To appear in the Proc. of the ACM SIGPLAN Conf. on Prog. Language Design and Implementation (PLDI '95), La Jolla, CA, June 18–21, 1995

Corpus-based Static Branch Prediction

Brad Calder, Dirk Grunwald, Donald Lindsay,
James Martin, Michael Mozer, and Benjamin Zorn
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

Abstract

Correctly predicting the direction that branches will take is increasingly important in today's wide-issue computer architectures. The name *program-based* branch prediction is given to static branch prediction techniques that base their prediction on a program's structure. In this paper, we investigate a new approach to program-based branch prediction that uses a body of existing programs to predict the branch behavior in a new program. We call this approach to program-based branch prediction, *evidence-based static prediction*, or ESP. The main idea of ESP is that the behavior of a *corpus* of programs can be used to infer the behavior of new programs. In this paper, we use a neural network to map static features associated with each branch to the probabilty that the branch will be taken. ESP shows significant advantages over other prediction mechanisms. Specifically, it is a program-based technique, it is effective across a range of programming languages and programming styles, and it does not rely on the use of expert-defined heuristics. In this paper, we describe the application of ESP to the problem of branch prediction and compare our results to existing program-based branch predictors. We also investigate the applicability of ESP across computer architectures, programming languages, compilers, and run-time systems. Averaging over a body of 43 C and Fortran programs, ESP branch prediction results in a miss rate of 20%, as compared with the 25% miss rate obtained using the best existing program-based heuristics.

1 Introduction

In this paper, we propose a new technique for program-based branch prediction based on a general approach that we have invented called *Evidence-based Static Prediction* (ESP). Our results show that using our new approach results in better branch prediction than all existing program-based techniques. In addition, our ESP approach is very general, and can be applied to a wide range of program behavior estimation problems. In this paper, we describe ESP and its successful application to the problem of program-based branch prediction.

Branch prediction is the process of correctly predicting whether branches will be taken or not before they are actually executed. Branch prediction is important, both for computer architectures and compilers. Compilers rely on branch prediction and execution estimation to implement optimizations such as trace-scheduling [12, 13] and other profile-directed optimizations [8, 9].

Wide-issue computer architectures rely on predictable control flow, and failure to correctly predict a branch results in delays for fetching and decoding the instructions along the incorrect path of execution. The penalty for a mispredicted branch may be several cycles long. For example, the mispredict penalty is 4 to 5 cycles on the Digital Alpha AXP 21064 processor. In previous studies, we found that conditional branches in C programs were executed approximately every 8 instructions on the Alpha architecture [7]. Current wide-issue architectures can execute four or more instructions per cycle. As a result, such architectures are likely to execute branch instructions every two cycles or less and effective branch prediction on such architectures is extremely important. Many approaches have been taken to branch prediction, some of which involve hardware [5, 23] while others involve software [3, 6, 11]. Software methods usually work in tandem with hardware methods. For example, some architectures have a "likely" bit that can be set by a compiler if a branch is determined to be likely taken by a compiler.

Compilers typically rely on two general approaches for branch prediction. Profile-based methods use program profiles to determine the frequency that branch paths are executed. Fisher and Freudenberger showed that profile-based branch prediction can be extremely successful in reducing the number of instructions executed between mis-predicted branches [11]. The main drawback of profile-based methods is that additional work is required on the part of the programmer to generate the program profiles.

Program-based branch prediction methods attempt to predict branch behavior in the absence of profile information and are based only on a program's structure. Some of these techniques use heuristics based on local knowledge that can be encoded in the architecture [14, 18]. Other techniques rely on applying heuristics based on less local program structure in an effort to predict branch behavior [3]. In this paper, we describe a new approach to program-based branch prediction that does not rely on such heuristics. Our branch prediction relies on a general program-based prediction framework that we call ESP. The main idea of ESP is that the behavior of a corpus of programs can be used to infer the behavior of new programs. That is, instead of using a different execution of a program to predict its own behavior (as is done with profile-based methods), we use the behavior of a large body of different programs (the training set, or corpus) to identify and infer common behavior. Then we use this knowledge to predict branches for programs that were not included in the training set. In particular, in this paper we use a neural network to map static features associated with each branch to the probability that the branch will be taken.

Branch prediction using ESP has several important advantages over existing program-based branch prediction methods. First, because the technique automatically maps static features of branches to a probability that they will be taken, our technique is suitable for program-based branch prediction across different languages, compilers, and computer architectures. Existing techniques rely on compiler-writer defined heuristics that are based on an intuition about common programming idioms. Second, given a large amount of static information about each branch, the technique automatically determines what parts of that information are useful. Thus, it does not rely on trial-and-error on the part of the compiler writer searching for good heuristics. Finally, our results show that ESP branch prediction outperforms existing heuristic program-based branch prediction techniques over a body of 43 C and Fortran programs. In particular, our heuristics have an average overall miss rate of 20%, which compares to the 25% miss rate of the best existing heuristic technique, and the 8% miss rate of the perfect static predictor.

This paper has the following organization. In Section 2 we discuss previous approaches to program-based branch prediction and other knowledge-based approaches to program optimization. In Section 3 we discuss the details of our ESP branch prediction method. Section 4 describes the methods we used to evaluate and compare ESP prediction with previous approaches, and Section 5 presents our results. We summarize our conclusions in Section 6 and also discuss possible future directions to take with this research.

2 Background

In this section, we discuss the existing approaches to static branch prediction and also discuss other knowledge-based approaches to compiler optimization.

2.1 Program-Based Branch Prediction Methods

One of the most simple program-based methods for branch prediction is called "backward-taken/forward-not-taken" (BTFNT). This technique relies on the heuristic that backward branches are usually loop branches, and as such are likely to be taken. One of the main advantages of this technique is that it relies solely on the sign bit of the branch displacement, which is already encoded in the instruction. While simple, BTFNT is also quite successful. Our results in Section 5 show it has an overall miss rate in our experiments of 34%. Any more sophisticated program-based prediction techniques must do better than BTFNT to be viable.

To facilitate program-based methods for branch prediction, some modern architectures provide a "branch-likely" bit in each branch instruction [1]. In these architectures, compilers can employ either profile-based [11] or program-based techniques to determine what branches are likely to be taken. In recent work, Ball and Larus [3] showed that applying a number of simple program-based heuristics can significantly improve the branch prediction miss rate over BTFNT on tests based on the conditional branch operation. A complete summary of the Ball and Larus heuristics is given in Table 1 (as described in [22]). Their heuristics use information about the branch opcode, operands, and characteristics of the branch successor blocks, and encode knowledge about common programming idioms.

Two questions arise when employing an approach like that taken by Ball and Larus. First, an important question is which heuristics should be used. In their paper, they describe seven heuristics that they considered successful, but also noted that "We tried many heuristics that were unsuccessful. [3]" A second issue that arises with heuristic methods is how to decide what to do when more than one heuristic applies to a given branch. This problem has existed in the artificial intelligence community for many years and is commonly known as the "evidence combination" problem. Ball and Larus considered this problem in their paper and decided that the heuristics should be applied in a fixed order; thus the first heuristic that applied to a particular branch was used to determine what direction it would take. They determined the "best" fixed order by conducting an experiment in which all possible orders were considered. We call using this pre-determined order for heuristic combination the *A Priori Heuristic Combination* (APHC) method. Using APHC, Ball and Larus report an average overall miss rate on the MIPS architecture of 20%.

In a related paper, Wu and Larus refined the APHC method of Ball and Larus [22]. In that paper, their goal was to determine branch *probabilities* instead of simple branch prediction. Whereas with branch prediction, the goal is to determine a single bit of information per branch (likely versus unlikely), with branch probabilities, the goal is to determine the numeric probability that a branch is taken or not taken. Wu and Larus abandoned the simplistic evidence combination function of APHC in favor of an evidence combination function borrowed from Dempster-Shafer theory [10, 17]. We call this form of evidence combination *Dempster-Shafer Heuristic Combination* (DSHC). By making some fairly strong independence assumptions, the Dempster-Shafer evidence combination function can produce an estimate of the branch probability from any number of sources of evidence. For example, if one heuristic indicates that a branch is likely to be taken with probability X%, while another says it is likely to be taken with probability Y%, then DSHC allows these two probabilities to be combined. The probabilities X% and Y% that Wu and Larus use are taken directly from the paper of Ball and Larus [3]. We refer to a DSHC algorithm based on this data as DSHC(B&L).

Heuristic Name	Heuristic Description
Loop Branch	Predict that the edge back to the loop's head is taken and the edge exiting the loop is not taken.
Pointer	If a branch compares a pointer against null or compares two pointers, predict the branch on false condition as taken.
Opcode	If a branch checks an integer for less than zero, less than or equal to zero, or equal to a constant, predict the branch on false condition.
Guard	If a register is an operand of the branch comparison, the register is used before being defined in a successor block, and the successor block does not post-dominate the branch, predict the successor block as taken.
Loop Exit	If a comparison is inside a loop and no successor is a loop head, predict the edge exiting the loop as not taken.
Loop Header	Predict the successor that does not post-dominate and is a loop header or a loop pre-header as taken.
Call	Predict the successor that contains a call and does not post-dominate the branch as taken.
Store	Predict the successor that contains a store instruction and does not post-dominate the branch as not taken.
Return	Predict the successor that contains a return as not taken.

Table 1: Summary of the Ball/Larus Heuristics

Because the goal of Wu and Larus was to perform program-based profile estimation, they give no results about how the DSHC method works for program-based branch prediction. One of the contributions of our paper is that we quantify the effectiveness of the DSHC method for branch prediction.

Wagner et al. [21] also used heuristics similar to those of Ball and Larus to perform program-based profile estimation. They also applied the heuristics in a fixed order. They report branch prediction miss rate results similar to those of Ball and Larus.

2.2 Knowledge-Base Approaches to Optimization

Our ESP method relies on collecting data from a corpus of program behavior and using that data to perform program-based prediction. There is little other work in compiler optimization that has taken this approach. We summarize the work we are aware of here.

In [2], Balasundaram et al. address a somewhat different program-based estimation problem. The authors wanted to make compile-time decisions about data partitioning across a parallel computer. They report on the idea of using profile data to "train" an estimator. This training, an offline step, generates code which is then incorporated into their compiler. Training only needs to be done once per compilation target, and is reported to be better than using a parameterized theoretical model. While the strategy they employ is similar to ESP, their application domain is quite different. In addition, our results show that this general approach of knowledge-based "training" can be used to enhance a wide class of optimizations based on program behavior estimation.

3 Evidence-based Branch Prediction

In this section, we propose a general framework for program-based prediction. Our method, ESP, is generally described as follows. A body of programs and program input is gathered (the *corpus*). Particular static information (the *static feature set*) about important static elements of the corpus (e.g., instructions) are recorded. The programs in the corpus are executed, and the corresponding dynamic behavior is associated with each static element (e.g., the number of times a branch is taken and not-taken is associated with each branch). At this point, we have accumulated a body of knowledge about the relationship between static program elements and dynamic behavior. This body of knowledge can then be used at a later time to predict the behavior of instructions with similar static features for programs not in the corpus. With this broad definition of our framework in mind, we now describe how we apply this general framework to the specific problem of branch prediction.

3.1 ESP Branch Prediction

In applying ESP to the problem of branch prediction, we instantiate the above framework in the following way. The static program elements we are interested in are the program branch instructions. For this study, we consider only two-way branches. For each branch instruction in the program text, we record a large static feature set (see Table 2).

Some of the features are properties of the branch instruction itself (e.g., the branch opcode), others are properties of the registers used to define the register in the branch instruction (e.g., the opcode of the instructions that defined them), while others are properties of the procedure that the branch is in (leaf versus non-leaf). The existence of some features is dependent on the values of other features. For example, feature 4 is only

Feat. Num.	Feature Name	Feature Description
1	Br. opcode	The opcode of branch instruction.
2	Br. direction	F — Forward branch, B — Backwards branch
3	Br. operand opcode	The opcode of the insruction that defines the register used in the branch instruction (or ?, if the branch operand is defined in a previous basic block).
4	RA opcode	If the instruction in (3) uses an RA register, this is the opcode of the instruction that defines that register (? otherwise).
5	RB opcode	If the instruction in (3) uses an RB register, this is the opcode of the instruction that defines that register (? otherwise).
6	Loop header	LH — the basic block is a loop header, NLH - not a loop header
7	Language	The language of the procedure the branch is in (C or FORT).
8	Procedure type	The branches' procedure is a Leaf, NonLeaf or calls itself recursively (CallSelf)
9–16	Features of the Taken Successor of the Branch	
9	Br. dominates	D — basic block dominates this successor, or ND — does not dominate
10	Br. postdominates	PD — the successor basic block post-dominates the basic block with the branch, or NPD — does not post-dominate
11	Succ. Ends	Branch type ending successor basic block, possible values (FT — fall through, CBR — conditional branch, UBR — unconditional branch, BSR — branch subroutine, JUMP — jump, IJUMP — indirect jump, JSR — jump subroutine, IJSR — indirect jump subroutine, RETURN, COROUTINE, LAST_JUMP_KIND, or NOTHING)
12	Succ. Loop	LH — the successor basic block is a loop header or unconditionally passes control to a basic block which is a loop header, NLH — not a loop header
13	Succ. Backedge	LB — the edge getting to the successor is a loop back edge, NLB — not a loop back edge
14	Succ. Exit	LE — the edge getting to the successor is a loop exit edge, NLE — not a loop exit edge
15	Succ. UseDef	UBD — the successor basic block has a use of a register before defining it and that register was used to determine the destination of the current conditional branch instruction. NU — no use before def in successor
16	Succ Call	PC — the successor basic block contains a procedure call or unconditionally passes control to a basic block with a procedure call, NPC — no procedure call down here
17–24	Features of the Not Taken Successor of the Branch	
	As above features 9–16	

Table 2: Static Feature Set Used in the ESP Branch Prediction Study.

meaningful if feature 3 has an RA operand. We call such features *dependent static features*.

We chose the feature set shown in Table 2 based on several criteria. First, we encoded information that we believed would likely be predictive of behavior. This information included some of the information used to define the Ball/Larus heuristics (e.g., information about whether a call appears in a successor of the branch). Second, we encode other information that was easily available. For example, since the opcodes that define the branch instruction register are readily available, we include them as well. Similarly, information about the procedure type is readily available. We note that the feature set listed here is the only one we have yet tried. We have made no effort to identify a particularly good feature set, and our positive results suggest that such "feature tuning" is unnecessary.

Having defined the static feature set, we then determine the static feature set for each branch in the corpus of programs. We next run the programs in the corpus and collect information about how often each branch is taken and not taken. The goal is to associate two pieces of dynamic information with each branch instruction: how frequently the branch was executed and how often was it taken. Because execution frequency is program dependent, we normalize the branch frequency by the total number of branches executed in the program. We compute the *normalized branch weight* by dividing how many times the branch was executed by the total number of branches executed by the program (resulting in a number between zero and one). Finally, we associate with each branch instruction in the corpus its static feature set, its normalized branch weight, and its branch probability (percentage of the time the branch was taken).

3.1.1 Prediction using Neural Nets

Our goal is to have a system that can predict the branch probability for a particular branch from its static feature set. This system should accurately predict not just for the programs in the corpus, but also for previously unseen programs.

One way of doing such prediction is via a *feedforward neural network* [19]. A feedforward neural network maps a numerical input vector to a numerical output. Here, the input vector consists of the feature values in the static feature set, and the output is a scalar indicating the branch probability.

Figure 1 depicts the branch prediction neural network. A neural network is composed of *processing units*, depicted in the Figure by circles. Each processing unit conveys a scalar value known as its *activity*. The activity pattern over the bottom row of units is the input to the network. The activity of the top unit is the output of the network. Activity in the network flows from input to output, through a layer of intermediate or *hidden units*, via weighted connections. These connections are depicted in the figure by links with arrows indicating the direction of activity flow.

This is a standard neural network architecture. We also use a fairly standard neural network dynamics in which the activity of hidden unit i, denoted h_i, is computed as:

$$h_i = \tanh(\sum_j w_{ij} x_j + b_i),$$

where x_j is the activity of input unit j, w_{ij} is the connection weight from input unit j to hidden unit i, b_i is a bias weight associated with the unit, and tanh is the hyperbolic tangent function,

$$\tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}.$$

Similarly, the output unit activity, denoted y, is computed from the hidden unit activities:

$$y = .5 \tanh(\sum_i v_i h_i + a) + 1,$$

where v_i is the connection weight from hidden unit i to the output unit and a is a bias weight associated with the output unit. The tanh function is normalized to achieve an activity range of $[0, 1]$ for the output unit.

The input-output behavior of the neural network is determined by its free parameters, the weights **w** and **v** and biases **b** and a. These parameters are set by an algorithm known as *back propagation* [16]. This is a gradient descent procedure for adjusting the parameters such that performance of the network on a *training corpus* is optimized. The standard measure of performance is the sum of squared errors,

$$E = \sum_k (y^k - t^k)^2,$$

where k is an index over examples in the training corpus, y^k is the actual output of the network when training input k is presented, and t^k is the *target output*—the output indicated for that example in the training corpus.

In this application, however, we have a different criterion for good performance. We want to minimize two sorts of errors, *missed branches* (MB) and *branches incorrectly taken* (BIT). MB occur when the predictor says that the branch will be taken with probability less than .5 when the branch is in reality taken; BIT occur when the predictor says that the branch will be taken with probability greater than .5 when the branch is in reality not taken. If the network output for example k, is binary—1 if the predicate "the branch probability is greater than .5" is believed to be true, 0 otherwise—then the total number of errors due to MB for example k is

$$E_{MB} = (1 - y^k) t^k n^k,$$

where n^k is the normalized branch weight. The product $t^k n^k$ gives the (relative) number of cases where the branch is taken. All of these branches are missed if $y^k = 0$ (or equivalently, $1 - y^k = 1$). Similarly, the total number of errors due to BIT is

$$E_{BIT} = y^k (1 - t^k) n^k.$$

Because these two types of errors have equal cost, the total error is simply

$$E = \sum_k E_{MB} + E_{BIT} = \sum_k n^k [y^k (1 - t^k) + t^k (1 - y^k)].$$

This is used as the error measure to be minimized by the neural net training procedure. That is, the free parameters in the neural

Figure 1: The branch prediction neural network. Each circle represents a processing unit in the network, and the links between units depict the flow of activity.

net are adjusted such that the network will produce outputs y^k such that E is minimized. Note that this does not require that the network accurately predict branch probabilities per se, as we were assuming previously.[1]

Each input unit's activity is normalized over the training set to have zero mean and standard deviation 1. The same normalization is applied for test cases. We deal with nonmeaningful dependent static features by setting their input activity to 0 after the normalization step; this prevents the nonmeaningful features from having any effect on the computation, and is equivalent to gating the flow of activity from these features by another feature that indicates the relevance of the dependent features for a particular example. We use a "batch" training procedure in which weights are updated following a sweep through the entire training corpus, and an adaptive learning rate procedure wherein the learning rate for the network is increased if error drops regularly or is decreased otherwise. Momentum is not used. Training of the network continues until the *thresholded error* of the net no longer decreases. By thresholded error, we mean the error computed when the output is first thresholded to values 0 or 1. This achieves a form of early stopping, and thereby helps to prevent overfitting.

3.1.2 Discussion

In Section 2, we noted that there were two inherent problems with heuristic-based approaches to program-based prediction. First there is the problem of determining what heuristics to use. In particular, the search for successful heuristics requires a significant amount of effort and cannot be easily automated. Furthermore, the effectiveness of particular heuristics (e.g., the "return heuristic" which predicts the successor without the return instruction to be taken) will depend on the programming language, compiler, programming style, and architecture being used.

[1]In the above discussion, we assumed that the network output will be either 0 or 1. However, the output must be continuous-valued in order to apply gradient-based training procedures. Thus, we use the continuous activation rule for y presented earlier, and simply interpret the continuous output as the network's confidence that the true branch probability is greater than .5.

For example, one might think that the return heuristic is likely to be more effective when applied to languages, such as Scheme, where recursion is the most commonly used mechanism for performing iteration. Likewise, the pointer heuristic, which assumes pointer comparisons to null and for equality will fail, is more likely to be applicable in a "pointerful" language like Scheme. We found, however, that when we applied these heuristics to three Scheme programs (boyer, corewar, and sccomp, all compiled with the Scheme-to-C compiler)[2], the results show that the return heuristic had an average 56% miss rate and the pointer heuristic had a miss rate of 89%. These results show that applying heuristics based on intuition is both difficult and can often result in incorrect conclusions. Thus, new heuristics will be required for new architectures, programming languages, and even compilers.

A second problem with heuristic approaches is determining how to combine them when more than one apply to the same situation. While Wu and Larus attempted to solve this problem using Dempster-Shafer theory, our results show that using the DSHC method results in slightly higher miss rates than the more *ad hoc* APHC method. This is likely a result of the strong independence assumptions embodied in the Dempster-Shafer evidence combination function [15].

Our ESP method addresses these two disadvantages directly. Instead of relying on experts to think of heuristics and to test them to determine if they are effective, our method extracts features associated with predictable behavior automatically. The ESP method also has disadvantages, as well. First, a corpus of programs must be available. For our results in Section 5, we initially had only 8 C programs to examine. Our average, ESP prediction results for these 8 programs were the same as the APHC and DSHC results. After we increased the corpus of 8 C programs to 23 C programs, the average misprediction rate for ESP was 5% lower than the average miss rates for the APHC and DSHC techniques. Second, our approach requires that the feature set be defined. Our results indicate that having too much information does not degrade the ESP

[2]In the future, we plan to investigate how the ESP approach works for languages such as C++ and Scheme as well.

predictions (we have not investigated the impact of not having enough data in the feature set). Third, our current implementation of ESP requires that the neural net be trained. Such training requires someone who understands neural nets fairly well, probably at the level of a person who has taken a course in neural nets. We envision that if the ESP approach becomes sufficiently widespread, then tools that facilitate such training would be made available. We also note that preliminary results we have obtained using decision trees instead of neural networks are comparable to the neural net results presented here. Moreover, decision trees are easier to use and the knowledge they encode can be automatically translated into simple if-then rules.

4 Evaluation Methods

To perform our evaluation, we collected information from 43 C and Fortran programs. During our study, we instrumented the programs from the SPEC92 benchmark suite and other programs, including many from the Perfect Club [4] suite. We used ATOM [20] to instrument the programs. Due to the structure of ATOM, we did not need to record traces and could trace very long-running programs. The programs were compiled on a DEC 3000-400 using the Alpha AXP-21064 processor using either the DEC C or FORTRAN compilers. Most programs were compiled using the standard OSF/1 V1.2 operating systems; other programs were compiled using different compilers and different versions of the operating system. Most programs were compiled with standard optimization (`-O`). Each program was run once to collect information about branch frequency and the percentage of "taken" branches. For the SPEC92 programs, we used the largest input distributed with the SPEC92 suite.

Table 3 shows the basic statistics for the programs we instrumented. The first column lists the number of instructions traced and the second column gives the percentage of instructions that are conditional branches. The third column gives the percentage of conditional branches that are taken. The columns labeled 'Q-50', 'Q-75', 'Q-90', 'Q-95', 'Q-99', and 'Q-100' show the number of branch instruction sites that contribute 50, 75, 90, 95, 99 and 100% of all the executed conditional branches in the program. The next column 'Static' shows the total number of conditional branch sites in each program. Thus, in Alvinn, two branch instructions constitute over 90% of all executed branches and correctly predicting these two conditional branches is very important.

The ATOM instrumentation tool provides a concrete representation of the program, and we used this information to construct a control flow graph. Using the control flow graph, we computed the dominator and post-dominator trees. Following this, we determined the natural loop headers and applied the same definition of natural loops used by Ball and Larus to determine the loop bodies [3]. We used ATOM to reproduce the Ball and Larus APHC results, and to generate the static feature sets with the corresponding branch probabilities which are used to train the neural net for ESP.

For ESP, we did not use the information gathered about a given program to predict the branches for that same program; rather, we used a *cross validation study*. We took all of the programs, except the one program for which we want to gather prediction results and fed the corpus of programs into the neural net. We then use the neural net's branch probabilities to predict branches for that program not included in the corpus. This provides a conservative estimate of how well ESP will perform since we are predicting the behavior of a program that the neural net has not seen. For the ESP results shown in Section 5, we performed the cross validation breaking the programs into two groups – C programs and FORTRAN programs. We performed cross validation feeding the feature sets for 22 of the C programs at a time into the neural net, predicting branches for the 23rd C program not included in initial 22. We did the same for FORTRAN programs feeding into the neural net the feature sets for 19 of the 20 programs in order to predict branches for the 20th program.

5 Results

We now compare the prediction accuracy of *a priori* heuristic combination (APHC) branch prediction [3], the Dempster-Shafer heuristic combination (DSHC) proposed by Wu and Larus [22], and our ESP technique. Following this, we show that the APHC and DSHC techniques are sensitive to differences in system architecture and compilers.

5.1 Comparison: APHC, DSHC and ESP

Table 4 shows the branch misprediction rate for the methods we implemented. The first column shows the results for the BTFNT architecture, the second column shows the results for our implementation of the Ball and Larus heuristics, and the third and fourth columns show the results when applying Dempster-Shafer to those heuristics. In implementing DSHC, we use both the original prediction rates specified in [3], DSHC(B&L), and the prediction rates produced by our implementation, DSHC(Ours). Later, we compare the similarity between these two sets of prediction heuristics as seen in Table 6. The fifth column in Table 4 shows the results for our ESP method and the last column shows the results for the perfect static profile prediction. Table 4 reveals several interesting points. First, the overall average shows that the Dempster-Shafer method performs no better than the fixed order of heuristics. Wu and Larus [22] said

> *When more than one heuristic applies to a branch, combining the probabilities estimated by the applicable heuristics should produce an overall branch probability that is more accurate than the individual probabilities.*

However, there was no comparison to the earlier results of Ball and Larus. In 6 cases (`flex, sort, mdljsp2, CSS, NAS, TFS`), the Dempster-Shafer method is more than 5% worse than the simple APHC ordering, while the APHC ordering method is 5% worse in only three cases (`wdiff, SDS, LWS`). The intuition in [22] was correct; however, the Dempster-Shafer theory does not combine the evidence well enough to improve branch prediction. The ESP technique performs significantly better than the Dempster-Shafer and the APHC method in 15 cases (`burg`, `flex`, `gzip`, `indent`,

	# Insn's	% Cond		Conditional Branch Quantiles						
Program	Traced	Branches	%Taken	Q-50	Q-75	Q-90	Q-95	Q-99	Q-100	Static
bc	93,395,683	10.06	42.43	41	97	160	204	273	753	1,956
bison	6,344,388	10.02	76.83	16	89	197	311	654	1,348	2,905
burg	721,029	12.17	62.32	30	84	153	220	465	802	1,766
flex	15,458,984	12.89	68.37	29	102	190	260	421	1,204	2,969
grep	745,131	19.35	72.40	6	25	94	196	422	910	3,310
gzip	309,547,166	11.08	60.75	3	13	29	36	49	342	2,476
indent	32,569,634	14.72	51.91	27	74	159	244	457	1,065	2,272
od	210,341,272	12.88	45.72	30	56	76	84	118	433	1,702
perl	181,256,552	10.26	39.89	28	88	233	342	719	2,690	12,288
sed	85,604,071	10.63	65.55	16	59	91	109	151	863	2,570
siod	28,750,877	13.04	56.85	14	38	95	128	186	684	2,156
sort	10,301,164	14.01	59.12	13	24	51	63	77	352	1,810
tex	147,820,930	7.58	57.47	39	111	259	416	790	2,365	6,050
wdiff	76,185,396	13.21	53.65	7	11	19	24	29	502	1,618
yacr	1,017,126,630	19.24	70.73	11	33	88	127	345	1,673	3,442
alvinn	5,240,969,586	8.93	97.77	2	2	2	3	102	430	1,622
compress	92,629,658	12.31	68.25	4	7	12	14	16	230	1,124
ear	17,005,801,014	4.97	90.13	2	4	6	8	32	530	1,846
eqntott	1,810,540,418	10.78	90.30	2	2	14	42	72	466	1,536
espresso	513,008,174	15.96	61.90	44	104	163	221	470	1,737	4,568
gcc	143,737,915	12.60	59.42	245	804	1,612	2,309	3,724	7,640	16,294
li	1,355,059,387	11.30	47.30	16	33	52	80	127	556	2,428
sc	1,450,134,411	17.99	66.88	14	41	94	153	336	1,471	4,478
doduc	1,149,864,756	6.94	48.68	3	40	175	231	296	1,447	7,073
fpppp	4,333,190,877	2.44	47.74	10	28	51	73	109	744	6,260
hydro2d	5,682,546,752	6.02	73.34	14	43	74	111	230	1,613	7,088
mdljsp2	3,343,833,266	10.12	83.62	6	10	14	16	23	1,010	6,789
nasa7	6,128,388,651	2.51	79.29	8	21	55	94	277	1,083	6,581
ora	6,036,097,925	5.25	53.24	5	8	11	12	17	641	5,899
spice	16,148,172,565	11.51	71.63	2	12	38	63	116	1,762	9,089
su2cor	4,776,762,363	3.34	73.07	8	15	26	34	60	1,569	7,246
swm256	11,037,397,884	1.65	98.42	2	2	3	3	13	795	6,080
tomcatv	899,655,317	3.35	99.28	3	4	5	7	7	515	5,474
wave5	3,554,909,341	4.37	61.79	18	40	82	132	276	1,331	8,149
APS	1,490,454,770	3.99	50.64	44	123	283	357	524	1,617	8,926
CSS	379,319,722	7.32	55.63	32	109	211	262	467	2,202	9,670
LWS	14,183,394,882	7.92	66.34	3	9	18	26	38	1,148	6,927
NAS	3,603,798,937	3.43	60.67	5	14	34	69	125	1,663	7,614
OCS	5,187,329,629	3.02	88.57	3	10	46	79	197	1,447	7,084
SDS	1,108,675,255	6.77	53.05	9	25	43	67	169	1,669	7,585
TFS	1,694,450,064	3.17	77.42	15	38	122	220	464	1,598	7,270
TIS	1,722,430,820	5.27	51.08	8	20	31	36	66	863	6,292
WSS	5,422,412,141	4.76	62.36	41	145	275	344	533	1,756	7,592

Table 3: Measured attributes of the traced programs.

	Branch Prediction Miss Rates					
Program	BTFNT	APHC (B&L's)	DSHC (B&L's)	DSHC (Ours)	ESP	Perfect
bc	40	37	35	35	32	14
bison	52	15	16	16	14	4
burg	53	35	33	32	26	9
flex	43	33	39	38	19	9
grep	42	27	23	22	19	12
gzip	33	32	33	33	20	9
indent	42	27	28	27	19	6
od	44	44	40	40	30	8
perl	35	39	36	36	26	4
sed	45	22	22	23	25	5
siod	50	34	32	33	27	10
sort	44	35	41	42	21	8
tex	43	37	38	36	30	13
wdiff	42	32	11	11	4	3
yacr	32	14	11	12	14	6
Other C Avg	43	31	29	29	22	8
alvinn	2	2	2	2	1	0
ear	10	8	8	8	8	7
compress	44	25	26	28	30	14
eqntott	47	7	7	7	6	2
espresso	34	24	23	23	32	15
gcc	48	34	35	34	31	12
li	43	26	25	27	28	12
sc	39	29	31	29	24	9
SPEC C Avg	34	19	20	20	20	9
dodoc	23	19	20	19	16	5
fpppp	42	53	52	52	35	11
hydro2d	28	17	16	16	12	4
mdljsp2	69	41	62	62	64	10
nasa7	8	12	12	11	5	3
ora	46	18	18	18	18	5
spice	16	16	18	14	14	7
su2cor	17	21	20	20	12	10
swm256	1	1	1	1	1	1
tomcatv	44	44	44	44	1	1
wave5	19	27	24	23	21	6
SPEC Fortran Avg	29	24	26	25	18	6
APS	28	30	34	31	26	10
CSS	39	29	40	36	33	9
LWS	38	32	25	25	18	16
NAS	42	12	22	22	12	4
OCS	4	6	5	5	4	2
SDS	18	32	25	19	21	12
TFS	12	10	15	13	11	6
TIS	18	26	25	22	16	16
WSS	32	28	26	26	25	11
Perf Club Avg	26	23	24	22	18	10
Overall Avg	34	25	26	25	20	8

Table 4: Comparison of using Heuristics in Ball and Larus ordering, Dempster-Shafer Theory and ESP. The first column shows the misprediction rate of the BTFNT approach. The second column shows the miss rate for our implementation of the APHC method of Ball and Larus. We computed the Dempster-Shafer miss rates, shown in column three, with the same values for the heuristics used by Wu and Larus as well as the values we computed, shown in column four. The fifth column is the miss rate for the ESP technique, while the last column is the miss rate for perfect static profile prediction. In each case, smaller values are better.

od, perl, siod, sort, tex, wdiff, fpppp, su2cor, tomcatv, LWS, and TIS), and has significantly worse performance in only one case (mdljsp2).

We feel that the ESP results may be improved by expanding the feature sets used. We used a limited number of "features" in the feature set to distinguish branches, primarily using the features described by Ball and Larus. To extend the set of features, we need to determine what new features (e.g., information from the control dependence graph) we want to include, capture that information during program instrumentation, and pass those features to the neural net. This is a simple process, but we have only examined a small set of the possible features. Rather than rely on intuition about the appropriate features (e.g,. by using the Ball and Larus predictors), we should provide as much information to the neural network as possible and let it decide the importance.

5.2 Cross-Architectural Study of A Priori Heuristics

In the paper by Ball and Larus [3], a number of *prediction heuristics* were described. These heuristics were the foundation for the prediction scheme in both the study by Ball and Larus and the study by Wu and Larus. In the study by Wu and Larus, the values given in [3] were used for the Dempster-Shafer combination method, even though the study by Wu and Larus used a different architecture, compiler and runtime system. We wondered how sensitive these metrics were to differences in architecture, compiler, runtime system and selection of programs.

We use the CFG, dominator, post-dominator and loop information to implement the same heuristics in [3], summarized in Table 1. Our implementation results for these heuristics are shown in Table 5. This table shows detailed information about how the branch heuristics performed for each program. Some of the programs in our suite were also used in the earlier study by Ball [3], and the values in parenthesis show the equivalent metrics recorded in that study. In general, the values are quite similar, but there are some small differences that we believe arise from different runtime libraries. For example, a binary-buddy memory allocator would not contain any loops, while a coalescing implementation may contain several loops. These library routines are part of the native operating system, and not part of the distributed benchmark suite. Note that there are considerable differences, in the percentage of non-loop branches, particularly in eqntott. Some of these differences are caused by libraries and runtime systems, but others can be attributed to architectural features. For example, the Alpha has a "conditional move" operation that obliviates the need for many short conditional branches, reducing the number of conditional branches that are executed.

Table 5 further demonstrates that our implementation of the heuristics listed in [3] appear to be correct. The loop miss rates are roughly the same, the heuristics cover approximately the same percentage of branches and the overall branch prediction miss rates are similar. There are some differences, but after some investigation, we have attributed most of these to different architectures, operating systems and compilers.

Table 6 shows the comparison of the overall averages for the heuristics comparing the Ball and Larus results on the MIPS architecture to our results on the Alpha. This table also shows the probablilites used in the DSHC results shown in Table 4. The B&L miss rates were used for the DSHC(B&L) probabilities and our Overall miss rates in Table 6 were used for the DSHC(Ours) probablilities in Table 4.

We felt that the differences seen in Table 6 were to be expected, because the two studies used a different collection of programs with different compilers that implement different optimizations for different architectures and used different runtime libraries. Table 6 supports our position that at least some of Ball and Larus heuristics are quite language dependent. First, we point out that pointers are very rare in FORTRAN, and as such the great success of the Pointer heuristic in FORTRAN is of little consequence because it applies to very few branches. Next, we see that while the Store heuristic appears successful in our FORTRAN programs, it performs much worse in our C programs. Conversely, the Loop Header heuristic performs well in C programs, but poorly in FORTRAN programs. Overall, four of the nine heuristics show a difference of greater than 10% in their miss rates when our C and Fortran programs are compared.

5.2.1 The Influence of Architectures

In certain cases, we had slightly different implementations of heuristics than Ball and Larus because the Alpha architecture did not allow us to implement the heuristics as originally stated. For example, with respect to the Opcode heuristic, the Alpha architecture has two types of branch instructions; one compares floating point numbers to zero and the other integer numbers to zero. The conditional branch instructions always compare a register to zero. On the MIPS architecture, the "branch if equal" (BEQ) and "branch if not-equal" (BNE) instructions compares two registers. To accomplish the same task on the Alpha, an earlier comparison must be made between the two registers, and the resulting value is then compared to zero.

However, our implementation of the heuristics took these factors into account, constructing an abstract syntax tree from the program binary and using that to determine the outcome of the conditional branch. Clearly, determining this information at compile time would simplify the analysis, because we could use more information from the program. However, both Ball and Larus [3] and our study used binary instrumentation, so we felt that other factors must also contribute to the prediction differences. We examined one program for which the Ball and Larus heuristics provided good prediction accuracy, tomcatv in more detail, since our implementation of those heuristics provided worse prediction accuracy (see Table 5). On the Alpha, tomcatv spends 99% of its execution in one procedure. Furthermore, most of the basic block transitions in that procedure involve three basic blocks, shown in Figure 2. The edge from block 32 $\rightarrow$ 28 is a loop back edge, and our heuristics indicate this correctly. However, the remaining two conditional branches only match the "guard" heuristic in the heuristics described by Ball and Larus. However, their study indicated that tomcatv benefited from the "store" heuristic, which predicts that basic blocks with store instructions following a conditional branch are not taken. By comparison, on the Alpha, none of the successors of block 28 (blocks 29

	Loop Branches	Non-Loop Branches				
Miss Rate	Miss Rate For Loops	%Non-Loop Branches	%Branches Covered By Heuristics	Miss Rate For Hueristics	Miss Rate With Default	Overall Miss Rate
bc	39	74	80	30	36	37
bison	12	64	84	15	18	15
burg	22	66	80	39	42	35
flex	15	60	58	38	46	33
grep	9	60	89	36	39	27
gzip	4	48	31	45	62	32
indent	27	69	77	23	27	27
od	56	83	74	43	42	44
perl	43	69	80	34	38	39
sed	19	54	78	19	25	22
siod	34	74	59	29	34	34
sort	17	63	66	50	45	35
tex	33	51	78	40	41	37
wdiff	11	65	100	44	44	32
yacr	4	37	85	24	31	14
Other C Avg	23	62	75	34	38	31
alvinn	0	3	65	40	42	2
compress	8 (12)	57 (66)	80 (90)	38 (39)	38 (40)	25 (30)
ear	2	17	96	41	41	8
eqntott	2 (3)	11 (49)	75 (5)	40 (37)	45 (50)	7 (26)
espresso	17 (18)	45 (37)	73 (44)	26 (25)	33 (26)	24 (21)
gcc	25 (22)	72 (73)	79 (79)	33 (32)	37 (37)	34 (33)
li	28 (28)	61 (62)	87 (90)	22 (25)	25 (28)	26 (28)
sc	10	64	76	40	40	29
SPEC C Avg	11	41	79	35	38	19
doduc	10 (8)	42 (52)	69 (92)	23 (31)	31 (33)	19 (21)
fpppp	28 (34)	70 (86)	61 (82)	63 (40)	64 (42)	53 (41)
hydro2d	3	52	88	25	31	17
mdljsp2	9	81	33	38	49	41
nasa7	3 (1)	24 (10)	66 (95)	33 (29)	38 (32)	12 (4)
ora	3	64	57	15	27	18
spice	9 (9)	23 (21)	61 (75)	27 (33)	38 (36)	16 (14)
su2cor	1	44	78	46	47	21
swm256	1	1	65	9	13	1
tomcatv	1 (1)	43 (38)	100 (100)	99 (1)	99 (2)	44 (1)
wave5	10	50	82	45	44	27
SPEC Fortran Avg	7	45	69	39	44	24
APS	26	52	62	25	33	30
CSS	22	62	57	35	34	29
LWS	15	60	62	26	44	32
NAS	5	74	38	10	14	12
OCS	3	10	54	15	31	6
SDS	22	36	58	26	48	32
TFS	6	24	76	14	23	10
TIS	22	40	44	20	32	26
WSS	18	40	56	33	43	28
Perf Club Avg	16	44	56	23	34	23
Common Avg	13 (14)	45 (49)	75 (75)	41 (29)	45 (33)	26 (22)
Overall Avg	15	50	70	33	38	25

Table 5: Results for the Program-Based Heuristic Approaches. The first column lists the miss rate for loop branches. The second column shows the percentage of non-loop branches. The third column shows the dynamic percentage of non-loop branches that can be predicted using one of the heuristics, while the fourth column shows the miss rate achieved when using those heuristics. For example, 80% of the non-loop branches in `compress` can be predicted using some heuristic, and those heuristics have a 38% miss rate. Branches that can not be predicted using the heuristics are predicted using a uniform random distribution. The fifth column shows the prediction miss rate for the execution of all non-loop branches, combining the predictions from the heuristics and the random predictions. Lastly, the sixth column lists the misprediction rate when both loop and non-loop branches are included.

Heuristic	Branch Prediction Miss Rates			
	B&L (MIPS)	Our Implementation (ALPHA)		
		C	FORTRAN	Overall
Loop Branch	12%	17%	12%	15%
Pointer	40%	58%	1%	55%
Call	22%	23%	44%	31%
Opcode	16%	33%	29%	32%
Loop Exit	20%	28%	30%	29%
Return	28%	29%	30%	30%
Store	45%	52%	30%	42%
Loop Header	25%	33%	48%	40%
Guard	38%	34%	31%	33%

Table 6: Comparison of Branch Miss Rates for Prediction Heuristics. These averages are for all the programs we simulated and a program is only included in a heuristic's average if the heuristic applies to at least 1% of the branches in the program.

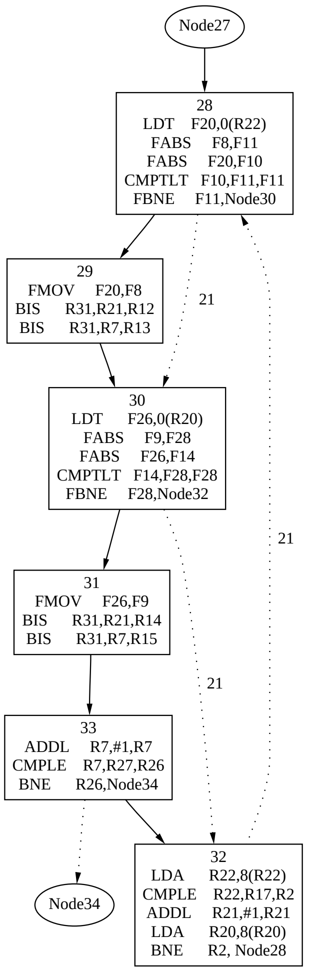

Figure 2: Sample code fragment from TOMCATV benchmark that continues most of the branches in the program. The numbers on the edges indicate the percentage of all edge transitions attributed to a particular edge. The dotted edges indicate taken branches.

and 30) or block 30 (blocks 31 and 32) contain store instructions. This difference may be attributed to different register scheduling or register saving conventions, requiring a store on the MIPS, but not on the Alpha. The "guard" heuristic still applies, but predicts both branches in blocks 28 and 30 incorrectly.

5.2.2 The Influence of Compilers and Optimizations

To further validate our belief that the choice of compilers influences the prediction accuracy of the various heuristics, we compiled one program, `espresso`, with the following compilers: `cc` on OSF/1 V1.2, `cc` on OSF/1 V2.0, the DEC GEM C compiler and the Gnu C compiler. The results are shown in Table 7. In terms of the overall miss rate, the compilers all show different behavior. Also note that the DEC GEM C compiler produced significantly fewer loop branches, and resulted in a program approximately 15% faster than the other compilers. The GEM compiler unrolled one loop in the main routine, inserting more forward branches and reducing the dynamic frequency of loop edges.

This simple optimization changed the characteristics of the branches in the program and the efficacy of the APHC branch prediction technique. The difference caused by loop-unrolling is significant if we want to use branch probabilities after traditional optimizations have been applied. However, many programmers unroll loops "by hand" and other programmers use source-to-source restructuring tools, such as KAP or VAST. The differences evinced by these applications may render the fixed ordering of heuristics inappropriate for some programs.

Our validation study confirmed an underlying assumption in our work: heuristic-based branch prediction rates vary with programs, program style, compiler, architecture, and runtime system. Rather than choosing a set of heuristics based on the intuition of a few people, we have devised a program-based prediction mechanism that can be adapted to the techniques, style and mechanisms of different programmers, languages and systems. Furthermore, the corpus-based approach means our prediction technique can be customized to specific groups or customers.

Program	O/S	Compiler	Loop Branches	Non-Loop Branches		Overall	Perfect
			Miss Rate	%Non-Loop Branches	Heuristic Miss Rate	Miss Rate	Miss Rate
Espresso	1.2	cc	17	45	26	24	15
Espresso	2.0	cc	18	46	27	25	15
Espresso	2.0	GEM C	25	57	26	32	12
Espresso	2.0	Gnu C	17	46	23	22	15

Table 7: Comparison of Accuracy of Prediction Heuristics Using Different Compilers

6 Summary

Branch prediction is very important in modern computer architectures. In this paper, we investigate methods for static program-based branch prediction. Such methods are important because they do not require complex hardware or time-consuming profiling. We propose a new, general approach to program-based behavior estimation called evidence-based static prediction (ESP). We then show how our general approach can be applied specifically to the problem of program-based branch prediction. The main idea of ESP is that the behavior of a corpus of programs can be used to infer the behavior of new programs. In this paper, we use a neural network to map static features associated with each branch to the probability that the branch will be taken.

ESP has the following advantages over existing program-based approaches to branch prediction. First, instead of being based on heuristics, it is based on a corpus of information about actual program behavior and structure. We have observed that the effectiveness of heuristic approaches to branch prediction can be architecture, compiler, and language dependent. Thus, ESP can be specialized easily to work with new and different programming languages, compilers, computer architectures, or runtime systems. It is our hope that it can even be customized for specific application domains, or workgroups with a modest amount of effort.

Second, the ESP approach does not require careful consideration when deciding what features to include in the training data. The neural net we use is capable of ignoring information that is irrelevant and such information does not degrade the performance of the predicted branch probabilities. On the other hand, with heuristic methods, trial-and-error is often required to find heuristics that are effective.

Finally, we have shown that the ESP approach results in branch prediction miss rates that are better than the best program-based heuristic approaches. Over a collection of 43 C and Fortran programs, the overall miss rate of ESP branch prediction was 20%, which compares against the 25% miss rate using a fixed ordering of the Ball and Larus heuristics (the best heuristic method), and the overall 8% miss rate of the perfect static-profile predictor.

We see many future directions to take with this work. Currently, the neural network we use not only provides a prediction for each branch, but also provides its estimate of the branch probability. If that probability is $> 50\%$ we estimate that the branch will be taken. Our next goal will be to incorporate this branch probability data to perform program-based profile estimation using ESP. It is simple to add more "features" into our training information; for example, we plan on indicating branches in library subroutines, since that those subroutines may have similar behavior across a number of programs. We also plan to gather large bodies of programs in other programming languages, such as C++ and Scheme, and evaluate how ESP branch prediction works for those languages. We are also interested in seeing how effective other classification techniques, such as memory-based reasoning or decision trees, will be for ESP prediction. Finally, we are interested in comparing the effectiveness of using ESP prediction techniques against using profile-based methods across a range of optimization problems.

We also see other possible uses of the ESP approach that supplement profile-based prediction techniques. We expect that organizations and workgroups might use their own programs to "train" the ESP system. They could then use program-based information for most compilations, and use profile-based information for performance-critical compilations. Likewise, computer vendors may provide several trained ESP predictors, based on program type or language.

Acknowledgements

We would like to thank Alan Eustace and Amitabh Srivastava for developing ATOM, and James Larus for motivating this paper. Brad Calder was supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland. This work was funded in part by NSF grant No. ASC-9217394, NSF grant No. CCR-9404669, ARPA contract ARMY DABT63-94-C-0029 and a software grant from Digital Equipment Corp.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[2] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 213–223, July 1991.

[3] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[4] M. Berry. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[5] Brad Calder and Dirk Grunwald. Fast & accurate instruction fetch and branch prediction. In *21st Annual International Symposium on Computer Architecture*, pages 2–11. ACM, April 1994.

[6] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Six International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251. ACM, 1994.

[7] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4), 1994. Also available as University of Colorado Technical Report CU-CS-698-94.

[8] P. P. Chang and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–376, 1992.

[9] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic compiler code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[10] A. P. Dempster. A generalization of bayesian inference. *Journal of the Royal Statistical Society*, 30:205–247, 1968.

[11] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 85–95, Boston, Mass., October 1992. ACM.

[12] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[13] Wen-mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual International Symposium on Computer Architecture*, pages 242–251. ACM, 1989.

[14] Scott McFarling and John Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. Association for Computing Machinery, 1986.

[15] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Parallel distributed processing: Explorations in the microstructure of cognition. Volume I: Foundations*, chapter Learning internal representations by error propagation, pages 318–362. MIT Press/Bradford Books, Cambridge, MA, 1986. D. E. Rumelhart and J. L. McClelland, editors.

[17] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, NJ, 1976.

[18] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.

[19] P. Smolensky, M. C. Mozer, and D. E. Rumelhart, editors. *Mathematical perspectives on neural networks*. Erlbaum, 1994. In press.

[20] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[21] Tim A. Wagner, Vance Maverick, Susan Graham, and Michael Harrison. Accurate static estimators for program optimization. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida, June 1994. ACM.

[22] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *27th International Symposium on Microarchitecture*, San Jose, Ca, November 1994. IEEE.

[23] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.