Evidence-based Static Branch Prediction using Machine Learning

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

September 19, 1996

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 neural networks and decision trees to map static features associated with each branch to a prediction 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 static 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. We provide results showing how sensitive ESP is to the number and type of static features and programs included in the ESP training sets, and compare the efficacy of static branch prediction for subroutine libraries. 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

---

[0] Currently at Department of Computer Science and Engineering 0114, University of California, San Diego, 9500 Gilman Road La Jolla, CA 92093-0114.

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 [15, 14, 17] and other profile-based optimizations [10, 11].

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 cycles on the Digital Alpha AXP 21064 processor and 5 cycles in the Alpha AXP 21164 processor. In previous studies, we found that conditional branches in C programs were executed approximately every 8 instructions on the Alpha architecture [9]. 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 [6, 28] while others involve software [4, 7, 13]. 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 estimating the behavior of branches at compile-time: profile-based and program-based 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 at predicting the future behavior of branches [13]. 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 [18, 23]. Other techniques rely on applying heuristics based on less local program structure in an effort to predict branch behavior [4]. In [15], Hank et al. showed that these program-based heuristics can be used to accurately guide profile-based compiler optimizations achieving performance improvements close to what is achieved if real profiles were used.

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. Thus, our technique has two phases. The first phase, performed once (e.g., when the compiler is constructed) involves profiling the corpus of programs. The second phase, which is entirely program-based but which uses the results of the first phase, predicts the branches in a new program. In this paper we use neural networks and decision trees to map static features associated with each branch to a prediction 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 generates predictions automatically, the predictions can be specialized based on specific languages, compilers, and computer architectures. Existing techniques rely on heuristics defined by compiler writers that are based on intuition and empirical studies (e.g., [23]) 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. We discuss some problems that occur with the previous program-based techniques in §3. In Section 4 we discuss the details of our ESP branch prediction method. Section 5 describes the methods we used to evaluate and compare ESP prediction with previous approaches, and Section 6 presents our results. We summarize our conclusions in Section 7 and also discuss possible future directions to take with this research.

2 Background

In this section, we discuss the existing approaches to program-based static branch prediction and also discuss other knowledge-based approaches to compiler optimization. Most of the previous work in program-based branch prediction uses the final program binary to determine the outcome of branches; following those decisions, the program can be reorganized [19, 7] to take advantage of the program-based branch prediction information.

2.1 Program-Based Branch Prediction Methods

One of the simplest 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. Our results in Section 6 show it has an overall miss rate in our experiments of 34%. While simple, BTFNT is also quite successful,

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

since many programs spend a lot of time executing inside of loops and the backwards branch in a loop is correctly predicted as taken when using the BTFNT heuristic.

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 [13] or program-based techniques to determine what branches are likely to be taken. In recent work, Ball and Larus [4] 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 [27]). 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. [4]” 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 predetermined 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%; when their technique is applied to the DEC Alpha architecture, the prediction accuracy worsens, becoming 25%.

In a related paper, Wu and Larus refined the APHC method of Ball and Larus [27]. In that paper, their goal was to determine branch *probabilities* instead of simple branch prediction. 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 [12, 22]. We call this form of evidence combination *Dempster-Shafer Heuristic Combination* (DSHC). By making some fairly strong assumptions concerning the independence of different attributes, 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.

Unless it is shown that the data is truly independent, and that other restrictions described in [12, 22] are observed, the Dempster-Shafer mechanism may not provide any useful information. The probabilities that Wu and Larus use are taken directly from the paper of Ball and Larus [4]. We refer to a DSHC algorithm based on this data as DSHC(B&L). 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. As we show later, the DSHC method provides worse prediction than the simple APHC method.

Wagner et al. [26] 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 [3], 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 off-line 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 Problems with Apriori Program Estimation

In the paper by Ball and Larus [4], 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 [4] were used for the Dempster-Shafer combination method, even though the study by Wu and Larus used a different architecture, compiler and run-time system. In this section, we show that the apriori heuristics described in [4] are sensitive to differences in architecture, compiler, run-time system and selection of programs. This sensitivity implies that it is important to develop automatic techniques for branch estimation, and an understanding of the sensitivity of those automatic techniques.

We use the CFG, dominator, post-dominator and loop information to implement the same heuristics used in [4], summarized in Table 1. Our implementation results for these heuristics are shown in Table 2. 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 [4], 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 run-time 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 run-time systems, but others can be attributed to architectural features. For example, the Alpha has a “conditional move” that avoids the need for many short conditional branches, reducing the number of conditional branches that are executed. Table 2 further demonstrates that our implementation of the heuristics listed in [4] appear to be correctly implemented. 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.

	Loop Branches	Non-Loop Branches				
Miss Rate	Miss Rate For Loops	%Non-Loop Branches (dynamic)	%Branches Covered By Heuristics (dynamic)	Miss Rate For Heuristics	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 2: 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.

	Branch Prediction Miss Rates			
Heuristic	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 3: 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.

Table 3 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. We felt that the differences seen in Table 3 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 run-time libraries. Table 3 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.

3.1 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, consider implementing 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

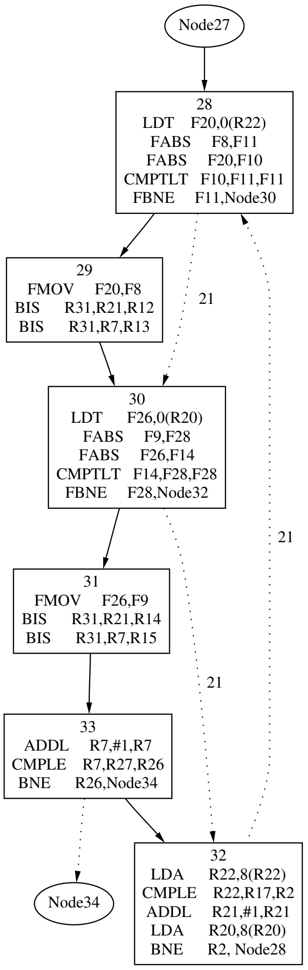

Figure 1: Sample code fragment from TOMCATV benchmark that continues most of the branches in the program. The numbers on the edges indicate the percentage of all edge transitions attributed to a particular edge. The dotted edges indicate taken branches.

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.

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, since more program information would be available. Both Ball and Larus [4] 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 2). 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 1. The edge from block 32 → 28 is a loop back edge, and our implementations identify and predict this correctly. Of the the remaining three conditional branches, both of the two important ones (labelled 28 and 30 in the figure) only match the "guard" heuristic in the heuristics described by Ball and Larus (see Table 1). 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 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.

3.2 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 4. The last column of the table is the miss rate for perfect static profile prediction. In terms of the overall miss rate, the compilers all show different behavior. 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.

Program	O/S	Compiler	Loop Branches	Non-Loop Branches		Overall Miss Rate	Perfect Miss Rate
			Miss Rate	%Non-Loop Branches (dynamic)	Heuristic 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 4: Comparison of Accuracy of Prediction Heuristics Using Different Compilers

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 [16] or VAST [2]. The differences evinced by these applications may render the fixed ordering of heuristics ineffective for some programs.

4 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.

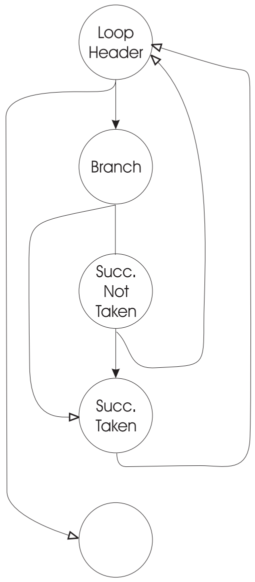

Figure 2: Simple control flow graph used to describe the feature set of branch instructions. The vertical ordering of the nodes indicates their order in the program binary. The conditional branch is followed by the "successor not taken" – the node in the CFG that is the "fall through" of the conditional branch. Likewise, the "successor taken" is the node representing the target of the branch.

With this broad definition of our framework in mind, we now describe how we apply this general framework to the specific problem of static branch prediction. We first describe the *features* we extract from the test programs for branch prediction. We then describe two machine-learning systems (based on neural networks and decision trees) that use the data to make decisions on branches seen in other programs.

4.1 ESP Branch Prediction

In applying ESP to the problem of branch prediction, we instantiate the above framework in the following way. For each branch instruction in the program text, we record a large static feature set (see Table 5). 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). Despite being a RISC processor, the Alpha instruction set contains a wealth of instructions. Some machine learning algorithms are confused by too many choices when fitting empirical data, and this results in no

Feat. Num.	Feature Name	Feature Description
1	Br. Instruction	The opcode of branch instruction.
2	Br. direction	F — Forward branch, B — Backwards branch
3	Br. operand opcode	The opcode of the instruction that defines the register used in the branch instruction (or ?, if the branch operand is defined in a previous basic block).
4	Br. operand Func.	The branch function (see text).
5	Br. operand type	The operation type (see text).
6	RA opcode	If the instruction in (3) uses an RA register, this is the opcode of the instruction that defines that register (? otherwise).
7	RA function	The RA function (see text).
8	RA type	The RA type (see text).
9	RB opcode	If the instruction in (3) uses an RB register, this is the opcode of the instruction that defines that register (? otherwise).
10	RB function	(As "RA function" above, but for RB register)
11	RB type	(As "RA type" above, but for RB register)
12	Loop header	LH — the basic block is a loop header, NLH - not a loop header
13	Language	The language of the procedure the branch is in (C or FORT).
14	Procedure type	The branches' procedure is a Leaf, NonLeaf or calls itself recursively (CallSelf)
15–22	Features of the Taken Successor of the Branch	
15	Br. dominates	D — basic block dominates this successor, or ND — does not dominate
16	Br. postdominates	PD — the successor basic block post-dominates the basic block with the branch, or NPD — does not post-dominate
17	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, or NOTHING)
18	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
19	Succ. Backedge	LB — the edge getting to the successor is a loop back edge, NLB — not a loop back edge
20	Succ. Exit	LE — the edge getting to the successor is a loop exit edge, NLE — not a loop exit edge
21	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
22	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
23–30	Features of the Not Taken Successor of the Branch Same as above features 15–22	

Table 5: Static Feature Set Used in the ESP Branch Prediction Study.

apparent pattern emerging from the data. For example, the instructions `subq` and `subl` both perform a subtraction, but the first uses quad-words and the latter long-words. It may be that the important aspect of those instructions is that a subtraction occurs. We aggregated certain instructions into an "operand function" that captures that information. Likewise, the important aspect may be that a quadword is used, rather than the specific arithmetic function that is applied. The "operand type" field encodes the type of the operand. A conditional branch may arise from a complex instruction sequence used to produce the left and right hand side of the comparisons. We constructed and encoded an abstract syntax tree representing those comparisons as the "RA" and "RB" features. Although not precisely correct, it is useful to thing to think of this as encoding comparisons of the form "if ($(RA_1 op_1 RA_2)$ op_2 $(RB_1 op_3 RB_2)$) . . ." – the abstract syntax tree includes information establishing the "context" of the conditional branch.

The majority of the features are related to the instructions following the branch (features 15–30). It is easiest to understand those features by considering the control flow graph in Figure 2. In that figure, the vertical ordering of the nodes indicates their order in the program binary. The conditional branch is followed by the "successor not taken" – the node in the CFG that is the "fall through" of the conditional branch. Likewise, the "successor taken" is the node representing the target of the branch. The feature "Successor not taken ends" encodes the branch type of the instruction ending the "successor not taken" node. The existence of some features is dependent on the values of other features. For example, feature 4 is only meaningful if feature 3 has an RA operand. We call such features *dependent static features*.

We chose the feature set shown in Table 5 based on several criteria. First, we encoded information that we believed would likely be predictive of behavior, even though we conducted no initial experimentation to confirm our intuition. 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 encoded 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. Later in the paper, we examine the predictive power of individual features in the feature set, and combinations of a subset of those features. We will show that the machine-learning algorithms are able to effectively select and combine the useful features from the set we have chosen. We note that the feature set listed here is the only one we have tried. It may be possible that extending the number of available features would improve the resulting branch prediction.

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 the static feature set, the normalized branch weight and the branch probability (percentage of the time the branch was taken) with each branch instruction in the corpus.

4.2 Prediction using Neural Nets

Our goal is to have a system that can effectively predict that a branch will be taken based on 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* [24]. 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 whether the branch will be taken.

Figure 3 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*.

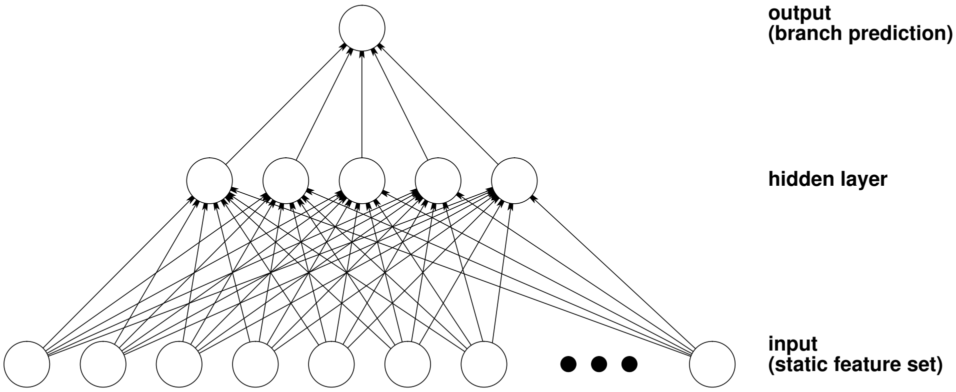

Figure 3: The branch prediction neural network. Each circle represents a processing unit in the network, and the links between units depict the flow of activity.

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* [21]. 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 relative 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 (e.g., the proportion of branches with that particular feature set in the corpus). 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 relative 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 overall 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 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.

4.3 Prediction using Decision Trees

As described in the previous section, the branch prediction task can be formulated as a probability prediction task. Given a branch characterized as a feature vector, we would like to predict the percentage of times such a branch is likely to be taken. It is, however, also possible to cast the branch prediction problem as a *classification problem*; given a branch characterized as a feature vector, we would like to assign that branch to either the class "branch" (taken) or "no branch" (not taken) based on the classification of similar

---

[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.

branches in the training corpus. Casting the problem in these terms permits the use of some simple, yet effective, techniques to build classifiers from training data. Decision tree induction systems represent one such approach.

Decision trees consist of internal test nodes that examine single features of objects, branches emanating from these internal nodes that correspond to each possible result of the test, and leaf nodes that denote object classifications. Operationally, decision trees are used to classify objects by first performing the test specified at the root node and then following the branch indicated by the result of the test down to the next level of the tree. The result of the tests at each successive level determine an object's unique path to a leaf node. When a leaf node is reached, the classification associated with that leaf is assigned to the object.

Note that such decision trees are directly interpretable by experts in the domain of interest, since they are expressed in terms of the features used to describe the objects. Indeed, each path in the tree from the root to a leaf can be thought of as a logical rule where a conjunction of the tests forms the antecedent and the classification specified at the leaf is the consequent. This direct interpretability was one of our primary motivations for investigating the use of decision trees.

Fortunately, there are effective and efficient algorithms for learning decision trees directly from a corpus of data represented as feature vectors with assigned categories (See Quinlan [20] for a survey of such methods.) There are two key notions underlying all of these algorithms. The first is the notion that any diverse collection of objects can be assigned a value based on the heterogeneity of the collection. The second is that a given collection can be partitioned into two or more sub-partitions based on the results of testing a single feature of all the objects in the collection.

Taken together, these two notions suggest a greedy divide-and-conquer approach to tree-construction given a training corpus. The algorithm first assigns as the root node the feature test that results in subpartitions that are maximally homogeneous. In other words, the single feature that best creates a set of partitions that group objects with the same category together and separate objects with differing classifications. Since the resulting partitions are still likely to be diverse, it next recursively builds sub-trees for each partition created by the application of a test to an earlier partition. This recursion typically halts when all the partitions at the leaves of the tree have uniform membership, when there are no more features left to be tested, or when the size of the subpartition is so small that it is unlikely that any further splitting would be reliable.

As with most data-intensive machine learning methods, it is important to avoid having the system memorize, or overfit, the training data. Two approaches have traditionally been taken to address this problem: the system is only allowed to view a fraction of the available training data with the remainder being held back as a test set, and the tree resulting from the training is "pruned" back from the leaves to avoid idiosyncratic overfitting of the data.

The decision tree induction experiments reported here were performed using the well-studied, and widely distributed, C4.5 system [20]. The results reported here were achieved using C4.5's default training settings, along with its standard tree pruning mechanism.

To be more specific, C4.5 requires a training set made up of individual feature vectors labeled with correct classifications and outputs a decision tree capable of classifying such vectors. In our experiments, the first step taken was to assign each feature vector to the category "branch" or "nobranch" based on whether its recorded branch probability was greater than or less than .5. The next task was to capture the fact that the branches contained in the training set displayed widely varying frequencies during the original data gathering. To reflect these differences, copies of each labeled branch were created in sufficient numbers to reflect the proportion of dynamic branches that each branch was responsible for. Due to practical limitations, we limited this copying to $1000 * normalized - frequency$ copies of each branch. This has the effect of eliminating a number of low-frequency branches from the training set while still allowing C4.5 to focus on the important frequent branches.

4.4 Discussion

To summarize, the ESP method automatically extracts the important features that can be used to predict the direction that branches will take. Because the method is automatic, it can be used to specialize predictions based on specific programming languages, compilers, run-time systems, architectures, and application domains. Thus, ESP has advantages over existing methods based on heuristics. In particular, the effectiveness of heuristics may depend on the context in which they are developed and evaluated, such as the programming language or architecture (as we have shown), and given a set of heuristics, there is no clear insight that guides how they can be combined effectively.

The ESP method also has disadvantages, as well. First, a corpus of programs must be available. For our results in Section 6, we initially had only 8 C programs to examine. On 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 in Section 6.3.2 show that as more features are used the miss rate improves, although not monotonically. Finally, if the neural net ESP approach is to be used, it 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 this approach becomes sufficiently widespread, then tools that facilitate such training would be made available. In contrast, decision trees are easier to use for ESP and the knowledge they encode can be automatically translated into relatively transparent if-then rules. The results we have obtained using decision trees are comparable to the neural net results. The C4.5 system was able to build trees with performance on average 1% worse than the corresponding neural net. Therefore, a decision tree approach may be more practical for most users of ESP.

5 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 [5] suite. We used ATOM [25] 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. We used versions of the SPEC benchmark suite compiled on the OSF/1 V1.2 operating system using the same compiler options used to report the official SPEC benchmark suite. During the period we conducted this study, we migrated from the OSF/1 V1.2 operating system to the OSF/1 V2.0 system, and most of the non-SPEC programs were compiled on that system. Unless otherwise noted, the programs were compiled using the normal C compiler, which was a derivative of the MIPS C compiler, or the DEC Fortran compiler, which uses a different code generator. All 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 6 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 last column contains the percentage represented by Q-100/Static, which indicates the fraction of total static branches exercised by the particular input data used.

Program	# Insn's Traced	% Cond Branches	%Taken	Conditional Branch Quantiles						Static	Q-100/ Static (%)
				Q-50	Q-75	Q-90	Q-95	Q-99	Q-100		
bc	93,395,683	10.06	42.43	41	97	160	204	273	753	1,956	38
bison	6,344,388	10.02	76.83	16	89	197	311	654	1,348	2,905	46
burg	721,029	12.17	62.32	30	84	153	220	465	802	1,766	45
flex	15,458,984	12.89	68.37	29	102	190	260	421	1,204	2,969	41
grep	745,131	19.35	72.40	6	25	94	196	422	910	3,310	27
gzip	309,547,166	11.08	60.75	3	13	29	36	49	342	2,476	14
indent	32,569,634	14.72	51.91	27	74	159	244	457	1,065	2,272	47
od	210,341,272	12.88	45.72	30	56	76	84	118	433	1,702	25
perl	181,256,552	10.26	39.89	28	88	233	342	719	2,690	12,288	22
sed	85,604,071	10.63	65.55	16	59	91	109	151	863	2,570	34
siod	28,750,877	13.04	56.85	14	38	95	128	186	684	2,156	32
sort	10,301,164	14.01	59.12	13	24	51	63	77	352	1,810	19
tex	147,820,930	7.58	57.47	39	111	259	416	790	2,365	6,050	39
wdiff	76,185,396	13.21	53.65	7	11	19	24	29	502	1,618	31
yacr	1,017,126,630	19.24	70.73	11	33	88	127	345	1,673	3,442	49
alvinn	5,240,969,586	8.93	97.77	2	2	2	3	102	430	1,622	27
compress	92,629,658	12.31	68.25	4	7	12	14	16	230	1,124	20
ear	17,005,801,014	4.97	90.13	2	4	6	8	32	530	1,846	29
eqntott	1,810,540,418	10.78	90.30	2	2	14	42	72	466	1,536	30
espresso	513,008,174	15.96	61.90	44	104	163	221	470	1,737	4,568	38
gcc	143,737,915	12.60	59.42	245	804	1,612	2,309	3,724	7,640	16,294	47
li	1,355,059,387	11.30	47.30	16	33	52	80	127	556	2,428	23
sc	1,450,134,411	17.99	66.88	14	41	94	153	336	1,471	4,478	33
doduc	1,149,864,756	6.94	48.68	3	40	175	231	296	1,447	7,073	20
fpppp	4,333,190,877	2.44	47.74	10	28	51	73	109	744	6,260	12
hydro2d	5,682,546,752	6.02	73.34	14	43	74	111	230	1,613	7,088	23
mdljsp2	3,343,833,266	10.12	83.62	6	10	14	16	23	1,010	6,789	15
nasa7	6,128,388,651	2.51	79.29	8	21	55	94	277	1,083	6,581	16
ora	6,036,097,925	5.25	53.24	5	8	11	12	17	641	5,899	11
spice	16,148,172,565	11.51	71.63	2	12	38	63	116	1,762	9,089	19
su2cor	4,776,762,363	3.34	73.07	8	15	26	34	60	1,569	7,246	22
swm256	11,037,397,884	1.65	98.42	2	2	3	3	13	795	6,080	14
tomcatv	899,655,317	3.35	99.28	3	4	5	7	7	515	5,474	9
wave5	3,554,909,341	4.37	61.79	18	40	82	132	276	1,331	8,149	16
APS	1,490,454,770	3.99	50.64	44	123	283	357	524	1,617	8,926	18
CSS	379,319,722	7.32	55.63	32	109	211	262	467	2,202	9,670	23
LWS	14,183,394,882	7.92	66.34	3	9	18	26	38	1,148	6,927	17
NAS	3,603,798,937	3.43	60.67	5	14	34	69	125	1,663	7,614	22
OCS	5,187,329,629	3.02	88.57	3	10	46	79	197	1,447	7,084	20
SDS	1,108,675,255	6.77	53.05	9	25	43	67	169	1,669	7,585	22
TFS	1,694,450,064	3.17	77.42	15	38	122	220	464	1,598	7,270	22
TIS	1,722,430,820	5.27	51.08	8	20	31	36	66	863	6,292	14
WSS	5,422,412,141	4.76	62.36	41	145	275	344	533	1,756	7,592	23

Table 6: Measured attributes of the traced program executables including libraries.

The ATOM instrumentation tool has an interface that allows the elements of the program executable (including all libraries), such as instructions, basic blocks, and procedures, to be queried and manipulated. In particular, ATOM allows an "instrumentation" program to navigate through the basic blocks of a program executable, and collect information about registers used, opcodes, branch conditions, control flow, etc. During this instrumentation phase, our tools collect all the static feature information described in Table 5. By gathering information about the targets of branches, 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 [4].

As the instrumentation program runs, it can also add procedure calls to the executable that allow information about the executing program to be collected. Thus, we also use ATOM to determine the execution frequency and branch probability for each branch in the corpus for the purpose of training, and in the test programs for the purpose of determining the miss rate of the methods.

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 and decision tree. We then use the neural net's and decision tree's branch classifications 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 and decision tree has not seen. For the ESP results shown in Section 6, we performed the cross-validation study 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 and decision tree, predicting branches for the 23rd C program not included in the initial 22. We did the same for Fortran programs feeding into the neural net and decision tree the feature sets for 19 of the 20 programs in order to predict branches for the 20th program.

6 Results

We now compare the prediction accuracy of *a priori* heuristic combination (APHC) branch prediction [4], the Dempster-Shafer heuristic combination (DSHC) proposed by Wu and Larus [27], and our ESP techniques. Following this, we show that the APHC and DSHC techniques are sensitive to differences in system architecture and compilers.

6.1 Comparison: APHC, DSHC and ESP

Table 7 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 [4], 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 3. The fifth column in Table 7 shows the results for ESP using Neural Nets, and the sixth column shows results for ESP using Decision Trees. The last column shows the results for the perfect static profile prediction. By *perfect profile prediction*, we mean the miss rate achieved when the same input is used to both profile a program, in order to predict the program's branches, and used when recording the miss rate. Table 7 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 [27] 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.*

	Branch Prediction Miss Rates							
Program	BTFNT	APHC (B&L's)	DSHC (B&L's)	DSHC (Ours)	ESP NN	ESP DT	ESP DT (self)	Perfect
bc	40	37	35	35	32	30	27	14
bison	52	15	16	16	14	16	12	4
burg	53	35	33	32	26	22	16	9
flex	43	33	39	38	19	27	23	9
grep	42	27	23	22	19	23	15	12
gzip	33	32	33	33	20	32	11	9
indent	42	27	28	27	19	22	13	6
od	44	44	40	40	30	21	12	8
perl	35	39	36	36	26	27	13	4
sed	45	22	22	23	25	26	8	5
siod	50	34	32	33	27	30	15	10
sort	44	35	41	42	21	19	18	8
tex	43	37	38	36	30	31	19	13
wdiff	42	32	11	11	4	11	8	3
yacr	32	14	11	12	14	13	8	6
Other C Avg	43	31	29	29	22	23	15	8
alvinn	2	2	2	2	1	2	2	0
compress	44	25	26	28	30	28	14	14
ear	10	8	8	8	8	8	7	7
eqntott	47	7	7	7	6	5	3	2
espresso	34	24	23	23	32	25	18	15
gcc	48	34	35	34	31	32	27	12
li	43	26	25	27	28	29	13	12
sc	39	29	31	29	24	26	16	9
SPEC C Avg	34	19	20	20	20	19	12	9
doduc	23	19	20	19	16	39	10	5
fpppp	42	53	52	52	35	22	14	11
hydro2d	28	17	16	16	12	10	5	4
mdljsp2	69	41	62	62	64	65	10	10
nasa7	8	12	12	11	5	5	5	3
ora	46	18	18	18	18	12	5	5
spice	16	16	18	14	14	11	11	7
su2cor	17	21	20	20	12	12	11	10
swm256	1	1	1	1	1	1	1	1
tomcatv	44	44	44	44	1	44	1	1
wave5	19	27	24	23	21	24	9	6
SPEC Fortran Avg	29	24	26	25	18	22	7	6
APS	28	30	34	31	26	29	17	10
CSS	39	29	40	36	33	25	24	9
LWS	38	32	25	25	18	17	17	16
NAS	42	12	22	22	12	12	6	4
OCS	4	6	5	5	4	6	4	2
SDS	18	32	25	19	21	22	15	12
TFS	12	10	15	13	11	11	8	6
TIS	18	26	25	22	16	16	16	16
WSS	32	28	26	26	25	24	25	11
Perf Club Avg	26	23	24	22	18	18	15	10
Overall Avg	34	25	26	25	20	21	12	8

Table 7: 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 Neural Net ESP technique, and the sixth column is the miss rate for ESP using Decision Trees. Both columns use a cross-validation training method. The seventh column shows the miss rate using ESP and Decision Trees where the program is used both as the training and the test set. The last column is the miss rate for perfect static profile prediction. In all cases, smaller values are better.

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's miss rate is more than 5% higher (worse) than the simple APHC ordering, while the APHC ordering method is 5% worse in only three cases (`wdiff`, `SDS`, `LWS`). The intuition in [27] was correct; however, the Dempster-Shafer theory does not combine the evidence well enough to improve branch prediction. The ESP techniques perform significantly better than the Dempster-Shafer and the APHC methods for most of the programs.

The ESP DT (self) column provides insight into the upper limit of the effectiveness of the ESP approach. In this case, the same data set was used for training as well as testing. Ideally, such a method should approach the performance of the perfect static predictor, and in many cases, but not all, it does. In most cases, self-training performs significantly better than cross-validation training, suggesting that the choice of training set and its closeness to the application being predicted can have a significant effect on the performance of the method.

Table 7 also allows us to compare the effectiveness of the decision tree and neural network prediction methods. In many cases the methods produce quite similar results, and this similarity is also reflected in the values of the overall average miss rates. The largest difference seen between the methods occurs in the `tomcatv` application, which we have already discussed in the context of the heuristic prediction methods. `tomcatv` exhibits such large differences between methods because there are a very small number of important branches that must be correctly predicted for good performance. The table shows that the neural network correctly predicts all of these important branches, whereas the decision tree method misses two of them. One reason that the neural network can perform better than the decision tree is that the decision tree method uses a greedy algorithm and selects one feature at a time in determining what features to use in categorizing branches. The neural net, on the other hand, is capable of identifying multi-feature combinations that could correlate highly with the branch direction.

To better understand why the decision tree method fails to predict the branches of `tomcatv`, we can look in detail at the different programs used to predict its behavior. In Figures 4 and 5 (discussed in more detail in Section 6.3.1), we show the miss rates that result from using one program to predict another (i.e., using one program as the training set and another as the test set). In Figure 5, in the “to” column (3rd to last) we show how effective the other Fortran programs in the corpus are at predicting `tomcatv` using a decision tree. It is clear from this figure that only five of the other 19 programs contain information that the decision tree method can use to predict the branches of `tomcatv`. This suggests that the branch behavior in `tomcatv` is not highly typical of the other Fortran programs we used.

6.2 Predicting Library and Main Program Branches

The miss rates for ESP in Table 7 were gathered using a cross-validation study in order to achieve a realistic estimate of ESP’s performance. A cross-validation study allows this since the program being measured is not included in the set of programs used to create the cross-validation training set. Therefore, the code for the program being examined is not profiled in the cross-validation study. This is true for the main program branches, but not necessarily true for the library code, since the same library routines may have been used by the programs included in the cross-validation study. If a majority of the library branches are executed in the cross-validation training set and the library routines have similar behavior between different programs, then ESP may be achieving a lot of its performance improvement by more accurately predicting the library branches than main program branches.

Tables 8 and 9 show the breakdown of the average miss rates in terms of library branches and branches executed in the main program. For ESP, results are only shown for using the Decision Tree approach (ESP-DT). The library branches are all the branches executed by the programs for the standard Digital-Unix libraries. The libraries used by the programs we examined are: `libc`, `libm`, `libFutil`, `libUfor`, `libfor`, `libcursors`, `libots`, and `libtermcap`. In Table 8, **Lib** shows the miss rates for library

		Library Prediction Miss Rates						
Program	% Lib Branches (dynamic)	BTFNT	APHC (B&L's)	DSHC (B&L's)	DSHC (Ours)	ESP-DT		Perfect
						Lib	All	
Other C Avg	29	39	34	31	31	26	25	10
SPEC C Avg	11	44	39	36	37	33	24	4
SPEC Fortran Avg	11	38	33	35	34	14	13	8
Perf Club Avg	23	46	36	35	35	20	23	7
Overall Avg	20	41	35	34	34	23	21	8

Table 8: Effectiveness of Predicting Library Branches Only. ESP results are only shown for the Decision Tree approach. Lib represents the miss rate when only features for library branches are included in the cross-validation training set. All represents the miss rate when all the branches executed in a program are included in the cross-validation training set used to predict the library branches.

		Main Program Miss Rates						
Program	% Main Branches (dynamic)	BTFNT	APHC (B&L's)	DSHC (B&L's)	DSHC (Ours)	ESP-DT		Perfect
						Main	All	
Other C Avg	71	43	28	28	28	26	24	7
SPEC C Avg	89	34	20	19	20	21	20	10
SPEC Fortran Avg	89	27	22	24	23	22	23	6
Perf Club Avg	77	21	20	21	20	18	20	11
Overall Avg	80	33	23	24	24	22	22	8

Table 9: Effectiveness of Predicting Non-library Branches Only. Main represents the miss rate when only features for main program branches are included in the cross-validation training set. All represents the miss rate when all the branches executed in a program are included in the cross-validation training set used to predict the main program branches.

	Combined Overall Miss Rates						
Program	BTFNT	APHC (B&L's)	DSHC (B&L's)	DSHC (Ours)	ESP-DT		Perfect
					Sep	All	
Other C Avg	43	31	29	29	25	23	8
SPEC C Avg	34	19	20	20	20	19	9
SPEC Fortran Avg	29	24	26	25	21	22	6
Perf Club Avg	26	23	24	22	18	18	10
Overall Avg	34	25	26	25	22	21	8

Table 10: Effectiveness of Predicting All Branches. Sep represents the miss rate when only features for library branches are used to predict library branches and only features for main program branches are used to predict main program branches. All represents the miss rate when all the branches executed in a program are included in the cross-validation training set for prediction.

branches for ESP-DT when only library branches are included in the cross-validation training set, and **All** represents when all branches are included in the cross-validation training set. Table 9 shows the miss rates for the main program branches for ESP-DT when only **Main** program branches are included in the training set and when **All** branches are included in the cross-validation training set. The reason for using two different training sets was to examine the difference in performance when excluding either the main program branches or the library branches from the training profiles.

Tables 8 and 9 show that the miss rate difference between the B&L heuristics and ESP-DT is significantly larger for the library branches, from 35% down to 23%, than for the main program branches, from 23% down to 22%. This difference is largest for the branches occurring in Fortran library routines. This decrease in miss rate for the library branches most likely comes from the library branches having very similar behavior between different programs. In a related study [8], we found this to be the case. In that study we found that when using a cross-validation profile of library code to predict library branches for a program not included in the profile, the miss rate for the library branches was 12% which was close to the perfect profile miss rate of 6%, and significantly lower than the B&L heuristic miss rate of 47%. This indicates that these library branches had very predictable behavior between different programs, and the results in Table 8 show that ESP was also able to provide a better automatic identification of the heuristics that identify these predictable library branches.

Table 10 shows the overall average miss rates from Table 7. The ESP-DT results are shown for when separate (**Sep**) cross-validation training sets are used to predict the library and main program branches, and when one combined cross-validation training set is used to predict all (**All**) branches. These results and the results in Tables 8 and 9 show that it is better to use the combined training including all the branches when predicting either library or main program branches, especially for the C programs. When using the combined training set in Table 10 the miss rate is lowered from 22% to 21% in comparison to using separate training sets. Table 8 shows that including the main program branch feature sets into the cross-validation training set when predicting library branches reduces the miss rate from 23% down to 21%. These results indicate that it is better to include all program branches in the training sets in order to achieve the best prediction accuracy for either library or main program branches.

Even though ESP shows only a small improvement in miss rate, 23% down to 22%, over the APHC heuristics in Table 9 for main program branches, ESP is still a much more attractive automated solution for finding program-based static branch prediction for a given architecture, programming language, and run-time system over using expert defined heuristics.

6.3 Sensitivity of Decision Tree ESP Results

We now examine how sensitive ESP is to the number of programs used in the training set for prediction and the types of features included in the feature set when using decision tree ESP prediction.

6.3.1 Effect of Size and Content of Training Set

Figures 4 and 5 show a matrix of miss rates using the ESP training set from one program to predict branches in all the other programs. The miss rate for each box in the matrix is for the corresponding program listed at the top of the matrix when the ESP training set from the program listed on the left hand side of the matrix is used to predict the branches. For example in Figure 4, when using the ESP training set from alvinn to predict the branches in ear (ea), ear has an 8% miss rate. The miss rates with a darkened box around them are the highest miss rates for each program listed in the columns. The miss rates that are shaded are the lowest miss rates for each program listed in the columns.

Not surprisingly, both of the matrices show that the best predictor for most programs was the own program's training set. In Figure 4, the C program training sets that provided the worst prediction came from `li`, `wdiff`, and `alvinn`. In Figure 5, the Fortran programs that provided the worst training set for prediction are not as concentrated as in the C programs, and they include `mdljsp2`, `SDS`, `fpppp`,

Programs	al	bc	bi	bu	co	ea	eq	es	fl	gc	gr	gz	in	li	od	pe	sc	se	si	so	te	wd	ya	Avg
alvinn	2	41	23	43	33	8	48	35	38	50	43	31	51	52	55	57	44	44	38	43	45	17	24	38
bc	2	27	42	39	49	10	49	42	28	37	55	21	39	34	41	35	53	51	38	57	37	44	31	37
bison	2	52	12	35	40	9	8	31	34	45	44	18	37	42	38	41	34	33	56	43	39	14	15	31
burg	97	45	23	16	36	9	48	33	26	35	28	24	25	28	20	35	49	43	40	34	38	17	25	34
compress	2	48	20	35	14	11	8	35	41	41	41	53	26	32	41	43	33	31	41	41	36	33	35	32
ear	2	42	26	43	31	7	48	40	41	48	47	38	42	45	53	47	34	46	41	48	44	36	47	39
eqntott	1	49	21	34	29	11	3	32	43	38	42	34	28	46	47	50	32	34	40	44	40	29	14	32
espresso	2	41	30	37	44	77	49	18	47	41	25	43	32	24	40	32	29	33	33	41	47	13	26	35
flex	2	40	20	40	39	10	7	36	23	39	34	19	33	23	36	26	33	32	33	39	34	4	14	27
gcc	49	40	16	31	30	9	49	49	36	27	26	21	28	27	35	39	42	33	28	34	36	46	57	34
grep	1	39	30	43	32	10	7	37	31	39	15	19	37	31	46	29	33	31	36	22	37	15	23	28
gzip	2	41	22	31	28	11	49	39	33	39	22	11	41	46	41	47	39	35	44	45	39	5	26	32
indent	97	48	18	30	38	9	50	47	27	35	32	20	13	41	33	35	49	37	36	39	32	17	20	35
li	2	42	59	49	47	90	91	50	58	46	44	59	41	13	44	43	59	40	38	36	42	59	56	48
od	2	48	57	31	53	64	90	49	30	39	22	40	46	35	12	44	50	34	42	43	38	56	48	42
perl	49	42	26	37	26	53	87	38	28	34	49	19	28	25	31	13	30	30	37	46	39	22	56	37
sc	2	44	20	41	27	25	48	36	37	39	35	21	23	44	34	29	16	31	31	43	41	21	36	31
sed	2	42	27	42	18	37	8	34	52	39	29	36	34	25	47	35	34	8	54	34	42	56	25	33
siod	49	43	24	28	30	9	48	44	30	34	38	37	24	31	30	34	33	41	15	36	39	25	37	33
sort	2	43	19	40	26	9	47	33	28	36	26	20	42	37	52	37	28	28	38	18	43	46	32	32
tex	2	43	16	43	26	38	49	36	30	36	31	34	27	39	39	32	38	36	39	45	19	5	30	32
wdiff	2	47	26	46	39	61	89	34	56	50	59	73	38	50	53	49	34	36	56	37	51	8	31	45
yacr	1	40	18	38	28	10	7	23	37	44	33	21	40	32	52	47	30	30	52	39	39	11	8	30

Figure 4: Using One C Program to Predict Another. Each row shows the miss rates obtained using one program to predict the behavior of the others. The results for individual programs are listed in each column.

Programs	AP	CS	LW	NA	OC	SD	TF	TI	WS	do	fp	hy	md	na	or	sp	su	sw	to	wa	Avg
APS	17	37	48	27	5	25	11	22	33	67	51	16	46	11	33	26	35	1	1	22	27
CSS	36	24	34	21	7	35	16	37	29	47	33	10	46	13	31	12	24	1	65	35	28
LWS	41	43	17	17	10	36	17	46	42	46	62	29	45	12	33	70	18	1	44	21	32
NAS	34	41	31	6	6	29	17	32	21	42	31	22	45	8	27	26	14	1	44	26	25
OCS	28	50	38	32	4	28	10	34	32	45	66	28	59	12	48	26	21	1	44	25	31
SDS	29	41	38	42	9	15	30	41	38	23	44	30	70	14	46	31	27	87	44	27	36
TFS	29	49	38	40	5	25	8	30	31	42	64	29	30	9	63	27	18	1	65	21	31
TIS	31	41	16	17	10	16	27	16	37	16	19	11	59	14	18	30	24	87	1	32	26
WSS	29	39	38	38	5	18	11	18	25	23	46	28	68	8	46	16	17	1	44	21	27
doduc	27	44	30	15	10	18	26	19	33	10	44	10	50	12	26	67	23	87	44	33	31
fpppp	31	42	43	19	7	29	22	22	41	42	14	17	38	27	18	22	37	1	31	40	27
hydro2d	38	32	21	19	7	35	19	37	32	45	41	5	27	11	33	20	15	1	44	24	25
mdljsp2	51	50	34	43	11	47	23	49	37	52	61	27	10	21	53	74	28	2	1	38	36
nasa7	36	37	31	17	6	27	9	27	32	24	36	23	53	5	24	15	11	1	22	26	23
ora	29	42	28	28	4	21	15	25	32	46	43	8	16	8	5	21	13	1	1	24	20
spice	28	36	41	42	4	18	11	18	34	23	43	29	70	8	40	11	18	1	44	19	27
su2cor	40	39	20	19	7	34	13	34	37	46	29	12	45	6	32	15	11	1	1	26	23
swm256	31	47	38	44	5	30	12	34	36	44	65	30	30	10	67	26	19	1	44	20	32
tomcatv	49	44	34	39	11	47	23	49	38	51	53	27	16	21	47	28	27	2	1	38	32
wave5	31	44	38	39	5	24	12	24	34	24	34	22	61	8	55	27	19	1	44	9	28

Figure 5: Using One Fortran Program to Predict Another. Each row shows the miss rates obtained using one program to predict the behavior of the others. The results for individual programs are listed in each column.

`swm256`, and `tomcatv`. What makes these programs particularly poor training sets? The answer can be seen in the "branch quantile" values in Table 6. Each of the programs that are poorly predictive tend to have a small number of branches that dominate the branch activity in the program, as indicated by the "Q95" value in Table 6. A small number of branches provides little training information for the machine learning algorithms.

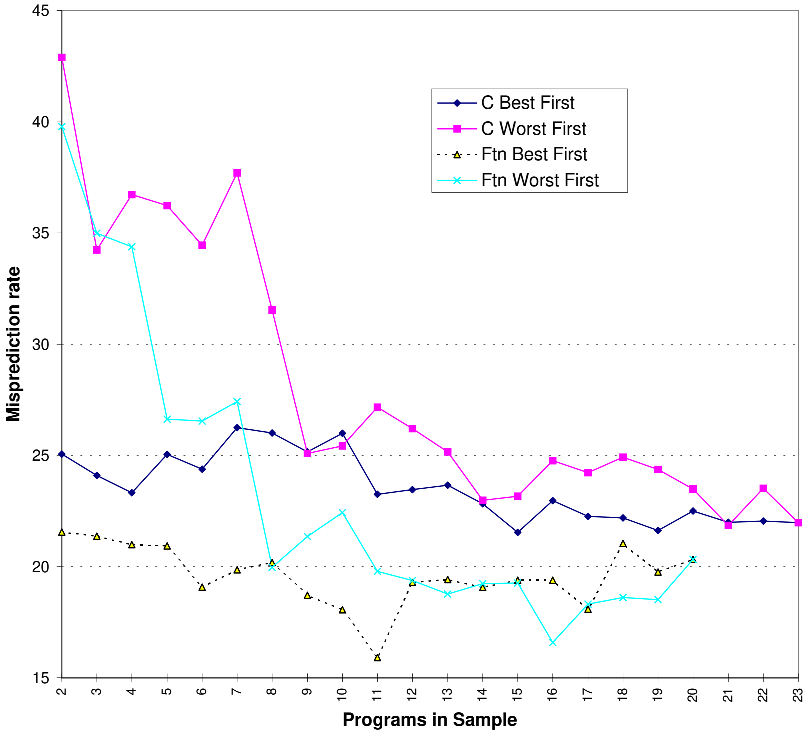

Figure 6: Effect of Adding Programs to Training Set Incrementally.

Figure 6 shows the average miss rates for adding one program at a time into the cross-validation training set. The "Best First" order adds one program at a time to the training set, starting with the program that achieved the lowest average miss rate in Figures 4 and 5 (e.g., the C program `flex` and the Fortran program `ora`) and ending with the program that had the highest miss rate. In Figure 6, the "Worst First" order adds the programs to the cross-validation training set in the reverse order of "Best First" (i.e., adding the program with the highest average miss rate to the training set first).

While the curves are not monotonic, the trend for the first half of all curves in Figure 6 is that the miss rate of the training set decreases as the size of the training set increases. Interesting, for the "Best First" lines, beyond the first half, the miss rate stays relatively constant, and in the case of Fortran actually increases. This effect is not as apparent in the "Worst First" curves. Based on the previous figures, it is clear that some of the programs in the corpus are quite atypical. Atypical programs in the corpus would have a negative effect on the ESP method, especially if they formed a majority of the programs in the corpus. The results in Figure 6 suggest that both our C and Fortran corpuses were sufficiently large and diverse that the atypical programs did not have large negative effect on their predictive ability. But the figure also indicates that care must be taken that the programs in the corpus are "typical" of the programs being predicted, an admittedly difficult quality to measure.

6.3.2 Effect of Size and Content of Feature Set

Table 11 shows the miss rates for each of the features described in Table 5 in sorted order of the feature's performance for both the C and Fortran programs. One interesting result is that the best feature for predicting the Fortran programs, `Branch Instruction`, is the worst predictor for the C programs. The features with the best prediction between the Fortran and C programs are `NotTaken Successor Exit`, `Taken Successor Exit`, and `Taken Successor Ends`. The first two features correspond to the loop exiting heuristic which predicts that conditional branches with edges exiting a loop will not have that edge taken.

The best single predictor for the C programs is `Taken Successor Ends`. This feature encodes the branch type of the instruction that ends the "taken successor" block (see Figure 2 for an explanation). Basic blocks can be terminated by a number of instructions, the most frequent of which is another conditional branch, which occurs 65% of the time. We examined the decision tree formed by the `Taken Successor Ends` heuristic in more detail; while the branch prediction contribution of the conditional branches was

C Sorted Features			Fortran Sorted Features		
Num	Heuristic	Miss Rate	Num	Heuristic	Miss Rate
17	Taken Successor Ends	30.4	1	Branch Instruction	23.2
20	Taken Successor Exit	32.1	28	NotTaken Successor Exit	26.4
22	Taken Successor Call	34.8	2	Branch Direction	27.3
26	NotTaken Successor Loop	35.4	19	Taken Successor Backedge	27.4
16	Taken Block PostDominates	35.8	24	NotTaken Block PostDominates	28.5
28	NotTaken Successor Exit	35.8	17	Taken Successor Ends	28.9
4	Branch Operand Function	36.3	20	Taken Successor Exit	29.0
5	Branch Operand Type	36.3	10	RB Function	29.1
6	RA Opcode	36.3	18	Taken Successor Loop	30.0
7	RA Function	36.3	3	Branch Operand Opcode	30.1
8	RA Type	36.3	6	RA Opcode	30.1
9	RB Opcode	36.3	26	NotTaken Successor Loop	31.1
10	RB Function	36.3	25	NotTaken Successor Ends	31.7
11	RB Type	36.3	8	RA Type	31.9
12	Loop Header	36.3	9	RB Opcode	32.0
13	Language	36.3	5	Branch Operand Type	32.2
15	Taken Block Dominates	36.3	7	RA Function	32.2
21	Taken Successor UseDef	36.3	11	RB Type	32.2
23	NotTaken Block Dominates	36.3	12	Loop Header	32.2
24	NotTaken Block PostDominates	36.3	13	Language	32.2
27	NotTaken Successor Backedge	36.3	16	Taken Block PostDominates	32.2
30	NotTaken Successor Call	36.3	21	Taken Successor UseDef	32.2
14	Procedure Type	36.4	23	NotTaken Block Dominates	32.2
29	NotTaken Successor UseDef	37.0	27	NotTaken Successor Backedge	32.2
2	Branch Direction	37.9	29	NotTaken Successor UseDef	32.2
18	Taken Successor Loop	37.9	30	NotTaken Successor Call	32.2
25	NotTaken Successor Ends	38.3	14	Procedure Type	32.4
3	Branch Operand Opcode	39.1	22	Taken Successor Call	32.7
19	Taken Successor Backedge	39.2	4	Branch Operand Function	34.2
1	Branch Instruction	42.5	15	Taken Block Dominates	35.0

Table 11: Effectiveness of Using Individual Features for Prediction. For each feature there is a number, a short description of the feature and the feature's miss rate for the C and Fortran program. Num represents the feature's number.

C Combinations		Fortran Combinations	
Features	Miss Rate	Features	Miss Rate
17,25	26.3	02,25	18.2
01,17	26.8	19,25	18.2
02,17	27.6	25,28	18.4
17,19	27.6	24,25	18.9
17,24	27.6	01,25	20.4
17,28	27.8	18,25	20.5
17,20	28.1	01,26	21.6
06,17	28.6	01,28	21.6
28,30	28.6	01,02	22.0
08,17	28.7	01,16	22.0
17,20,25	24.0	01,25,28	15.8
01,17,20	24.7	03,25,28	16.5
01,03,28	25.3	17,25,28	16.8
02,15,17	25.3	03,19,25	17.0
15,17,19	25.3	01,02,25	17.2
02,17,20	25.5	01,19,25	17.2
15,17,18	25.5	02,17,25	17.2
15,17,28	25.5	17,19,25	17.2
17,19,20	25.5	02,03,25	17.3
17,20,28	25.5	02,16,25	17.3
01,03,28,25	22.0	01,25,28,18	15.6
01,03,28,30	23.8	01,25,28,05	15.7
17,20,25,28	23.9	01,25,28,17	15.7
01,03,28,15	24.0	17,25,28,01	15.7
01,03,28,18	24.0	01,25,28,01	15.8
17,20,25,08	24.0	01,25,28,11	15.8
17,20,25,10	24.0	01,25,28,13	15.8
17,20,25,13	24.0	01,25,28,22	15.8
17,20,25,17	24.0	01,25,28,25	15.8
17,20,25,20	24.0	01,25,28,27	15.8

Table 12: Effectiveness of Using the Best Small Combinations of Features for Prediction. Top 10 double, triple, quadruple combinations are shown.

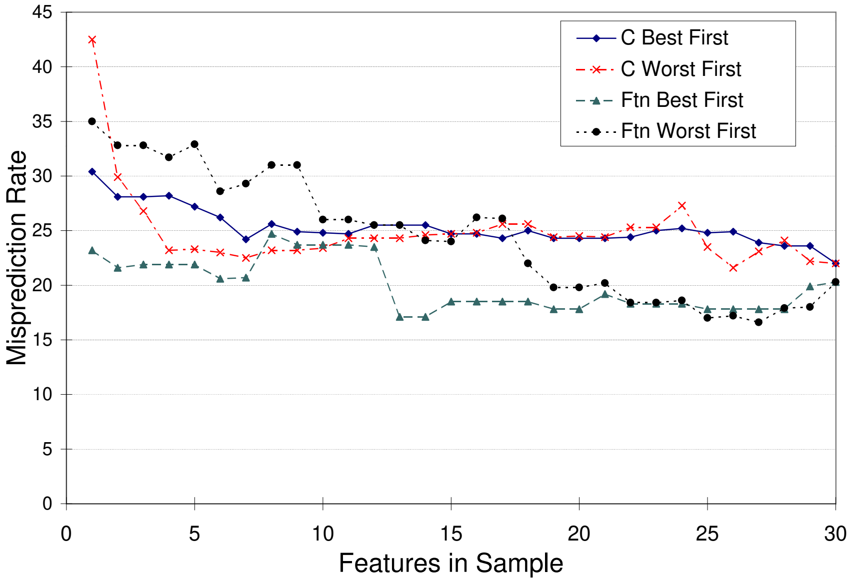

Figure 7: Effect of Adding Features to Training Set Incrementally

largest, the seven other branch types contributed significantly to the overall branch prediction rate—no one feature stood out as particularly more important than the others. Simply ignoring the extra information, as a human expert may be tempted to do, would have increased the miss rate by a few percent. Using an automated expert system, we were able to benefit from the additional information with little effort.

Table 12 gives the top 10 double, triple and quadruple feature combinations, which provided the lowest miss rates. The feature numbers listed in the table correspond to the feature numbers in Table 11. The table shows that combining the top 4 features in Table 11 does not necessarily achieve the lowest miss rate. For example, 3 of the features in the best quadruple feature combination (01,03,28,25) for the C programs have the worst miss rate for individual feature performance in Table 11. It is also striking that using the best 4-feature combination in Fortran can result in miss rates significantly lower than those achieved when using the entire feature set (15.6% versus 20.3%).

Figure 7 shows the miss rate for adding features one at a time into the cross-validation profile when using the decision tree ESP approach. Similar to Figure 6, the “Best First” order for the C and Fortran programs represents adding the features in the order shown in Table 11 going from the features with the lowest miss rate to the highest miss rate. The “Worst First” order in Figure 7 adds the features in the reverse order of “Best First”.

The figure shows a slight downward trend in the lines as more features are added, although as with Figure 6, the trends are not monotonic. One striking aspect of some of the lines is how adding a single feature (and not necessarily an important one by itself) can significantly reduce the miss rate. This is particularly true when the 13th feature (NotTaken Successor Ends, feature 25) is added in the “Best First” Fortran line. This result, in combination with the data in Table 12, strongly suggests that for Fortran, at least, a small feature set containing the most important features, results in the best miss rates for the technique. Any additional features appear to have little effect and sometimes reduce the effectiveness of the technique.

7 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 since they can be used to estimate branch behavior at compile-time when performing compiler optimizations [15].

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 and decision tree to map static features associated with each branch to a prediction that the branch will be taken.

ESP has the following advantages over existing program-based branch prediction approaches. 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 run-time 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 and decision tree are capable of ignoring information that is irrelevant and such information does not degrade the performance of the branch predictions. 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 previous 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, and the overall 8% miss rate of the perfect static-profile predictor.

We see many future directions to take with this work. Currently, we have investigated how effective the neural network and decision tree are at providing a prediction for each branch. But both methods provide additional information, and the the decision tree in particular provides an estimate of the branch probability. If that probability is $> 50\%$ we estimate that the branch will be taken. One future goal is 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. We also plan to gather large bodies of programs in other programming languages, such as C++, Java, 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, will perform 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.

In order to achieve full instruction level parallelism on future processors, profile-based compiler optimizations need to be performed. Since not all users will take the time and effort to profile their programs, techniques that estimate program behavior at compile time, such as ESP, will become increasingly important.

Acknowledgements

We would like to thank Alan Eustace and Amitabh Srivastava for developing ATOM, and James Larus for motivating this paper. We also thank the anonymous referees for their insightful comments. 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, a hardware and software grant from Digital Equipment Corp and an equipment gift from Hewlett-Packard.

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] C. N. Arnold. Performance evaluation of three automatic vectorizer packages. *Proceedings of the 1982 International Conference on Parallel Processing*, pages 235–242, 1982.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] Brad Calder, Dirk Grunwald, and Amitabh Srivastava. The predictability of branches in libraries. In *28th International Symposium on Microarchitecture*, pages 24–34, November 1995.

[9] 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.

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

[11] 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.

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

[13] 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.

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

[15] Richard Hank, Scott Mahlke, Roger Bringmann, John Gyllenhaal, and Wen mei Hwu. Superblock formation using static program analysis. In *26th International Symposium on Microarchitecture*, pages 247–256. IEEE, 1993.

[16] C. Huson, T. Macke, J. Davies, M. Wolfe, and B. Leasure. The **kap**/205: An advanced source-to-source vectorizer for the Cyber 205 supercomputer. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 827–835, 1986.

[17] 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.

[18] 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.

[19] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, ACM, June 1990.

[20] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[21] 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.

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.