Evaluation of the Gini-index for Studying Branch Prediction Features

Veerle Desmet   Lieven Eeckhout   Koen De Bosschere

Ghent University
Dept. of Electronics and Information Systems (ELIS)
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
Fax. +32 9 264 35 94
{vdesmet,leeckhou,kdb}@elis.UGent.be
`http://www.elis.ugent.be/paris`

Abstract. Predicting future outcomes based on past observational data is a common application in data mining. While the primary goal is usually to achieve the highest possible prediction accuracy, the interpretation of the resulting prediction model is important to understand its shortcomings for further improvements.

Throughout this paper we focus on branch prediction, where the (binary) outcome of a test is needed for enhancing the performance of pipelined computer architectures. Many research has been done in this domain and different branch prediction solutions are described in the literature.

The quality of a prediction model is highly dependent on the quality of the available data. Especially the choice of the related variables or *features* to base the prediction on is important. In this paper we evaluate the predictive power of different branch prediction features using the metric *Gini-index*, which is used as feature selection measure in the construction of decision trees. We observe that through this Gini-metric an explanation can be provided for the performance of existing branch predictors. We show that the Gini-index is a good metric for comparing branch prediction features. Further, we found that a feature can have good discriminative capacities, although this does not result in very good accuracies because of shortcomings in the predictor implementation.

Keywords: computer architecture, branch prediction, data mining, decision tree, Gini-index

1. INTRODUCTION

Deeply pipelined computer architectures as we know them today rely on a fetch mechanism that provides one or more useful instructions every clock cycle. This constant feeding process encounters difficulties because of the control dependencies, which determine the flow of the program at run time. Especially conditional branches cause difficulties: the next instruction to be executed is not known until the branch condition (e.g., argument equals zero) is computed. This computation typically completes 3-14 cycles after the branch has entered the pipeline, meanwhile no further instructions can be fetched.

To solve this situation, an essential part of modern microarchitectures consists of branch prediction. A branch predictor predicts the outcome of the branch condition so that instructions on the predicted path can enter the pipeline the next cycle. This method enables a constant pressure on the pipeline but involves additional complications for handling speculative instructions and verifying the prediction. Of course the branch instruction itself is executed to verify the prediction. On a correct prediction all speculative instructions are useful and finish earlier compared to no branch prediction. On a misprediction however, all speculative work has to be undone before the correct path executes. This means that on a misprediction there is even an extra penalty compared to not predicting. As pipelines deepen and the number of instructions issued per cycle increases, the penalty for a misprediction also increases. Current branch predictors already reach 95% prediction accuracy and continuous improvements and new directions are exploited in this domain. The driving force behind these efforts is the large improvement in the average number of instructions executed per cycle (IPC) that is possible by even a small improvement in prediction accuracy [1].

2. BRANCH PREDICTORS

Branch prediction distinguishes two classes of prediction schemes: static and dynamic. Static branch prediction associates a fixed prediction for each static branch at compile-time, while dynamic prediction makes a prediction at run time. As backward branches (towards lower addresses, e.g., in loops) tend to be taken mostly, the static strategy 'predict backward taken, forward not taken' [2] achieves prediction accuracies up to 63%. Improved static predictors use profile information and program-based heuristics [3] to get accuracies up to 75%. More recent research on static prediction in [4] does not step over 80% accuracy, which is worse than almost any dynamic scheme.

Dynamic branch prediction schemes exist in many variations. In what follows we give an overview of classical prediction schemes: bimodal, local, global, gshare and hybrid branch predictors. Figure 5 presents the prediction accuracies that can be obtained with these dynamic branch predictors as a function of their hardware budget measured in bytes of storage. The results were obtained using the SimpleScalar/Alpha sim-bpred simulator [5]. We simulated the 12 programs in the SPEC CPU 2000 suite of integer benchmarks which are compiled with the Compaq C compiler version V6.3-025 with optimization flags `-arch ev6 -fast -O4`. As several benchmarks have an initialization period during which branch prediction accuracy is unusually high, we skip the first 50 million conditional branches. For each benchmark we measure for the next 250 million conditional branches and the presented results show the average over all the SPEC benchmarks.

2.1. Bimodal branch predictor

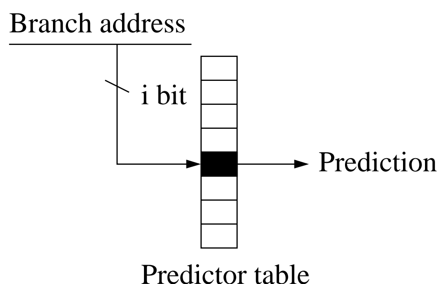

FIGURE 1. Bimodal predictor

The bimodal prediction scheme [2] in Figure 1 is very simple: a prediction table is indexed by the program counter or branch address and the contents is used to generate the prediction. Each table entry keeps a 2-bit saturating counter. The branch is predicted taken if the counter value is above a certain threshold, e.g., 2, and not-taken if it is not. The idea behind this indexing scheme is that all instances of a static branch use the same entry and this performs well because previous outcomes provide useful information to predict the next occurrence. When the branch resolves, the effective direction is known and the used counter is incremented or decremented on a taken or not-taken branch, respectively.

2.2. Local history branch predictor

Figure 2 illustrates the local prediction scheme where two prediction tables are managed. As in the bimodal predictor the first table is indexed by the program counter, but now a *local history* is kept in the table. The local history indexes the second table where a saturating counter value indicates the branch direction prediction. The extra level allows different local histories of a particular static branch to use different saturating counters and therefore the prediction can be made more accurate. For example for a loop as `for (i=0; i<4; i++) {...}` where the loop test is done at the end of the body, the corresponding branch executes the pattern *taken–taken–taken–not-taken*. When the loop is executed a number of times, the next outcome follows from the previous three directions and a local history predictor makes correct predictions. In Figure 5 a local history predictor is considered where both prediction tables have the same number of entries, or $i = k$. When the branch resolves, counter value and local history are updated.

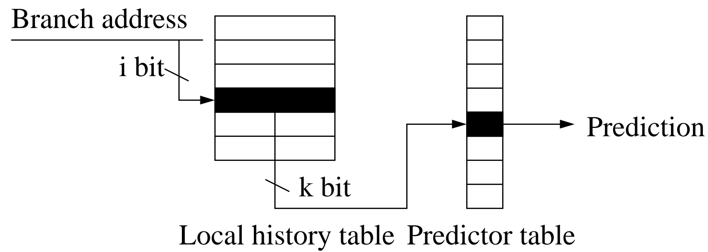

FIGURE 2. Local history predictor

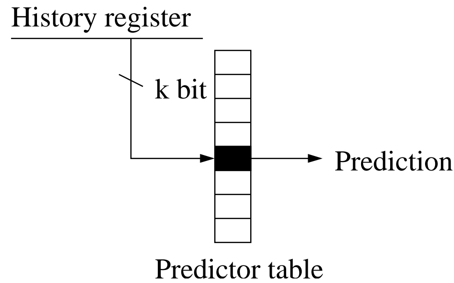

FIGURE 3. Global history predictor

2.3. Global history branch predictor

A global history predictor, in Figure 3, uses the *global history*, i.e., the outcomes of recently resolved branches, for indexing the counter table. The global history is stored in a register in which the outcome of a resolved branch is shifted during update. This scheme is able to take advantage of other recent branches to make a prediction and it performs better for sufficiently large predictors. This is illustrated in Figure 5, where the prediction accuracy is plotted for varying predictor sizes.

2.4. Gshare branch predictor

Another indexing strategy is used in the gshare predictor proposed by McFarling [6]. The scheme in Figure 4 tries to combine good points of both bimodal and global prediction by using the XOR-result of branch address and global history as index in the prediction table. This XOR-operation gives more information than either component alone. Especially for large table sizes the gshare predictor performs very well with accuracies up to 95% as shown in Figure 5. An 8KiB-gshare predictor is used in the AMD K6 architecture and achieves a prediction accuracy of 95%.

2.5. Hybrid branch predictors

Each of the presented prediction schemes has specific advantages and disadvantages. A natural question is whether they can be combined to make a new predictor with better prediction accuracy. In a hybrid predictor, two different predictors generate a prediction while a third predictor serves to select which predictor to use. Hybrid predictors achieve higher prediction accuracies: this is illustrated in Figure 5 for a combination of bimodal and gshare predictor as described by McFarling [6]. In the Alpha 21264 microprocessor [7] the branch predictor is a hybrid prediction scheme that dynamically chooses between local and global history to predict the direction of a given branch.

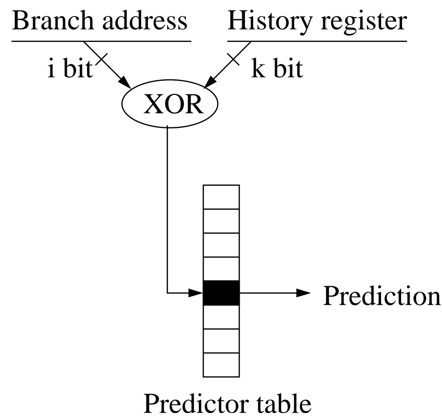

FIGURE 4. Gshare predictor

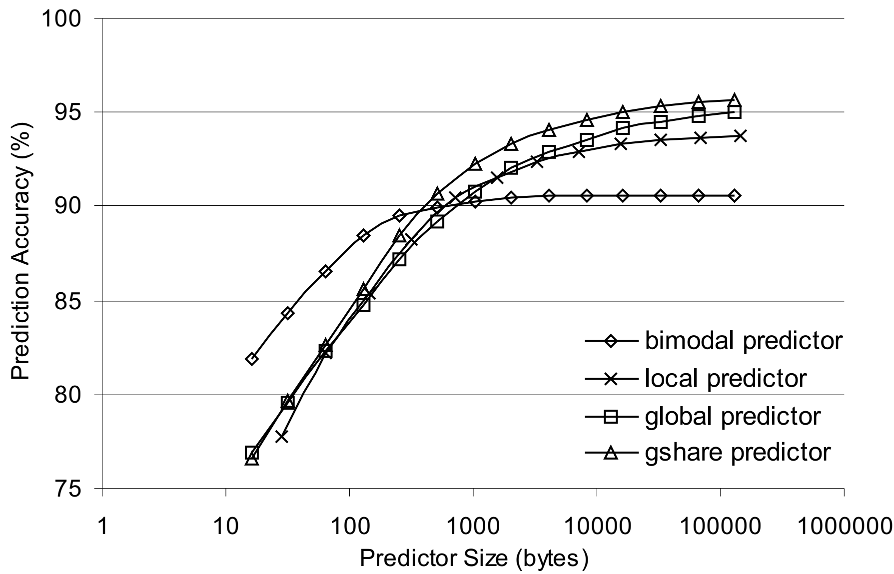

FIGURE 5. Branch prediction accuracy in function of predictor size for bimodal, global history, local history, and gshare prediction scheme.

2.6. Branch prediction accuracy

In Figure 5 the prediction accuracy is shown as a function of the predictor size for the different above-described branch prediction schemes. We note that for small predictor sizes a bimodal prediction scheme outperforms all others, whereas at large sizes the gshare predictor performs best. Note that in current microprocessor designs the predictor sizes mostly range between 1,000 and 10,000 bytes: smaller predictors are not accurate enough and larger predictors require too much hardware compared to the small improvement. In the remainder of this paper we try to find the underlying reason for the behaviour of these predictors by evaluating the different features behind their prediction strategies. To achieve this we apply some data mining aspects, more specifically we use the Gini-index as metric for feature selection.

3. DATA MINING

Data mining is concerned with all those techniques that extract hidden and unexpected relationships and knowledge in large data sets. Today data mining techniques are used in a broad range of applications, e.g., identifying buying patterns of customers, detecting patterns of fraudulent credit card use, identifying successful medical therapies for different illnesses...

Classification is a data analysis process of finding a model for describing and distinguishing data classes . Its purpose is to use the model for predicting new observations whose class is unknown. In a first step, a model is built upon some strategy by analyzing the data whose class is known (supervised learning). In the second step, the model is evaluated (e.g., prediction accuracy) while it is being used for classifying previously unseen data.

There exist two main techniques for performing classification: decision tree induction and neural (network) induction. Each of them has its own strengths and weaknesses. Decision trees are easily understandable but they are sensitive to outliers and noisy observations. In contrast, a neural approach can find more robust prediction models but is in general hard to interprete. In this paper we use the construction of decision trees because our primary goal is to gain insight in the prediction process.

3.1. Decision tree

The construction of a decision tree is a recursive algorithm:

1. The tree construction starts with a single node representing all the available observations in the data set.
2. If all observations belong to the same class, the node becomes a leaf node for that class.
3. Otherwise, a mechanism selects the feature that will best separate the observations into the different classes. This becomes the decision at the node.
4. For each occurring value of the decision a branch is created and the observations are partitioned accordingly.
5. For each new node recursively apply steps 2–5.
6. The recursion stops if and only if one of the following holds: (i) the observations belong to the same class, or (ii) there are no features left, or (iii) there are not enough (predefined number, e.g., 10) observations left to be general. In the second and third case the majority vote is assigned at the leaf node.

Different data mining approaches use different mechanisms for selecting the best feature to split on. A commonly used criteria (e.g., in CART [8]) is based on the Gini-index, which we discuss in the next section. This metric is derived from the decision construction process and will be used to compare the discriminative power of different prediction features.

3.2. Feature selection measure

The Gini-index is used to select the feature at each internal node of the decision tree.

We define the Gini-index for a data set S as follows:

$$Gini(S) = 1 - \sum_{i=0}^{c-1} p_i^2 \tag{1}$$

where,

c the number of predefined classes,
C_i the classes for $i = 0, \ldots, c-1$,
s_i the number of examples in C_i,
$p_i = \frac{s_i}{S}$ the relative frequency of class i in set S.

This metric indicates the partition purity of the data set S.

For branch prediction we have two classes (corresponding to taken and not-taken branches) and therefore the Gini-index lies within [0, 0.5]. If all the data in S belong to the same class, $Gini(S)$ equals the minimum value 0, which means that S is pure. If $Gini(S)$ equals 0.5, all observations in S are equally distributed among the classes.

The quality of a split on a feature into k subsets S_i is then computed as the weighted sum of the Gini-indices of the resulting subsets:

$$Gini_{split} = \sum_{i=0}^{k-1} \frac{n_i}{n} Gini(S_i) \tag{2}$$

where,

n_i is the number of examples in subset S_i,
$n = \sum_{i=0}^{k-1} n_i$.

Thus $Gini_{split}$ is calculated for all possible features and the feature with minimum $Gini_{split}$ is selected as split-point.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the predictive power of several features for branch prediction. We start by reporting the discriminative capacity through the Gini-index for 1 bit dynamic features and some static features. Then we compare the different features as they are used in existing dynamic branch predictors.

4.1. Individual feature bits

Although there are many similarities between the discussed prediction schemes (e.g., use of tables with saturating counter mechanisms), their prediction behaviour is quite different. This is because the input to the predictors is different, so their prediction is based on different features.

The input to the bimodal predictor is the branch address, for the global history predictor it is the global history, while gshare uses the XOR-result of both previously mentioned features. A local history branch predictor produces a prediction based on local history information. The branch address and the global history (kept in a shift register) are directly available at the time the prediction has to be made. This is not the case for the local history because it depends on the branch address. Moreover, as the local history table from Figure 2 is limited, several static branches use the same table entry and thus they interfere in the local history. As our aim is to evaluate the different features without considering any implementation constraint, we measure the *perfect local history* as they would be available from an infinite local history table.

Besides the just mentioned dynamic features, some static features (available at compile-time) are used. They include target direction (forward or backward), branch type (e.g., bne, bgt, ble,. . .), ending type of taken-successor-basic-block (e.g., fall trough, return, conditional branch,. . .) and ending type of not-taken successor basic block. The latter static features were proposed in [4] and are properties of the basic block partitioning of a program. The target direction can be represented by just one bit, while the other static features require each 3 bits [1].

Figure 6 compares the individual bits from the different dynamic features against the static features by the Gini-index. E.g., for the global history every bit in the 96-bit long history was evaluated separately. Recall that the lower the Gini-index, the better the discriminative power of the feature. The Gini-index is about the same for all bits of global history, branch address and gshare-index, but we notice a much lower Gini-index for the (perfect) local history. This is because the perfect local history is kept per branch address. But as previously explained the local history as used in the local history branch predictor cannot be modeled independently from a specific implementation or prediction table size.

Another point is that the static features perform better (except for the perfect local history) than the 1-bit dynamic features. In particular this holds for the feature *target direction*, which also involves 1 bit and thus its predictive power is better than e.g., a bit from the global history. The other static features require more bits and therefore they are likely to have better discriminatory properties.

Of course, we are not limited in using one bit as prediction feature but it gives an idea of feature power in terms of Gini-index.

---

[1] This depends on how many different possibilities are considered for these features, which is closely related to the defined instruction set architecture.

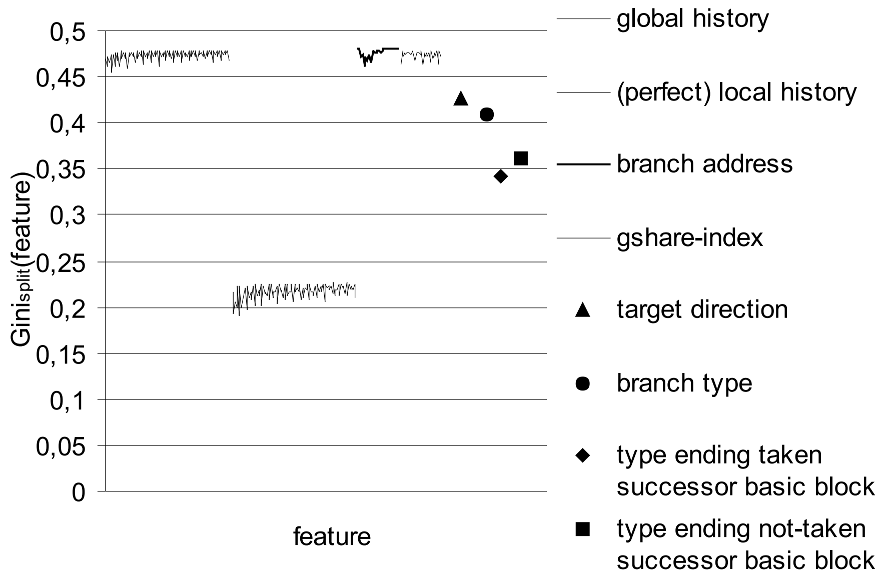

FIGURE 6. Gini-index for the individual bits in the dynamic features and for some static features.

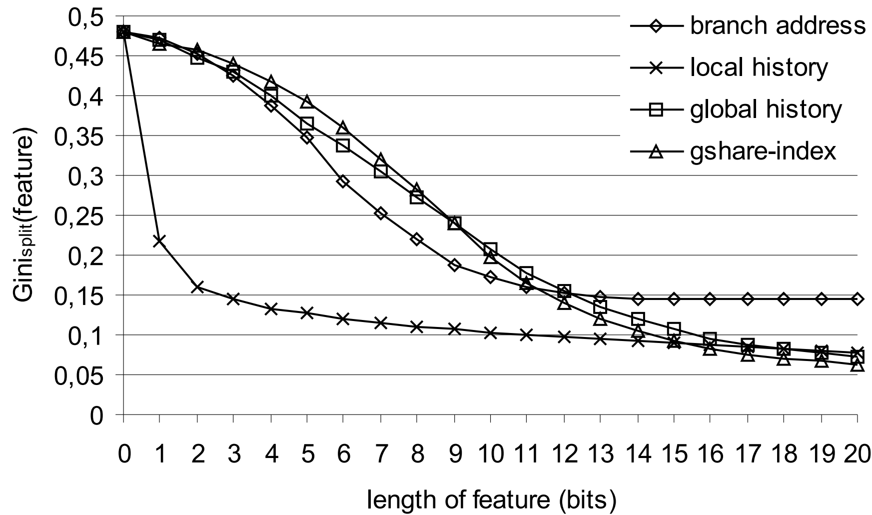

FIGURE 7. Evolution of Gini-index as more bits are used for the global history, branch address, gshare-index and (perfect) local history.

4.2. Features as used in existing branch predictors

In this section we evaluate the different features as they are used in existing branch predictors. Therefore we extend the previous analysis to combinations of consecutive bits of dynamic features.

In Figure 7 the resulting $Gini_{split}$ is shown as function of the number of bits used in the branch address, the global history, the (perfect) local history and the gshare-index, respectively. We note that these curves differ from the corresponding ones in Figure 5, where the prediction accuracy is plotted for a particular implementation of a branch

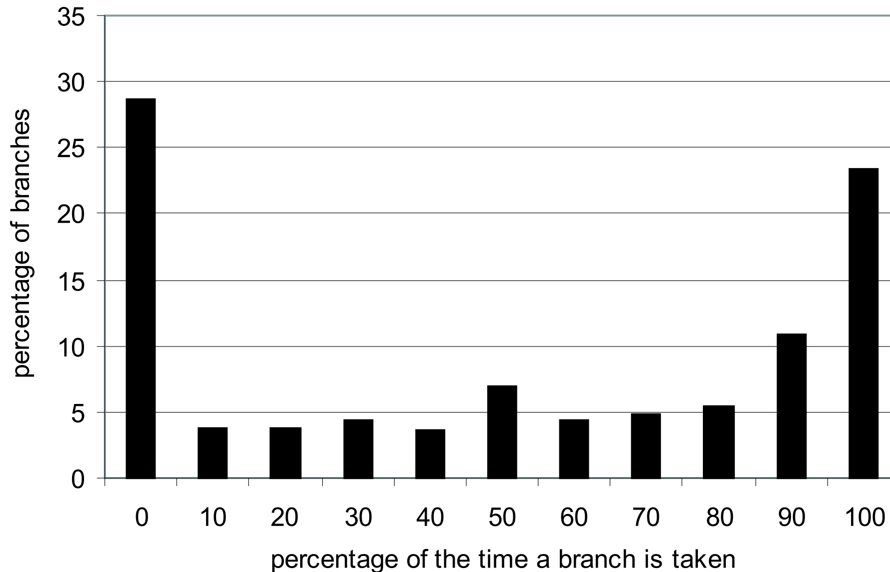

FIGURE 8. Branch behaviour histogram: for each static branch we measure the taken rate and the number of dynamical occurrences of that branch. The graph shows that the majority of branches are either mostly taken, either mostly not-taken.

prediction scheme. Here, the discriminative capacity of the different features is considered without any particular implementation in mind. e.g., the point with a 10-bit global history was constructed in the following manner: for each of the 2^{10} possibilities of the global history we count how many branches were taken and not-taken, respectively; then we calculate the value for $Gini_{split}$ corresponding with these 1024 (k in formula) subsets. Compared to the curves in terms of prediction accuracy, now the saturating counter mechanism and the limitation of table sizes are eliminated.

The common point for all the curves represents the situation when no feature is used. Note that this point does not necessarily reach the maximum Gini-index 0.5, as it represents the purity of the complete data set. In our case, this point reaches 0.48 because there were slightly more taken branches.

Again, the best result is obtained for the (perfect) local history. While the individual bits of this local history all reach a very low Gini-index (Figure 6), the combination shows that additional bits only slightly improve its quality. This means that the bits in the perfect local history are not very complementary. The reason hereto is that many branches are strongly biased (i.e., having a specific usually direction) as shown in Figure 8.

The branch address feature in Figure 7 is the best (except for the local history) up to 10 bits. This exactly corresponds to the point where the bimodal predictor, which is an implementation based on the branch address feature, in Figure 5 reaches the highest accuracy at that hardware budget: a 10-bit use of the branch address corresponds to a predictor size of 256 bytes. Starting from 11 bits, the feature gshare-index achieves the lowest Gini-index, even better than the (perfect) local history at large feature lengths. This also corresponds to the highest prediction accuracy of a gshare predictor at large hardware budgets in Figure 5.

Using less than 4 bits in global history, branch address or gshare-index makes no sense, as the static feature target direction reaches a lower Gini-index and thus has better discriminative capacities. The other static features, requiring 3 bits each, also perform better than dynamic features of 3-bit. This also holds for the combination of both ending types for the successor basic blocks (6 bits) that reaches a Gini-index of 0.27 and perform better than most dynamic features at a 6-bit length. The previous conclusions suggest that dynamic features mainly profit from their long history lengths. In current branch predictor designs however, the required hardware budget grows exponentially with the length of the dynamic features. Alternative designs such as the perceptron predictor [9] are a solution for this exponential growth.

It is interesting to observe that the results presented in Figure 7 provide an explanation for the results in Figure 5 on the performance of real branch predictors. Indeed, in the region of hardware budgets where the branch address features attain the lowest Gini-index corresponds to the highest prediction accuracy of the bimodal branch predictor; likewise for the gshare-index feature and the gshare branch predictor. As such, we can conclude from these experiments that the Gini-index is a good indicator for studying branch prediction features. In future work we will try to combine various

branch features and evaluate its discriminating capacity using the Gini-index. As such, we hope that we will be capable of designing better branch predictors.

5. RELATED WORK

In [4], Calder *et al.* measured the effectiveness of static branch prediction features by constructing decision trees and training neural networks. Our work is more general because we are not limited to static features and we explore relationships between the discriminative capacity of a several features and the performance of existing prediction schemes in which these features are used.

Fern *et al.* propose in [10] a prediction mechanism that dynamically selects the most predictive features from a large feature set. Therefore they associate a so-called correlation counter to each feature. Upon observing a feature vector/target outcome pair, the feature counters are incremented or decremented if the corresponding feature agrees or disagrees with the target outcome. The most predictive feature at a certain moment is the feature with the largest counter magnitude. Thus in [10] temporal correlation is used, while we are looking for features that correlate highly with the branch direction during the complete run of the program.

6. CONCLUSION

In this paper we evaluated the Gini-index as a metric for studying features for branch prediction. We analyzed the discriminative power of dynamic features as they are used in existing branch predictor implementations. For small feature lenghts static features outperform dynamic prediction features. The longer the features, the better the discriminative power of e.g., global history and gshare-index. For the local history we found that using a few bits already achieve a high discriminative capacity, and that additional local history bits can not improve a lot. If we compare our analysis in terms of Gini-index with predictor implementations that use these features, we found that low values for the Gini-index correspond to highly accurate predictors. As such, we conclude that the Gini-index is a good indicator for studying branch prediction features.

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 Fellow of the Fund for Scientific Research—Flanders (Belgium) (F.W.O.—Vlaanderen).

REFERENCES

1. Neefs, H., De Bosschere, K., and Van Campenhout, J., “An Analytical Model for Performance Estimation of Modern Data-Flow Style Scheduling Microprocessors,” in *Proceedings of the 22nd Euromicro Conference: Short Contributions*, 1996, pp. 2–7.
2. Smith, J. E., “A Study of Branch Prediction Strategies,” in *Proceedings of the Eighth International Symposium on Computer Architecture*, 1981, pp. 135–148.
3. Ball, T., and Larus, J. R., Branch prediction for free, ACM SIGPLAN Notices, 28, 300–313 (1993).
4. Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., and Zorn, B., Evidence-based static branch prediction using machine learning, ACM Transactions on Programming Languages and Systems, 19, 188–222 (1997).
5. Burger, D., Austin, T. M., and Bennett, S., Evaluating future microprocessors: The SimpleScalar Tool Set, Tech. rep., Computer Sciences Department, University of Wisconsin-Madison (1996).
6. McFarling, S., Combining branch predictors, Tech. Rep. TN-36, Digital Western Research Laboratory (1993).
7. Kessler, R., McLellan, E., and Webb, D., “The Alpha 21264 microprocessor architecture,” in *Proceedings of International Conference on Computer Design*, 1998, pp. 90–105.
8. Breiman, L., Friedman, J., Olsen, R., and Stone, C., Classifi cation and regression trees (1984).
9. Jiménez, D. A., and Lin, C., “Dynamic Branch Prediction with Perceptrons,” in *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, 2001, pp. 197–206.
10. Fern, A., Givan, R., Falsafi , B., and Vijaykumar, T. N., Dynamic feature selection for hardware prediction, Tech. Rep. TR-ECE 00-12, School of Electrical and Computer Engineering, Purdue University (2000).