figure 10: Misprediction Rates for Indirect Branches in Gcc

5.3 Why Variable Length Path Prediction Works So Well

As shown in Section 5.2, the variable length path predictor has a clear performance advantage over the baseline predictors. This is due to two different effects. The first is the ability of the variable length path predictor to represent the actual path leading up to the branch more fully than the competing predictors. This can be seen clearly by comparing the fixed length path predictor to gshare for conditional branches and to the pattern and path based 2-level predictors for indirect branches. This advantage is only slight for conditional branches, but great for indirect branches.

The second effect is due to the variable nature of the path length. Evers et. al. [8] showed that only a small amount of the path information leading up to a branch is needed for prediction. If parts of the path that have no bearing on the outcome of the current branch are included in the history, an unnecessarily high number of predictor table entries will be used for the branch. This leads to longer training times and more interference in the predictor table. By selecting the history length individually for each branch, the variable length path predictor is able to discard unimportant path prefixes. This results in shorter training times and less interference.

6 Conclusions

In this paper, we proposed a new path based predictor for both conditional branches and indirect branches. By constructing the index into the predictor table as a function of the last N target addresses, and using profiling information to select the proper value of N for each branch, prediction accuracy was improved substantially over existing predictors.

For conditional branches, the new predictor was as accurate as gshare even when the value of N was fixed. This value for N can be selected without the aid of any profiling information or compiler heuristics. When profiling information was used to help select the value of N for each branch, the new predictor performed much better than gshare, reducing the number of mispredictions on average by 28.6% for a 16K byte predictor.

For indirect branches, the new predictor performed even better. When compared to the best predictors from previous work, at an implementation cost of 2K bytes, the new predictor reduced the number of mispredictions on average by 36.4% even when the value of N was fixed. When also taking advantage of profiling, the number of mispredictions was reduced on average by 54.3%.

Although this predictor already represents a substantial improvement over previous work, especially for indirect branch prediction, there are some ways to potentially improve it further. One promising idea is to save some history from before certain control structures such as loops or subroutines and then restore the history after the control structure ends. For instance, Jacobson et. al. [11] proposed to store some of the history information on a stack whenever a subroutine call was made. When returning from a subroutine, the old history would be popped from the stack. The old history would be combined with the more recent history to make any further predictions. Another avenue for improving this predictor would be to develop compiler heuristics to select the hash function in lieu of profiling.

Acknowledgements

This paper is one result of our ongoing research in high performance computer implementation at the University of Michigan. The support of our industrial partners, in particular Intel and HAL computer systems, is greatly appreciated. We specially thank Dan Friendly, Paul Racunas, Sanjay Patel, and the anonymous referees for their insights and comments on drafts of this paper. In addition, we wish to gratefully acknowledge the other members of our HPS research group for the stimulating environment they provide.

References

[1] B. Calder and D. Grunwald, "Reducing indirect function call overhead in C++ programs," in *21st Symposium on Principles of Programming Languages*, pp. 397–408, 1994.

[2] P.-Y. Chang, M. Evers, and Y. N. Patt, "Improving branch prediction accuracy by reducing pattern history table interference," in *Proceedings of the 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1996.

[3] P.-Y. Chang, E. Hao, and Y. N. Patt, "Predicting indirect jumps using a target cache," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 274 – 283, 1997.

[4] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt, "Branch classification: A new mechanism for improving branch predictor performance," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 22–31, 1994.

[5] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 292 – 302, 1997.

[6] K. Driesen and U. Hölzle, "Accurate indirect branch prediction," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 167–178, 1998.

[7] K. Driesen and U. Hölzle, "Improving indirect branch prediction with source- and arity-based classification and cascaded prediction," Technical Report TRCS98-07, Department of Computer Science, University of California, Santa Barbara, March 1998.

[8] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt, "An analysis of correlation and predictability: What makes two-level branch predictors work," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 52–61, 1998.

[9] L. Gwennap, "Intel's MMX speeds multimedia," *Microprocessor Report*, March 1996.

[10] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith, "Control flow speculation in multiscalar processors," in *Proceedings of the Third IEEE International Symposium on High Performance Computer Architecture*, 1997.

[11] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[12] T. Juan, S. Sanjeevan, and J. J. Navarro, "Dynamic history-length fitting: A third level of adaptivity for branch prediction," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 155–166, 1998.

[13] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The bimode branch predictor," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 4 – 13, 1997.

[14] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[15] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 292–303, 1997.

[16] R. Nair, "Dynamic path-based branch correlation," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 15–23, 1995.

[17] R. L. Sites, *Alpha Architecture Reference Manual*, Digital Press, Burlington, MA, 1992.

[18] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The agree predictor: A mechanism for reducing negative branch history interference," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 284–291, 1997.

[19] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 196–205, 1994.

[20] A. R. Talcott, M. Nemirovsky, and R. C. Wood, "The influence of branch prediction table interference on branch prediction scheme performance," in *Proceedings of the 1995 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1995.

[21] M.-D. Tarlescu, K. B. Theobald, and G. R. Gao, "Elastic history buffer: A low-cost method to improve branch prediction accuracy," in *Proceedings of the 1997 International IEEE Conference on Computer Design*, pp. 82–87, 1997.

[22] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 51–61, 1991.

[23] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 124–134, 1992.

[24] C. Young, N. Gloy, and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 276–286, 1995.

[25] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232–241, 1994.