Decision Trees for Improving Heuristic-Based Static Branch Prediction

Veerle Desmet

Promoter(s): Koen De Bosschere

Abstract—**Static branch prediction determines the most frequent direction of control flow in programs, e.g. predicting whether `if` or `else` will be chosen most of the time. Improving static branch prediction is an important problem with various interesting applications ranging from compiler optimizations, to improving dynamic branch predictors, to improving performance of embedded microprocessors lacking a dynamic branch predictor.**

This paper builds on previous work done on evidence-based static branch prediction which uses decision trees to classify branches [1]. We demonstrate how decision trees can be used to improve the Ball and Larus heuristics [2] by optimizing the sequence of applying the heuristics and by adding two new heuristics, namely one based on the postdomination 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% for SPECint95 and SPECint2000.

I. Heuristic-Based Prediction

We first give a brief discussion on the Ball and Larus heuristics as proposed in [2]. The Ball and Larus heuristics start by classifying branches as loop and non-loop branches. Loop branches, e.g. `for, while` can be predicted very accurately by the **loop heuristic** which predicts that loops are repeated. These loop branches account for 35% of the dynamic conditional branches. The rest of the heuristics concern non-loop branches:

- The **pointer heuristic** predicts that pointers are mostly non-null, and that two pointers are occasionally equal.
- The **opcode heuristic** assumes many programs use negative numbers to denote error values.
- The **guard heuristic** predicts the successor that uses the branch register before defining it. The intuition is that guards usually catch exceptional conditions.
- The **loop header heuristic** predicts that loops are executed rather than avoided.
- The **call heuristic** predicts that a branch avoids executing a function call.
- The **store heuristic** predicts the successor containing a store instruction.
- The **return heuristic** predicts that a successor with a return will be not taken.

A heuristic applies if it predicts exactly one successor.

Coverage—measured as the number of static branches to which the heuristic applies—and miss rate of the individual heuristics along with the highest possible (perfect) static miss rate are listed in Table I.

V. Desmet is with the Electronics and Information Systems (ELIS) department, Ghent University, and supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT). E-mail: Veerle.Desmet@UGent.be .

Heuristic	Coverage	Miss 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%

TABLE I

COVERAGE AND MISS RATES FOR THE BALL AND LARUS HEURISTICS.

II. Optimal Heuristic Ordering

In case more than one heuristic applies, problems occur when different directions are predicted. An *ordering* solves this: 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. As already pointed by Ball and Larus [2], the ordering of the heuristics can have an important impact on the overall miss rate. Ball and Larus came up with a fixed ordering for applying their heuristics, which is: $Loop \rightarrow Pointer \rightarrow Call \rightarrow Opcode \rightarrow Return \rightarrow Store \rightarrow LoopHeader \rightarrow Guard$. To identify the optimal ordering, Ball and Larus evaluated all possible combinations and thus time-consuming. In addition, determining the optimal ordering for one particular instruction set architecture (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 automated 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 [3] on this data set, and the ordering (no real tree!) obtained from this analysis is: $Loop \rightarrow Opcode \rightarrow Call \rightarrow Return \rightarrow LoopHeader \rightarrow Store \rightarrow Pointer$. When no heuristic can be applied it chooses the $NotTaken$ decision. This new ordering is slightly better (2.5%) than the ordering proposed by Ball and Larus. Note that the $Guard$ heuristic, which has the lowest priority in the Ball and Larus order, is completely ignored by the decision tree.

III. Adding heuristics

Given the automated way for ordering heuristics, we further investigate whether additional heuristics would help improving

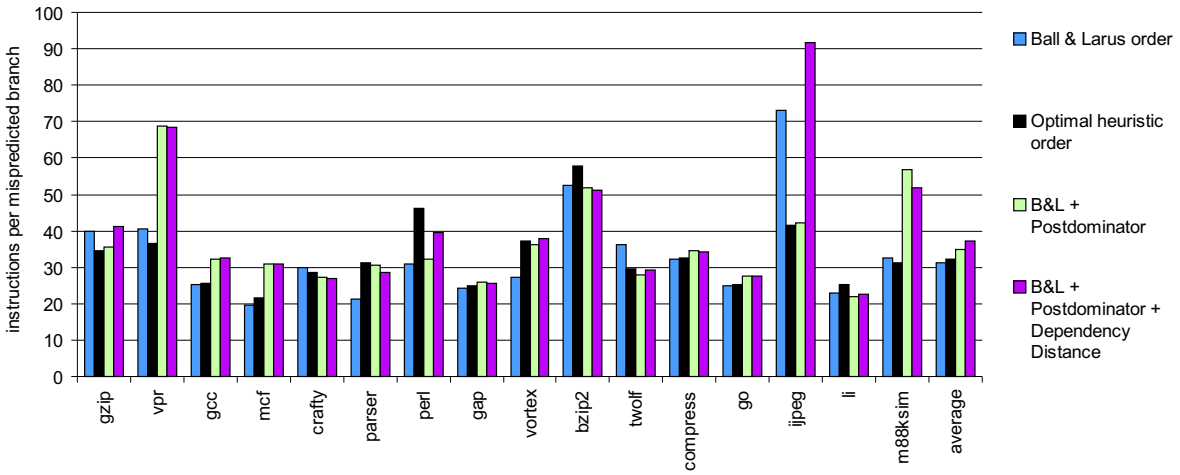

Fig. 1. The number of instructions per mispredicted branch for heuristic-based branch prediction.

the prediction accuracy. Therefore, we have evaluated several (new) information elements available in program structure. 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. The simplest example to which this *predict-non-postdominating-successor* heuristic applies is an if-statement (without else-block); the heuristic then predicts 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 short distances result in more likely taken branches.

Figure 2 shows the final 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.

For fair comparison of the benchmarks, we have to use the *higher-is-better metric IPM*, the average number of instructions per mispredicted branch. Details are shown in Figure 1. 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.

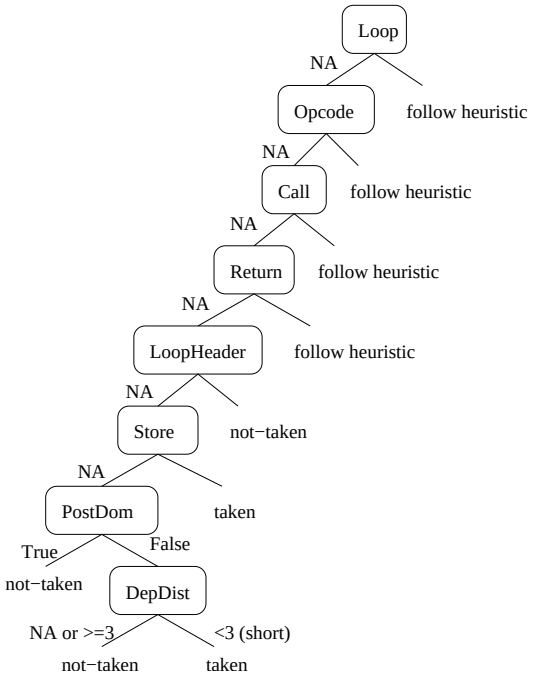

Fig. 2. Final decision tree for the extended set of heuristics.

IV. Conclusion

Static branch prediction is an important issue with several applications ranging from compiler optimizations, to improving dynamic branch predictors, to improving performance of embedded microprocessors. This paper showed how decision trees can be used to improve heuristic-based static branch prediction for two reasons: (i) by automatically finding a well performing ordering for applying the heuristics, and (ii) by automatically finding additional heuristics. 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 branch and its successors—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.

References

[1] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, pp. 188–222, Jan. 1997.

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

[3] J.R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.