# Micro-Architecture Independent Branch Behavior Characterization

Sander De Pestel Stijn Eyerman Lieven Eeckhout

Department of Electronics and Information Systems, Ghent University, Belgium
{smdpeste,seyerman,leeckhou}@elis.UGent.be

***Abstract*—In this paper, we propose linear branch entropy, a new metric for characterizing branch behavior. The metric is independent of the configuration of a specific branch predictor, but it is highly correlated with the branch miss rate of any predictor. In particular, we show that there is a linear relationship between linear branch entropy and the branch miss rate. This means that the metric can be used to estimate branch miss rates without simulating a branch predictor by constructing a linear function between entropy and miss rate.**

**The resulting model is more accurate than previously proposed branch classification models, such as taken rate and transition rate. Furthermore, linear branch entropy can be used to analyze the branch behavior of applications, independent of specific branch predictor implementations, and the linear branch miss rate function enables comparing branch predictors on how well they perform on easy-to-predict versus hard-to-predict branches. As a case study, we find that the winner of the latest branch predictor competition performs worse on hard-to-predict branches, compared to the third runner-up; however, since the benchmark suite mainly consisted of easy branches, a predictor that performs well on easy-to-predict branches has a lower average miss rate.**

## I. Introduction

Branch prediction is and will remain an important performance contributor. Branch mispredictions disrupt the continuous flow of instructions in a processor, and although advanced branch predictors succeed in correctly predicting the vast majority of the branches, the impact of branch mispredictions on overall performance is non-negligible for many applications. Therefore, performance analysis modeling methods should carefully consider branch prediction effects.

Previous work has quantified the penalty of a branch misprediction [4], [5], i.e., how many cycles are lost when an incorrect prediction occurs. However, to the best of our knowledge, there is currently no technique to profile branch behavior in a branch predictor independent way. To be meaningful, a predictor-independent branch profile should have a good correlation with branch miss rates of specific branch predictors. This is not straightforward, because branch prediction accuracy depends both on particular branch patterns in the application and the specific structure of the predictor. Given the multitude of different branch predictors, each having different patterns that they can or cannot detect, finding a single profile that correlates with miss rates for all of them is challenging. We propose *linear branch entropy*, a metric that quantifies how regular the branch behavior of an application is. An entropy of 0 means that the branches are highly predictable, resulting in few branch mispredictions; on the other hand, an application with an entropy close to 1 will have a large number of branch mispredictions. We define *global entropy* as the entropy based on global branch history, and *local entropy* as the entropy based on local branch history. Furthermore, there is one entropy number per history length. All of these entropies can be measured through a single profiling run over the program.

Linear branch entropy can be used to characterize and classify applications depending on their branch behavior. Furthermore, because entropy correlates well with miss rate, we can construct a model for a specific branch predictor that translates entropy into miss rate. We find that a linear relationship between entropy and miss rate fits best and is intuitive to understand. We therefore use a training set of benchmarks, for which we know both the entropy and the miss rate, to fit a line and use that model to estimate the miss rates of other applications. In summary, we have one entropy profile per application and one miss rate versus entropy model per branch predictor (two parameters to define a linear relationship). Combined, the miss rates for all applications and all predictors can be estimated. The linear model itself can also be used to analyze and compare branch predictors (without the input of specific applications): the two parameters of the linear model (constant factor and slope) indicate how well a specific branch predictor predicts regular (low-entropy) and irregular (high entropy) branches.

We make the following novel contributions:

- We propose linear branch entropy, which correlates better to branch miss rates than the classic definition of binary entropy, because it more closely resembles the organization of a branch predictor.
- We show that linear branch entropy indeed has a better correlation: we find that there is a linear relationship between entropy and miss rate, and show that this relationship results in more accurate branch miss rate estimations than prior work (on average 38% lower error compared to the best prior work, i.e., a combination of taken and transition rate).
- We use linear branch entropy to classify benchmarks based on branch behavior, and show that we are able to obtain the same branch predictor miss rate ordering by evaluating only 5 representative benchmarks out of a set of 40 benchmarks.
- We compare the-top four branch predictors of the latest branch competition using the new branch predictor model, and show that the third runner-up is better

at predicting high-entropy branches than the winner. However, most of the evaluated benchmarks contain low-entropy branches, and the winning predictor performs better on these branches.

We begin by discussing prior work on branch classification and predictor analysis in Section II. We then introduce our entropy metric and discuss how to build the profiler and the branch predictor model in Sections III and IV. In Section V, we evaluate the correlation of linear branch entropy to branch miss rates of different predictors, and compare against prior work. We explore classifying applications based on branch entropy, and show how the branch predictor model can be used for analyzing and comparing branch predictors in Section VI. We end this paper with our conclusions and future work.

## II. Prior Work

In this section, we discuss the previously proposed taken rate and transition rate to classify branches, and discuss related work on using entropy to characterize branches. We also show the similarities between our way of measuring entropy and prior work on analyzing branch predictors as a Markov predictor. We begin by summarizing the basics about branch prediction.

### A. Branch Predictors

Branch predictors predict the outcome of a conditional branch (and also the branch target address, which is not in the scope of this paper). They do that by using information from the past, e.g., whether the branch has been previously taken or not. Because there can be correlations between the outcome of the current branch and the outcomes of previous branches, branch predictors often keep track of history information. These predictors are called two-level predictors [16], because they use history (first level) to index a pattern history table (PHT, second level). The first level can be global, containing the outcomes of all past branches, or local, recording the previous outcomes of each static branch instruction separately (per-address history). Indexing the second level can be done using history only (called global indexing) or combined with branch address bits (called per-address indexing). This leads to multiple different combinations, the most common are a GAg predictor (global history, global indexing), a GAp predictor (global history, per-address indexing), a PAg predictor (local history, global indexing) and a PAp predictor (local history, per-address indexing). A gshare predictor is a variant of a GAp predictor, which XORs global history and branch address bits, instead of concatenating them. A tournament or hybrid predictor combines multiple predictors, and uses a metapredictor to choose which predictor performs best for which branch instruction [11].

### B. Taken Rate

Taken rate is defined as the number of times a given branch is taken divided by the total number of times the branch is executed [1]. Branches with a high (close to 1) or low (close to 0) taken rate are easy to predict, because they are highly biased towards a particular direction (taken or not taken). A taken rate of around 0.5 typically indicates a branch that is difficult to predict, because its outcome varies between taken and not taken. Taken rate can detect branches that are easy to predict, but classifies some branches that are easy to predict with some history information (e.g., a periodic switch between taken and not taken) as difficult to predict (because its taken rate is close to 0.5).

### C. Transition Rate

To solve this problem, Haungs et al. [6] propose to measure transition rate to classify branches. Transition rate is defined as the number of times a given branch switches between taken and not-taken over the total number of times the branch is executed. A low or high transition rate indicates a highly predictable branch: a low transition rate means that the branch has a bias towards a certain direction, and therefore comprises both high and low taken rate branches. A high transition rate indicates a branch that switches frequently, and might therefore have a regular pattern that can be recognized by a predictor that keeps track of branch history.

The authors also show that a combination of taken and transition rate has a slightly better correlation with miss rate than each of the metrics separately. An important advantage of taken and transition rate is that they are very easy to measure. However, they cannot detect branches that have a regular but slightly more complex pattern (e.g., a repeating pattern of two times taken and one time not taken has a taken and transition rate of 67%, but can be accurately predicted with two bits of history).

### D. Branch Entropy

Yokota et al. [17] measure the entropy of branch outcomes using the standard entropy formula from information theory: $E = -\sum_i p(S_i) \log_2 p(S_i)$, where $S_i$ denotes all possible branch outcome patterns and $p(S_i)$ the probability for pattern $S_i$. They define local and global history entropy, as well as a predictor-dependent entropy. They show that entropy correlates well with miss rate and that inverting the entropy function to a binary probability (i.e., solving $E = -p \log_2 p - (1-p) \log_2 (1-p)$ to $p$) yields an upper bound for the hit rate of a predictor. We find that a linearized version of entropy leads to a simpler model to estimate branch miss rates, and we are able to estimate the actual miss rate of a specific predictor, instead of a lower bound.

### E. Branch Predictor as a Markov Predictor

Chen et al. [2] show that a branch predictor implements a simplified prediction-by-partial-matching algorithm, which is a set of Markov predictors of different orders. A Markov predictor of order $m$ predicts the outcome of the next branch as the most frequent outcome seen in the past after the same outcome history of the last $m$ branches. Our way of profiling is similar to building a Markov predictor: it collects the distribution of the outcome of a branch per history of $m$ previous branches. This distribution is the input for our entropy metric. We then reduce the history by one bit to calculate the entropy for $m-1$ history bits, similar to a partial matcher, which also reduces the order to find new patterns. However, we use this information to characterize branches and estimate miss rates, while [2] uses this insight to propose better branch predictors.

### F. Micro-Architecture Independent Characterization

Several researchers use micro-architecture independent metrics to characterize and analyze programs. Joshi et al. [10] use taken rate and forward branch taken rate to characterize

branch behavior, while Hoste and Eeckhout [7] use the miss rate of a theoretic prediction-by-partial-matching algorithm. Shao and Brooks [14] show that application profiling depends on the ISA, and present an ISA-independent profiler. They use the entropy model of Yokota et al. [17] to characterize branch behavior.

## III. Linear Branch Entropy

The general idea behind branch prediction is that there is correlation between branch outcomes. Depending on the outcome of previous branches or previous outcomes of the same branch, a particular branch can have a higher probability to be taken or not[1]. The better the correlation is between the outcome of the current branch and the history of previous outcomes, the better the branch is predicted. The accuracy of a branch predictor thus depends on how stable the outcome is for each history pattern. We use this insight to quantify the predictability of a branch: for each branch $i$ and history pattern $H$ (which could be local or global history), we record the number of taken and not-taken branches, denoted $n_1(i,H)$ and $n_0(i,H)$, respectively. We then define the probability for a taken branch, given a specific history pattern, as

$$p(i,H) = \frac{n_1(i,H)}{n_0(i,H) + n_1(i,H)}. \quad (1)$$

This is similar to the definition of taken rate [1]. The main difference is that we do not calculate a single taken rate per static branch instruction, but a taken probability for each possible history pattern of each static branch.

Now we have to transform this taken probabilities into a branch predictability metric. As discussed before, a taken probability of 0 (never taken) and 1 (always taken) are highly predictable. An often used metric in physics and information theory to denote the amount of disorder or the amount of information in a system is entropy. The classic definition of binary entropy (also denoted Shannon entropy) is

$$E(p) = -p \times \log_2(p) - (1-p) \times \log_2(1-p). \quad (2)$$

Entropy is 0 if $p = 0$ or $p = 1$, and 1 if $p = 0.5$. This definition is used by Yokota et al. [17]. They showed that the inverse of this function, i.e., reconstructing $p$ given $E(p)$, forms an upper bound for branch prediction accuracy.

We use a alternative definition of entropy, which we call linear entropy, defined as

$$E_L(p) = 2 \times \min(p, 1-p). \quad (3)$$

This equation also equals 0 in $p = 0$ and $p = 1$, and 1 in $p = 0.5$ (hence the factor of 2). It is easy to calculate, and we find that miss rate is approximately a linear function of this entropy, resulting in a simple linear model for the branch predictor miss rate. The intuition behind this is that a branch predictor also performs a simple function: it selects the outcome with the highest occurrence in the recent past, by making use of (2-bit) saturating counters. Therefore, it is easy to see that miss rate is proportional to the probability of the least frequent outcome, hence the minimum of $p$ and $1-p$. On the other hand, Shannon entropy relates to information theory, and reflects the amount of bits needed to represent a certain amount of information, which is conceptually more complex than the functioning of a branch predictor. A downside of the linear entropy function is that it is not differentiable in $p = 0.5$, which could be a problem in mathematical optimization, but which is not a problem for our model.

After calculating the entropy of each branch and for each history pattern using Equations 1 and 3, we average the entropies over all branches and all history patterns. Let $n(i,H)$ be the number of occurrences of branch $i$ with history pattern $H$ (i.e., $n(i,H) = n_0(i,H) + n_1(i,H)$), and $N$ the total number of dynamically executed branches (i.e., $N = \sum_i \sum_H n(i,H)$). The average branch entropy of an application equals then

$$E = \frac{1}{N} \sum_i \sum_H n(i,H) \times E_L(p(i,H)). \quad (4)$$

This averaging method is sound, because branch entropy is linear in $p$. This is another advantage over Shannon entropy, which cannot be easily averaged due to the logarithms, and which also explains the detour Yokota et al. [17] had to take to find an upper bound for the hit rate (calculating entropy over all patterns, and then solving this entropy for a binary probability).

## IV. Branch Profiler and Predictor Model

This paper has three main contributions: (1) it proposes a branch behavior profile that is application-dependent but independent of a specific branch predictor, (2) it builds a model that relates branch entropy to branch miss rate for a specific branch predictor organization, and (3) it proposes a fast branch predictor design space exploration tool by combining the application-dependent profile and the predictor-dependent model. This section elaborates on each of these three contributions. For the ease of discussion, we begin with explaining how the design exploration tool (third contribution) is organized, because this provides a clear overview on how the contributions relate to each other.

### A. Overview

Figure 1 gives a high-level overview of the proposed framework. The calculation of the branch entropy is illustrated at the bottom: the application is executed on a profiler, resulting in a (set of) entropy number(s). The profiler is discussed in Sections IV-B and IV-C. The top part of the figure shows the predictor-dependent model. The dashed box shows the steps that need to be done once for each predictor to construct the model that relates branch entropy to miss rate. We first need to select some benchmarks for training. For this training set, we measure both the entropy and branch predictor accuracy. The branch predictor miss rate is measured using a simulator. A simulation for every different type of branch predictor that we want to model is required, hence the multiple simulations in Figure 1.

The entropy and branch miss rates for the training set are then used to construct a model for each of the branch predictors. There is one model per branch predictor type, which takes as an input the branch entropy of an application and estimates branch miss rates. Conceptually, a branch predictor model is tied to a specific branch predictor only, and it can be used to estimate miss rates for all applications.

Once the models are built, the miss rates for a new application can be estimated by profiling that application once,

[1]We only consider branch predictors that use the history of outcomes of previous branches to predict future branches. The methodology can also be applied to branch predictors that have other inputs (e.g., call stack depth), by defining corresponding patterns.

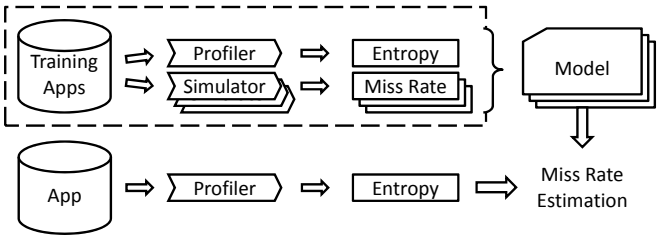



Fig. 1: Overview of the branch predictor model. The dashed box represents the training phase, which has to be done once for each branch predictor type.

obtaining its branch entropy number. This entropy number is then used to estimate the miss rates for all modeled branch predictors instantaneously, by just evaluating a linear function. Introducing a new type of branch predictor requires simulating this branch predictor for the training benchmarks, and constructing a new model. Afterwards, this model can be used for all applications, for which we can reuse their already measured entropy numbers.

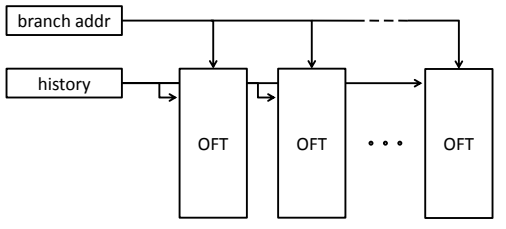
Building a branch predictor model consists of two main steps: we have to profile an application to measure its branch entropy and we have to construct a model using entropies and miss rates from the training set. The profiler itself also consists of two phases: recording branch histories and outcomes, and calculating entropy. The next sections discuss each of these steps.

### B. Recording Branch Behavior

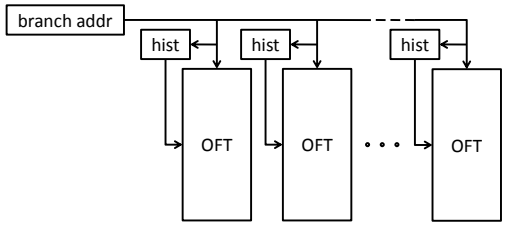
First, we need a profiling run to determine the outcome (taken or not) for every branch and record its history. We do this by maintaining an outcome frequency table (OFT) that is indexed by a history pattern, and that has two counters per entry: the number of not-taken branches $n_0$ and taken branches $n_1$ when this history pattern is encountered.

We keep two versions of the tables, one indexed by global history (outcome of all previous branches), and one indexed by local history (outcome of the previous occurrences of the same static branch instruction), see Figures 2a and 2b. We make this distinction because current branch predictors use either local or global history, or a combination of both, and it is impossible to reconstruct global history from local history information or vice-versa without detailed information on how branches interleave. Furthermore, we keep track of one OFT per static branch instruction, to be able to model branch predictors that use branch address bits to index the second level of the branch predictor. The history pointers keep track of a history of $m$ bits, which is the largest history we want to model. From this, we can extract the entropy for all smaller history sizes, as we will explain in the next section.

Although this seems a lot of data, the tables are actually very sparsely occupied. Each branch has only a limited number of actually encountered history patterns, so only a fraction of the $2^m$ entries is used. Therefore, we use an associative array instead of a full table to record the information. In all of our profiling runs, our tool uses at most 10 MB of memory for $m = 20$ (compared to 52 GB if we had allocated the full table).




(a) Global History




(b) Local History

Fig. 2: The profiler records branch outcomes and history of every unique branch, both for global and local history.

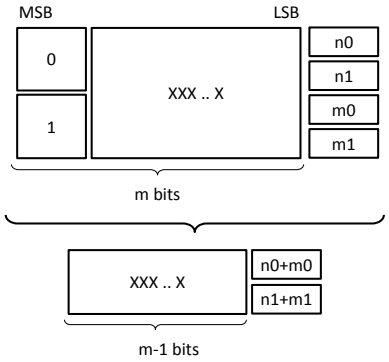



Fig. 3: Reducing the number of history bits.

### C. Calculating Branch Entropy

After recording the information in the tables, we calculate branch entropy. Formula 3 is used to calculate the entropy of every OFT entry of every unique branch. Then we take a weighted average over all entropy values using Formula 4. This leads to two numbers, representing the branch entropy for global and local history and $m$ history bits.

To get the branch entropy for a smaller number of history bits, we collapse the tables to reduce the history size. To get the table for $m - 1$ history bits, the oldest history bit, represented by the most significant bit of the index, needs to be removed. After removing this bit, the two entries with the same index need to be merged to one entry. This is done by adding the taken and not-taken counters of the different entries together (see Figure 3). The table now represents the OFT for $m - 1$ history bits and can be used to calculate the new entropy number. This process can be repeated until the number of history bits is 0 (e.g., to model a simple bimodal predictor). The result of this step is two sets of entropy numbers, one for the local history and one for the global history, containing entropy numbers for history sizes from 0 to $m$.

These entropy numbers reflect two basic branch predictor organizations: a GAp predictor, which uses global history and

branch address bits to index its PHT, and a PAp predictor, which uses local history and branch address bits. Tournament predictors [11] combine multiple predictors to obtain better prediction accuracy. They aim at selecting the optimal history (local or global) per individual branch. Therefore, next to global and local history entropy, we also calculate *tournament entropy*: we calculate global and local entropy per branch, take the minimum of the two, and average that minimum over all branches. This adds a third set of entropies to the application branch profile. Note that this does not change the way the tables are recorded, this only affects the entropy calculation step.

A PHT entry in a branch predictor contains meaningful information only after a few updates to that entry. So, even if a branch has a stable behavior, the first predictions can be wrong, depending on the initial value of the saturating counter. To account for this warming effect, we assign an entropy of 1 to the first access, which boils down to modeling a probability of 50% to predict the first branch correctly. This only has a noticeable impact on the final entropy number if there are many different branches and branch histories that occur only a few times.

A branch predictor can also suffer from aliasing, because only a few bits of the branch address are used. That means that different branches can map to the same entry, which can lead to positive (all branches have the same outcome) or negative (the branches have different outcomes) interference. To model this, we gradually reduce the number of address bits, and add all tables that have the same truncated address element by element, similar to collapsing the tables to model a smaller number of history bits. In the extreme, we use no address bits, meaning that all tables are added together. This models a predictor that uses no address bits, such as a GAg predictor. Modeling aliasing into the entropy adds multiple extra sets of entropy numbers, one for each setting of the number of address bits used to index the branch predictor PHT. Modeling aliasing also affects only the entropy calculation step, and has no impact on the way the tables are recorded. We will show the necessity of modeling branch aliasing in the entropy in Section V-E.

The resulting branch entropy profile for an application consists of multiple sets of entropy numbers: a set for local, one for global, and one for tournament history. Each set has an entropy number for each considered history length and number of address bits. Recording all of these values is necessary, as each application behaves differently when the history length and number of address bits is changed. Note that all of these numbers are calculated using a single profiling step. One can also choose not to model some effects, such as not taking into account warmup for predictors with adaptable history length (that combine the short warmup time for small histories and the accuracy of long histories, see SectionVI-A), or not modeling aliasing for predictors that have aliasing-reducing hashing functions.

### *D. Building the Model*

The last component is the model that transforms entropies into miss rate. Because different branch predictors will have a different miss rate for the same application, each type of branch predictor will have its own model. We build this model using the entropy numbers and the branch predictor miss rates from the benchmarks in the training set. Selecting the training set is straightforward: because we have to measure the branch entropy profile for each application anyway, we can select a subset of applications that covers the full range of entropy values, leaving out applications that have a similar entropy to that of already selected applications.

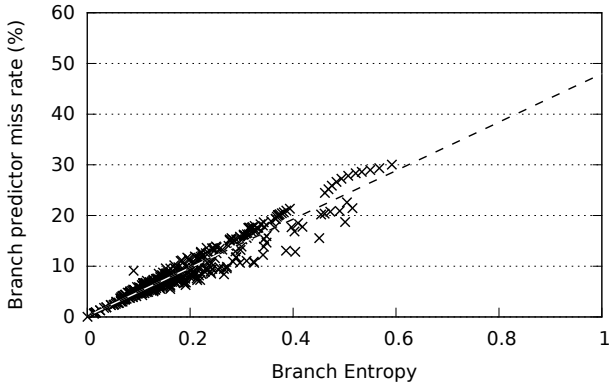


Fig. 4: Training input with fitted line.

We first select the set of entropy numbers that is conceptually close to the predictor we want to model: local, global or tournament history. For common predictors (e.g., bimodal predictor, GAg, GAp, PAp, etc.), this is straightforward. For other, more complex predictors (e.g., perceptron-based predictors), this may be not immediately clear. For these predictors, we can build a model for each set of entropy numbers, and select the one that has the smallest error on the training set. Building the model only takes a fraction of time once the entropies and miss rates are known.

Because we have an entropy number for each number of history bits, we do not need to construct a new model for each history length of a particular type of branch predictor. Instead, we can fit the entropies and miss rates for all history sizes and for all benchmarks in the training set to a single model. This reduces the number of models, and also increases the number of points to fit the model (or it reduces the number of simulations required to produce enough training data). Furthermore, because we incorporate branch address aliasing in the entropy calculation, we can also incorporate the entropies and miss rates for a varying number of address bits used for indexing into a single model.

We find that a linear relationship between entropy and miss rate fits best. This can be explained intuitively by our choice of using a linear entropy function: a branch predictor usually predicts the outcome most encountered in the same context in the past, so the fraction of the least encountered outcome correlates well with the miss rate. We use the following equation to calculate the miss rate $M$ from the entropy $E$:

$$M(E) = \alpha + \beta \times E. \tag{5}$$

Parameters $\alpha$ and $\beta$ are determined using a least-squares fit.

Figure 4 shows the result of such a fit. Every point represents the branch predictor miss rate (y-axis) and the corresponding entropy value (x-axis). There are 40 benchmarks and 20 different history lenghts, resulting in 800 points. The fitted model is indicated by the dashed line. It is clear that a linear model is appropriate for this data, and that the fit is relatively good.

## V. Results

### A. Experimental Setup

To construct and evaluate our technique, we use the framework provided by the 2011 Championship Branch Predictor competition (CBP)[2]. This framework consists of a set of 40 benchmarks from different domains (client, server, work station, multimedia and integer) and a simulation environment in which we can easily implement different branch predictors. We also implemented our profiler to record the tables for calculating entropy in this tool.

Recording the OFTs takes approximately the same time as simulating a branch predictor, so measuring entropy has the same overhead as one branch predictor simulation, but enables predicting miss rates for all modeled branch predictors. Using the model as a replacement for simulation therefore leads to a speedup equal to the number of evaluated branch predictor configurations. Note that the training phase needs to be done only once per predictor, on a limited set of applications and history lengths (e.g., 20 points per predictor to estimate the two parameters of the model should be sufficient if a good entropy coverage is selected).

We also considered the most recent 2014 Championship Branch Predictor competition, but found that the provided benchmarks have a lower entropy range than those of the 2011 competition (the benchmarks of the 2014 competition are within $[0, 0.21]$ entropy, while those of 2011 are within $[0, 0.39]$). Because our model is constructed using regression, a larger range will result in a more general model. Therefore, we choose to use the 2011 benchmarks to evaluate the model (we could not simulate the 2014 benchmarks on the 2011 infrastructure and vice versa, preventing the evaluation of superset consisting of the 2011 and 2014 benchmarks). Nevertheless, we ported the predictor code of the 2014 winning predictors to the 2011 framework to evaluate our model on the latest state-of-the art predictors (see Section VI).

### B. Model Accuracy

We first show that linear branch entropy correlates well with branch miss rates, by evaluating the accuracy of the branch predictor model. Because the model is a linear function, model accuracy is a good indicator for correlation: if the model is accurate, branch entropy correlates well with miss rate, and vice versa. We evaluate the accuracy of the model for a few common two-level predictors: a GAg predictor, a GAp predictor, a PAp predictor and a gshare predictor. Furthermore, we also evaluate a tournament predictor, consisting of a GAp and a PAp predictor and a metapredictor (indexed with address bits only) to choose between the two predictors.

We evaluate the model using leave-one-out cross-validation: we train the model on all but one benchmark, and evaluate the accuracy for the left-out benchmark, and repeat this for all benchmarks as the left-out benchmark. We report the average difference in MPKI (misses per 1,000 instructions) between the prediction and the simulation. Using MPKI instead of miss rate avoids inflating numbers when there are few branches. Furthermore, MPKI is proportional to the branch miss CPI penalty of an application [3].

We train the model for each of the predictors using simulation results for all training benchmarks and for history sizes between 0 and 20 bits. Figure 4 shows the individual points (40 benchmarks $\times$ 20 history settings) and the fitted model for the GAg predictor. For GAg, gshare and GAp, the global history entropy is used to fit the model to the miss rates; for PAp, we use the local history entropy. For the tournament predictor, we use the tournament entropy. For GAg, the number of address bits for calculating entropy is set to 0, i.e., all tables for all branches are added together. For gshare, we found that setting the number of address bits for calculating entropy equal to the number of address bits used in the indexing leads to relatively large errors. By XOR-ing address bits and history, we loose some of the information of the address bits. We found that fitting the miss rates of gshare to the global entropy with no address bits (as for GAg) provides the best results. This can be explained by the fact that we use the same amount of history bits as the GAg to index the PHT, and that the XOR with the address bits partly solves the problem of GAg that the same history for different branches is mapped to the same entry (history aliasing). This aliasing reduction effect is now visible in a low (even negative) parameter $\alpha$ of the gshare model, which is a measure for aliasing, see also the next section. Table I shows the fitted parameters $\alpha$ and $\beta$ for each of the predictors (in % miss rate). Note that when $\alpha$ is negative, the model can result in negative miss rates for low entropy numbers. Obviously, this makes no sense, so we estimate a zero miss rate if the model predicts a negative miss rate.

| | $\alpha$ | $\beta$ |
|---|---|---|
| GAg | 0.168 | 47.814 |
| GAp | 0.421 | 44.801 |
| PAp | 2.460 | 49.211 |
| gshare | -1.235 | 47.107 |
| Tournament | 0.335 | 53.952 |

TABLE I: Model parameters for the common branch predictors (in % miss rate).

Figure 5 shows the prediction error as a box-and-whiskers plot[3] for the five branch predictors. These numbers are for one specific configuration (i.e., number of history and address bits) for each predictor. These configurations have approximately the same hardware cost ($2^{12}$ entries in the second level), in order to have miss rates and errors that are in the same range. The error of the tournament predictor is the smallest with an average absolute error of 1.36 MPKI. For all modeled predictors, the average absolute error is around 2.1 MPKI, with a maximum error of 8.64 MPKI (for gshare). The average MPKI for all predictors is 10.8, which means that the model has a relative error of less than 20%, and the errors are both negative and positive, as the error boxes show. The highest average errors are for PAp (2.38 MPKI) and gshare (2.58 MPKI). This is because these predictors suffer from aliasing effects, other than pure branch address aliasing, that are not modeled in our entropy calculation. The PAp predictor has aliasing in its history table: we use 10 bits to index the history table, which means that it can happen that different branches update the same history, which pollutes this history. This effect is not modeled in our entropy calculation. Modeling this would require separate tables for every setting of the number of

[2]Available at http://www.jilp.org/jwac-2/

[3]The box covers the second and third quartile, with a line at the median. The whiskers cover all points within 1.5 interquartile distance outside of the box, and the crosses are the outliers. The circels represent the average of the absolute value of the error.

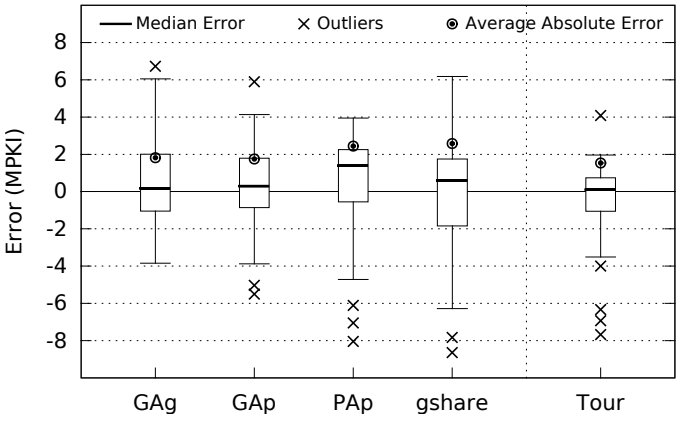



Fig. 5: Prediction error in MPKI for the common predictors.

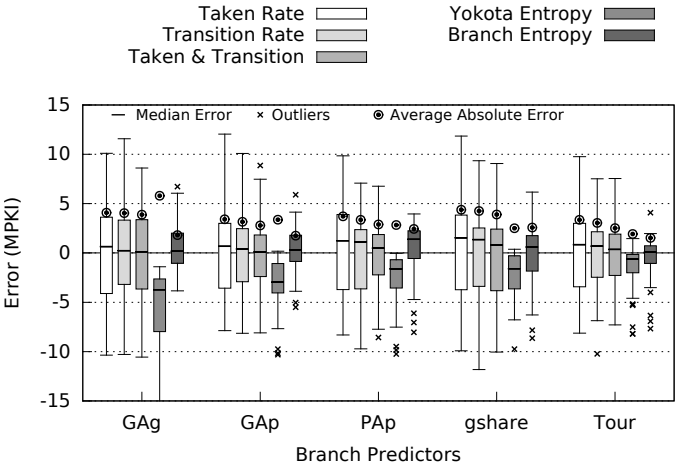



Fig. 6: Model error in MPKI for taken rate, transition rate, the two combined, Yokota lower bound, and branch entropy.

bits used to index the branch history table, which would be too much overhead. The gshare predictor XORs history and address bits, which also leads to unpredictable aliasing effects (branches with a different history and instruction address can map to the same entry).

### C. Meaning of the Model Parameters

The branch predictor model has two parameters, $\alpha$ and $\beta$. $\alpha$ is the miss rate for an entropy of zero, and determines the miss rate for low-entropy branches, while $\beta$ is the slope of the line, and therefore is an indicator for the miss rate for high-entropy branches. These numbers allow for comparing branch predictors. For example, the GAg predictor is good at predicting low-entropy branches (low $\alpha$, see Table I), but has difficulties with high-entropy branches (high $\beta$), while the GAp predicts difficult branches better (low $\beta$), but has a higher miss rate for low-entropy branches (high $\alpha$), due to warmup effects.

Another explanation for the $\alpha$ parameter is that it quantifies the impact of the intrinsic miss rate of the predictor due to aliasing: $\alpha$ is the offset that is added to all miss rates. For example, in Table I, we can see that PAp has the largest $\alpha$ of all predictors. On the other hand, for some predictors, $\alpha$ is negative, e.g., for gshare. In this case, we estimate a zero miss rate for low-entropy branches. These predictors almost perfectly predict low-entropy branches, for example because they solve some aliasing issues that are not modeled in the entropy model, as discussed in the previous section.

### D. Comparison to Prior Work

Prior work proposed taken and transition rate for classifying branches (see Section II). Although they do not specifically target estimating branch miss rates, they show that taken and transition rate correlate well with miss rate. Therefore, we also build models that use taken rate, transition rate and a combination of both. The models are built by dividing taken and transition rate numbers into bins, and using the average miss rate in a bin as the branch miss rate estimation (requiring a similar training phase as our technique). For the combination of taken and transition rate, we construct two-dimensional bins, combining the bins from taken and transition rate (for example, a bin contains all branches that have a taken rate between 0.1 and 0.2, *and* a transition rate between 0.6 and 0.7; this is similar to what is described in [6]).

Figure 6 shows the resulting error box plots for our example predictors. The figure shows that the combination of taken and transition rate is indeed better than taken or transition rate separately. It is also clear that our branch entropy method proposed in this work outperforms all three other methods. For the GAg predictor, the improvement is the highest: the average absolute error drops by more than $2\times$ to 1.82 MPKI. For the other predictors the average absolute error drops from an average of 3.2 MPKI to 2 MPKI (an 38% reduction).

Other prior work by Yokota et al. [17] shows that branch entropy can provide an upper bound on the hit rate (or a lower bound on the miss rate), by reconstructing the $p$ (binary probability) from the entropy number. Translated to our model, this means that only entropy is used, and there are no different models per branch predictor type. Figure 6 also shows the error when we use only entropy (i.e., we divide linear entropy by 2 to reconstruct the $p$), and no specific fitted model per predictor. It shows that all errors are negative, i.e., the miss rate is always underestimated, which is consistent with the notion of a lower bound. The average absolute error for this method is 3.71 MPKI, which is higher than our model and than the model using the combination of taken and transition rate.

Figure 7 shows the average MPKI estimation error for different sizes of the predictors (using different history sizes and numbers of address bits used to index the PHT), both for our model and the model using the combination of taken and transition rate (which is the most accurate of all prior proposals). As a reference, the average MPKI of the predictors is also shown. This graph shows that our model is more accurate than taken and transition rate across all sizes, and that the relative error remains approximately the same for small and large predictors.

### E. Modeling Aliasing

As described in Section IV-C, we model branch address aliasing in our entropy model. This has some impact on the complexity of calculating entropy, because we have to look for branch addresses that, when truncated, map to the same entry, and add the tables element-wise. Furthermore, it makes the entropy profile of an application bigger: instead of having $m$ entropy numbers for local, global and tournament history

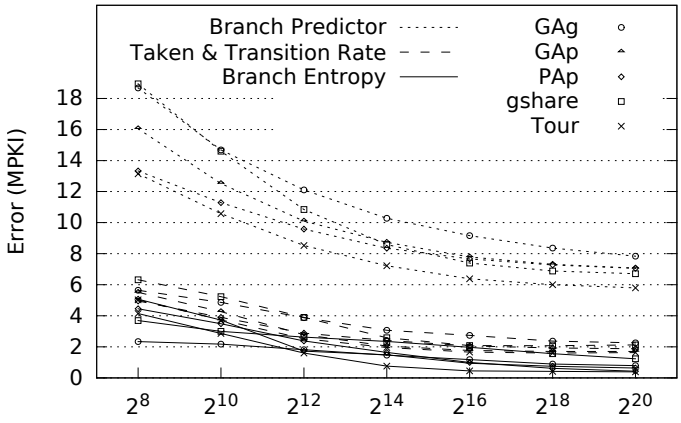



Fig. 7: Average absolute MPKI error for different predictor sizes (the X-axis shows the number of entries in the second-level PHT).

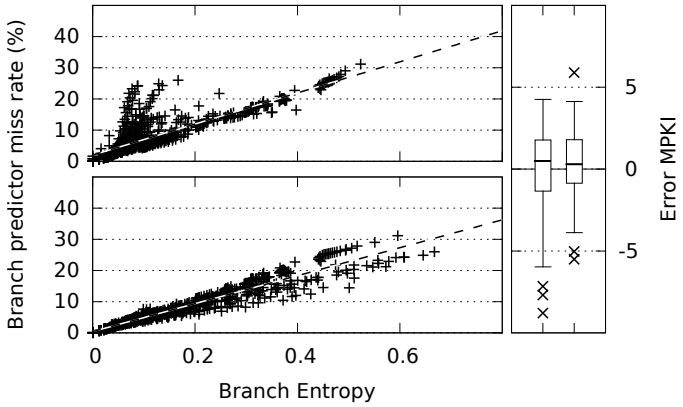



Fig. 8: GAp model without taken into account aliasing (top graph) and with accounting aliasing (bottom graph). The error box on the left is without modeling aliasing, the right one includes aliasing.

(with $m$ the number of history bit settings), we now have $m \times a$ entropy numbers, with $a$ the number of address bit settings. However, assuming 20 history bit and 20 address bit settings – which is already a huge design space – we need $3 \times 400$ entropy numbers in the profile. Assuming a 4-byte fixed point format (because entropy is always between 0 and 1), this results in a profile of only 4.7 KB. Furthermore, for most applications, the entropy does not change much once a certain number of history and address bits is reached, meaning that the profile can be further reduced or easily compressed.

Figure 8 shows the impact of (not) modeling aliasing for the GAp predictor. The top figure shows the model fit where no aliasing is incorporated in the entropy profile; the bottom figure includes modeling aliasing. It is clear that the fit is much better if we include aliasing. The average absolute MPKI error decreases from 2.38 to 1.73.

However, we should note that modeling aliasing has no big impact on the estimation accuracy for the PAp predictor, compared to not modeling aliasing. This is because the PAp predictor mainly suffers from aliasing in the branch history table, as discussed before. Simulations with an (unrealistic) perfect branch history table (one entry for each unique branch) revealed that the estimation of the miss rate is now much more accurate for PAp, and including aliasing in the PHTs improves the model here too.

## VI. Applications

Now that we have shown that our model to estimate branch miss rates using a predictor-independent profile is accurate, the most straightforward way to use it is to estimate miss rates for several different branch predictors to perform a design space exploration. In the results section, we have already shown that our model is accurate across a large design space. However, because we have a separate application profile (entropy) and branch predictor model (the linear model), we can also use the model to compare branch predictors, which we will do in the next section. We also show how entropy can be used to classify benchmarks and select a representative subset of benchmarks for branch predictor studies.

### A. Comparing State-of-the-Art Predictors

As discussed in Section V-C, the parameters $\alpha$ and $\beta$ of the linear model can be used to compare branch predictors. As a case study, we now compare the top four branch predictors from the 2014 Championship Branch Predictor competition in the 4KB hardware budget category. These are, ranked on misses per thousand instructions (MPKI):

1) Seznec: TAGE-SC-L [13] (5.29 MPKI)
2) Otiv: H-Pattern [12] (5.47 MPKI)
3) Ishii: GL-TAGE [8] (5.58 MPKI)
4) Jiménez: Strided Sampling HPP [9] (5.81 MPKI)

Note that the MPKI values and the ordering slightly differs from the results presented at the CBP-4 workshop, because we evaluated these predictors on the 2011 CBP benchmarks. We will discuss the results using the 2014 benchmarks at the end of this section.

First, we need to select the entropy numbers that should be used to model the miss rate of these predictors. Since they all claim to be able to handle large branch histories, we use 25 history bits, which is the largest history we used for measuring entropy[4]. Furthermore, we do not model branch address aliasing and warmup effects in calculating the entropy, assuming that these advanced predictors are able to eliminate these effects (e.g., by keeping multiple tables with different hashing and history lengths).

We also conjecture that tournament entropy should fit best, because all four use both global and local history. We checked this claim by constructing models using global history entropy, local history entropy and tournament entropy, and noticed that tournament entropy indeed leads to the lowest error on the training set. The resulting average MPKI error (again using leave-one-out cross-validation) is 0.91 for Seznec's predictor, 1.14 for Otiv's predictor, 1.26 for Ishii's predictor and 1.39 for Jiménez' predictor (compared to an average MPKI of 5.54 for all four predictors).

Table II shows the parameters of the linear model for the four predictors. Looking at the table we can see that Seznec has the lowest $\alpha$, but also the second highest $\beta$. For Jiménez, it is the other way around: it has the highest $\alpha$, but the lowest $\beta$. Figure 9 shows the models for the four predictors.

[4]We see no significant decrease in entropy when we increase the history beyond 25 bits.

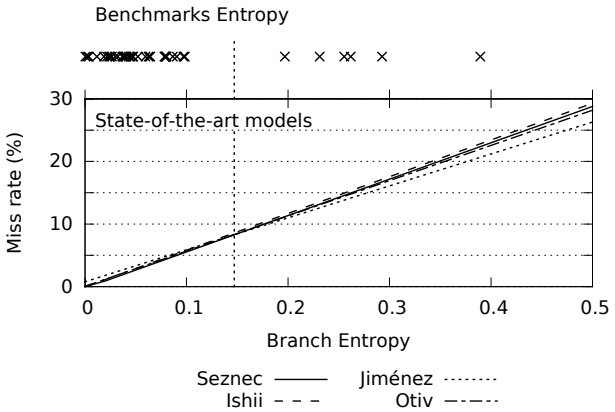



Fig. 9: Models for the state-of-the art predictors of the 2014 CBP championship. The entropies of all benchmarks are shown at the top.

| | $\alpha$ | $\beta$ |
|---|---|---|
| Seznec | -0.251 | 58.039 |
| Otiv | 0.063 | 56.238 |
| Ishii | -0.035 | 58.629 |
| Jiménez | 0.782 | 51.016 |

TABLE II: Model parameters for the state-of-the-art branch predictors (in %).

Again, we assume a zero miss rate if the model estimates a negative miss rate. The figure shows that Seznec's miss rate is lower than that of Jiménez if the entropy is lower than 0.147, but for larger entropies, Jiménez performs better, with a 2.47% lower miss rate if the entropy equals 0.5. The similarity between the models of the predictors of Seznec, Otiv and Ishii can be explained by the fact that they all extend upon the TAGE predictor, while Jiménez proposes a perceptron-based predictor.

| | Entropy | Seznec | Otiv | Ishii | Jiménez |
|---|---|---|---|---|---|
| INT04 | 0.0022 | 0.09 | 0.10 | 0.10 | 0.63 |
| INT06 | 0.0985 | 4.90 | 4.86 | 4.99 | 4.80 |
| INT01 | 0.196 | 10.19 | 10.21 | 10.29 | 9.89 |
| INT02 | 0.258 | 12.85 | 12.80 | 12.78 | 12.08 |
| WS04 | 0.389 | 24.45 | 23.87 | 24.99 | 22.57 |

TABLE III: Entropy and simulated miss rate (%) for the top 4 CBP predictors

To ensure that this behavior is real and not an anomaly of our model, we show entropies and simulated miss rates for some of the benchmarks in Table III. We can see that for the benchmark INT04, which has a very low entropy, the Seznec predictor outperforms the Jiménez predictor. For benchmarks close to the cross-over point (INT06 and INT01), the miss rates for all predictors are very similar. For the high-entropy benchmark WS04, the Jiménez predictor has a 1.9% lower miss rate than the Seznec predictor. Of all benchmarks that have an entropy higher than the cross-over point at 0.147 entropy, the average miss rate of Jiménez is almost 1% lower than the miss rate of Seznec.

By looking at the entropy distribution of all benchmarks, shown on top of Figure 9, it is clear why Seznec's predictor performs best on average: only 6 out of 40 benchmarks have an entropy that is higher than 0.147 (the cross-over point is indicated by the dashed line). The average miss rate is therefore mostly determined by low-entropy benchmarks, for which Seznec's predictor performs best. This is further exacerbated by the choice of benchmarks in the 2014 competition: only 3 out of 40 have an entropy higher than 0.147, with a maximum entropy of 0.21, which is only slightly higher than the cross-over point. Our branch entropy metric can be used by branch competition organizers to select a more diverse and balanced set of benchmarks.

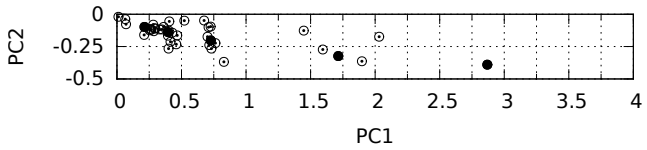



Fig. 10: Result of PCA analysis and clustering for the 40 benchmarks. Cluster representatives are highlighted.

*B. Choosing a Representative Subset of Benchmarks for Branch Predictor Studies*

In Section V, we show that our linear branch entropy metric correlates well with branch miss rate for a large range of predictors. This means that benchmarks with similar entropy also have similar miss rate, and just produce redundant results in a study where branch predictors are compared. By only simulating benchmarks that have different entropy, and therefore different branch behavior and different branch miss rates, we can reduce the number of benchmarks that need to be evaluated.

To validate this, we cluster the 40 benchmarks in our setup based on entropy. Because we have multiple entropy numbers per benchmark (local, global, and tournament, and for different history sizes), we first perform principal component analysis (PCA) to reduce the number of dimensions. We find that 2 principal components (PC) already account for 98.3% of the variance, so we selected 2 dimensions. The result of PCA is illustrated in Figure 10. The first principal component (PC1) basically weight each entropy equally, and is therefore proportional to the average entropy across all history sizes. PC2 gives a negative weight to small-history entropies and a positive weight to large-history entropies: a PC2 close to 0 means that large and small-history entropy is approximately equal, while a negative PC2 means that large-history entropy is lower than small-history entropy, so these benchmarks see more benefit when history is increased. The PCA makes no particular difference between local, global and tournament entropy in its first two components.

We then cluster the benchmarks using these two principal components, by performing K-means clustering. We select an optimal number of clusters using BIC (Bayesian information criterion), which results in 5 clusters. For these 5 clusters, we select the benchmark that is closest to the cluster center as the representative benchmark; these are highlighted in Figure 10. We then simulate these 5 benchmarks on a branch predictor simulator, and weight the miss rates with the number of benchmarks in its respective cluster. This number is then compared to the average miss rate over the whole set of benchmarks. Table IV shows the average miss rate over all benchmarks and the weighted average miss rate of the 5 representative

| Predictor | Miss rate | Cluster average using entropy | Cluster average using taken/trans. rate |
|---|---|---|---|
| GAp | 14.69 | 14.62 | 15.71 |
| PAp | 10.26 | 9.58 | 11.04 |
| GAg | 8.54 | 8.29 | 9.82 |
| gshare | 7.12 | 7.56 | 8.27 |
| Jiménez | 4.95 | 5.46 | 5.57 |
| Ishii | 4.76 | 4.94 | 5.46 |
| Otiv | 4.66 | 4.93 | 5.54 |
| Seznec | 4.49 | 4.50 | 5.59 |

TABLE IV: Average miss rates (in %) over all benchmarks and average miss rate using entropy-based clustering and taken/transition rate based clustering. Predictors are ordered in decreasing average miss rate.

benchmarks for different predictors. It also shows the average miss rate for a similar method using taken and transition rate as an input to the clustering mechanism (also for 5 clusters). For clustering using entropy, the difference between the actual and estimated miss rate is 4.5%, compared to 14.5% when using taken and transition rate. Furthermore, the clustering using entropy preserves the ordering of the branch competition top four, while using taken and transition rate classifies the Seznec predictor as the worst.

## VII. Conclusions and Future work

In this paper, we propose linear branch entropy, a branch predictor independent characterization of the branch behavior of an application. Linear branch entropy correlates well with miss rates, which enables building a simple linear model that estimates branch miss rates for multiple branch predictors using a single branch entropy profile of an application. The model is more accurate than previously proposed branch classification methods, and allows for exploring a large design space using a single profiling run, as opposed to performing multiple branch predictor simulations.

We show how branch entropy can be used to classify and select benchmarks. Furthermore, the branch predictor model enables comparing branch predictors. In particular, we analyze the top-four predictors of the latest branch predictor competition, and found that they perform differently on easy-to-predict and hard-to-predict branches.

In our follow-on work we have incorporated the model into a comprehensive and completely micro-architecture independent processor performance model [15]. The miss rates are used to estimate the branch miss component for estimating the total cycle count, in the interval model [4]. Results show that estimating branch miss rates with our model has no considerable negative impact on the full model's accuracy, compared to simulating the branch predictor, and it significantly reduces the profiling overhead by using a single profile run.

As future work, we will use the knowledge and insight gained by the model to optimize branch behavior at compile time , e.g., by if-converting difficult-to-predict branches.

## Acknowledgments

We thank the reviewers for their constructive and insightful feedback. Sander De Pestel is supported through a doctoral fellowship by the Agency for Innovation by Science and Technology in Flanders (IWT). Stijn Eyerman is supported through a postdoctoral fellowship by the Research Foundation – Flanders (FWO). Additional support is provided by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant Agreement no. 259295, as well as by the European Commission under the Seventh Framework Programme, Grant Agreement no. 610490.

## References

[1] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th annual international symposium on Microarchitecture (MICRO)*, pages 22–31, 1994.

[2] I. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 128–137, Oct. 1996.

[3] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *Proceedings of The Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184, Oct. 2006.

[4] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):42–53, May 2009.

[5] S. Eyerman, J. E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 48–58, Mar. 2006.

[6] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: a new metric for improved branch classification analysis. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 241–250, 2000.

[7] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[8] Y. Ishii. Global-local combined branch history: The alternative way to improve TAGE branch predictor. In *JWAC-4: Championship Branch Prediction*. JILP, June 2014.

[9] D. Jiménez. Strided sampling hashed perceptron predictor. In *JWAC-4: Championship Branch Prediction*. JILP, June 2014.

[10] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.

[11] S. McFarling. Combining branch predictors. Technical Report WRL TN-36, Digital Western Research Laboratory, June 1993.

[12] S. Otiv, K. Garikipati, M. Patnaik, and V. Kamakoti. H-pattern: A hybrid pattern based dynamic branch predictor with performance based adaptation. In *JWAC-4: Championship Branch Prediction*. JILP, June 2014.

[13] A. Seznec. TAGE-SC-L branch predictors. In *JWAC-4: Championship Branch Prediction*. JILP, June 2014.

[14] Y. S. Shao and D. Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255, 2013.

[15] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, L. Eeckhout, E. Hagersten, and D. Black-Schaffer. Micro-architecture independent analytical processor performance and power modeling. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015.

[16] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th annual international symposium on computer architecture (ISCA '93)*, pages 257–266, 1993.

[17] T. Yokota, K. Ootsu, and T. Baba. Potentials of branch predictors: From entropy viewpoints. In *Proceedings of the 21st International Conference on Architecture of Computing Systems*, ARCS'08, pages 273–285, 2008.