BranchSherpa: A Fast and Accurate Analytical Modeling Framework for Branch Predictors

Dongin Lee
National University of Singapore
Singapore
dongin.lee@u.nus.edu

Trevor E. Carlson
National University of Singapore
Singapore
tcarlson@nus.edu.sg

***Abstract*—As modern branch predictors become increasingly complex, cycle-level simulators impose a substantial burden on early-stage design exploration. Analytical modeling provides a promising alternative by enabling fast design exploration and offering architectural insights. However, prior analytical approaches suffer from several limitations that reduce modeling accuracy. To address these limitations, we propose BranchSherpa, a fast and accurate analytical modeling framework for branch predictor design exploration. BranchSherpa improves modeling accuracy over prior work through three key techniques: (i) jointly using taken and transition rates to derive branch entropy, (ii) leveraging multiple branch entropy values to model modern multi-component predictors, and (iii) employing an exponential saturation model to characterize branch predictor behavior. Our evaluations show that BranchSherpa reduces mean absolute error (MAE) and miss per kilo instructions (MPKI) error by 68.75% and 52.00% on average compared to prior work.**

***Index Terms*—Branch prediction; analytical modeling; design space exploration**

I. Introduction

Branch prediction remains critical to the performance of modern out-of-order superscalar processors. Over the past decades, branch prediction has evolved from early dynamic branch predictors [1]–[5] to sophisticated multi-component predictors [6]–[10]. To evaluate these increasingly complex predictors, prior studies [6]–[15] predominantly rely on architectural cycle-level simulators [16]–[21]. Cycle-level simulators are widely used because they can model detailed microarchitecture designs and provide accurate simulation results.

However, cycle-level simulators impose significant challenges for early-stage branch predictor design exploration. State-of-the-art predictors such as TAGE [6], [7] and TAGE-SC-L [8], [9] employ multiple prediction tables and auxiliary components to improve prediction accuracy on diverse conditional branches. Several branch predictors [12]–[15], [22]–[24] have been proposed as variants of existing state-of-the-art designs to address their limitations. Exploring such a large design space can be time-consuming due to the extremely slow speed of cycle-level simulators. Although sampling techniques [25]–[32] have been proposed to reduce simulation time, cycle-level simulations can still take several months to a year for complete applications [30], [33]. Furthermore, cycle-level simulators provide limited insights into branch predictor behavior.

As an alternative, analytical modeling serves as a complementary approach for early-stage design exploration. Analytical models employ mathematical expressions to estimate performance based on program characteristics and microarchitectural design. Given these characteristics, analytical modeling offers two advantages compared to cycle-level simulators. Previous studies demonstrate that analytical approaches can achieve one to two orders of magnitude speedup over detailed cycle-level simulators [34]–[36]. In addition, analytical models provide insights into microarchitectural behavior through interpretable mathematical formulas [34], [36]–[38]. However, analytical models are often considered less accurate than simulators [34], [36], [39] due to their high-level abstraction.

Several analytical approaches have been proposed for branch predictors [39]–[45]. As a first step, branch classification methods [40], [41] provide insights into branch predictability by classifying conditional branches using metrics such as taken rate and transition rate. More advanced techniques, such as Linear Branch Entropy (LBE) [42], [43], leverage branch classification methods to enable branch predictor design exploration. LBE derives branch entropy from taken rate and uses a linear relationship between branch entropy and branch miss rate for design exploration. Although LBE provides essential insights into branch predictor design, we identify several limitations in the methodology that lead to reduced modeling accuracy. First, we find that branch entropy derived solely from the taken rate can mischaracterize branch behavior for certain workloads. Second, we observe that using a single entropy value can be insufficient to model modern multi-component predictors. Third, our analysis shows that a linear model cannot fully capture the non-linear relationship between branch entropy and branch miss rate.

To address these limitations, we propose **BranchSherpa**[*], a fast and accurate analytical modeling framework for early-stage branch predictor design exploration. BranchSherpa addresses the limitations of LBE through three key techniques. First, BranchSherpa incorporates both the taken rate and transition rate to derive branch entropy values for certain workloads. This approach improves the accuracy of branch entropy deriva-

[*]Sherpas are renowned for guiding climbers through the high-altitude terrain of the Himalayas. Similarly, BranchSherpa is designed to guide the development of branch predictors by providing a fast and accurate analytical modeling methodology.

tion for these workloads. Second, BranchSherpa models multi-component predictors using multiple branch entropy values aligned with their structures. This enables BranchSherpa to accurately reflect the behavior of modern predictors across different configurations. Third, BranchSherpa employs an exponential saturation model to characterize branch predictor behavior. This model more accurately captures the non-linear relationship between branch entropy and branch miss rate.

We evaluate our methodology for diverse branch predictors [2]–[5], [7]–[9] using a variety of benchmark suites [46]–[49]. Experimental results demonstrate that BranchSherpa reduces mean absolute error (MAE) by 68.75% and miss per kilo instructions (MPKI) error by 52.00% on average compared to LBE. Beyond early-stage design exploration, we further present workload characterization as a practical use case of BranchSherpa.

II. Related Work

This section discusses prior studies related to BranchSherpa. We first describe branch classification methods used to characterize branch behavior. We then discuss previous analytical approaches that leverage branch classification methods for branch predictor modeling. Finally, we describe LBE, which serves as the primary baseline for our work.

A. Branch Classification

Branch classification methods [40], [41] categorize static conditional branches based on their dynamic behavior. These approaches use metrics such as taken rate and transition rate to classify branches into several classes. Such classifications provide insights into branch predictability.

Taken Rate. Branch Classification [40] defines taken rate as the ratio of taken outcomes to the total number of executions of a given branch. Branches with low (close to 0) or high (close to 1) taken rate are considered easy-to-predict because they are highly biased toward a certain direction (i.e., not-taken or taken). Branches with a taken rate near 0.5 are classified as hard-to-predict.

Transition Rate. Branch Transition Rate [41] proposes transition rate as a complementary metric to taken rate. Transition rate is defined as the ratio of transitions between taken and not-taken of a given branch to the total number of executions of the given branch. Branches with low or high transition rates are considered easy-to-predict since they are biased to a certain direction or represent regular patterns in transitions, respectively. By contrast, branches with a transition rate near 0.5 are classified as hard-to-predict.

Branch Transition Rate [41] addresses a limitation of the taken rate-based classification. The paper identifies that the taken rate alone can misclassify branches with regular transitions in their branch outcomes as hard-to-predict. This work further shows that combining taken and transition rates provides a clearer characterization of branch behavior.

B. Previous Analytical Approaches for Design Exploration

Previous studies [39], [42]–[44] propose several analytical approaches for branch predictor design exploration. Yokota et al. [44] use information entropy [50] to quantitatively analyze the potential performance of branch predictors. This work derives branch entropy values from branch outcome patterns. The entropy values are then used to estimate an upper bound on the branch predictor hit ratio. LBE further proposes a branch entropy-based analytical approach for branch predictor design exploration. It derives branch entropy values from the taken rate and uses a linear correlation between branch entropy and branch miss rate to estimate branch miss rate. Wang et al. [39] propose a modular framework for branch predictor design exploration. The framework characterizes applications using branch behavior metrics and selects the most suitable branch predictor. It then applies predictor-specific analytical models for design exploration.

We select LBE as the primary baseline because its modeling methodology closely aligns with our goal of early-stage branch predictor design exploration. Wang et al. [39] leverage predictor-specific analytical models for different predictors. In contrast, LBE follows a structured modeling process with microarchitecture-independent and microarchitecture-dependent stages, enabling a consistent modeling approach across different branch predictors. This consistent process enables the same modeling methodology to be reused across different predictors, facilitating early-stage design exploration. Moreover, its microarchitecture-independent profiling stage supports broader use cases such as application cloning [51]–[54] and workload characterization [45].

C. Linear Branch Entropy (LBE)

LBE introduces a novel technique for estimating precise branch miss rate as described in Section II-B. To achieve this, it introduces a microarchitecture-independent metric called linear branch entropy and a linear model that relates branch entropy to branch miss rate. The outcome frequency table (OFT) is used to capture branch behavior based on the taken rate. While profiling an application, LBE collects information about conditional branch instructions in the OFTs as shown in Fig. 1. Each branch address has its own OFT and OFT entries are indexed by a 20-bit branch history. There are two types of OFT, global and local, depending on the entry indexing scheme. Each OFT entry contains two counters that record the number of taken (i.e., $n_t(i,H)$) and not-taken (i.e., $n_{nt}(i,H)$) outcomes.

After profiling an application, branch entropy is computed in several steps, as represented in Fig. 1. The first step is to compute the taken rate for each OFT entry using (1). This equation is similar to the taken rate explained in Section II-A, while it is defined for a static branch i and history pattern H. The second step is to calculate the linear branch entropy of each OFT entry using (2).

$$p(i,H) = \frac{n_t\big(i,H\big)}{n_t\big(i,H\big) + n_{nt}\big(i,H\big)}\,, \tag{1}$$

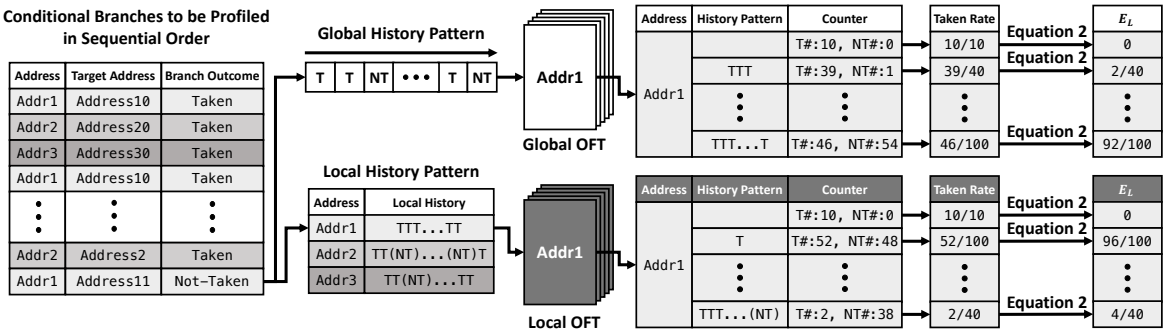

Fig. 1. An overview of application profiling and the entropy derivation process in Linear Branch Entropy [42], [43].

$$E_L(p) \;=\; 2 \times \min\big(p,\; 1-p\big)\,, \tag{2}$$

The final step is to calculate a weighted average of all linear branch entropy values across all branch instructions and their branch histories using (5). In (5), N denotes the total number of conditional branch instructions as defined in (4) and $n(i,H)$ represents the total number of taken and not-taken branches for a specific branch instruction under a history pattern as shown in (3). After deriving the weighted average in (5), LBE provides two branch entropy values from global and local OFTs for 20-bit branch histories of full branch address.

$$n(i,H) = n_t\big(i,H\big) + n_{nt}\big(i,H\big)\,, \tag{3}$$

$$N = \sum_i \sum_H n\big(i,H\big)\,, \tag{4}$$

$$E = \frac{1}{N}\sum_i \sum_H n\big(i,H\big) \times E_L\big(t(i,H)\big)\,, \tag{5}$$

LBE introduces two optimization techniques to improve modeling accuracy. The first optimization is the derivation of tournament entropy. To model a `tournament predictor`, LBE derives a tournament entropy by using both global and local OFTs. Global and local entropy are first computed for each branch instruction, and the smaller value of these two is selected as the static branch instruction's effective entropy. The tournament entropy is then computed by averaging these per-branch values. The second optimization is table entry merging for reducing both branch history and branch address lengths. For branch history, LBE removes the most significant bit and merges the two OFT entries by adding their taken and not taken counters into a single entry if these entries map to the same history. This process is repeated from 20 history bits down to 0 bits. Table entry merging of address length follows a similar process of history lengths, reducing the address bits from the full address width to zero. Thus, LBE derives $(address\ length + 1) \times 21$ entropy values for global, local, and tournament history after profiling an application.

III. Limitations of Previous Work

This section discusses the limitations of LBE. We identify three limitations that reduce its modeling accuracy.

A. Can taken rate alone accurately capture branch entropy?

To assess the modeling accuracy of branch predictors, we revisit the methodology for deriving branch entropy values in prior work [42], [43]. During profiling, LBE derives branch entropy from the taken rate (i.e., (1)). *However, entropy derived solely from the taken rate can mischaracterize branch behavior.* As described in Section II-A, taken rate often misclassifies easy-to-predict branches as hard-to-predict. Such misclassifications can lead to inaccurate branch entropy values, overestimating the inherent unpredictability.

We first quantify the degree of misclassification in taken rate-based classification. We measure the distributions of static conditional branches (i.e., `Number of Instructions`) and dynamic branch executions (i.e., `Number of Executions`) across taken rate classes. Each class is divided into 5% intervals over the full 100% range. For instance, Class 1, Class 2, Class 19, and Class 20 correspond to the 0–5%, 5–10%, 90–95%, and 95–100% ranges, respectively. Table IV lists the benchmarks used in the analysis. We focus our analysis on the distribution of dynamic branches because the number of branch executions has a practical impact on branch predictors.

Fig. 2 shows the distribution of taken rate classes across benchmark suites. In Fig. 2a and 2b, Class 1 and Class 20, which correspond to low and high taken rates, respectively, account for more than 92% of static branches and 97% of dynamic branches in PolyBenchC and NPB-SER. Similar trends (i.e., 89.64% and 91.60%) are observed for static branches in SPEC CPU 2017 and GAP Benchmark Suite (GAPBS). However, the proportion of dynamic branches in Classes 1 and 20 decreases to 85.80% and 76.60% for SPEC CPU 2017 and GAPBS, respectively, as shown in Fig. 2c and 2d.

The distribution analysis shows that most dynamic branches in PolyBenchC and NPB-SER have taken rates close to 0 or 1. In contrast, the proportions of dynamic branches in these taken rates are reduced in SPEC CPU 2017 and GAPBS. As discussed in Section II-A, taken rate alone can misclassify branches with regular branch outcomes as hard-to-predict branches. These observations show that deriving branch entropy solely from the taken rate can produce inaccurate entropy values for such branches.

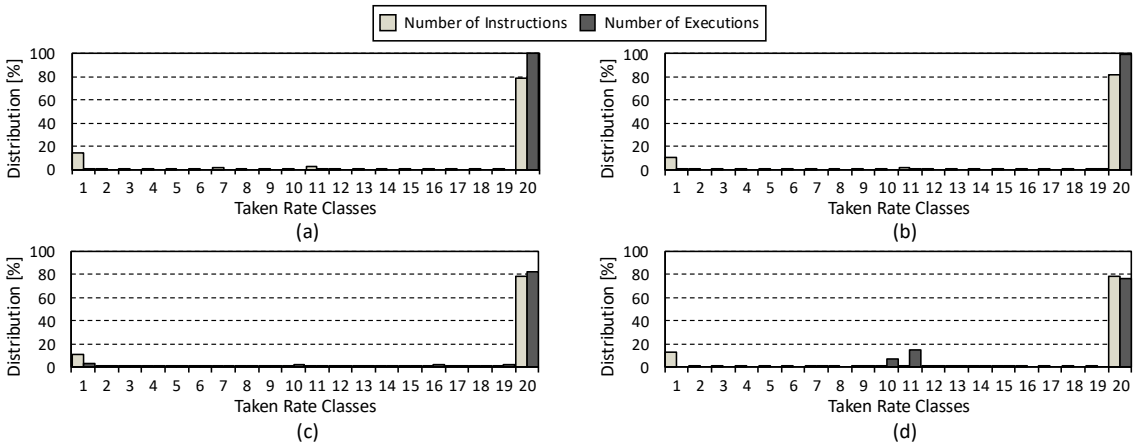

Fig. 2. Distribution of static and dynamic branches (i.e., Number of Instructions and Number of Executions) across taken-rate classes. Subfigures (a), (b), (c), and (d) show the distributions for PolyBenchC, NPB-SER, SPEC CPU 2017 Rate, and GAPBS, respectively.

TABLE I
PERCENTAGE OF DYNAMIC BRANCH DISTRIBUTIONS IN JOINT CLASS FOR SPEC CPU 2017 RATE. "-" DENOTES 0%.

	Taken Rate																				
Trans. Rate	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Total
1	3.36%	**0.85%**	**0.01%**	-	**0.02%**	**0.07%**	**0.01%**	-	**0.04%**	**0.17%**	**0.01%**	**0.01%**	**0.02%**	**0.05%**	**0.03%**	**0.53%**	**0.16%**	**0.18%**	**0.7%**	81.34%	87.56%
2	0.02%	0.01%	**0.05%**	**0.04%**	**0.01%**	**0.03%**	**0.03%**	**0.01%**	**0.03%**	**0.09%**	-	**0.12%**	**0.02%**	-	-	**0.28%**	**0.2%**	**0.2%**	0.42%	1.08%	2.64%
3	-	0.01%	0.07%	-	-	0.01%	0.03%	-	0.03%	0.01%	0.14%	0.04%	0.02%	0.01%	0.01%	0.02%	0.07%	0.19%	0.67%	-	1.33%
4	-	0.01%	0.01%	0.01%	-	-	0.03%	-	-	0.05%	0.03%	-	0.05%	0.01%	0.09%	0.04%	0.04%	0.21%	0.41%	-	0.99%
5	-	-	0.01%	-	0.01%	0.01%	-	0.02%	0.04%	0.01%	-	0.03%	0.08%	0.02%	-	0.64%	0.02%	0.21%	-	-	1.1%
6	-	-	-	0.05%	0.03%	0.01%	0.01%	0.01%	0.01%	0.01%	0.04%	0.01%	-	0.45%	0.02%	0.01%	0.25%	0.02%	-	-	0.93%
7	-	-	-	-	-	-	0.22%	-	0.06%	0.04%	0.01%	0.04%	0.01%	0.1%	0.54%	-	0.03%	-	-	-	1.05%
8	-	-	-	-	-	-	-	0.01%	0.03%	0.03%	0.13%	0.08%	0.11%	0.06%	0.6%	0.02%	0.13%	-	-	-	1.2%
9	-	-	-	-	-	0.01%	-	0.24%	-	0.09%	0.14%	0.06%	-	0.1%	-	0.12%	-	-	-	-	0.76%
10	-	-	-	-	0.01%	0.03%	0.01%	-	-	0.01%	0.04%	-	0.04%	0.01%	-	0.04%	-	-	-	-	0.19%
11	-	-	-	-	-	0.02%	-	-	-	0.01%	0.26%	-	0.02%	-	-	-	-	-	-	-	0.31%
12	-	-	-	-	-	-	-	-	0.03%	-	-	0.02%	0.01%	0.18%	0.01%	-	-	-	-	-	0.25%
13	-	-	-	-	-	-	-	-	0.01%	-	0.02%	-	-	-	-	-	-	-	-	-	0.03%
14	-	-	-	-	-	-	0.02%	-	0.03%	-	0.01%	-	-	0.03%	-	-	-	-	-	-	0.09%
15	-	-	-	-	-	-	-	0.14%	-	0.32%	0.05%	-	0.13%	-	-	-	-	-	-	-	0.64%
16	-	-	-	-	-	-	-	-	-	-	-	0.03%	-	-	-	-	-	-	-	-	0.03%
17	-	-	-	-	-	-	-	-	0.02%	-	-	-	-	-	-	-	-	-	-	-	0.02%
18	-	-	-	-	-	-	-	-	0.02%	0.03%	0.01%	-	-	-	-	-	-	-	-	-	0.06%
19	-	-	-	-	-	-	-	-	-	-	**0.01%**	-	-	-	-	-	-	-	-	-	0.01%
20	-	-	-	-	-	-	-	-	-	**0.69%**	**0.12%**	-	-	-	-	-	-	-	-	-	0.81%
Total	3.38%	0.88%	0.15%	0.1%	0.08%	0.19%	0.36%	0.43%	0.35%	1.56%	1.02%	0.44%	0.51%	1.02%	1.3%	1.7%	0.9%	1.01%	2.2%	82.42%	

We further analyze the proportion of misclassified dynamic branches in SPEC CPU 2017 and GAPBS using a joint table, following a similar approach to previous work [41]. In this analysis, each dynamic branch is mapped to a pair of taken rate and transition rate classes. Table I and Table II represent the distribution of dynamic branches for all class combinations in SPEC CPU 2017 and GAPBS, respectively. The values highlighted in bold indicate the proportions of dynamic branches that are misclassified by taken rate. In SPEC CPU 2017, misclassified branch executions account for 4.79% of total dynamic branch executions. In contrast, GAPBS exhibits a significantly higher misclassification rate of 16.35%. From joint class analysis, we observe that relying solely on the taken rate fails to accurately capture the branch behavior in GAPBS, leading to inaccurate entropy values.

B. Can a single entropy value model the behavior of multi-component branch predictors?

We examine the methodology used in prior work [42], [43] to model modern branch predictors. State-of-the-art predictors [6]–[10], [12], [13], [15], [22] combine multiple prediction tables to improve accuracy on diverse conditional branches. These tables use different combinations of address and history information to capture the various correlations between previous and current branch outcomes. *However, LBE models multi-component predictors using a single entropy value derived with a fixed history length (i.e., 25 history bits for tournament entropy).*

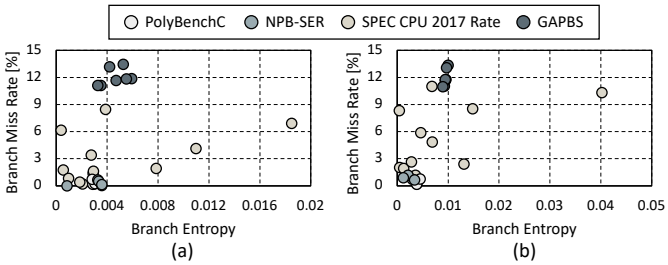

Fig. 3. Results for (a) TAGE and (b) TAGE-SC-L predictors when applying the modeling methodology from prior work. The dataset shows low correlation between branch entropy and branch miss rate for a single configuration.

LBE methodology presents two challenges for modeling

TABLE II
PERCENTAGE OF DYNAMIC BRANCH DISTRIBUTIONS IN JOINT CLASS FOR GAPBS. "-" DENOTES 0%.

	Taken Rate																				
Trans. Rate	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Total
1	-	-	-	-	-	-	**0.37%**	-	**0.41%**	**0.12%**	**15.12%**	**0.06%**	**0.04%**	-	**0.23%**	-	-	-	-	76.6%	92.95%
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.01%	-	-	-	-	-	0.01%
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-	-	-	0.01%	-	-	-	-	-	-	-	0.01%
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-	0.01%	0.03%	-	-	-	-	-	-	-	-	-	0.04%
11	-	-	-	-	-	-	-	-	-	6.98%	0.01%	-	-	-	-	-	-	-	-	-	6.99%
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Total	0%	0%	0%	0%	0%	0%	0.37%	0%	0.41%	7.12%	15.16%	0.06%	0.05%	0%	0.24%	0%	0%	0%	0%	76.6%	

multi-component predictors. First, prior work cannot accurately capture predictor behavior across different configurations using a single entropy value. Modern branch predictors employ diverse predictor configurations, including a variety of history lengths, indexing schemes, and the number of prediction tables. However, different configurations are mapped to a single entropy value, making it difficult to establish a distinct correlation between branch entropy and branch miss rate. Second, even for a single predictor configuration, the method often fails to establish a clear relationship between branch entropy and branch miss rate. As shown in Fig. 3, the methodology used in prior work fails to demonstrate a clear correlation for `TAGE` [6], [7] and `TAGE-SC-L` [8], [9].

C. Can a linear function accurately describe the relationship between entropy and miss rate?

Previous work [42], [43] identifies a linear correlation between branch entropy and branch miss rate. LBE employs a least-squares linear regression fit to derive a mathematical expression for the relationship. We conduct experiments to evaluate the correlation for several branch predictors as listed in Table III. *As shown in Fig. 8, we observe that a linear function is insufficient to accurately model branch predictor behavior.*

IV. BRANCHSHERPA METHODOLOGY

This section presents the BranchSherpa framework and its three key techniques for improving modeling accuracy. These techniques address the limitations of prior work [42], [43] discussed in Section III.

A. Overview of BranchSherpa Framework

BranchSherpa framework consists of three stages: *profiling*, *simulation*, and *recursive profiling*, as depicted in Fig. 4. The *profiling* and *simulation* stages follow a methodology similar to that of LBE (see Section II-C). The *profiling* stage derives branch entropy values from intrinsic branch behavior without any dependence on branch predictor configurations, making the stage microarchitecture-independent. This stage

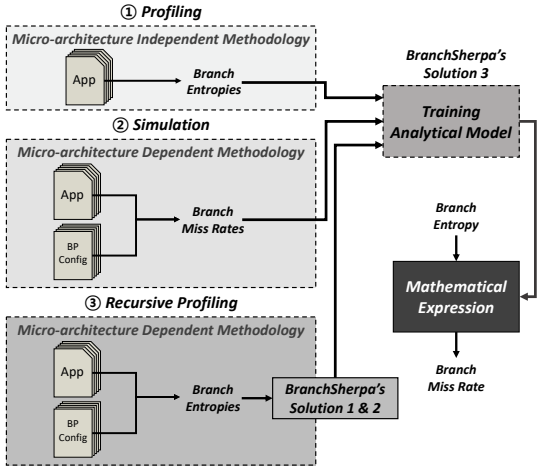

Fig. 4. An overview of the BranchSherpa framework.

selects multiple branch entropy values to cover a wide range of branch behaviors relevant to the target branch predictor. In the *simulation* stage, branch miss rates are collected using a cycle-level simulator. The *simulation* stage obtains branch miss rates for multiple configurations using diverse address and history lengths corresponding to the selected branch entropy values. Therefore, the *simulation* stage is microarchitecture-dependent because it evaluates specific predictor configurations.

We propose *recursive profiling* to address the limitations of prior work [42], [43]. As discussed in Section III-A and III-B, LBE exhibits two limitations: inaccurate branch entropy computation and simplified modeling of modern branch predictors. To address these limitations, *recursive profiling* performs additional profiling in two cases. When the taken rate alone is insufficient to characterize branch behavior, the framework jointly considers the taken and transition rates to construct more accurate entropy values. For multi-component predictors, *recursive profiling* profiles additional branch entropies using history lengths beyond 20 bits to capture the behavior of different predictor components and configurations.

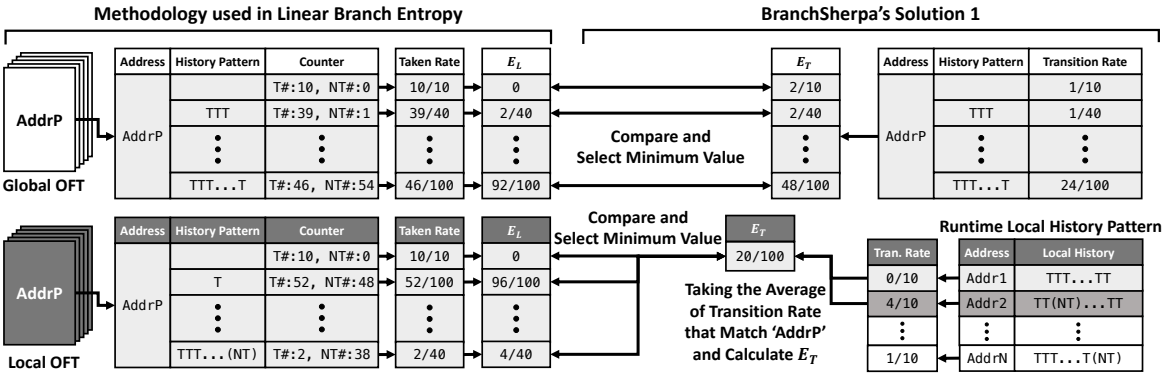

Fig. 5. An overview of the entropy derivation process in Linear Branch Entropy, extended by BranchSherpa's Solution 1 to incorporate both the taken rate and transition rate. Addr_N represents a full 48-bit address, while Addr_P is a partial address with 1–48 bits.

B. BranchSherpa's Solution 1

We propose a solution to address the limitations of branch entropy estimation in prior work [42], [43]. LBE can inaccurately characterize the predictability of conditional branches, leading to inaccurate branch entropy values (see Section III-A). Building on the insight from previous work [41], we address the misclassification problem by incorporating both the taken rate and transition rate into the entropy derivation for certain applications prone to misclassification.

There are two key aspects to consider when integrating the transition rate into branch entropy derivation based on the taken rate. These two aspects arise from the intrinsic characteristics of the transition rate, which depend on runtime information about the transitions of branch outcomes. The first aspect involves determining which historical data should be used to calculate the transition rate. These data should be aligned with the definitions of global and local histories to ensure that the entropy values are accurately captured. The second aspect is preserving the runtime information consistent with the definitions of the transition rate. If the runtime information of transitions deviates from the definitions, it may no longer represent branch transition behavior accurately.

We define the transition rate for both global and local histories based on the two essential aspects. For global history, the transition rate is defined as a value computed by sequentially considering the runtime transitions of a specific address and history pattern within a program. For local history, the transition rate is defined on a per-branch basis, where each branch instruction maintains its own transition runtime information. These definitions adhere to the intrinsic nature of each history type: global history captures collective transition behaviors across all branches, while local history preserves per-branch transition characteristics.

We extend the entropy derivation methodology of prior work [42], [43] based on the aforementioned definitions. To achieve this, BranchSherpa adopts distinct techniques for global and local histories, as illustrated in Fig. 5. For the global history, we profile all conditional branch instructions in the program to collect taken and not-taken counts, following the methodology of LBE [42], [43]. In addition, BranchSherpa captures runtime transition information of a specific address and history pattern in the program. After profiling, we first compute the taken rate for each entry, followed by the calculation of the linear branch entropy using (2). Subsequently, BranchSherpa derives the transition rate for each entry and introduces a new metric, the **linear transition rate**. The linear transition rate is expressed in (6) and q represents the transition rate calculated depending on the definition of transition rate for global history. We then compare the linear branch entropy of each OFT entry with its corresponding linear transition rate and select the smaller value as the genuine linear branch entropy, as expressed in (7). Using these genuine linear branch entropy values, we compute the overall global entropy of the application following (5).

$$E_T(q) \;=\; 2 \times \min\big(q,\; 1-q\big)\,, \tag{6}$$

$$E_L(p,q) \;=\; \min\big(E_L(p),\; E_T(q)\big)\,, \tag{7}$$

For the entropy calculation based on local history, we profile an application to collect taken and not-taken counts along with runtime transition information for each static branch. After profiling, we derive the linear branch entropy for every OFT entry based on the taken rate, and compute the transition rate for each branch address using (6). We then compare the linear branch entropy of each OFT entry with a representative linear transition rate as shown in Fig. 5. Unlike global history, local history maintains transition rates for each branch address. To derive a representative linear transition rate for entropy calculation, we perform the following process. Because the address length used for entropy computation is typically shorter than the full branch address, we first identify the transition rates of all branch addresses that match the address bits used for entropy computation. We then compute a representative linear transition rate by averaging the transition rate of the corresponding branch addresses. As shown in (7), we compare the representative linear transition rate with the linear branch entropy of every OFT entry and select the smaller value as the genuine linear branch entropy. The subsequent entropy

derivation process follows the same procedure as in the global history. Fig. 6 presents the results of applying our first solution to both global and local history-based entropy derivations. The results demonstrate that BranchSherpa's methodology effectively corrects the inaccurate entropy values. While the impact is modest for `BiMode` in Fig. 6a, the correction is more evident for `TAGE` in Fig. 6d due to its use of longer histories.

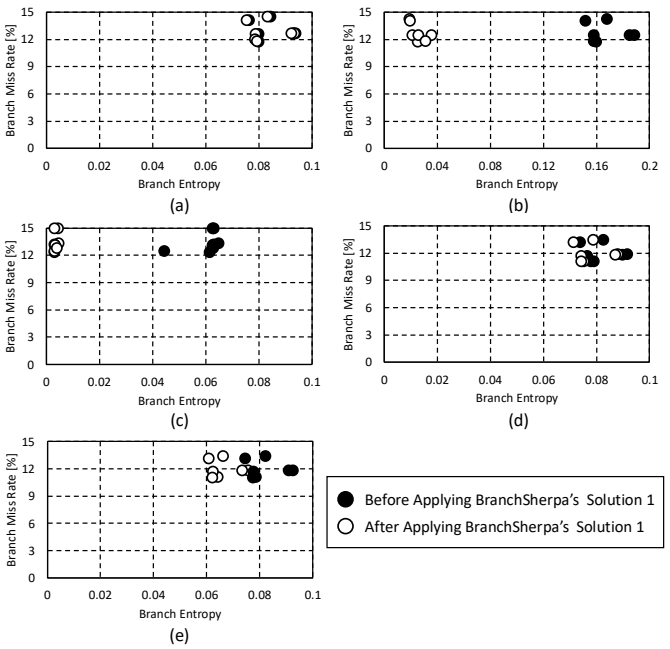

Fig. 6. Results of applying BranchSherpa and Linear Branch Entropy methodologies for entropy derivation in (a) BiMode, (b) PAp, (c) Tournament, (d) TAGE, and (e) TAGE-SC-L predictors, respectively.

C. BranchSherpa's Solution 2

BranchSherpa provides a methodology for accurately modeling modern branch predictors [7]–[9]. Prior work [42], [43] relies on a single entropy value to describe the behavior of modern multi-component predictors (see Section III-B). To address the limitation, BranchSherpa leverages multiple branch entropy values and derives a representative entropy value based on the predictor structure and behavior. When additional branch entropy values are required, BranchSherpa performs `recursive profiling`.

We classify multi-component predictors into two categories: predictors composed of homogeneous prediction tables and heterogeneous prediction components. BranchSherpa applies different representative entropy derivation methods for each category. For predictors composed of homogeneous prediction tables, we first derive branch entropies for each table using the corresponding address and history length. The representative entropy value is then computed according to the predictor behavior. `TAGE` [6], [7], for instance, employs eight tables and each table is indexed with a different global history length (i.e., 0, 2, 4, 8, 16, 32, 64, 128 bits). Considering the structure of `TAGE`, BranchSherpa introduces a weighted

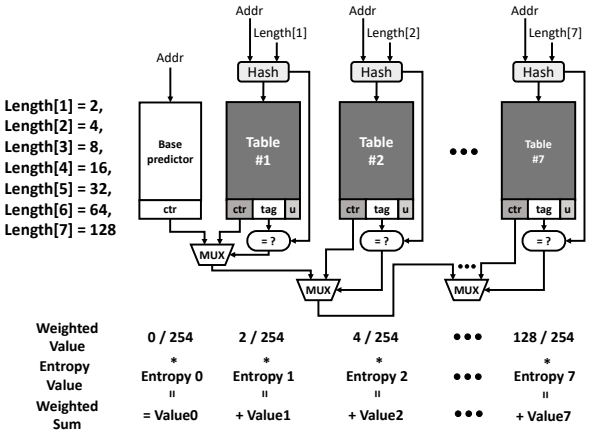

Fig. 7. Procedure for deriving the representative entropy for TAGE using BranchSherpa's solution 2

sum for computing a representative entropy value based on two observations. First, `TAGE` selects the prediction from the table with the longest history length in case of multiple tag matching. Second, `TAGE` uses substantially longer global histories because future branch outcomes can be correlated with earlier branch outcomes across long branch histories [6], [7]. As illustrated in Fig. 7, BranchSherpa first aggregates history lengths of all tables and computes a weight for each table by dividing its history length by the sum of all lengths. The representative entropy is calculated as a weighted sum, where each table's global entropy is multiplied by its corresponding weight and the results are added together.

For predictors composed of heterogeneous components, BranchSherpa first assigns a weight to each component based on predictor structure and derives a single branch entropy value for each component according to predictor behavior. The representative entropy value is then computed using a weighted sum of the selected entropy values. Our work presents a method for deriving a representative entropy of the `TAGE-SC-L` predictor with an 8KB storage budget [9] and it can be extended to predictors with larger storage budgets. `TAGE-SC-L` [8], [9] combines heterogeneous components to predict branches that are not effectively predicted by `TAGE`. It uses `TAGE` as the main predictor and incorporates a loop predictor and a statistical corrector to improve prediction accuracy. We first assign a weight to each component based on the number of bits used in its prediction tables, as each component in `TAGE-SC-L` is allocated a different budget according to its importance [8], [9]. For the TAGE component, BranchSherpa computes its representative entropy using the same methodology described in Fig. 7. For the loop predictor component, we use a single local entropy corresponding to the address and history lengths of its prediction table. The statistical corrector consists of four types of prediction tables: three PC-indexed tables, two tables indexed by global history, two tables indexed by global history of backward branches, and two tables indexed by local history. In our study, the two tables dedicated to backward branches are disabled, as

our modeling framework does not model backward-branch behavior. For the statistical corrector, we select the minimum entropy value among the tables as a representative value. Since the statistical corrector reverts the predictions from `TAGE` only when the statistical corrector provides a prediction with confidence [8], [9], we select the minimum entropy value as a proxy for such high-confidence behavior. BranchSherpa then computes a representative entropy for `TAGE-SC-L` by taking a weighted sum of the entropies of each component using their assigned weights.

D. BranchSherpa's Solution 3

BranchSherpa introduces a methodology to address the limitation of the descriptive model used in the previous work [42], [43]. LBE relies on a simple linear model to represent the relationship between branch entropy and branch miss rate (see Section III-C). However, we observe that such a linear function fails to capture the correlation that characterizes the behavior of branch predictors. As shown in Fig. 8, branch miss rates exhibit an exponential growth trend in the low entropy region, but gradually saturate as entropy values increase in the majority of the evaluated predictors. To model this non-linear behavior, we propose to use an exponential saturation model. The exponential saturation model is a non-linear regression function and the mathematical expression used in BranchSherpa is represented as (8).

$$m(x) = a + b(1 - e^{-cx}) \tag{8}$$

In the exponential saturation model, three parameters (i.e., a, b, and c) are fitted from the training dataset. In general, a defines the baseline value as x approaches zero, b determines the saturation height, and c controls the rate at which the function approaches its saturation point. When applied to branch predictor behavior, a represents the minimum branch miss rate, $a + b$ gives the upper bound of the miss rate, and c controls how quickly the miss rate grows as branch entropy increases. Thus, BranchSherpa provides insights into branch predictor behavior, including the expected lower and upper bounds of branch miss rates, as well as the growth tendency of the miss rate from the fitted model.

V. Experimental Setup

A. Profiling and Simulation Environment

We use DynamoRIO [55]–[57] version `11.90.20199` to profile conditional branches and capture entropy values. DynamoRIO provides runtime instrumentation to capture the branch behavior during program execution. Branch miss rates are collected using gem5 [16], [17] version `24.1.0.2`. We calculate the branch miss rate for committed conditional branches. We consider the results of the cycle-level simulator as the ground truth because CPU vendors do not reveal the internal structure of their branch predictors due to security concerns, such as Spectre [58]. Several branch predictors are evaluated with Skylake-like [59] CPU configuration, as listed in Table III. Representative simulation regions of each benchmark are selected using SimPoint [25], [33]. We conduct our experiment on Intel Xeon Gold 6242R using gcc/g++ version `11.1.0`. Exponential saturation and linear regression models are fitted using Python `3.9.23` and PyTorch v.`2.2.2`.

TABLE III
Evaluated Branch Predictors

Branch Predictor	Branch Entropy	Configuration
`BiMode` [4]	Global	Address, History = n [bit]
`PAp` [2], [3]	Local	Address = m, History = n [bit]
`Tournament` [5]	Tournament	Address, History = n [bit]
`TAGE` [6], [7]	Global	Address, Geometric Histories
`TAGE-SC-L` [8], [9]	Global, Local	Multiple Histories

TABLE IV
Evaluated Benchmarks

Benchmark Suite	Benchmarks	Dyn. Branch
PolyBenchC [46]	covariance, gemm, symm, syr2k, syrk, trmm, 2mm, 3mm, atax, bicg, doitgen, mvt, ludcmp, deriche, floyd-warshall, nussinov, adi, fdtd-2d, jacobi-2d, seidel-2d	7M–21B
NPB-SER [47], [60]	cg, ep, ft, is, mg, bt, sp, lu	1–34B
SPEC CPU 2017 Rate [48]	500.perlbench_r, 505.mcf_r, 520.omnetpp_r, 523.xalancbmk_r, 525.x264_r, 531.deepsjeng_r, 541.leela_r, 548.exchange2_r, 557.xz_r, 507.cactuBSSN_r, 519.lbm_r, 521.wrf_r, 527.cam4_r, 544.nab_r, 549.fotonik3d_r	11–439B
GAPBS [49], [61]	bc, cc, cc_sv, pr, pr_spmv, tc, bfs, sssp	21–283B

B. Workloads

We use diverse benchmark suites [47]–[49], [60], [61] for the evaluation of our methodology, as shown in Table IV. The benchmarks are executed with `ref`, `Class B`, and `-g 23 -n 2` inputs for SPEC CPU 2017, NPB-SER, and GAPBS, respectively. All benchmarks are compiled with `-O3` optimization level. These benchmark suites exhibit a wide range of static and dynamic branches. The average number of static branch instructions is approximately 1510, 2106, 7616, and 2483, while the average number of dynamic branches is approximately 2B, 10B, 160B, and 56B for PolyBenchC, NPB-SER, SPEC CPU 2017, and GAPBS, respectively. When selecting representative simulation regions using SimPoint, warmup and region lengths are configured based on the total number of instructions of each benchmark. For benchmarks with fewer than 100 million instructions, both warmup and region lengths are set to 10 million instructions. For benchmarks exceeding 100 million instructions, both parameters are set to 100 million instructions. We select 30 representative SimPoint regions from the total regions.

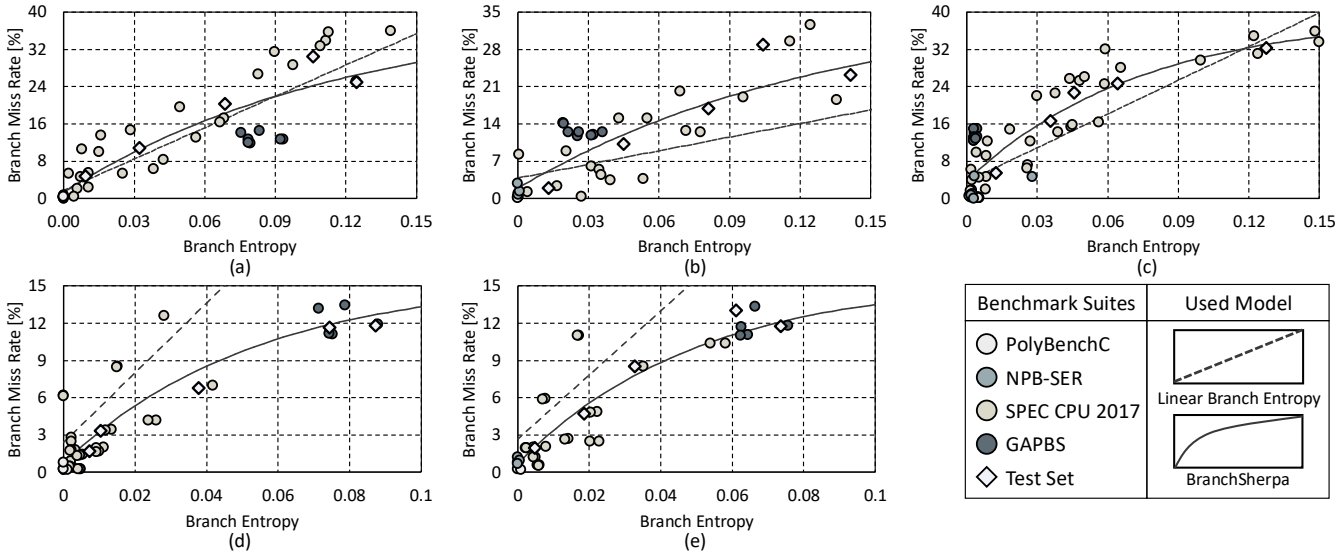

Fig. 8. Datasets representing the relationship between branch entropy and branch miss rate for (a) BiMode, (b) PAp, (c) Tournament, (d) TAGE, and (e) TAGE-SC-L predictors, respectively, along with the exponential saturation and linear models fitted using the training datasets of BranchSherpa and LBE, respectively.

VI. Evaluation

This section evaluates the modeling accuracy and profiling speed of BranchSherpa compared to LBE. We first compare the modeling accuracy of BranchSherpa and LBE using branch miss rate and MPKI. We then evaluate the profiling time for modeling the `TAGE` predictor. Finally, we discuss the insights provided by our analytical models.

A. Comparison of Accuracy to Previous Work

Fig. 8 plots the relationship between branch entropy and branch miss rate for the evaluated branch predictors. The figure also presents the trained analytical models from BranchSherpa and LBE. BranchSherpa applies *Solution 1 and 3* for `BiMode`, `PAp`, and `Tournament` predictors, while applying all three solutions to `TAGE` and `TAGE-SC-L`. Compared to prior work, BranchSherpa shows a clearer correlation between branch entropy and miss rate for all evaluated predictors. In particular, the results indicate that the proposed methodology more accurately captures modern predictor behavior such as `TAGE` and `TAGE-SC-L`. For each predictor, we use 36 to 55 datasets to train the analytical model and hold out five additional datasets for testing. The test datasets are selected to cover diverse branch entropy regions, rather than based on their fit to the trained model.

Fig. 9 shows the modeling accuracy of BranchSherpa and LBE regarding branch miss rate and MPKI for the test sets. As discussed in Section V-A, we consider the result from cycle-level simulations as the ground truth. Fig. 9a presents the MAE of the evaluated predictors. BranchSherpa reduces MAE by 68.75% on average compared to LBE for all evaluated predictors. The improvement becomes particularly significant for modern branch predictors, representing 90.82% and 84.04% lower MAE for `TAGE` and `TAGE-SC-L`, respectively. Overall, our methodology improves the modeling accuracy of branch miss rates across all evaluated branch predictors.

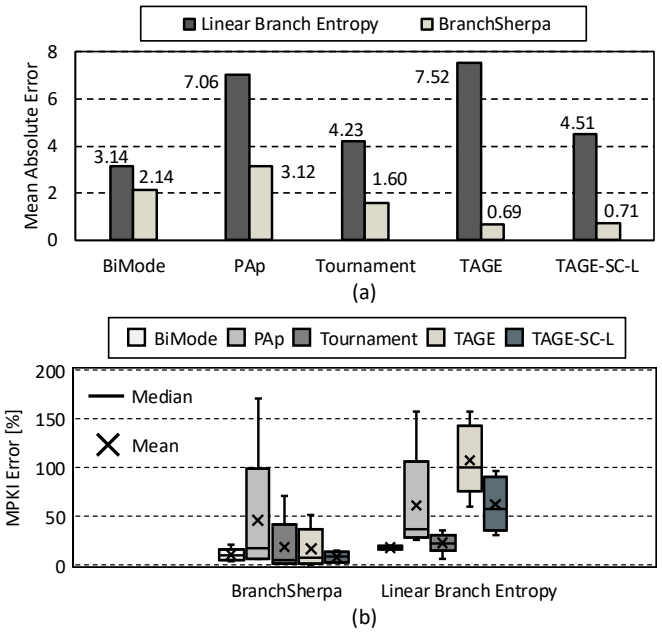

Fig. 9. A comparison of BranchSherpa and LBE in (a) MAE and (b) MPKI error across the evaluated branch predictors

Fig. 9b presents the MPKI error of the evaluated branch predictors. MPKI values are estimated using the analytical models of BranchSherpa and LBE for the same test sets shown in Fig. 8. The error is measured against the ground truth MPKI obtained from cycle-level simulations. BranchSherpa substantially reduces MPKI error for all evaluated branch predictors. Our methodology achieves, on average, 52.00% lower MPKI

error than LBE. The improvement is particularly significant for modern branch predictors, where BranchSherpa reduces MPKI error by 84.61% and 87.10% for `TAGE` and `TAGE-SC-L`, respectively. Our work also reduces the error by 42.14%, 25.50%, and 20.63% for `BiMode`, `PAp`, and `Tournament` predictors. These results demonstrate that BranchSherpa provides more accurate MPKI estimation than prior analytical approaches. BranchSherpa further demonstrates a significant improvement in MPKI error over LBE.

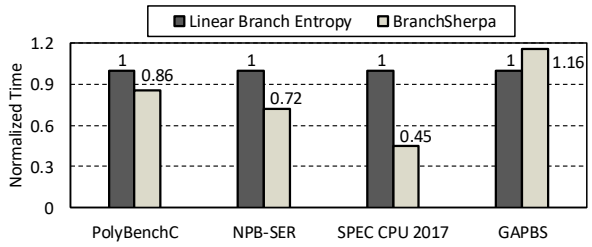

Fig. 10. A comparison of BranchSherpa and LBE in profiling time across the evaluated benchmarks

B. Comparison of Speed to Previous Work

Fig. 10 compares the profiling time of BranchSherpa and LBE for modeling the `TAGE` predictor. As described in Section III-B, LBE models multi-component predictors using tournament entropy with a fixed 25-bit history length. In contrast, BranchSherpa uses *profiling* and *recursive profiling* stages to collect branch entropy values at multiple history lengths. Since entropy profiles at different history lengths can be collected in parallel, the overall profiling time is determined by the longest-running profile. We therefore use the profiling time at the maximum history length of TAGE to represent the profiling time of BranchSherpa.

BranchSherpa reduces the average profiling time by 14%, 28%, and 55% on PolyBenchC, NPB-SER, and SPEC CPU 2017 Rate, respectively, while showing 16% higher profiling time on GAPBS. BranchSherpa achieves lower profiling time than LBE on PolyBenchC, NPB-SER, and SPEC CPU 2017 because it requires less profiling work for entropy profiling. LBE computes tournament entropy by comparing the global and local branch entropy values of each branch, whereas BranchSherpa profiles branch entropy using a longer global history. The difference between these profiling methods becomes larger as the number of static and dynamic branches increases. This trend can be seen from PolyBenchC to NPB-SER and SPEC CPU 2017, where BranchSherpa shows progressively lower profiling time relative to LBE. However, BranchSherpa requires more profiling time for GAPBS because it additionally profiles the taken and transition rates of each branch to improve the entropy estimation accuracy. The average profiling times for PolyBenchC, NPB-SER, SPEC CPU 2017 Rate, and GAPBS are 3.12 minutes, 14.68 minutes, 4.40 hours, and 2.54 hours for BranchSherpa, and 3.65 minutes, 20.45 minutes, 9.86 hours, and 2.19 hours for LBE, respectively.

TABLE V
DERIVED ANALYTICAL MODELS AND METRICS FROM THE MODELS APPLYING BRANCHSHERPA METHODOLOGY.

Predictor	Mathematical Expression	Upper Bound of Miss Rate	Miss Rate Increasing Rate
`BiMode`	$1.33 + 38.98(1 - e^{-8.33x})$	40.31	8.33
`PAp`	$1.91 + 35.40(1 - e^{-7.43x})$	37.31	7.43
`Tournament`	$3.75 + 35.61(1 - e^{-13.61x})$	39.36	13.61
`TAGE`	$0.72 + 14.91(1 - e^{-18.61x})$	15.63	18.61
`TAGE-SC-L`	$0.68 + 14.86(1 - e^{-19.92x})$	15.54	19.92

C. Insights of Our Analytical Models

Table V shows the analytical models for the evaluated branch predictors and insights derived from these models. From these mathematical expressions, we can derive the expected upper bounds of the branch miss rate and the rate at which the miss rate approaches its saturation point (see Section IV-D). Table V indicates that the state-of-the-art predictors such as `TAGE` and `TAGE-SC-L` have significantly lower upper bounds of the miss rate than conventional predictors. We also observe that the miss rates of state-of-the-art predictors approach their upper bounds more quickly as entropy increases. The lower bound of branch miss rate provides limited insight because the exponential saturation model does not accurately capture datasets with very low entropy and miss rates, as shown in Fig. 8.

VII. USE CASE: WORKLOAD CHARACTERIZATION

Beyond early-stage design exploration, BranchSherpa can support several practical use cases. Its microarchitecture-independent profiling can be applied to application cloning [51]–[54], while analytical models derived from its microarchitecture-dependent stage can be used for analytical CPU modeling [34], [37], [62] and architectural security analysis [63]. In this section, we demonstrate workload characterization as a representative use case of our analytical methodology. The goal for workload characterization is to quickly identify applications that are likely to suffer frequent branch mispredictions using branch entropy measurements.

To evaluate whether BranchSherpa methodology can support this use case, we measure the branch entropy of server workloads using Google Workload Traces [64] as listed in Table VI. Several prior studies [13], [14], [22], [24] propose specialized variants of state-of-the-art predictors for server workloads, showing that server applications exhibit frequent branch mispredictions. Therefore, we compare the branch entropy distribution of server workloads with that of other benchmark suites to examine whether our methodology supports this observation.

We first measure branch behavior distributions to obtain accurate branch entropy measurements for the Google Workload Traces [64] following the process described in Section III-A. Our results show that dynamic branches in taken rate Classes 1 and 20 account for 74.27% of all dynamic branches, while

TABLE VI
GOOGLE WORKLOAD TRACES USED FOR WORKLOAD CHARACTERIZATION.

Google Workload Traces [64]
Benchmarks: arizona, bravo, charlie, delta, merced, sierra.a.3, sierra.a.4, sierra.a.6, tahoe, tango, whiskey, yankee
Number of Dynamic Branches: 0.4–11B (AVG: 6B)

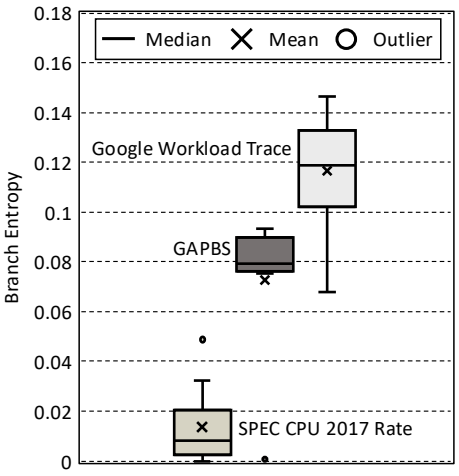

Fig. 11. Branch entropy distribution for the evaluated benchmark suites.

only 3.44% are misclassified according to the joint class table. Based on this branch behavior analysis, we use the taken rate-based method to compute the branch entropy of Google Workload Traces. Fig. 11 presents box and whisker plots of global branch entropy values for SPEC CPU 2017, GAPBS, and Google workload traces. The entropy values are derived using a fixed 10-bit address length and a 10-bit global history length. Compared to conventional benchmark suites such as SPEC CPU 2017 and GAPBS, Google workload traces exhibit significantly higher branch entropy values. As shown in Table V, higher branch entropy corresponds to higher branch miss rates. Thus, the results suggest that server workloads are likely to experience more frequent branch mispredictions, consistent with previous studies [13], [14], [22], [24]. Our analysis demonstrates that BranchSherpa can effectively characterize workload predictability using branch entropy.

VIII. CONCLUSIONS

In this paper, we present BranchSherpa, an analytical framework for early-stage branch predictor design exploration. We identify three limitations in prior analytical approaches that reduce modeling accuracy. To address these limitations, BranchSherpa introduces three modeling techniques. Our experimental results show that BranchSherpa reduces MAE and MPKI error by 68.75% and 52.00% on average compared to prior work, while achieving faster profiling time for modeling a multi-component predictor in most benchmark suites.

REFERENCES

[1] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*, 1981, p. 135–148.

[2] 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)*, 1993, pp. 257–266.

[3] ——, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO)*, 1991, pp. 51–61.

[4] C.-C. Lee, I.-C. Chen, and T. N. Mudge, "The bi-mode branch predictor," in *Proceedings of 30th Annual International Symposium on Microarchitecture (MICRO)*, 1997, pp. 4–13.

[5] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[6] A. Seznec, "A new case for the TAGE branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 117–127.

[7] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism (JILP)*, vol. 8, p. 23, 2006.

[8] A. Seznec, "TAGE-SC-L branch predictors," in *JILP-Championship Branch Prediction*, 2014.

[9] ——, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[10] D. A. Jiménez, "Multiperspective perceptron predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[11] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, 2001, pp. 197–206.

[12] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "BranchNet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 118–130.

[13] D. Schall, A. Sandberg, and B. Grot, "The last-level branch predictor," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 464–479.

[14] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, "Whisper: Profile-guided branch misprediction elimination for data center applications," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 19–34.

[15] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, "Improving branch prediction by modeling global history with convolutional neural networks," *arXiv preprint arXiv:1906.09889*, 2019.

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[17] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[18] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[19] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.

[20] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[21] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *ACM SIGARCH computer architecture news*, vol. 25, no. 3, pp. 13–25, 1997.

[22] M. Sadooghi-Alvandi, K. Aasaraai, and A. Moshovos, "Toward virtualizing branch direction prediction," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 455–460.

[23] C.-K. Lin and S. J. Tarsa, "Branch prediction is not a solved problem: Measurements, opportunities, and future directions," *arXiv preprint arXiv:1906.08170*, 2019.

[24] D. Schall, M. Ďuračková, and B. Grot, "The last-level branch predictor revisited," in *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2026, pp. 1–16.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, p. 45–57.

[26] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003, pp. 84–97.

[27] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[28] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "Barrierpoint: Sampled simulation of multi-threaded applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 2–12.

[29] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé, "Taskpoint: Sampled simulation of task-based programs," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 296–306.

[30] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson, "Looppoint: Checkpoint-driven sampled simulation for multi-threaded applications," in *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2022, pp. 604–618.

[31] C. Liu, A. Sabu, A. Chaudhari, Q. Kang, and T. E. Carlson, "Pac-Sim: Simulation of multi-threaded workloads using intelligent, live sampling," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 21, no. 4, pp. 1–26, 2024.

[32] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 2–12.

[33] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001, pp. 3–14.

[34] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, "Micro-architecture independent analytical processor performance and power modeling," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 32–41.

[35] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout, "RPPM: Rapid performance prediction of multithreaded workloads on multicore processors," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 257–267.

[36] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 3, pp. 1–28, 2011.

[37] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004, p. 338.

[38] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, 1994, pp. 52–62.

[39] Y. Wang, H. Fan, S. Li, T. Liang, and W. Zhang, "A modular branch predictor performance analysis framework for fast design space exploration," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.

[40] 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)*, 1994, pp. 22–31.

[41] M. Haungs, P. Sallee, and M. Farrens, "Branch transition rate: A new metric for improved branch classification analysis," in *Proceedings Sixth International Symposium on High Performance Computer Architecture. (HPCA)*, 2000, pp. 241–250.

[42] S. De Pestel, S. Eyerman, and L. Eeckhout, "Micro-architecture independent branch behavior characterization," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 135–144.

[43] ——, "Linear branch entropy: Characterizing and optimizing branch behavior in a micro-architecture independent way," *IEEE Transactions on Computers (TC)*, vol. 66, no. 3, pp. 458–472, 2016.

[44] T. Yokota, K. Ootsu, and T. Baba, "Potentials of branch predictors: From entropy viewpoints," in *International Conference on Architecture of Computing Systems (ARCS)*, 2008, pp. 273–285.

[45] F. Vikas, P. Gratz, and D. Jiménez, "Workload characterization for branch predictability," *arXiv preprint arXiv:2512.15827*, 2025.

[46] M. Reisinger, "PolyBenchC-4.2.1," https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1, 2016.

[47] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski *et al.*, "The NAS parallel benchmarks," Technical Report RNR-94-007, NASA Ames Research Center, Tech. Rep., 2010.

[48] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2018, pp. 41–42.

[49] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[50] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[51] M. Liang, Y. Gan, Y. Li, C. Torres, A. Dhanotia, M. Ketkar, and C. Delimitrou, "Ditto: End-to-end application cloning for networked cloud services," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, 2023, pp. 222–236.

[52] A. Joshi, L. Eeckhout, R. H. Bell Jr, and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 2, pp. 1–33, 2008.

[53] R. Panda and L. K. John, "Proxy benchmarks for emerging big-data workloads," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 105–116.

[54] G. S. Ravi, R. Bertran, P. Bose, and M. Lipasti, "Micrograd: A centralized framework for workload cloning and stress testing," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 70–72.

[55] D. Bruening, Q. Zhao, and R. Kleckner, "DynamoRIO: Dynamic instrumentation tool platform," *URL http://www.dynamorio.org*, 2020.

[56] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization (CGO)*, 2003, pp. 265–275.

[57] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing binary translation of dynamically generated code," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 68–78.

[58] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&half: Demystifying intel's directional branch predictors for fast, secure partitioned execution," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1220–1237.

[59] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation Intel Core: New microarchitecture code-named Skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[60] NASA Advanced Supercomputing Division, "NAS parallel benchmarks (NPB) 3.3.1," https://www.nas.nasa.gov/software/npb.html.

[61] S. Beamer, "GAP benchmark suite," https://github.com/sbeamer/gapbs, 2015.

[62] A. Nasr-Esfahany, M. Alizadeh, V. Lee, H. Alam, B. W. Coon, D. Culler, V. Dadu, M. Dixon, H. M. Levy, S. Pandey *et al.*, "Concorde: Fast and accurate CPU performance modeling with compositional analytical-ml fusion," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025, pp. 1480–1494.

[63] B. Amornpaisannon, A. Diavastos, L.-S. Peh, and T. E. Carlson, "Secure run-time hardware trojan detection using lightweight analytical models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 43, no. 2, pp. 431–441, 2023.

[64] "Google workload traces version 2," https://console.cloud.google.com/storage/browser/external-traces-v2, accessed: 2026-03-11.

IX. Artifact Appendix

A. Abstract

In this artifact, we provide the BranchSherpa framework and the relevant data to reproduce the experimental results shown in the paper. The artifact consists of two main components.

First, we provide profiling tools for deriving branch entropy values from target applications. The artifact includes separate tools for profiling applications online and analyzing offline traces. The implementations of the online and offline profiling tools are provided in the `online_profiler` and `offline_profiler` directories, respectively.

Second, the artifact provides the datasets and modeling scripts used to reproduce the analytical modeling workflow. The provided datasets contain branch entropy and branch miss rate pairs, divided into training and test sets. These datasets are used by the accompanying Jupyter notebooks to fit and evaluate the LBE and BranchSherpa methods. The data is provided in the `data` directory.

B. Artifact check-list (meta-information)

- **Program:** BranchSherpa and LBE frameworks.
- **Compilation:** CMake v3.16.3, GCC/G++ 11.1.0.
- **Data set:** Training and test datasets containing branch entropy and branch miss rate pairs for the evaluated branch predictors.
- **Run-time environment:** Ubuntu 20.04.5 LTS, Python3.9.23, PyTorch v.2.2.2.
- **Hardware:** A standard x86-64 CPU.
- **Experiments:** Profiling branch behavior, deriving branch entropy values, fitting the LBE and BranchSherpa models.
- **How much disk space required (approximately)?:** ∼2GB
- **How much time is needed to prepare workflow (approximately)?:** Approximately 10–30 minutes, depending on the machine and network connection
- **How much time is needed to complete experiments (approximately)?:** Approximately an hour for profiling 50 billion dynamic branches.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD 3-Clause License.
- **Data licenses (if publicly available)?:** MIT License.
- **Archived (provide DOI)?:** 10.5281/zenodo.21848462

C. Description

1) How to access: The artifact is publicly available at GitHub (https://github.com/nus-comparch/branchsherpa) or Zenodo (https://doi.org/10.5281/zenodo.21848462).

2) Hardware dependencies: The artifact does not require specialized hardware. A standard x86-64 Linux machine is sufficient to run the profiling tools and analytical models.

3) Software dependencies: The profiling tools require DynamoRIO, CMake, and a C/C++ compiler. The Jupyter notebooks provide the implementation of the analytical modeling using Python 3.9.23 and PyTorch 2.2.2. Additional Python dependencies, including NumPy, Matplotlib, and Jupyter, are listed in the provided `requirements.txt`.

4) Data sets: The artifact provides training and test datasets containing branch entropy and branch miss rate pairs for the evaluated branch predictors.

5) Models: The artifact includes Jupyter notebooks implementing the LBE linear regression model and BranchSherpa's exponential saturation model. The notebooks fit the models using the provided training datasets and evaluate them using the corresponding test datasets.

D. Installation

To evaluate BranchSherpa, DynamoRIO should first be installed. DynamoRIO version used in our experiments is no longer publicly available. We therefore recommend using version 11.90.20203, the next publicly available version. DynamoRIO can be installed using the provided script:

```
# Install DynamoRIO
$ ./install_dynamorio.sh
```

Once DynamoRIO has been successfully installed, the profiling tools can be built. The online profiler can be built using the following:

```
# Build online profiler
# Update the DynamoRIO path in the script
$ ./<path/to/online_profiler>/build/build.sh
```

The offline profiler can be built using the following:

```
# Build offline profiler
# Update the DynamoRIO path in the script
$ ./<path/to/offline_profiler>/build/build.sh
```

To run the Jupyter notebooks, we recommend using a virtual environment. The environment used in our experiments can be created using Conda as follows:

```
# Create the conda environment
$ conda create -n branchsherpa python=3.9.23
# Activate the environment
$ conda activate branchsherpa
```

After creating and activating the environment, the required Python packages can be installed as follows:

```
# Install the required Python packages
$ pip install -r data/requirements.txt
```

E. Experiment workflow

Before measuring the branch entropy of target applications, we first analyze the taken-rate class distribution. As described in the paper, if the combined proportion of the high and low classes does not account for the majority of the taken rate distribution, we further examine the joint class distribution. These analyses can be performed using `offline_profiler/branch_classification.{h,cpp}` for offline traces and `online_profiler/branch_classification.cpp` for online execution. The classification profiler can be tested using the provided example script:

```
# The following is an example,
# so please modify the script
$ ./benchmarks/test_classification.sh
```

After branch behavior analysis, we compute the branch entropy values. The profiling procedure depends on the target predictor. For single-component predictors, branch entropy is computed using the address and history lengths corresponding to the target predictor. The baseline entropy can be derived using the following profilers:

- `online_profiler/global_lbe_opt.cpp`
- `online_profiler/local_lbe_opt.cpp`
- `online_profiler/tour_lbe_opt.cpp`
- `offline_profiler/branch_entropy.{cpp, h}`

If the branch analysis indicates that the BranchSherpa entropy derivation should be used, the following profilers are used instead:

- `online_profiler/global_full_sherpa_tage.cpp`
- `online_profiler/local_sherpa.cpp`
- `online_profiler/tour_sherpa.cpp`

These tools can be executed as follows:

```
# Please modify the script
$ ./benchmarks/test_global.sh
```

For multi-component predictors, multiple branch entropy values are collected to derive a single representative branch entropy value. For TAGE and TAGE-SC-L, entropy values corresponding to longer history lengths are computed using the following profiler:

- `online_profiler/global_longer_sherpa_tage.cpp`

When both taken rate and transition rate are required for entropy derivation, the following BranchSherpa profilers are used:

- `online_profiler/global_full_sherpa_tage.cpp`
- `online_profiler/local_sherpa.cpp`

The address and history lengths in these profilers can be configured according to the target predictor's configuration. These configurations can be changed by modifying the following macros:

```
#define ADDRESS_LENGTH <integer>
#define HISTORY_LENGTH <integer>
```

The profiling scripts can be executed as follows:

```
# Please modify the script
$ ./benchmarks/test_tage.sh
```

F. Evaluation and expected results

The provided profiling tools produce one or multiple branch entropy values, depending on the profiler used. To construct an analytical model using the BranchSherpa methodology, we obtain branch entropy and branch miss rate pairs. We collect branch entropy values across a wide range using different predictor configurations and applications. For each predictor configuration and application used to obtain the branch entropy values, the corresponding branch miss rate is obtained from cycle-level simulators. This process produces training and test datasets in the same format (i.e., branch entropy and branch miss rate pairs) as those provided in `data/dataset`.

Once the datasets are prepared, the Jupyter notebooks under `data/{predictor}` can be used to evaluate the LBE and BranchSherpa methodologies. Each notebook fits the corresponding analytical model using the training dataset and evaluates its prediction accuracy on the held-out test dataset. The modeling error is then reported using MAE, allowing the modeling accuracy of LBE and BranchSherpa to be directly compared.