Sampling-Based Binary-Level Cross-Platform Performance Estimation

Xinnian Zheng, Haris Vikalo, Shuang Song, Lizy K. John, Andreas Gerstlauer
The University of Texas at Austin, TX, USA
{xzheng1, hvikalo, songshuang1990, ljohn, gerstl}@utexas.edu

***Abstract*—Fast and accurate performance estimation is a key challenge in modern system design. Recently, machine learning-based approaches have emerged that allow predicting the performance of an application on a target platform from executions on a different host. However, existing approaches rely on expensive instrumentation that requires source code to be available. We propose a novel sampling-based, binary-level cross-platform prediction method that accurately predicts performance of a workload on a target by relying on various performance statistics sampled on a host using built-in hardware counters. In our proposed framework, samples acquired from the host and target do not satisfy straightforward one-to-one correspondence that characterizes prior instrumentation-based approaches. The resulting alignment problem is NP-hard; to solve it efficiently, we develop a *stochastic dynamic coupling* (SDC) algorithm which, under mild assumptions, with high probability closely approximates optimal alignment. The prediction model constructed using SDC-aligned samples achieves on average 96.5% accuracy for 45 benchmarks at speeds of over 3 GIPS. At similar accuracies, this is up to 6× faster than instrumentation-based prediction, and approximately twice the speed of executing the same applications natively on our ARM target.**

I. Introduction

Estimating performance of complex software on hardware that is not readily available is among the key challenges in modern system-level design. Due to increasing complexity of software and hardware, performance modeling and prediction are difficult tasks. Widely adopted simulation-based approaches, such as cycle-accurate instruction set simulators (ISSs), excel in accuracy yet lack significantly in speed. By contrast, traditional analytical models are computationally efficient but their accuracy is limited. To this end, novel machine learning-based cross-platform prediction techniques have recently been proposed [1], [2]. Compared to traditional cycle-accurate ISSs, cross-platform methods offer several orders of magnitude speedup while maintaining similar accuracy. The key idea behind such approaches is the simple intuition that performance of the same application running on two different hardware platforms is correlated, and that this latent correlation can be extracted by supervised learning methods to ultimately predict performance of applications running on a target while executing them natively on a host. Resulting prediction models trained by executing small micro-benchmarks on an early reference implementation or model of the target can be used by software and hardware developers to evaluate large, real-world application behavior that would otherwise require target access or be too slow if not infeasible to obtain.

Existing cross-platform prediction approaches achieve high accuracy using a setup where training and prediction are both performed at program phase granularity. Such approaches require intrusive instrumentation of a compiler-generated intermediate representation (IR) with profiling calls at basic block boundaries to capture phase-level performance features of a program on the host as well as, during training, reference performance on the target. Instrumentation at the IR basic block level leads to three major problems, however: (1) it introduces performance overhead, (2) application source code is required, and (3) prediction is only possible for a single standalone application at a time. These inherently limit practical applicability, e.g. to predict performance of an actual application and system code mix running on a platform.

In this paper, we instead propose a sampling-based, binary-level technique that addresses the drawbacks of existing instrumentation-based cross-platform prediction methods. Our approach is source-oblivious and can perform background prediction for arbitrary binary code while maintaining similar prediction accuracy with a significant increase in speed. A key challenge in sampling-based learning methods is proper alignment of samples during training. In instrumentation-based approaches, since the IR is architecture-independent, profiling every fixed number of dynamic basic blocks guarantees an exact correspondence between the performance features on the host and reference performance on the target for each captured phase. By contrast, when host and target measurements are sampled with respect to time rather than to fixed block or phase boundaries, the one-to-one correspondence between host and target measurements is lost. Due to a different pace of application progress, sampling an application at the same rate on different hardware platforms almost always results in a vastly different number of samples. Hence, the main challenge is to align and synchronize the host and target samples during training such that they approximately correspond to the same section of executed code. To solve this alignment problem, we propose a novel, efficient and effective *stochastic dynamic coupling* (SDC) heuristic. Crucially, numerical measurements from the host and target can be viewed as two stochastic signals, and our aim is to align them such that their cross-covariance is maximized. Using the aligned samples from the host and target, we then train prediction models, which correlate performance from the host to the target.

The rest of the paper is organized as follows: after a discussion of related work and an overview of our approach, Section II presents the formulation of the alignment problem and the SDC algorithm as well as theoretical performance guarantees. Section III discusses the experimental setup and Section IV presents results. Finally, Section V concludes the paper with a summary of key contributions and results.

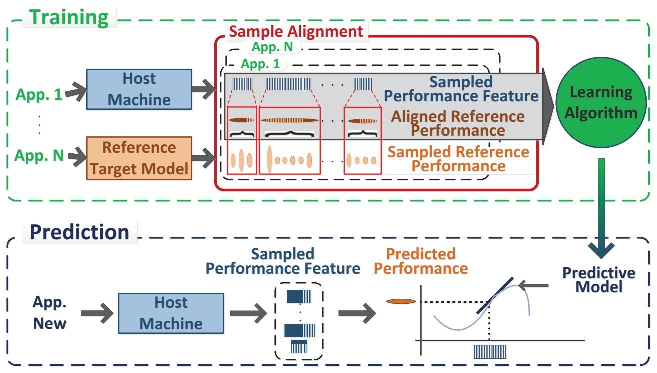


Fig. 1: Sampling-based cross-platform prediction framework.

A. Related Work

Simulation-based approaches such as cycle-accurate or cycle-approximate ISSs [3], [4] have been proposed and widely adapted for performance estimation. However, throughput of most ISSs is on the order of several hundred KIPS to several MIPS. Recently, source-level host-compiled and transaction level modeling (TLM) techniques [5] have been proposed for improving simulation speed while maintaining accuracy close to an ISS, but they often require tedious back-annotation of source code with static or dynamic performance estimates, which makes them source dependent and inherently limits speed and accuracy.

Many analytical models based upon statistical methods have been proposed [6]–[8]. However, all of these approaches try to obtain performance models for some target architecture of interest from measurements performed on the same base architecture. By contrast, in earlier work [1] we introduced performance and power prediction by establishing analytical models that correlate two distinct architectures. Ardalani *et al.* [9] proposed similar concepts for estimating GPU performance from CPU executions. These approaches were limited to predicting performance of whole programs only, where errors of more than 50% were shown. Phase-based approaches [2] were later proposed to support estimation of both performance and power at finer temporal granularities with more than 95% accuracy when targeting complex benchmarks and architectures. However, all these instrumentation-based approaches are limited to prediction of single applications for which source code is available. Furthermore, instrumentation comes with non-negligible execution overhead.

B. Overview

An overview of our approach is shown in Fig. 1. Like in previous instrumentation-based cross-platform prediction approaches, our sampling-based approach consists of a training and a prediction phase. During training, a set of representative programs is first executed on both the host and the target to obtain a training set. The target reference can be either a simulator or real hardware. During the execution of each training program, instead of instrumenting profiling calls at the phase level to measure various hardware performance features on the host and reference performance on the target, we sample them periodically. Samples are then aligned and a

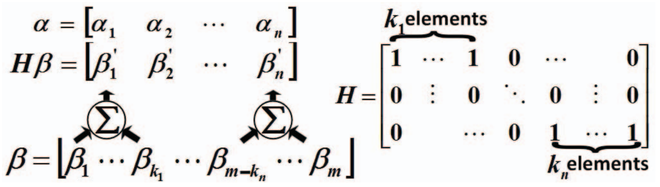

$\alpha = [\alpha_1 \quad \alpha_2 \quad \cdots \quad \alpha_n]$
$H\beta = [\beta'_1 \quad \beta'_2 \quad \cdots \quad \beta'_n]$
Σ
Σ
$\beta = [\beta_1 \cdots \beta_{k_1} \cdots \beta_{m-k_n} \cdots \beta_m]$
$k_1$elements
$H = \begin{bmatrix} 1 & \cdots & 1 & 0 & \cdots & & 0 \\ 0 & \vdots & 0 & \ddots & 0 & \vdots & 0 \\ 0 & & \cdots & 0 & 1 & \cdots & 1 \end{bmatrix}$
$k_n$elements

Fig. 2: Training sample alignment formulation.

cross-platform prediction model is learned from them. Finally, during the prediction phase, an arbitrary binary is executed and sampled periodically on the host. Sampled features are then fed into the model to predict performance on the target.

II. Training and Prediction

In this section, we describe sample alignment employed in the training phase and formulate the regression model used for performance prediction. We demonstrate that the sample alignment problem is NP-hard, which motivates the search for computationally efficient heuristics. Moreover, we propose one such heuristic, a linear-time approximation scheme that we refer to as stochastic dynamic coupling, and show it is guaranteed to converge under mild and realistic assumptions.

A. The Sample Alignment Problem

During training, we choose one representative feature on host and target to be used for sample alignment. Let $\alpha \in \mathbb{R}^n_+$ and $\beta \in \mathbb{R}^m_+$ denote positive vectors whose entries are samples of the time series of chosen base measurements acquired on a host and target, respectively, that need to be aligned. Without a loss of generality, assume that $n < m$; this assumption holds for the rest of the paper. The goal of sample alignment is to find a binary coupling matrix $H \in \{0,1\}^{n\times m}$ such that the empirical covariance between the host samples α and the aligned target samples $\beta' = H\beta$ is maximized. As illustrated in Figure 2, the coupling matrix H belongs to the space $\mathcal{H}$ of binary matrices characterized by descending staircase pattern of consecutive ones and non-overlapping support of rows. Since the i^{th} row of H specifies how consecutive target samples in β need to be aggregated to form the i^{th} element of the aligned vector β', every element in the original vector β must correspond to only one element of α. Formally, the alignment problem can be expressed as the optimization

$$\underset{H\in\mathcal{H}}{\text{maximize}}\; cov(\alpha, \beta') = \frac{1}{n}(\alpha - E[\alpha])^T(\beta' - E[\beta']). \quad \text{(II.1)}$$

If α and β are normalized so that $\sum_{i=1}^n \alpha_i = 1$ and $\sum_{j=1}^m \beta_j = 1$, then $E[\alpha] = (\frac{1}{n}\sum_{i=1}^n \alpha_i, \ldots, \frac{1}{n}\sum_{i=1}^n \alpha_i)^T = (\frac{1}{n}, \ldots, \frac{1}{n})^T$ and similarly, due to the structure of H, $E[\beta'] = (\frac{1}{n}\sum_{i=1}^n (H\beta)_i, \ldots, \frac{1}{n}\sum_{i=1}^n (H\beta)_i)^T = (\frac{1}{n}\sum_{j=1}^m \beta_j, \ldots, \frac{1}{n}\sum_{j=1}^m \beta_j)^T = (\frac{1}{n}, \ldots, \frac{1}{n})^T$. Therefore, expectations $E[\alpha]$ and $E[\beta']$ are constants that do not depend on matrix H and the original optimization problem (II.1) can hence be simplified as

$$\underset{H\in\mathcal{H}}{\text{maximize}} \quad \alpha^T H\beta. \quad \text{(II.2)}$$

Due to the combinatorially high number of potential solutions to the alignment problem, optimization (II.2) is NP-hard and thus finding its exact solution for large n and m is generally intractable. As an alternative, we propose a practically feasible

Algorithm 1 Stochastic Dynamic Coupling

```
Input: $\alpha \in \mathbb{R}_+^n$, $\beta \in \mathbb{R}_+^m$;
Output: $H$, total_error
1: Initialize $H_{ij} = 0 \ \forall i, j$ and total_error = 0;
2: Normalize $\alpha = \frac{\alpha}{\sum_{i=1}^n \alpha_i}$, $\beta = \frac{\beta}{\sum_{j=1}^m \beta_j}$;
3: k = 1;
4: for $i = 1, \ldots, n$ do
5:     residual = $\alpha_i$;
6:     for $j = k, \ldots, m$ do
7:         if |residual| > |residual $- \beta_j$| then
8:             residual = residual - $\beta_j$;
9:             $H_{ij}$ = 1;
10:        else
11:            $r_i$ = |residual|;
12:            total_error = total_error + $r_i$
13:            k = j;
14:            break;
15:        end if
16:    end for
17: end for
18: return $H$, total_error
```

greedy strategy with time complexity $O(\max(m, n))$, hence only linear in m and n, and show that its solution converges asymptotically in probability to the optimal alignment.

B. Stochastic Dynamic Coupling

The stochastic dynamic coupling (SDC) procedure is formalized as Algorithm 1. SDC starts with the first component of α and aligns to it multiple consecutive components (starting from the first one) of β; specifically, SDC proceeds to aggregate components of β and align them to α_1 for as long as the absolute value of the alignment residual keeps decreasing (line 7 to line 16 in Algorithm 1). The residual represents the remainder of α_1 that has not yet been accounted for by the sum of aligned β components. Once no more β components can be aligned, SDC moves to α_2 and aligns to it the next subset of consecutive β components, and so on. The SDC procedure keeps track of the total residual error during the execution (line 12) as a quality measure of the overall alignment. The total residual error accumulates the leftover α_i residuals that are unaccounted for during each iteration of the outer *for* loop (line 4). Ideally, we want the total residual error to be as close to zero as possible. In the next section we show that the total residual error from SDC indeed converges in probability to zero asymptotically with sampling parameters.

C. Optimality of SDC

Before arguing asymptotic optimality of SDC, we impose the following boundedness assumption on vector β.

Assumption 1. *There exist a constant $C > 0$ such that $\|\beta\|_\infty \in [0, \frac{C}{\lambda}]$, where $\lambda = \frac{m}{n}$ characterizes the relative sampling rate between the two input vectors α and β.*

To see that Assumption 1 is intuitive and realistic, note that in the alignment problem each component of vector β corresponds to the total number of hardware performance events of a certain type that occur during a sampling period. Since sampling periods are finite, the total number of hardware performance events in each period is bounded. Moreover, if we assume application runtime is finite, the number of performance events in each sampling period decreases with sampling frequency. The ratio λ between the total number of samples on two platforms may therefore serve as a proxy for the ratio of sampling frequencies.

Theorem 1. *Under the boundedness Assumption 1, the total residual error $R = \sum_{i=1}^n r_i$ of the SDC procedure satisfies*

$$\mathbb{P}(R \geq \frac{nC}{2\lambda} + \epsilon) \leq \exp(-\frac{\lambda^2 \epsilon^2}{2n^3 C^2}) \qquad \text{(II.3)}$$

for all $\epsilon \geq 0$.

Intuitively, Theorem 1 states that as the relative sampling rate λ increases, the probability of having large total residual error decreases exponentially with λ^2. Asymptotically, as $\lambda \to \infty$, (II.3) becomes

$$\lim_{\lambda\to\infty} \mathbb{P}(R \geq \frac{nC}{2\lambda} + \epsilon) \leq \lim_{\lambda\to\infty} \exp(-\frac{\lambda^2\epsilon^2}{2n^3C^2})$$
$$\Rightarrow \lim_{\lambda\to\infty} \mathbb{P}(R \geq \epsilon) \to 0 \Rightarrow R \xrightarrow{p} 0,$$

and the total residual SDC error converges in probability to 0.

Proof. It follows from Assumption 1 that $\beta_j \in [0, \frac{C}{\lambda}]$. The SDC procedure guarantees that the residual error satisfies $r_i \in [0, \frac{C}{2\lambda}]$. This is straightforward to prove by a contradiction argument. Assume the contrary, which means that at some iteration i of the inner loop (line 7 in Algorithm 1) SDC in the *else* branch (line 11 in Algorithm 1) assigns $r_i > \frac{C}{2\lambda}$. However, since all the elements of β are bounded above by $\frac{C}{\lambda}$, it follows that $|r_i - \beta_j| \leq |\frac{C}{2\lambda} - \frac{C}{\lambda}| \leq \frac{C}{2\lambda} \leq r_i$, which means the condition for the previous *if* branch (Algorithm 1, line 8) is satisfied and the algorithm could not have entered the *else* branch (line 11), hence a contradiction.

Define a random variable $Z_i = E[\sum_{k=1}^n r_k | r_1, \ldots, r_i]$ and note that the random sequence $\{Z_i : i = 0, \ldots, n\}$ forms a martingale with respect to the sequence $r_1, r_2, \ldots, r_n$. From the definition of a discrete-time martingale [10] it follows that $E[Z_{i+1}|r_1, \ldots r_i] = E[E[\sum_{k=1}^n r_k|r_1, \ldots, r_{i+1}]|r_1, \ldots r_i] = E[\sum_{k=1}^n r_k|r_1, \ldots, r_i] = Z_i \ \forall i = 0, \ldots, n$, where we applied the smoothing property of conditional expectation since $\mathcal{G}(r_1, \ldots, r_i) \subseteq \mathcal{G}(r_1, \ldots, r_{i+1})$. Here, $\mathcal{G}(S)$ denotes the σ-algebra generated by set S. Such a construction is also known as a Doob backward martingale [10].

Now, since $r_i \in [0, \frac{C}{2\lambda}]$, the martingale difference can be bounded as follows: $|Z_i - Z_{i-1}| = |E[\sum_{k=1}^n r_k|r_1, \ldots, r_i] - E[\sum_{k=1}^n r_k|r_1, \ldots, r_{i-1}]| \leq |E[\sum_{k=1}^n r_k|r_1, \ldots, r_i]| + |E[\sum_{k=1}^n r_k|r_1, \ldots, r_{i-1}]| \leq \frac{2nC}{2\lambda} = \frac{nC}{\lambda}$, where we used the triangle inequality and the fact that both Z_i and Z_{i-1} are bounded above by $\frac{nC}{2\lambda}$. Therefore, sequence $\{Z_i\}$ is a martingale with bounded increments and hence the Azuma-Heoffding inequality [10] implies that, for all $\epsilon \geq 0$,

$$\mathbb{P}(Z_n - Z_0 \geq \epsilon) = \mathbb{P}(\sum_{k=1}^n r_k - E[\sum_{k=1}^n r_k] \geq \epsilon) \leq \exp(-\frac{\epsilon^2\lambda^2}{2n(nC)^2}).$$

From $E[r_k] \leq \frac{C}{2\lambda}$ it follows $E[\sum_{k=1}^n r_k] \leq \frac{nC}{2\lambda}$. By rearranging both sides of the inequality we arrive at (II.3). □

The martingale analysis outlined above applies to the most general case where no independence assumption is imposed on the random variables r_i. With certain independence assumption on r_i, we can achieve significantly tighter bounds on the concentration behavior of R. In what follows, we describe a special case which yield better concentration results.

Remark 1. ***(Independence)*** *Under independence assumptions on random variables r_i, the Heoffding inequality [10] for the sum of independent sub-Gaussian random variables yields*

$$\mathbb{P}(R \geq \frac{nC}{2\lambda} + \epsilon) \leq \exp(-\frac{8\lambda^2\epsilon^2}{nC^2})$$

for all $\epsilon \geq 0$. Here we used the fact that $r_i \in [0, \frac{C}{2\lambda}]$ implies r_i is a sub-Gaussian random variable with parameter $\sigma = \frac{C}{4\lambda}$.

Note that in this case, the multiplicative constant in the exponential term ($\frac{8}{nC^2}$) is larger than in the general case ($\frac{1}{2n^3C^2}$), which implies faster convergence of the total residual error R to $\frac{nC}{2\lambda}$. In practice, we may not be able to drive parameter λ to extremely large values, which is the setting on which asymptotic results are predicated. We discuss in Section IV practical aspects of tuning parameter λ by sampling one platform at a faster rate than the other.

D. Prediction

We use sampled feature vectors consisting of selected hardware performance counters from the host for prediction. During training, we sample host features and target reference performance for each workload in the training set, and perform SDC alignment using selected host and target measurements to determine the coupling matrix H. Similar to existing cross-platform prediction approaches, we then apply a constrained locally linear regression (CLLR) on the the aligned host feature vectors and the target performance to obtain a local LASSO-like prediction model specific to each period.

To formalize the prediction procedure, for each sample t let $x_t \in \mathbb{R}^d$ denote the performance feature vector obtained from the host and $y_t \in \mathbb{R}$ the performance on the target. The goal of prediction is to extrapolate a mapping $\mathfrak{F} : \mathbb{R}^d \to \mathbb{R}$ such that for all t, $\mathfrak{F}(x_t) \approx y_t$. Results in [1] suggest that performance features on one platform and timing on another follow a non-linear relationship. Instead of assuming that $\mathfrak{F}$ is globally linear, one can only impose a differentiability assumption, i.e. that $\mathfrak{F}$ is differentiable everywhere in its domain. Although $\mathfrak{F}$ can no longer be expressed explicitly in closed form as in an ordinary linear regression case, the differentiability assumption allows approximating $\mathfrak{F}$ point-wise via a first-order linear approximation, which we denote here as $\widehat{\mathfrak{F}}$.

Formally, given the feature vector $\hat{x}_{\hat{t}}$ at sample $\hat{t}$ to be predicted, let $\{(x_l, y'_l), l = 1 \ldots q\}$ with $y'_l = \sum_j H_{lj} y_j$ be the set of feature vector and H-aligned reference performance pairs in the training set that are close to $\hat{x}_t$ based upon the distance criteria, $\|\hat{x}_{\hat{t}} - x_l\|_2 \leq \mu$, where μ is a parameter for determining the size of the local neighborhood of interest. Let $X \in \mathbb{R}^{q \times d}$ be the matrix that contains all the x_l^T as its row vectors, and $Y' \in \mathbb{R}^q$ be the column vector that contains all the y'_l as its elements. The CLLR then solves the following

TABLE I: Performance counters profiled on the host.

Total Cycles	Total Instructions
Total Cache References	Total Cache Misses
Total Branches	Total Branch Misses

optimization problem,

$$\begin{aligned} \underset{w_{\hat{t}}}{\text{minimize}} \quad & \frac{1}{2m}\|Xw_{\hat{t}} - Y'\|_2^2 + \nu\|w_{\hat{t}}\|_2^2 \\ \text{subject to} \quad & w_{\hat{t}} \geq 0. \end{aligned} \tag{II.4}$$

The solution $w_{\hat{t}}$ is then used for prediction as parameter for the local linear approximation $\widehat{\mathfrak{F}}$ at $\hat{x}_{\hat{t}}$ (i.e, $\mathfrak{F}(\hat{x}_{\hat{t}}) \approx \widehat{\mathfrak{F}}(\hat{x}_{\hat{t}}) = w_{\hat{t}}^T \hat{x}_{\hat{t}}$). Notice that the optimization problem in (II.4) does not have an analytical solution. However, as the objective function is both smooth and convex, the solution can be computed efficiently via standard gradiant decent methods [11].

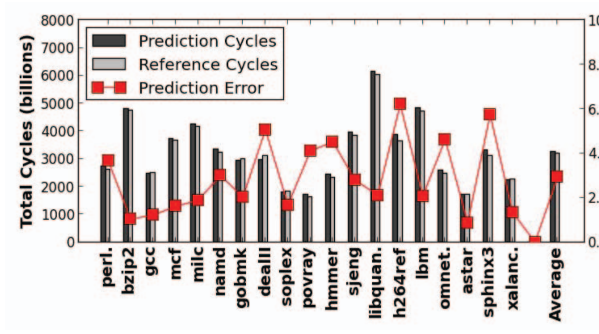
In the CLLR, we need to choose two tuning parameters, the regularization penalty ν and the parameter μ for controlling the size of the neighborhood to explore. We iteratively employ a standard cross-validation technique [12] over the training set to empirically determine their optimal values.

III. Prediction Setup

We have implemented a prediction setup using the Linux Perf tool infrastructure v3.16 for timer interrupt based counter sampling, and Python v2.7.3 and SciPy v0.15.1 as the main computation environment. We collect a total of 6 hardware performance counters, shown in Table I, on an Intel Core i7-920 machine with 24GB of memory used as the host. Note that host hardware limits simultaneously collectable counters. We use fewer features than prior work [1], [2], which relied on expensive, repeated profiling of each program to collect 14 counters in total. We sample at 5 different periods ranging from 100ms to 1000ms on the host to study the effect of sampling period on the prediction accuracy and speed. On the target, we sample only cycles and instructions as reference, where we choose the sampling period to be 100ms as the minimum allowed by the Perf tool.

We use the total number of host and target instructions in each sampling period as α and β base features to perform alignment in our setup. Unlike other performance features that are both microarchitecture and instruction set architecture (ISA) dependent, the number of instructions only depends on the ISA. We otherwise obtain sampled training data consisting of host counter vectors x_i and corresponding reference target performance y_j. The aligned training data is then used in prediction by solving the optimization problem in (II.4).

Similar to [2], our training set consists of 284 diverse and representative sample programs from the ACM International Collegiate Programming Contest (ICPC) database. We employ a larger training set than [2] to account for use of fewer counters and coarser sampling. We apply our trained prediction setup to a test set consisting of 19 CPU-bound C++ programs (*perlbench, bzip2, gcc, mcf, milc, namd, gobmk, dealII, soplex, povray, hmmer, sjeng, libquantum, h264ref, lbm, omnetpp, astar, sphinx3, xalancbmk*) from the SPEC 2006 CPU suite with "ref" input, 13 Java programs (*avrora, eclipse, fop, h2, jython, luindex, lusearch, pdm, sunflow, tomcat, tradebeans, trades-*


(a) 19 SPEC CPU C/C++ programs.

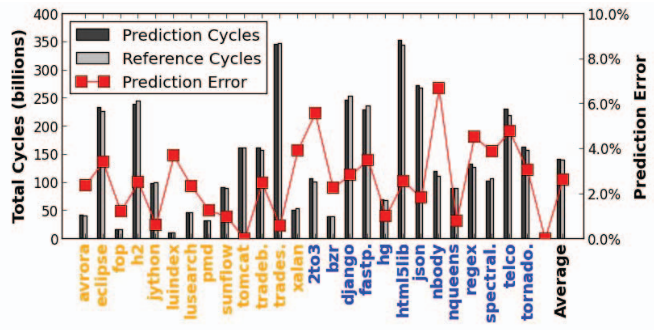
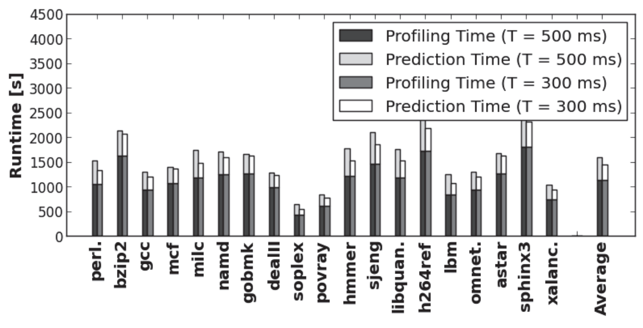

(b) 13 DaCapo Java and 13 Pybench Python programs.

Fig. 3: Predicted target cycles and prediction accuracy of 45 benchmarks (host sampling period T = 500ms).


(a) 19 SPEC CPU C/C++ programs.

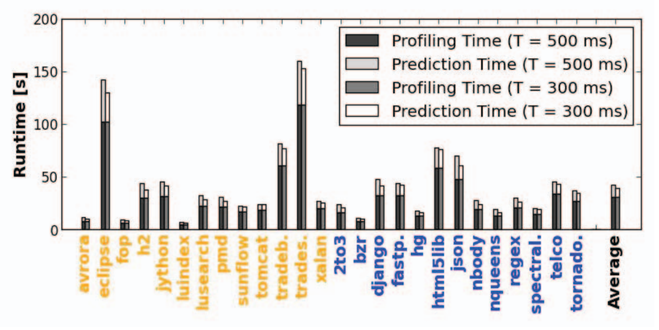

(b) 13 DaCapo Java and 13 Pybench Python programs.

Fig. 4: Runtime of 45 benchmarks.

oap, xalan) from the DaCapo Java benchmark v9.12 [13] and 13 Python programs (*2to3, bzr_startup, django_v3, fastpickle, hg_startup, html5lib, json_load, nbody, nqueen, regex_v8, spectral_norm, telco, tornado_http*) from the commercial Unified Python benchmark suite (PyBench) [14]. The Java and Python benchmarks include significant binary-only library, or virtual machine performance components. In our current setup, we are interested in predicting performance for single-core workloads excluding disturbances due to host/target operating system variations. Thus, all programs are restricted to run on one core till completion, which minimizes measurement noise due to core migration. On both the host and target, the Java programs are executed with the OpenJDK Runtime v2.6.7, and the Python test programs with Python v2.7.3.

To demonstrate the effectiveness of our approach on state-of-the-art, real-world mobile target platforms, we employ a physical hardware reference as target for training and prediction. We specifically use the ODROID-U3 development board with a quad-core ARM Cortex-A9 based Samsung Exynos 4412 SoC as our target platform for experiments.

IV. Experimental Results

To demonstrate the accuracy of the proposed approach, we apply our sampling-based prediction framework to the 45 test programs on the Intel host in order to predict performance of each benchmark on the ARM target. We compare predictions against actual measurements obtained from the U3 hardware. Fig. 3 shows the accuracy of predicting whole program performance with a host sampling period of $T = 500\text{ms}$. Predicted cycles are very close to actual hardware measurements. Worst-case prediction error is around 6%, with average errors of less than 3.5%. Despite using fewer counters and coarser sampling but with a larger training set, for the C++ benchmarks for which complete source code is available (Fig. 3a), this is comparable to results reported by existing instrumentation-based, source-level cross-platform prediction techniques [2], which achieve on average over 97% accuracy.

The total runtime of each test program is shown in Fig. 4. It consists of profiling and prediction times. The profiling time is the time it takes to collect various counters on the host. Our Intel host supports simultaneous reading of 3 counters at a time. Since the Linux Perf tool allows for over-sampled counter multiplexing, only one run of each program is necessary to collect all 6 counters with little overhead. Compared to prior work [2], which incurs instrumentation overhead at every basic block, samples at faster rates, and requires 5 separate runs of each program to collect 14 counters, profiling in our approach is significantly faster while achieving similar accuracies. The prediction time measures the total duration of solving the optimization problem (II.4) for all the samples obtained from the program. Solving time is governed by the dimension of the data matrix X and the neighborhood defined by distance threshold μ in II.4. Prior work [2] uses a convex but non-smooth objective function. By contrast, our objective function is both convex and smooth, which results in faster solving speed overall. Combined, shorter profiling and prediction times significantly increase the speed of our approach as compared to prior work [1], [2]. Overall simulation speed of our approach is on the order of 3 giga target instructions per second (GIPS). Related work runs at 500-800 MIPS counting instrumented

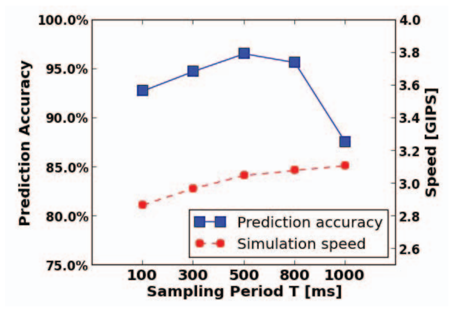


Fig. 5: Speed and accuracy tradeoff.

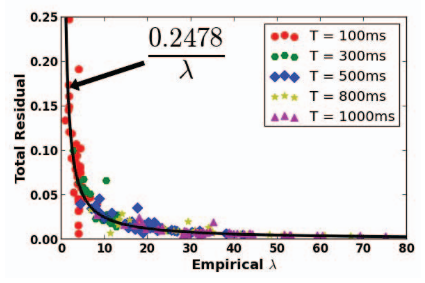


Fig. 6: Convergence behavior of total residual in training.

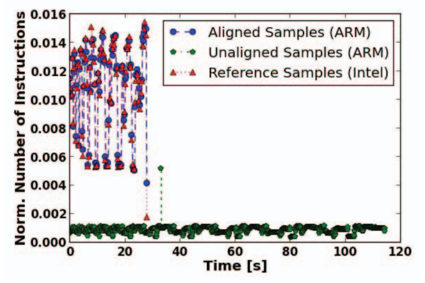


Fig. 7: Alignment example (447.dealII-train).

instructions, with a peak of around 1 GIPS excluding any prediction. Furthermore, as the Intel host is significantly more powerful than our ARM target, performance estimation using our approach is in fact on average twice as fast than running the test application natively on the target itself.

For the same sampling period T, comparing runtimes of the SPEC programs (Fig. 4b) with Java and Python programs (Fig. 4a), we see that as programs execute longer, the number of samples increases proportionally, which results in more time spend in solving the optimization problem (II.4). As sampling periods become smaller, since Perf introduces only a very small sampling overhead, the profiling time stays roughly the same. The prediction time, however, increases due to an increase in the total number of samples.

Fig. 5 shows the prediction accuracy and speed tradeoff with respect to the choice of host sampling period T. For reasons outlined above, as sampling period increases, speed increases proportionally. Prediction accuracy also increases initially until the sampling period reaches 500ms. Increasing the sampling period causes the relative sampling ratio λ to also increase, which results in a smaller alignment error as discussed in Section II-C. This is empirically confirmed in Fig. 6, which shows the convergence behavior of the total residual R with respect to λ for the training data. In general, an increase in host sampling rate T results in an increase of the effective λ. As λ increases, the total residual error deviates significantly less often from its mean, which in turn roughly scales as $O(\frac{nC}{2\lambda})$ and thus converges to zero as $\lambda \to \infty$. This agrees with our analysis of SDC optimality in Section II-C. In other words, the training data from the host and target are better aligned at larger sampling periods. However, this trend does not continue beyond a sampling period of 500ms. When the host sampling period becomes too large, the total amount of samples obtained during training decreases, and prediction accuracy suffers due to a lack of training data.

Finally, Fig. 7 illustrates SDC alignment during training using SPEC program *447.dealII* with "train" input as an example. As mentioned before, we perform SDC alignment on the total number of instructions due to their microarchitecture independence. The aligned samples measured on the ARM target couple closely with the reference samples obtained on the Intel host. Even when the underlying application exhibits drastic temporal variations, SDC is able to accurately track and map the application's behavior from the target to the host.

V. Summary and Conclusions

This paper proposes a novel sampling-based approach for source-oblivious, binary-level cross-platform performance prediction. We introduce a stochastic dynamic coupling (SDC) algorithm to address the training sample alignment problem unique to the proposed approach. Although solving such alignment exactly is NP-hard, we show that our algorithm approximates the optimal alignment with high probability under mild boundedness assumptions. Using SDC-aligned samples for prediction achieves on average over 96.5% accuracy across 45 diverse academic and commercial test programs in C/C++, Java and Python. With a host sampling period of 500ms, the average simulation speed is over 3 GIPS, which, at the same accuracy, is up to six times faster than state-of-the-art instrumentation-based cross-platform prediction, and twice the speed of executing the test applications natively on our ARM A9 target. In the future we plan to extend our approach to support both performance and power prediction for more complex multi-threaded and IO-intensive workloads, as well as other architectures, such as GPUs.

References

[1] X. Zheng et al. Learning-based analytical cross-platform performance prediction. In *SAMOS*, 2015.
[2] X. Zheng et al. Accurate phase-level cross-platform power and performance estimation. In *DAC*, 2016.
[3] Nathan Binkert et al. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.
[4] T. E. Carlson et al. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC*, 2011.
[5] O. Bringmann et al. The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems. In *DATE*, 2015.
[6] W. L. Bircher et al. Runtime identification of microprocessor energy saving opportunities. In *ISLPED*, 2005.
[7] B. C. Lee et al. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, 2008.
[8] J. C. McCullough et al. Evaluating the effectiveness of model-based power characterization. In *USENIX*, 2011.
[9] N. Ardalani et al. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *MICRO*, 2015.
[10] S. Boucheron et al. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford university press, 2013.
[11] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.
[12] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, 1995.
[13] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.
[14] The Grand Unified Python Benchmark Suite. https://www.openhub.net/p/python-benchmarks.