Using Decision Trees to Improve Program-Based and Profile-Based Static Branch Prediction

Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere

Department of Electronics and Information Systems,
member HiPEAC, Ghent University—UGent, Belgium
{veerle.desmet, lieven.eeckhout, koen.debosschere}@elis.UGent.be

Abstract. Improving static branch prediction accuracy is an important problem with various interesting applications. First, several compiler optimizations such as code layout, scheduling, predication, etc. rely on accurate static branch prediction. Second, branches that are statically accurately predictable can be removed from the dynamic branch predictor thereby reducing aliasing. Third, for embedded microprocessors which lack dynamic branch prediction, static branch prediction is the only alternative.

This paper builds on previous work done on evidence-based static branch prediction which uses decision trees to classify branches. We demonstrate how decision trees can be used to improve the Ball and Larus heuristics by optimizing the sequence of applying the heuristics and by discovering two new heuristics, namely one based on the post-domination relationship between the current basic block and its successor and one based on the dependency distance between the branch and its operand defining instruction. Experimental results indicate an increase in the number of instructions per mispredicted branch by 18.5% on average for SPECint95 and SPECint2000. In addition, we show that decision trees can improve profile-based static branch prediction by up to 11.7% by predicting branches that are unseen in the profile runs.

1 Introduction

Static branch prediction is an important research topic for several reasons. Compilers rely on accurate static branch prediction for applying various compiler optimizations, such as code layout, instruction scheduling, register allocation, function inlining, predication, etc. Moreover, in many cases the applicability or the effectiveness of the compiler optimizations is directly proportional to the branch prediction accuracy. Second, branches that are highly predictable using static branch prediction or hard to predict dynamically can be excluded from the dynamic branch predictor thereby reducing aliasing in the predictor and thus increasing the predictor's performance [9]. The IA-64 ISA for example, provides branch hints to communicate to the hardware whether the branch should be updated in the dynamic branch predictors. Third, several embedded processors lack a dynamic branch predictor, e.g. the Philips' TriMedia and the TI VLIW family processors. For these microprocessors, static branch prediction is the only source for reducing the number of branch mispredictions.

T. Srikanthan et al. (Eds.): ACSAC 2005, LNCS 3740, pp. 336–352, 2005.

1.1 Background

There exist two approaches to static branch prediction, namely *program-based* and *profile-based* branch prediction. The first approach only uses structural information of a computer program. A well known example of program-based branch prediction are the Ball and Larus heuristics. Ball and Larus [1] present a set of heuristics that are based on the branch opcode, the branch operands, and properties of the basic block successors of a branch. Ball and Larus propose to apply these heuristics in a fixed ordering which means that the first heuristic that applies to a particular branch will be used to predict the branch direction. To determine the best ordering they evaluate all possible combinations. Instead of using a fixed ordering for applying the Ball and Larus heuristics, Wu and Larus [17] propose using the Dempster-Shafer theory for combining heuristics in case multiple heuristics apply to a particular branch. The results in [3] however, indicate that this does not improve the branch prediction accuracy.

A second example of program-based branch prediction is evidence-based static branch prediction (ESP) proposed by Calder *et al.* [3]. In evidence-based branch prediction, machine learning techniques are used to classify branches based on a large number of branch features. We will use decision trees for classification since they are equally performing as neural networks while being easier to interpret [3]. The biggest advantage of this approach is that decision trees are generated automatically so that they can be specialized for a specific programming language, a specific compiler, a specific ISA, etc. Other program-based techniques rely on heuristics that are based on intuition and empirical studies and thus are not easily transformed to another environment.

The profile-based branch prediction approach uses information obtained from previous runs of the same program with different inputs. According to Fisher and Freudenberger [7], branches go in one direction most of the time; as such, one can predict the direction of a branch well using previous runs of the same program. Profile-based static branch prediction is widely recognized as being more accurate than program-based prediction [3,7]. However, profile-based static branch prediction has several disadvantages. First, gathering profile data is a time-consuming process that programmers are not always willing to do. Second, profiling is not always practical, e.g. for an operating system kernel, or even infeasible for real-time applications. Third, the selection of representative inputs might be difficult.

In this paper, we consider both static branch prediction approaches since both come with their advantages. Program-based branch prediction has a lower cost and profile-based branch prediction has a higher accuracy.

1.2 Contributions and Outline

This paper makes the following contributions. First, we demonstrate that decision trees can be used to improve the Ball and Larus heuristics by automatically optimizing the sequence of applying these heuristics. Second, we increase the number of instructions per mispredicted branch by 18.5% over the Ball

and Larus heuristics by proposing two new heuristics, one based on the post-dominator relationship of the current block and its successor and one based on the dependency distance between the branch and its operand defining instruction. Third, we show that decision trees can also be used to improve the accuracy of profile-based static branch prediction by up to 11.7%. In particular, for branches that are unseen in the profile runs, we propose using decision trees. The experimental results are obtained using the SPECint95 and SPECint2000 benchmarks.

This paper is organized as follows. After detailing the experimental setup in section 2, we will discuss in section 3 how evidence-based branch prediction can be improved by adding new static branch features. In section 4, we discuss the Ball and Larus heuristics and show how decision trees can be used to improve their performance. In section 5 we demonstrate that profile-based branch prediction can also benefit from decision trees when used in conjunction. Finally, we discuss related work in section 6 and conclude in section 7.

2 Experimental Setup

The experimental results in this paper are obtained for the SPECint95 and the SPECint2000 benchmarks. We did not include gcc, perl and vortex from SPECint95 because our evaluation methodology uses cross-validation which

Table 1. Benchmarks, their inputs and some branch statistics. 'Stat.' is the number of static branches executed; 'Dyn.' is the number of dynamic branches executed in millions; 'Best' is the upper limit for IPM for static branch prediction. Inputs with an asterix* are only used in Section 5 and are excluded from the average.

Program	Input	Stat.	Dyn.	Best	Program	Input	Stat.	Dyn.	Best
gzip	graphic*	830	7,939	102.14	vpr	place*	1,584	292	75.29
	log*	833	3,861	129.01		route	2,463	7,671	128.59
	program*	832	8,031	129.39	mcf	ref	844	10,758	50.08
	random	777	6,748	241.05	crafty	ref	2,677	15,990	67.93
	source*	851	9,351	104.94	gap	ref	4,757	21,975	169.42
gcc	166*	18,588	4,136	358.23	vortex	lendian1*	5,930	12,966	1037.62
	200*	17,581	12,011	122.58		lendian2	5,940	13,779	730.65
	expr*	17,904	1,243	146.16		lendian3*	5,923	14,500	1092.65
	integrate*	16,720	1,281	229.12	bzip2	graphic*	854	12,697	140.25
	scilab	17,771	6,516	113.30		program	854	11,612	106.70
parser	ref	2,624	60,851	64.02		source*	853	9,862	102.14
perlbmk	splitmail 850*	6,146	12,309	440.06	twolf	ref	2,976	34,249	63.44
	splitmail 704*	6,147	6,508	402.49	compress	bigtest	426	5,004	72.66
	splitmail 535	6,146	6,072	421.05	go	5stone21	4,963	3,872	39.00
	splitmail 957*	6,148	10,822	402.55	ijpeg	penguin	1455	1,485	134.80
	makerand*	2,029	177	318.10	li	ref	918	8,532	50.98
	diffmail*	5,598	3,774	154.18	m88ksim	ctl.raw.lit	1,113	1,969	202.79
average									115.94

means that the program being evaluated is not included in the train set; including these three benchmarks would have been unfair as they appear in both SPECint95 and SPECint2000. We did not include `eon` since it is the only SPECint2000 benchmark written in C++ whereas the other benchmarks are all written in C. Indeed, Calder *et al.* [3] showed that ESP is sensitive to the programming language in which the benchmarks are written and that therefore separate decision trees should be considered for different programming languages. All the benchmarks are compiled with the Compaq C compiler version V6.3-025 with optimization flags `-arch ev6 -fast -O4`. An overview of the benchmarks and their inputs is listed in Table 1. All the inputs are reference inputs and all the benchmarks were run to completion. In Table 1, for each benchmark-input pair the lower limit in branch misprediction rate is shown for static branch prediction. This lower limit is defined by the most likely direction for each branch. For example, for a branch that is executed 100 times of which 80 times taken, the best possible static branch prediction achieves a prediction accuracy of 80%. All the averages reported in this paper are geometric averages over all the benchmarks. Our primary metric for measuring the performance of the proposed branch prediction techniques is the number of instructions per mispredicted branch (IPM) which is a better metric than misprediction rate because IPM also captures the density of conditional branches in a program [7]. Detecting basic blocks and loops in the binaries was done using Squeeze [5], a binary rewriting tool for the Alpha architecture. Squeeze reads in statically linked binaries and builds a control flow graph from it. Computing the static branch features that will serve as input for the decision trees is done using a modified version of Squeeze.

3 Evidence-Based Prediction

This section presents a background on evidence-based branch prediction (ESP) using decision trees. We also show how ESP can be improved by stopping the growth during decision tree learning and by adding a set of newly proposed branch features.

3.1 Decision Trees

Decision trees consider static branch prediction as a classification problem: branches are classified into 'taken' and 'not-taken' classes based on a number of static branch features. A decision tree consists of a number of internal nodes in which each node discriminates on a given branch feature, and in which the leafs represent the branch classes 'taken' and 'not-taken'. A decision tree can thus be viewed as a hierarchical step-wise decision procedure.

In our experiments, the decision trees are generated using C4.5 [11]. Developed by J. Ross Quinlan in 1992, C4.5 is a widely used tool for constructing decision trees; C4.5 was also used by Calder *et al.* [3]. Our slightly modified version of C4.5 takes into account the branches' execution frequencies giving a

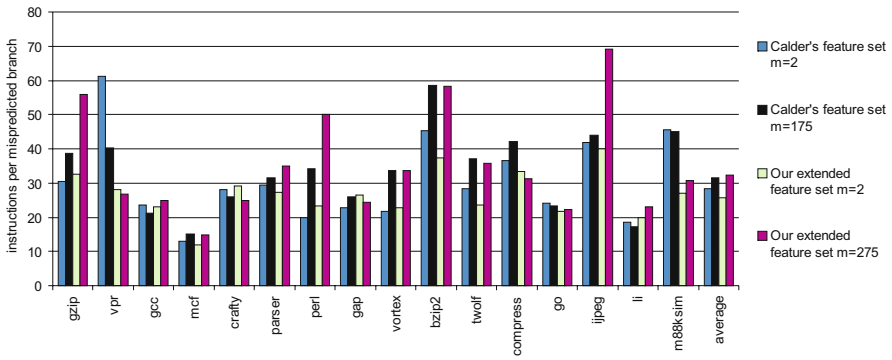

Fig. 1. Number of instructions per mispredicted branch for evidence-based branch prediction using decision trees

higher weight to more frequently executed branches; this is done by adding an extra attribute to the input of C4.5 for each static branch. To achieve the same result, Calder *et al.* [3] duplicates static branches proportional to their execution frequency. As done in [3], the execution frequencies were rescaled by multiplying the normalized frequencies by a factor 1,000; branches with a rescaled value smaller than 1 were excluded from the training set. This eliminates a number of low-frequency branches and allows C4.5 to focus on the important frequently executed branches. An additional advantage of eliminating low-frequency branches is that it seriously reduces the time to construct the decision trees. Further, the default C4.5 training and pruning settings are used.

All the results in this paper are obtained by performing a cross-validation study, meaning that the benchmark under evaluation is excluded from the training set. Cross-validation was done to provide fair results, since the predicted program is not seen during the construction of the decision tree.

3.2 Branch Features

We have applied decision tree prediction on the SPECint95 and SPECint2000 benchmarks in a cross-validation study using the feature set from Calder *et al.* [3]. The first bar in Figure 1 show the average number of instructions between two mispredicted branches equals 28.4.

3.3 Stopping the Growth of the Decision Tree

In his book on machine learning [8], Tom Mitchell describes approaches to avoid overfitting in decision tree learning. The first group of techniques allow the tree to overfit the data, and then post-prune the tree in a second phase. Pruning is the process of consecutively removing subtrees, making them leaf nodes, and assigning them the most common decision class of the training examples. Nodes

are pruned iteratively, at each step choosing the node whose removal affects the estimated classification accuracy the least. This post-pruning approach is the default strategy for C4.5.

An alternative to post-pruning is stopping the growth of the tree before it reaches the point of overfitting the data. In C4.5, this technique can be forced by setting *parameter* m to indicate the minimal number of examples in at least two subtrees when splitting a node. The default value in C4.5 for m equals 2. By setting this m, we directly tune the generalization degree to the training data. As m increases the tree concentrates only on splits with sufficient examples in the subtrees. and thus the prediction strategy becomes more general. To determine a good m, we evaluate program 1 (P1) based on data from P2 to P16 while varying m from 25 to 1000 with a step size of 25 and we repeat this experiment for P2 to P16 (cross-validation). If we would optimize the IPM as a function of m for each benchmark, we would systematically bias our results to better performance. To prevent the latter, for a program P1 we determine the average IPM for P2 to P16 for each m, and optimize for m. This technique potentially results in 16 different m values; however, for all except one the value equals $m = 175$. In addition, we observed that m-values within the same order of magnitude do not affect the results significantly. In Figure 1, the second bar displays the IPM when applying this growth stopping mechanism using the feature set by Calder *et al.* This graph shows that IPM is increased from 28.4 to 31.4 which is an increase by 10.6%. Most programs benefit substantially from stopping the growth of the tree; however, there are a few benchmarks that benefit from a specialized tree, the most notable example being `vpr`.

3.4 Additional Branch Features

In the next set of experiments we have extended the number of static branch features with the ones given in Table 2. Half of the additional features are static properties of the branch's basic block. The other half relates to the successor basic blocks. The experimental results given in Figure 1 indicate the extended set decreases the average IPM over the original feature set for the default $m = 2$. However, when the growth stopping mechanism is enabled, the average IPM increases to 32.3 which is 2.9% larger than the original set. From a detailed analysis we observed that the 'number of incoming edges to the taken successor' from Table 2 was particularly valuable. Moreover, most (11 out of 16) programs benefit from the combination of an extended feature set and a setup that stops the growth of the tree. On average, the IPM increases by 18.5% over the previous work done by Calder *et al.* [3].

3.5 Comparing with Previous ESP Work

During this analysis of ESP, we observed that our results showed a higher average misprediction rates—38%—than the 25% reported by Calder *et al.* There are three possible reasons for this. First, we used different benchmarks than Calder *et al.* did. They used SPECint92 plus a set of Unix utilities which we

Table 2. Additional static features

Feature	Description
Register	Register number used in branch instruction
Loop level	Loop level of the branch instruction (start with zero in non-loop part, increase the number for each nested loop)
Basic Block Size	Size in number of instructions
Distance	Distance between register defining instruction and the branch that uses it
RA Register	Register number for RA
RA Distance	Distance between both register defining instructions
RB Register	Register number for RB
RB Distance	Distance between both register defining instructions
Register definition	The register in the branch instruction was defined in that basic block, or not
Pointer	The pointer heuristic is applicable
Opcode	Opcode heuristic is applicable
Incoming Edges	Number of possible ways to enter basic block
	Features of the Taken and Not Taken Successor
Basic Block Size	Size of successor basic block
Incoming Edges	Number of possible ways to enter successor basic block
Pre-return	Successor contains a return or unconditionally passes control to a return
Pre-store	Successor contains a store instruction or unconditionally passes control to a block with a store instruction
Direct call	Successor contains a call
Loop Sequence	Only applicable if loop exit edge: successor is also a loop at the same level, or not

believe are much less complex than SPECint95 and SPECint2000. For several SPEC benchmarks, Calder *et al.* report a misprediction rate around 30% (32% for `gcc`), which is comparable to our results. Next to these benchmarks, Calder's study also includes several C-benchmarks (alvinn, ear and eqntott) with extremely low miss rates pulling down the average miss rate to 25%. A second possible explanation is the fact that the lower compiler optimization level used by Calder *et al.* (`-O`) results in better predictable branch behavior than when using a higher optimization level (`-O4` in our study). During investigation of how much the branch predictability is affected by the chosen compiler optimization level we found that the use of a lower optimization level in Calder's paper indeed results in better predictable branch behavior. Less optimization makes the program structure more generic so that branches are easier to predict. The latter drops the average misprediction rate by 10% between `-O4` and `-O1`, and by 3% between `-O4` and `-O`. A third explanation could be the use of a different and more recent compiler. We do not believe the difference in reported misprediction rates between this paper and Calder's work comes from the fact that we use a (slightly) smaller number of benchmarks in our analysis, namely 16 versus 23 C

programs and 20 Fortran programs used by Calder *et al.* [1], because the number of static branches in our analysis is 2 times higher than the number of static branches by Calder *et al.*

4 Heuristic-Based Prediction

In this section, we show how decision trees improve the Ball and Larus heuristics' accuracy and usability. Before going into detail on how this is done, we first give a brief discussion on the Ball and Larus heuristics as proposed in [1].

The Ball and Larus heuristics start by classifying branches as loop and non-loop branches. A branch is a loop branch if either of its outgoing edges is an exit edge or a loop backedge. A branch then is a non-loop branch if neither of its outgoing edges is an exit or a backedge. Loop branches can be predicted very accurately by the **loop heuristic** which predicts that the edge back to the loop's head is taken and that the edge exiting the loop is not taken. In our analysis these loop branches account for 35% of the dynamic branches and for 11% of the static branches. The rest of the heuristics concern non-loop branches:

- The **pointer heuristic** will predict that pointers are mostly non-null, and that pointers are typically not equal, i.e. comparing two pointers typically results in non-equality.
- The **opcode heuristic** will predict that comparisons of integers for less than zero, or less than or equal to zero will evaluate false, and that comparisons for greater than zero, or greater than or equal to zero will be true. This heuristic is based on the notion that many programs use negative numbers to denote error values.
- The **guard heuristic** applies if the register used in the branch instruction is used in the successor basic block before being defined, and the successor block does not postdominate the branch. If the heuristics applies, the successor *with* the property is predicted to be the next executed block. The intuition is that guards usually catch exceptional conditions.
- The **loop header heuristic** will predict the successor that is a loop header or pre-header and which does not postdominate the branch. This heuristic will predict that loops are executed rather than avoided.
- The **call heuristic** will predict the successor that contains a call and does not postdominate the branch as not taken. This heuristics predicts that a branch avoids executing the function call.
- The **store heuristic** will predict the successor containing a store instruction and does not postdominate the branch as not taken.
- The **return heuristic** will predict that a successor with a return will be not taken.

Coverage—measured as the number of static branches to which the heuristic applies—and misprediction rate of the individual heuristics are listed in Table 3

[1] Note that Calder *et al.* did a separate analysis for C programs and Fortran programs; he did not consider them together in one analysis.

Table 3. Coverage and misprediction rates for the Ball and Larus heuristics

Heuristic	Coverage	Misprediction rate	Perfect
Loop	35%	19%	12%
Pointer	21%	39%	9%
Opcode	9%	27%	11%
Guard	13%	37%	12%
Loop Header	26%	25%	10%
Call	22%	33%	8%
Store	25%	48%	9%
Return	22%	29%	12%

for the SPECint95 and SPECint2000 benchmarks given in section 2. The measured misprediction rates correspond to those presented by Wu and Larus [17], except for the opcode heuristic where we reach a higher misprediction rate. The reason is that we use an Alpha architecture which does not allow us to implement the opcode heuristic as originally stated: conditional branches in the Alpha ISA compare a register to zero, rather than comparing two registers as is the case in the MIPS ISA (for which the Ball and Larus heuristics were developed). The opcode heuristic was implemented by applying the heuristic to the compare instruction (`cmp`) that defines the branch's operand. Calder *et al.* [3] also obtain a higher misprediction rate for the opcode heuristic for the Alpha ISA than for the MIPS ISA.

4.1 Optimal Heuristic Ordering

As already pointed out by Ball and Larus [1], the ordering of the heuristics can have an important impact on the overall misprediction rate. Ball and Larus came up with a fixed ordering for applying their heuristics, which is: $Loop \rightarrow Pointer \rightarrow Call \rightarrow Opcode \rightarrow Return \rightarrow Store \rightarrow LoopHeader \rightarrow Guard$. As soon as one heuristic applies, the branch is predicted along that heuristic and all other heuristics possibly applying are ignored. If no heuristic applies a *Random* prediction is made. For the above ordering of heuristics, we measure an average IPM of 31.3, see Figure 2. The coverage for each heuristic in this ordering is 35%, 13%, 13%, 4.5%, 7%, 8%, 1.5%, 1%, respectively. This sums to a heuristic coverage of 83%, the remainder is randomly predicted.

To identify the optimal ordering, Ball and Larus evaluated all possible combinations. Note that the total number of orderings grows quickly as a function of the number of heuristics—more in particular, for n heuristics, there exist $n!$ orderings. As such, evaluating all possible combinations quickly becomes infeasible as the number of heuristics increases (as will be the case in the next subsection). In addition, determining the optimal ordering for one particular ISA, programming language, and compiler (optimization level), does not guarantee a well performing ordering under different circumstances embodying another ISA, compiler or programming language. Therefore, it is important to have an au-

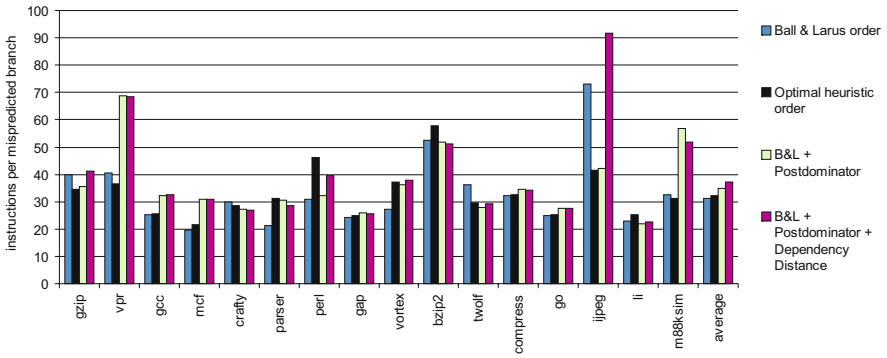

Fig. 2. The number of instructions per mispredicted branch for heuristic-based branch prediction

tomated and efficient way of finding a well performing ordering. This section proposes decision trees for this purpose.

As input during decision tree learning, we provide the evaluation of each heuristic for every static branch together with its most taken direction. We then applied our decision tree learning tool C4.5 on this data set in a cross-validation setup, i.e. for each benchmark under evaluation we build a separate tree using the information for the remaining 15 benchmarks. As such, we obtain 16 decision trees. Inspection of the trees however, revealed that most of them were quite similar to each other. The ordering obtained from this analysis is the following: $Loop \rightarrow Opcode \rightarrow Call \rightarrow Return \rightarrow LoopHeader \rightarrow Store \rightarrow Pointer$. When no heuristics can be applied it chooses the $NotTaken$ decision. The overall average IPM for this new ordering now is 32.1 which is slightly better (2.5%) than the ordering proposed by Ball and Larus. For the coverage of the heuristics we now get 35%, 5.5%, 16.5%, 9%, 4%, 8%, 4% respectively, summing up to 82%. By moving the *Pointer* heuristic to the end of the chain, the other heuristics predict a larger part more accurately, and finally the uncovered branches are mostly not taken. Note that the *Guard* heuristic, which has the lowest priority in the Ball and Larus order, is completely ignored by the decision tree. Although C4.5 is not forced to do so, the tree clearly identifies an ordering for applying the heuristics. Indeed, the decision tree could have been a 'real' tree with multiple paths from the root node to the leafs instead of a tree with one single path which is an ordering.

4.2 Adding Heuristics

Given the fact that we now have an automated way for ordering heuristics, it becomes feasible to investigate whether additional heuristics would help improving the prediction accuracy of heuristic-based static branch prediction. The question however remains to determine which heuristics should be added. To answer this

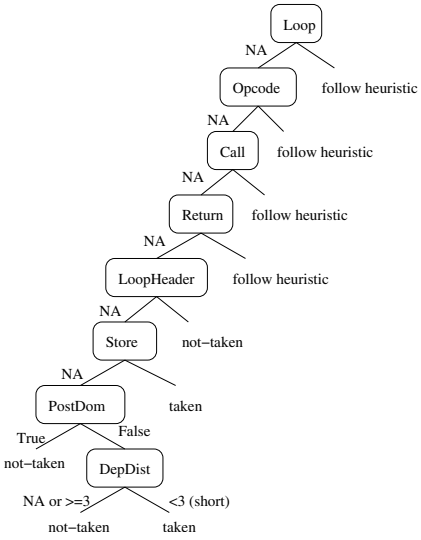

Fig. 3. Decision tree for the extended set of heuristics

question, we have done the following experiment. We have added the static features from Calder *et al.* [3] and Table 2 *one by one* to the set of Ball and Larus heuristics. Using this set of heuristics we have built up a decision tree using C4.5 and we have measured the resulting static branch prediction accuracy[2]. For each of the static features we thus have a prediction accuracy when added to the set of Ball and Larus. Subsequently, the static feature for which the extended set of heuristics achieved the highest prediction accuracy was selected for permanent inclusion in the extended set. Using this extended set we then iterate this algorithm using the remaining static features until the static prediction accuracy does no longer improve.

This experiment revealed two heuristics that when added to the set of Ball and Larus improve the overall prediction accuracy. The first one concerns the postdominator relationship between the successor and the current basic block. This heuristic states that if a branch has two successors of which one postdominates the current basic block, the successor that does not postdominate the current block should be predicted taken. The simplest example to which this *predict-non-postdominating-successor* heuristic applies is an if-statement (without else-block); the heuristic will then predict the if-block to be taken. The second heuristic is based on the dependency distance between the branch and its operand defining instruction, i.e. the number of instructions between producing a register value and consuming it by the branch. If the operand defining instruction is not part of the branch's basic block, the dependency distance is left undefined. This newly proposed *dependency distance* heuristic states that a

[2] Note that in this experiment we also use cross-validation. Further, we assume $m = 300$.

branch with an undefined dependency distance or a dependency distance larger than 3 should be predicted not-taken. The threshold on distance 3 was found empirically, but changing it to 2 or 4 does not significantly affect the results.

Figure 3 shows the decision tree provided by C4.5 for the extended set of heuristics, i.e. the set of Ball and Larus heuristics plus the predict-non-postdominating-successor and the dependency distance heuristics. This tree shows that the *Pointer* heuristic is replaced with our heuristic extensions. Replacing the *Pointer* heuristic by the postdominator heuristic results in a coverage increase of 2% together with a significant misprediction rate reduction for that specific class.

The number of instructions per mispredicted branch (IPM) is shown in Figure 2 with the extended set of heuristics. Adding the predict-non-postdominating-successor heuristic improves the IPM from 31.3 to 34.7; the dependency distance heuristic further improves the IPM to 37.1 which is an increase by 18.5% over the Ball and Larus heuristics. To conclude, the extended set of heuristics covers all branches because the remaining 16% are predicted by their dependency distance.

5 Profile-Based Prediction

As stated earlier in this paper, profile-based static branch prediction is more accurate than program-based branch prediction. There are however two important issues that need to be dealt with. The first issue is the selection and/or combination of profile inputs. The second issue is the prediction of branches that are unseen during profiling. The following two subsections show how decision trees can be used to address both issues. In these experiments we consider the test and train inputs as our profile inputs.

5.1 Comparing Decision Trees Versus Address-Based Prediction

The easiest way for selecting a profile input is by picking a randomly chosen input (in our settings, we randomly choose the test or train input) and to assign the most likely direction to each branch based on the observed behavior of the chosen profile input. The branch prediction accuracy for this approach is shown in the first column of Table 4 for all SPECint2000 benchmarks.

Previous work however has shown that if multiple profiling inputs are available, combining those can significantly improve prediction accuracy. Fisher and Freudenberger [7] propose three methods for combining multiple profiles: (i) *polling* gives each profile input one vote to predict the direction of a branch, decision by majority or taken in case of a tie; (ii) *unscaled* adds the taken (not taken) counts for the different profiles, majority direction is predicted; (iii) *scaled* also adds the taken (not taken) counts for the different profiles but scales these counts by the branch execution frequencies in each profile. The results in Table 4 clearly indicate that unscaled and scaled profile combining methods perform better than random selection. More in particular, the unscaled and scaled methods

Table 4. Number of instructions per mispredicted branch (IPM) for profile-based branch prediction: (from left to right) random, polling, unscaled, scaled, 'orig' decision trees using Calder's features [3] and 'extd' decision trees additionally using the features in 2, combining the extended decision trees with unscaled, combining the extended decision trees with scaled

benchmark	input	random	polling	unscaled	scaled	orig	extd	unscld+extd	scld+extd
gzip	graphic	94.22	31.90	93.32	93.33	53.31	61.82	93.32	93.33
	log	63.74	23.21	115.63	115.64	27.62	72.91	115.63	115.64
	program	68.83	40.11	79.97	79.97	50.89	67.93	79.97	79.96
	random	212.26	32.03	190.87	190.87	79.47	80.11	190.87	190.87
	source	44.34	21.31	97.00	97.00	26.23	72.88	97.00	97.00
vpr	place	29.49	25.96	29.49	29.49	35.50	35.98	36.76	36.76
	route	128.41	18.75	129.37	127.54	46.33	76.29	129.37	127.54
gcc	166	330.56	49.81	343.10	342.95	118.49	149.30	343.20	343.05
	200	103.41	24.64	108.07	106.78	52.48	67.32	108.33	107.03
	expr	141.88	27.17	142.57	142.55	70.57	89.41	142.57	142.55
	integrate	222.56	35.47	223.96	223.89	95.62	114.94	223.96	223.89
	scilab	104.86	25.14	106.61	106.56	57.26	71.44	106.86	106.81
mcf	ref	44.99	14.37	49.94	49.91	47.96	49.54	49.94	49.91
crafty	ref	62.19	28.02	61.71	64.89	53.05	58.09	61.72	64.90
parser	ref	61.58	17.57	63.91	63.77	50.82	62.60	63.91	63.77
perlbmk	splitmail 850	32.64	20.58	32.62	34.41	65.91	44.70	37.93	40.37
	splitmail 704	33.80	20.97	33.32	36.93	16.79	11.48	37.87	42.60
	splitmail 535	33.18	20.68	33.12	35.29	30.69	20.47	38.20	41.11
	splitmail 957	33.57	30.96	32.96	36.67	32.47	21.72	37.53	42.41
	makerand	58.83	42.56	63.63	63.63	63.16	19.82	63.63	63.63
	diffmail	39.70	23.69	42.89	43.92	34.38	22.63	42.89	43.92
gap	ref	89.65	24.37	94.11	87.43	73.02	72.74	94.05	87.39
vortex	lendian1	1016.59	16.14	1014.40	1018.95	121.13	249.08	1014.40	1018.95
	lendian2	724.35	17.80	726.31	725.93	105.97	203.37	726.31	725.93
	lendian3	1047.88	16.32	1045.09	1050.87	121.71	253.67	1045.09	1050.87
bzip2	graphic	133.99	27.83	133.43	134.71	90.73	55.98	133.43	134.71
	program	90.40	27.13	103.39	101.14	70.26	52.33	103.39	101.14
	source	63.73	28.26	80.18	78.81	54.08	45.46	80.19	78.82
twolf	ref	43.14	19.24	61.92	63.07	40.49	44.14	61.92	63.07
average		85.34	22.46	93.39	93.68	55.40	62.03	95.13	95.50

outperform polling and random selection. These four methods all assign branch predictions on a per-branch basis; we will therefore refer to them as address-based prediction techniques.

We now study how decision trees can be used to combine multiple profile inputs. For this purpose, we use the sets of static branch features from Calder and from Table 2 and build a decision tree based on the test and train inputs of the corresponding benchmark. [3] The accuracy of the decision tree is then evaluated for the available reference inputs, which were unseen during decision tree learning. The 'orig' column in Table 4 gives the IPM when using the feature set proposed by Calder *et al.* [3]; the 'extd' column gives the IPM using the extended set of static features (Calder's features and 2 together). These data illustrate that the extended set of features yields more accurate predictions than the original set by Calder *et al.* In addition, the decision trees perform better than polling. The comparison between unscaled/scaled and orig/extd also illustrate

[3] For these experiments m=2 (default) for providing maximum flexibility to the decision trees in order to approximate profile-based techniques.

the discrepancy between program-based and profile-based prediction. An important advantage decision trees have over address-based prediction techniques is that the decision trees can be interpreted so that programmers or compiler writers can learn more about their software's branching behavior. This information could be valuable for optimizing their software.

5.2 Combining Address-Based and Decision Tree Prediction

As stated earlier, an important problem with profile-based branch prediction is that it does not provide information about branches that are unseen in the profile runs. In our experiments, for seven inputs, i.e. one input for vpr and all for perlbmk, the profile runs did not cover all executed branches. For vpr–place, 14% static branches that account for 65% dynamic branches are uncovered; for perlbmk, 33% static branches accounting for 51% of the dynamic branches are uncovered. For the other benchmarks, the percentage dynamic branches that were uncovered by the profile runs was less than 1%.

We propose to use decision trees for unseen branches instead of randomly assigning a branch direction—in our experiments we assigned taken. For the other branches, that are seen in the profile runs, we use the unscaled profile combining approach by Fisher and Freudenberger [7]. Similar results for the scaled combination are shown in the rightmost column in Table 4. This approach increases IPM by 11.7% and 10.5% for vpr and perlbmk, respectively. As such, we conclude that decision trees can be successfully used in profile-based branch prediction for branches that are unseen in the profile runs.

6 Related Work

The first subsection on related work focuses on static branch prediction for which the primary goal is to guide compiler optimizations. The second subsection illustrates other applications of machine learning techniques.

6.1 Static Branch Prediction

Program-based static branch prediction. One of the simplest program-based heuristics is 'backward-taken/forward-not-taken' (BTFNT). This heuristic is based on the observation that loop branches are typically backward branches and as such are likely to be taken. Although simple, this heuristics was proven to be successful. Smith [13] discussed several static prediction strategies based on instruction opcodes. Bandyopadhyay *et al.* [2] used a table lookup using the branch opcode and operand types to determine the direction for non-loop branches. Wall [15] evaluated several heuristics for predicting basic block frequencies: the basic block loop nesting depth, a combination of the loop nesting depth and the distance to the call graph leaf, and two combinations of the loop nesting depth and the number of direct calls to the block's procedure.

Deitrich *et al.* [6] extended the Ball and Larus heuristics by incorporating source-level information available in a compiler when performing static branch prediction. The source-level information they used concerns I/O buffering, exiting, error processing, memory allocation and printing.

Wong [16] also investigated in source-level prediction by introducing the use of names (macro, function, variable) for static branch prediction.

Patterson [10] used value range propagation which tracks value ranges of variables through a program. Branch prediction is then done by consulting the value range of the appropriate variable.

Profile-based static branch prediction. Savari and Young [12] developed a technique for combining profiles using information theory, with the notion of entropy. Although this approach attained good prediction accuracies, extending this approach to more than two profiles is not straightforward.

Young and Smith [18] proposed profile-based code transformations that exploit branch correlation to improve the accuracy of static branch prediction. If a branch exhibited a different behavior on different paths, they duplicated the code and provided different static predictions along the different paths.

6.2 Machine Learning

As in this work, Calder *et al.* [3] utilized decision trees, cfr. section 3.1. However, they did neither discuss the structure of the decision trees learned nor identified the applicability of decision trees for heuristic-based branch prediction.

Cavazos *et al.* [4] applied supervised learning techniques for inducing heuristics to predict which blocks of code would benefit from scheduling. The static features they used are the number of instructions in the block and the fraction of instructions of a certain category. They showed that rule induction can successfully use these features to determine whether to schedule or not.

Stephenson *et al.* [14] employed genetic programming to automatically search for effective heuristic priority functions in various compiler optimizations. Given a set of heuristics they tuned the priority function by evolving it over several benchmarks.

7 Conclusion

Static branch prediction is an important issue with several important applications ranging from compiler optimizations, to improving dynamic branch predictors, to improving performance of embedded microprocessors lacking a dynamic branch predictor. There are two well known static branch prediction techniques, namely program-based and profile-based branch prediction. The benefit of program-based prediction is its low cost, whereas profile-based is more accurate. This paper showed how decision trees, previously proposed in the context of evidence-based branch prediction, can be used to improve both static branch prediction approaches.

It is important to emphasize that the biggest advantage of using decision trees is the fact that they can be used to automatically generate static branch predictors. In other words, they are optimized (by construction) for a given compiler, the given programming language in which the benchmarks are written and the given ISA. As such, we are aware of the fact that the experimental results that are obtained in this paper are sensitive to the chosen compiler, benchmarks and ISA. However, we strongly believe that the major contribution of this paper—showing how decision trees can be used to improve static branch prediction—will be applicable under different setups.

We have presented a set of static branch features that when added to the previously proposed set by Calder *et al.* also increases accuracy. We have shown that the use of decision trees improves the Ball and Larus heuristics for two reasons: (i) by automatically finding a well performing ordering for applying the heuristics, and (ii) by automatically finding additional heuristics. Our experimental results on SPECint95 and SPECint2000 show that these two contributions increase the IPM by 18.5%. The two additional heuristics identified in this paper are related to the postdomination relationship between the successor and the current basic block—the non-postdominating successor is predicted taken—and the dependency distance between the branch's operand and its defining instruction—short distances result in more likely taken branches. Finally, we have also shown that decision trees improve the accuracy of address-based techniques up to 11.7% when used in combination. Decision trees prove to be effective for branches that are unseen during profiling.

Acknowledgments

Veerle Desmet is supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT). Lieven Eeckhout is a postdoctoral researcher of the Fund for Scientific Research-Flanders (FWO). This research was also funded by Ghent University and HiPEAC. We are indebted to Bruno De Bus for his support and modifications to Squeeze.

References

1. T. Ball and J. R. Larus. Branch prediction for free. In *PLDI*, pages 300–313, June 1993.
2. S. Bandyopadhyay, V. S. Begwani, and R. B. Murray. Compiling for the CRISP microprocessor. In *Proc. of the Spring 1987 COMPCON*, pages 96–100, Feb. 1987.
3. B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, Jan. 1997.
4. J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI*, pages 183–194, June 2004.
5. S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM ToPLaS*, 22(2):378–415, Mar. 2000.

6. B. L. Deitrich, B.-C. Cheng, and W. mei W. Hwu. Improving static branch prediction in a compiler. In *PACT*, pages 214–221, Oct. 1998.
7. J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *5th ASPLOS*, pages 85–95, Oct. 1992.
8. T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.
9. H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *HPCA*, pages 251–262, Jan. 2000.
10. J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78, June 1995.
11. J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
12. S. Savari and C. Young. Comparing and combining profiles. *JILP*, 2, Apr. 2000.
13. J. E. Smith. A study of branch prediction strategies. In *ISCA*, pages 135–148, May 1981.
14. M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI*, pages 77–90, June 2003.
15. D. W. Wall. Predicting program behavior using real or estimated profiles. In *PLDI*, pages 59–70, June 1991.
16. W. F. Wong. Source level static branch prediction. *The Computer Journal*, 42(2):142–149, 1999.
17. Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *27th MICRO*, pages 1–11, Nov. 1994.
18. C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *6th ASPLOS*, pages 232–241, Oct. 1994.