Learning-based Analytical Cross-Platform Performance Prediction

Xinnian Zheng, Pradeep Ravikumar, Lizy K. John, and Andreas Gerstlauer
The University of Texas at Austin, Austin, TX, USA
xzheng1@utexas.edu, pradeepr@cs.utexas.edu, {ljohn, gerstl}@ece.utexas.edu

***Abstract*—As modern processors are becoming increasingly complex, fast and accurate performance prediction is crucial during the early phases of hardware and software co-development. To accurately and efficiently predict the performance of a given software workload is, however, a challenging problem. Traditional cycle-accurate simulation is often too slow, while analytical models are not sufficiently accurate or still require target-specific execution statistics that may be slow or difficult to obtain. In this paper, we propose a novel learning-based approach for synthesizing analytical models that can accurately predict the performance of a workload on a target platform from various performance statistics obtained directly on a host platform using built-in hardware counters. Our learning approach relies on a one-time training phase using a cycle-accurate reference of the chosen target processor. We train our models on over 15,000 program instances from the ACM-ICPC programming contest database, and demonstrate the prediction accuracy on standard benchmark suites. Result show that our approach achieves on average more than 90% accuracy at $160\times$ the speed compared to a cycle-accurate reference simulation.**

I. Introduction

Under ever increasing time to market pressures, software is often designed in parallel with the processor hardware during early development stages. Being able to predict performance of software running on a target processor that does not yet physically exist is a necessary component to enable such co-development of software and hardware. However, due to the increasing complexity of processors, modeling and predicting the performance of programs is a difficult task.

Simulation-based approaches, such as cycle-accurate instruction set simulators (ISSs) are widely used in obtaining accurate estimates of program performance on a given target. However, simulation speeds are typically very slow. By contrast, traditional analytical performance models in the form of closed-form expressions or functions are efficient to compute. However, they are typically not accurate enough or still require at least a partial execution model (such as a functional ISS) of the target processor to collect target-specific traces or execution statistics, such as instruction counts, memory traces, or branch statistics. This can still be slow, and even partial target models may not be available to the software developers. By contrast, we aim to perform true cross-platform performance prediction purely based on measurements directly taken from native execution on a host.

Although collecting performance of a program on an ISS is slow, executing the same program on some existing platform can be done at native speeds. Hence, if we are able to generate accurate analytical models that associate performance of programs on a target with execution statistics efficiently obtained on an existing platform, performance prediction can be achieved without speed penalty. This motivates the need for constructing analytical models that are capable of cross-platform and/or cross-ISA performance prediction.

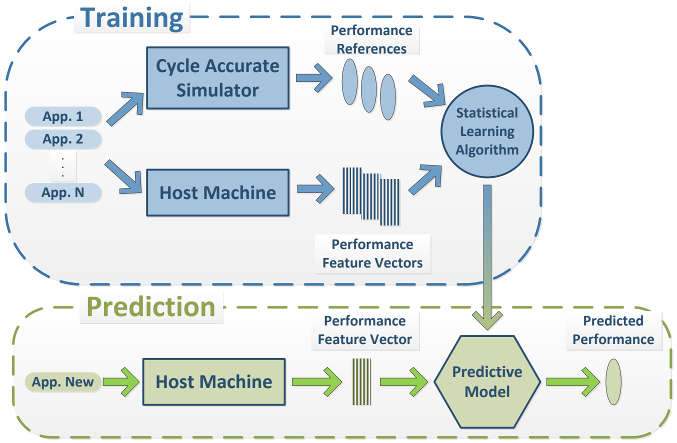

Fig. 1. Cross-platform performance prediction framwork.

In this paper, we introduce an approach for learning-based analytical cross-platform performance prediction between different architectures. From basic intuition, we know that there exists some latent relationship between the execution of a program on one platform and the performance of the same program on another. Consider the simple scenario of a program **A** that takes t seconds to finish its execution on a particular machine. We can then expect **A** to run longer on a less powerful machine. Conversely if we instead execute **A** on a more powerful machine, we are likely to expect it to finish quicker. For the remainder of the paper, we use "*target*" to denote the platform for which we want to predict performance, and "*host*" to denote the platform on which various execution statistics are collected. The goal of our work is to provide a systematic way of extracting the relationship between target and host, and use it for inferring the performance of programs running on the target by executing them on the host. Note that target and host do not necessarily have to be of similar architectures. In fact, as our results will show, it is possible to establish accurate prediction models between targets and hosts that are of vastly different hardware micro-architectures and even different instruction set architectures (ISA).

We employ a statistical, learning-based formulation, as shown in Figure 1. Our approach belongs to the supervised learning category and consists of a training and test phase. During the training phase, a large amount of sample programs (which we denote as the "*training set*") are collected and executed on the host. Various execution statistics are obtained using built-in hardware performance counters. The programs in the training set are then simulated on an cycle-accurate simulator to obtain the reference execution times on the target. The learning model takes the training data consisting of the hardware performance counter measurements and the simulated reference execution times to synthesize a model that maps host hardware performance counter data into execution times on the target. Note that the training process, is a one-time effort. Once the predictive model is constructed, we can use this model to make a prediction of the performance of a program running on the target given counter measurements efficiently obtained on the host.

The remainder of the paper is organized as follows: Section II provides a discussion on the construction of our training set. Section III introduces the profiling procedures on the host and target processors to collect the training data. This is followed by discussion of our learning model in Section IV. Section V presents empirical results for our cross-platform performance prediction as applied to the prediction of performance on an ARM processor from execution statistics observed on an x86 host. Finally, Section VI surveys the related work and Section VII concludes with a summary of the key contributions and results of this work.

II. Training Set

The validity of any learning-based approach is crucially dependent upon the choice of the training set. An ill-formed or insufficient training set affects the statistical model, causes over-interpretation of the data during the training phase and produces a model that overfits the data. A good training set should satisfy the following properties:

- Each workload inside the training set should individually be a good representative of the programs encountered during the later prediction phase.
- The variety of the workloads in the training set should be sufficiently large to cover the application space of interest.
- The overall number of program instances in the training set should be large enough to avoid overfitting problems in general.

For the purpose of our targeted cross-platform performance prediction, we are in need of a diverse variety and sufficient number of programs which contain algorithms that are used as typical building blocks in real-life software applications.

For our performance prediction approach, we utilize the programs from the ACM-ICPC (International Collegiate Programming Contest) [1] database. The ACM-ICPC is the largest and most prestigious programming contest, where hundreds of programming problems are created for the ICPC regional competitions every year. These programming problems are aimed at testing participants' knowledge on algorithms, programming as well as the ability to create new software applications. Solutions to all problems are made public in open source form. As such the ACM-ICPC database provides a great resource for large-scale program mining. More specifically, we chose 157 programs from the programming contests. Table I shows the breakdown of the programs in the training set with respect to their related application domains.

TABLE I
BREAKDOWN OF THE TRAINING SET.

Application Domains	Number of Programs
Simulation	14
Enumeration	16
String Manipuation	30
Graph Algorithm	26
Dynamic Programming	21
Geometry	25
Recursion	13
Miscellaneous	12

Programs in the simulation category perform step-wise traversal of the given input sequence and produce corresponding outputs (e.g, a replay of a chess game, a trace of a discrete event simulation, etc.) Enumeration programs often involve searching for some solutions to a problem through enumeration of all possible candidate solutions. String manipulation problems typically consist of tasks such as parsing, translation, encryption and decryption. Various graph algorithms (shortest path, graph search, connected components, network flow, etc.), dynamic programming and geometry related programs form a vast majority of our training set. Finally, problems that realize recursive solutions and other miscellaneous types complete the rest of the training set.

The large variety of programs in the ACM-ICPC database resolves the representativeness and diversity requirements of the training set. However, the overall number of program instances is still far from sufficient for training any statistical model. Since all problems were once used for the contest, input specifications are well documented and include many examples. Using this information, we generated approximately one hundred randomized inputs for each program in the training set for a total of about 15,700 program instances. In order to maintain a reasonable time for executing and profiling the entire training set, we restricted the size of each random input to complete in 1 to 20 seconds on the host machine.

III. Feature Extraction

In this section, we give an overview of the profiling and feature extraction procedure performed on the target instruction set simulator as well as the host machine. On the target side, all workloads are first cross-compiled and then simulated on a cycle-accurate instruction set simulator of the target platform to obtain the reference timings of the workloads executing on the target.

On the host side, the goal of profiling is to accurately and concisely capture and represent the execution of programs as sets of features. In our approach, we utilize various hardware performance counters as representative program features. For each workload in the training set, we obtain a feature vector

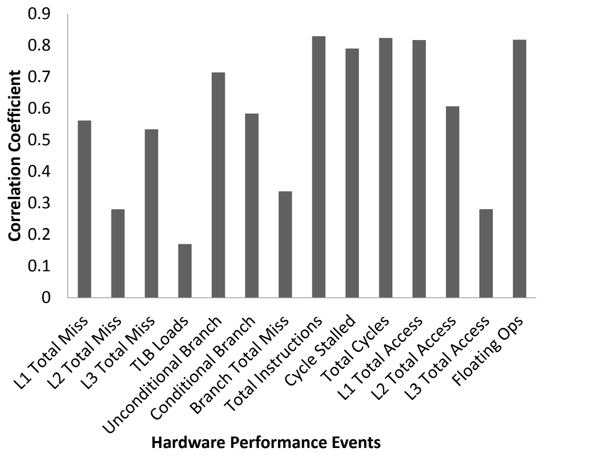

Fig. 2. Correlation of hardware performance events with reference target timing (Intel Core i7 920).

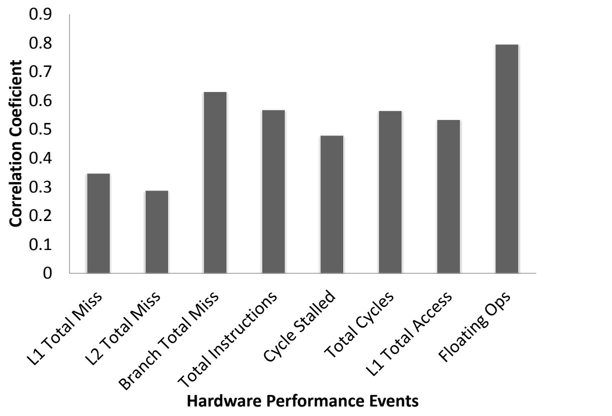

Fig. 3. Correlation of hardware performance events with reference target timing (AMD Phenom II X6 1055T).

which consists of selected hardware performance counter measurements. We perform profiling of the training set on two different host machines configurations: an Intel Core i7 920 processor with 24 GB of memory, and an AMD Phenom II X6 1055T processor with 8GB of memory. All hardware performance counter measurements were carried out using the PAPI [20] tool set.

Figure 2 shows the 14 hardware performance events that we collect on the Intel host machine, and the correlation coefficients between each individual events measured on the host and the target timing obtained from the cycle-accurate simulator. The number of CPU cycles, together with the number of instructions, the total number of L1 cache accesses and the number of floating point operations appears to be highly correlated with the target timing. Other events (i.e, number of unconditional branches and L2 cache related events) also influence the target timing substantially, whereas the rest of the events have relatively low impact on the target timing.

Similarly, Figure 3 shows the 8 hardware performance events we collected on the AMD host machine and their correlation coefficients. Note that due to the underlying difference in the implementation of the two host processors, some hardware performance counters available on the Intel i7 processor are not present on the AMD platform. Hence, only 8 out of the original 14 hardware performance events are measured on the Phenom II processor. Nevertheless, as indicated in Figure 3, the majority of the 8 hardware events continue to show strong correlation with the target timing.

A. Latent Semantics Extraction

As the various features measured on a host machine are multi-dimensional vectors, it is difficult to simultaneously look at all of their components and draw useful conclusions. Thus, we use dimensionality reduction techniques known as principal component analysis (PCA) [23] based on the Singular Value Decomposition (SVD) to analyze and extract latent semantics from the data. Formally, the SVD of any real matrix $A \in \mathbb{R}^{m\times n}$ is a factorization of the form:

$$A = U\Sigma V^T,$$

where $U \in \mathbb{R}^{m\times m}$ is a unitary matrix with its columns being the eigenvectors of AA^T, $\Sigma \in \mathbb{R}^{m\times n}$ is a rectangular diagonal matrix with positive real numbers on its diagonal, and $V \in \mathbb{R}^{n\times n}$ is also a unitary matrix with its columns being the eigenvectors of A^TA. The columns of U and V are called the **left-singular vectors** and **right-singular vectors**. The non-zero diagonal entries, the **singular values**, of Σ are the square roots of the non-zero eigenvalues of both A^TA and AA^T.

Furthermore, for each column $a_i \in \mathbb{R}^m$ in A, its k-dimensional projection $a'_i \in \mathbb{R}^k (k < m)$ is given by,

$$a'_i = \widehat{U}^T \times a_i, \tag{III.1}$$

where columns of $\widehat{U} \in \mathbb{R}^{m\times k}$ are the left-singular vectors corresponds to the largest k singular values. This transformation achieves dimensionality reduction by projecting the original m-dimensional input vectors onto a k-dimensional space, where $k < m$. The more dominant the largest k singular values are in comparison to the rest of the singular values, the more information the approximation retains.

In our case, we form input matrices D by placing each feature vector measured on the host as one column. $D_{i7} \in \mathbb{R}^{14\times N}$ and $D_{ph} \in \mathbb{R}^{8\times N}$ (N is the size of the training set) from the data obtained on the Intel i7 host and the AMD Phenom host, respectively. We normalize both data matrices D_{i7} and D_{ph} with respect to their columns and then perform the SVD to obtain $\widehat{U}_{i7}$ and $\widehat{U}_{ph}$. In both cases, we choose to keep 3 columns ($k = 3$) for the $\widehat{U}$s, where in all cases, the sum of the largest three singular values covers more than 90% of the sum of all the singular values of the normalized data matrix D_{i7} and D_{ph}. Hence, the rank-3 projection of D_{i7} and D_{ph} is computed as shown in (III.1). Shown in Figure 4 are the 3D projection of the 14-dimensional feature vectors obtained by executing all program instances in the training set on the Intel i7 host. Note the axes of such projection are also commonly known as the *principal components* (PCs). The projection points form various clusters, which is likely due to the inherent similarities between program instances. Across different clusters, some are close to others due to their inherent

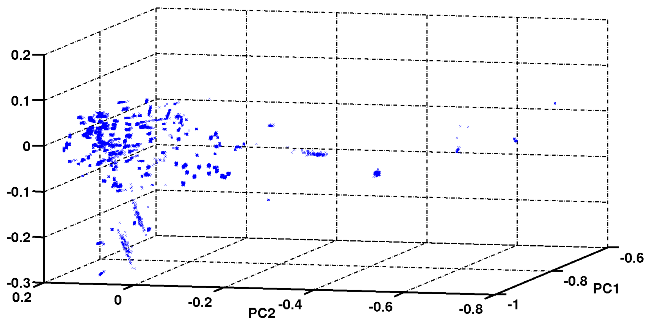

Fig. 4. Projection of the 14-dimensional hardware performance counter feature vectors (Intel Core i7) into three principal components.

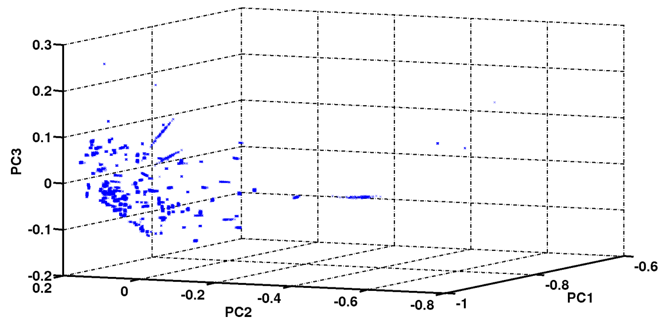

Fig. 5. Projection of the 8-dimensional hardware performance counter feature vectors (AMD Phenom II) into three principal components.

similarities. These notions of closeness (or similarity) are obtained by the latent semantic extraction during the SVD and the k-rank approximation operations. Similar results, in Figure 5, are seen for the 8-dimensional feature vector projections obtained by executing the training set on the AMD Phenom II machine. The same clustering behavior appears.

IV. Training and Prediction

Given the feature vectors obtained from the host machines and the corresponding reference timing information from the cycle accurate simulator of the target, our essential goal is to extrapolate mappings between the two, such that for any new feature vectors obtained on the same host machine, we may use such mapping to make a prediction of the target's timing.

Formally, for any given pair of a d-dimensional performance feature vector $x \in \mathbb{R}^d$ obtained on the host and its corresponding reference timing $y \in \mathbb{R}$ obtained on the target, such a mapping (or model) is a function $\mathfrak{F} : \mathbb{R}^d \to \mathbb{R}$, such that,

$$\mathfrak{F}(x) \approx y. \tag{IV.1}$$

In this section, we discuss the choice of "$\mathfrak{F}$" and the rationale behind it. We focus the discussion here at a conceptual level. A quantatitive evaluation will appear later in Section V.

A. Lasso Linear Regression

Linear regression is a widely used data fitting technique. It assumes an underlying linear relationship between the input and output, and seeks to find the solution that minimizes the sum of squared residuals. The residuals are the discrepancies between the observed and fitted values produced by the model.

In previous work, Joseph *et al.* and Lee *et al.* [11], [16] used linear models as effective ways of measuring processor performance based upon various micro-architectural parameter, such as cache sizes, associativities, branch predictor configuration, pipeline depth and so on. We adopt similar assumptions and use linear models and their variants to fit our data.

Formally, given a data set consisting of n data points (x_i, y_i), for $i = 1, \dots, n$, the ordinary linear regression problem is formulated as follows:

$$\underset{\theta}{\text{minimize}} \quad J(\theta) = \|X\theta - Y\|_2^2, \tag{IV.2}$$

where the rows of matrix $X \in \mathbb{R}^{n\times d}$ are the d-dimensional feature vectors x_i, and $Y \in \mathbb{R}^n$ is a column vector consisting of all the y_i corresponding to each x_i, and $\theta \in \mathbb{R}^d$ are the vector of coefficients of the model. The model $\mathfrak{F}$ in this case is linear (i.e. $\mathfrak{F}(x) = x^T\theta$). Such an optimization problem is also known as the ordinary least squares problem, for which an analytical solution exists as,

$$\theta = (X^TX)^{-1}X^TY.$$

Although, ordinary linear regression is applicable in our case, it is known to be sensitive to measurement noise and suffers from overfitting. Instead we consider a regularized version, the linear regression with l_1-regularization (also known as LASSO) [25], for a more robust fit. Different from the least square formulation in (IV.2), the lasso linear regression has a slightly different objective function:

$$\underset{\theta}{\text{minimize}} \quad J(\theta) = \frac{1}{2n}\|X\theta - Y\|_2^2 + \lambda\|\theta\|_1, \tag{IV.3}$$

where the rows of matrix $X \in \mathbb{R}^{n\times d}$ are the d-dimensional feature vectors x_i, and $Y \in \mathbb{R}^n$ is a column vector consisting of all the y_i corresponding to each x_i. The function $\mathfrak{F}$ also remains linear. Thus, the only difference between the ordinary linear regression is the l_1-penalty term applied to parameter θ. This penalty restricts the vector θ to be sparse. It allows the linear model to generalize better based on the principle of parsimony [5]. The variable $\lambda \in \mathbb{R}_+$ is known as the tuning parameter, which controls the degree of the l_1-penalty. Roughly speaking, the bigger the λ, the sparser θ becomes.

Due to the linearity assumption placed on $\mathfrak{F}$, lasso linear regression is expected to have good performance when the underlying relationship between inputs and outputs is in fact linear. By contrast, if the relationship is non-linear, the performance of a linear model will suffer.

B. Constrained Locally Sparse Linear Regression (CLSLR)

To deal with potential non-linearity problems, we relax the global linearity of $\mathfrak{F}$ to a local linearity assumption where $\mathfrak{F}(x)$ is assumed to be continuous and differentiable everywhere. Such an assumption relaxes $\mathfrak{F}$ to a point such that it may no longer be expressed as a linear function. Yet, the differentiability of $\mathfrak{F}$ can still be leveraged. For any target input vector x_t, base on our local linearity assumption, we can now use a first-order linear approximation $\widehat{\mathfrak{F}}$ around a close neighborhood of x_t to approximate the true $\mathfrak{F}$. For a more robust fit, we impose the l_1-penalty terms for finding a sparse solution of the parameters θ. This is known as a constrained locally sparse linear regression (CLSLR).

Before introducing the problem formulation of CLSLR, we first define the distance $\mathfrak{d}$ between any two input feature points x_i and x_j as follows:

$$\mathfrak{d} = \| \frac{x_i}{\|x_i\|_2} - \frac{x_j}{\|x_j\|_2} \|_2. \tag{IV.4}$$

The distance is essentially the Euclidean distance between the unit vector in the direction of x_i and x_j.

With this we can introduce the CLSLR formulation. Formally, let $\mathfrak{F}_{x_t}$ denote the true function value near an input point x_t, and let N_{x_t} be the set of the m pair of points $(x_1, y_1), \ldots, (x_m, y_m)$ in the training set that are closest to the input point x_t according to (IV.4). Then,

$$\mathfrak{F}_{x_t} \approx \widehat{\mathfrak{F}}(x_t) = x_t^T \theta_{x_t} \tag{IV.5}$$

is the approximated model at target point x_t. where $\theta_{x_t} \in \mathbb{R}^d$ is the parameter of the model at the target point of interest. The CLSLR near the input point x_t solves the following optimization problem,

$$\begin{aligned} \underset{\theta_{x_t}}{\text{minimize}} \quad & J(\theta_{x_t}) = \frac{1}{2m} \|X_{x_t}\theta_{x_t} - Y_{x_t}\|_2^2 + \lambda_{x_t}\|\theta_{x_t}\|_1 \\ \text{subject to} \quad & \theta_{x_t} \geq 0, \end{aligned} \tag{IV.6}$$

where each row x_i of the matrix X_{x_t} and each row y_i of the column vector Y_{x_t} corresponds to the points (x_i, y_i) in the neighborhood set N_{x_t} with respect to the input vector x_t. The positivity constraint placed upon θ_{x_t} arises from the correlation between the input features and the output timing in Section III. As indicated in both Figure 2 and Figure 3, the features are all positively correlated to the target timing. In other words, the inputs positively contribute to the output. Thus, we restrict the fitting parameter θ_{x_t} to be strictly non-negative. Notice that the optimization problem in (IV.6) does not have an analytical solution. In fact, it belongs to a particular type of convex optimization problems where the the objective function can be decomposed into a convex and smooth function (the least-square term) plus a convex but non-smooth function (the l_1-regularizer). The solution can be computed efficiently by first-order iterative algorithms, such as proximal gradient method [2], [21].

By solving (IV.6), we obtain a linear approximation to a non-linear function at input point x_t. As such, the CLSLR provides a powerful tool for modeling any generic non-linear function using a first-order local approximation. In principle, if the neighborhood is chosen to be close enough to the input point of interest, such techniques would give good prediction accuracy. However, if a close neighborhood does not exist, i.e. if all points are far way compared to the point of interest, the smoothness assumption breaks down (i.e. $\mathfrak{F}_{x_t} \not\approx \widehat{\mathfrak{F}}(x_t)$) and the CLSLR is likely to give erroneous predictions.

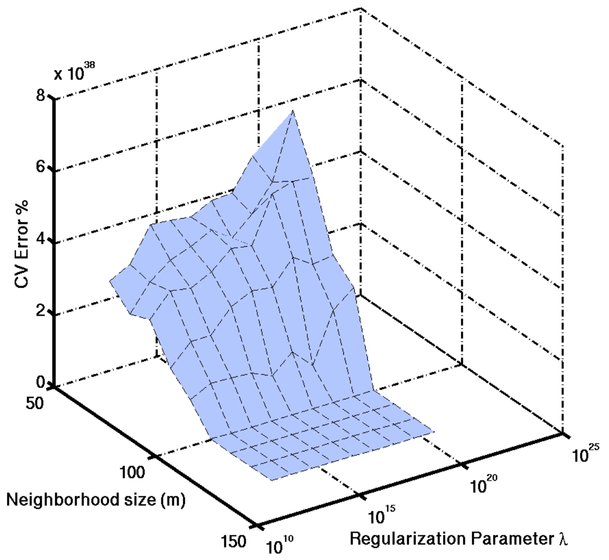

Fig. 6. Average cross-validation error.

C. Prediction and Parameter Tuning

To perform the prediction at any target point x_t of interest, we perform the following three steps in order:

1) Identify the m-nearest neighbors of x_t.
2) Solve the optimization problem in (IV.6) and obtain θ_{x_t}.
3) Compute the prediction,

$$\hat{y}_t = x_t^T \theta_{x_t}. \tag{IV.7}$$

Notice that in the process of CLSLR, we need to choose two tuning parameters, namely, the sparsity penalty λ_{x_t} and the number of nearest neighbors m to consider. We employ a standard technique known as cross-validation to determine their values. In particular, we divide the data set into a training set and a test set. We train using only data from the training set, and we use data from the test set to test how well we performed via computing the average prediction error percentage. We iteratively repeat this process applying different values for λ_{x_t} and m until the cross-validation error (CV error) is sufficiently small. Figure 6 shows the trade-off surface between the cross-validation error, the number of nearest neighbors considered and the size of λ. As λ becomes larger, the sparsity of the fitted linear model (θ) increases, the degree of freedom of the model decreases with respect to the number of parameters in the model, and the data becomes more and more difficult to fit (CV error increases).

As m grows bigger, more and more neighbors from further away are included in the computation of θ, which causes the local linearity assumption to breakdown.

Therefore, λ is chosen to be 5×10^{17} such that the local linear model produced by CLLLR is neither too complex nor too sparse, and m is chosen to be 80 such that the local linearity of the model is still maintained.

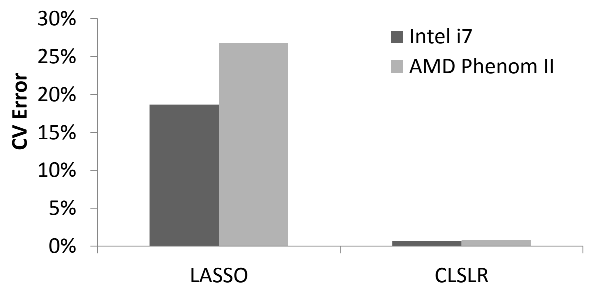

Fig. 7. Cross-validation error among different statistical models.

V. Empirical Results

We perform feature extraction of programs in the training set on both the Intel i7 and AMD Phenom hosts. On the target side, we use the default five-stage in-order version of the ARM CPU model in the gem5 simulator [3] to obtain the reference timings. For each x86 host, we generate models predicting the execution time of a program on the ARM target given hardware counter features obtained on the host. We evaluate the proposed approach from two aspects: the accuracy of the statistical models in terms of cross-validation error, and the overall accuracy of the approach on unseen programs.

A. Cross-Validation Error

We employ 10-fold cross-validation [15] of the training set used during the training phase as an estimate for the generalization error of different regression techniques. The comparison is shown in Figure 7. A Lasso Linear Regression method results in more than 15% cross-validation error, whereas CLSLR yields an 1% average prediction error. The fact that the non-linear CLSLR technique performs an order of magnitude better than the linear model strongly suggests that the underlying function "$\mathfrak{F}$" between the input hardware performance counters on the host and execution timings on the ARM target is likely to be non-linear. The accurate prediction of CLSLR provides us insights into the inherent nature of the training set. It reassures us that the assumption about the smoothness of the target function "$\mathfrak{F}$" is indeed valid empirically.

B. Prediction Accuracy

To demonstrate the accuracy of our approach, we test on 15 programs from standard benchmark suites that are not encountered in the training set. We use seven programs (**aes, crc, dijkstra, fft patricia, qsort, sha**) from the MiBench suite [9] and eight programs (**disparity, localization, mser, multi_ncut, sift, stitch, svm, tracking**) from the San Diego Vision Benchmark Suites (SD-VBS) [26]. These fifteen programs are first profiled on the Intel and AMD machines to obtain their 14-dimensional and 8-dimensional hardware performance counter feature vectors respectively. As mentioned earlier, the performance counter measurements are obtained using the PAPI tool set. These feature vectors then become

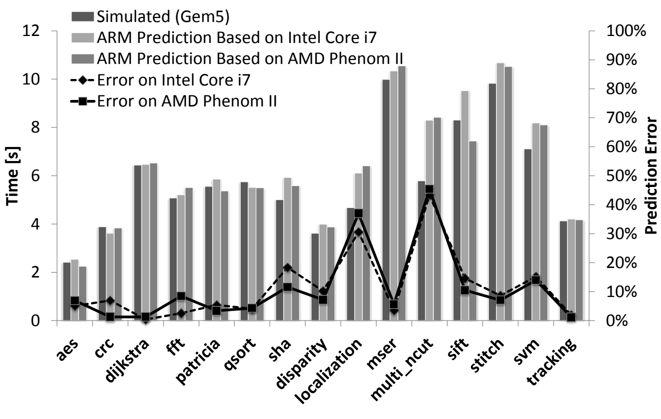

Fig. 8. Prediction accuracy of CLSLR on selected benchmarks.

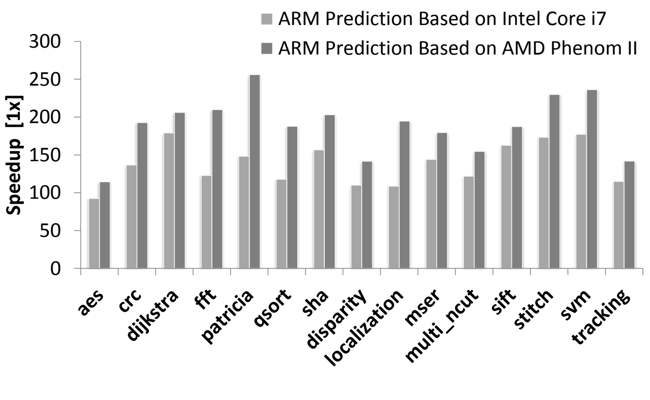

Fig. 9. Speedup over cycle-accurate ISS on selected benchmarks.

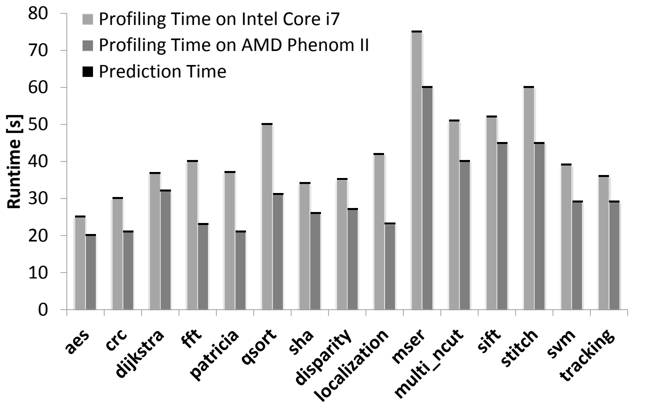

Fig. 10. Runtime on selected benchmarks.

inputs to the CLSLR algorithm which predicts performance of each benchmark on the ARM target. Finally, we compare the predictions with the actual simulated timing obtained from the cycle-accurate simulator for the ARM processor. Note that we only use the CLSLR method here. As we have shown in Section V-A, our target function $\mathfrak{F}$ is almost surely non-linear.

Performance prediction results for the 15 benchmark programs are shown in Figure 8. In general, for many of the test programs, the accuracy is only slightly worse than the performance during the cross-validation. Except for programs **sha**, **multi_ncut**, **localization** and **svm** whose prediction errors are more than 10%, performance of all benchmarks is predicted with an average error of less than 5% for both the Intel Core i7 as well as the AMD Phenom II host. As shown in Figure 11 and 12, the projected 2D features of 7 of the Mibench and 8 of the SD-VBS programs are indeed embedded in the "cloud" of the training data points. The principal component (PC) axes in Figure 11 and 12 are linear projections of the counter features obtained via PCA as described in Section III. As highlighted in Figure 11 and Figure 12, the projection of the feature vectors of program instances **sha**, **multi_ncut**, **localization** and **svm** all falls in a region somewhat isolated from the projected feature vectors of the training data. In other words, these programs behave differently from the programs inside the training set. All these programs had greater than 10% prediction error.

On the other hand, program instances such as **aes**, **dijkstra**, **fft**, **qsort**, **mser** and so on, all obtained very accurate predictions from CLSLR. As shown in both Figure 11 and 12, we notice that majority of these test programs are surrounded by an abundant number of neighboring points in the training set. This suggests that there exists some program instances inside the training set that resemble the behavior of these test programs. It is not surprising that ACM-ICPC contest questions does aim to cover a broad range of algorithms similar to, for example, the Dijkstra's shortest path algorithm, and sorting. Many string manipulation contest programs perform similar operations as the encryption and decryption algorithms we use in the test programs such as **aes** and **crc**.

In Figure 9, we compare the speedup of our approach over cycle-accurate simulation. The total runtime of our approach, as shown in Figure 10, consists of the profiling time and the prediction time. The profiling time is the time needed to collect various hardware performance counters on the host. Due to hardware limitations and differences in the the number of counters collected, this requires 5 separate runs of each program on the Intel host and 3 runs each on the AMD. The prediction time is the runtime for solving the optimization problem in (IV.6). Since solving (IV.6) scales mainly with the neighborhood size m, which in our case is a constant, the predictions take less than 1 second in all cases. Overall, the runtime is lower and hence the speedup is greater on the AMD host than on the Intel one. This is mainly due to the difference in the number of performance counters and hence profiling runs on the different hosts.

Overall, the proposed approach is accurate when there is a sufficient amount of data in the feature space that lies close to the input point of interest (i.e. for programs such as **aes** and **dijkstra**). Conversely, if the existing data lies far away from the input points (i.e. for programs such as **localization** and **multi_ncut**), the statistical model may not be able to capture the underlying structure of the non-linear target function, which directly leads to inaccuracies in the prediction. Such inaccuracies due to lack of data exist in all statistical learning problems.

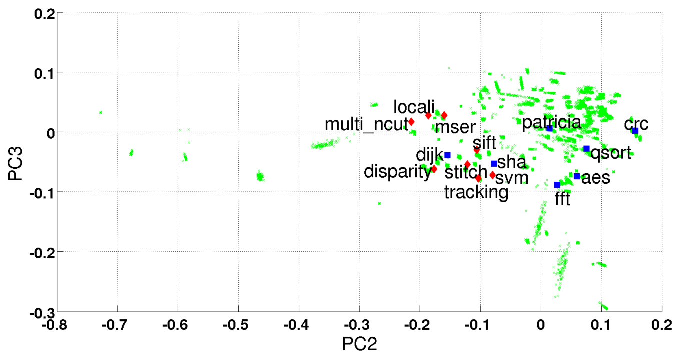

Fig. 11. Principal components (PC2 vs. PC3) of selected MiBench (🟦) and SD-VBS (♦) programs (Intel Core i7).

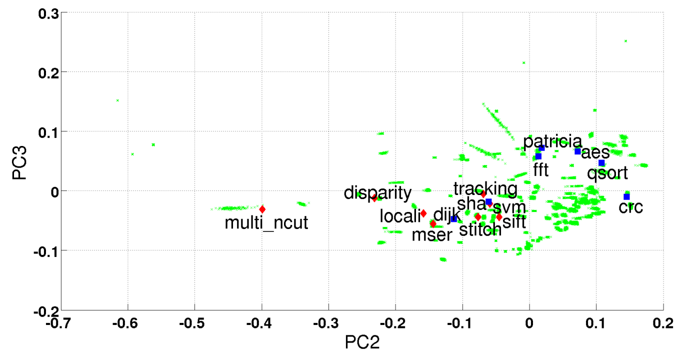

Fig. 12. Principal components (PC2 vs. PC3) of selected MiBench (🟦) and SD-VBS (♦) programs (AMD Phenom II).

VI. Related Work

Early analytical approaches [8], [22], [24] for processor performance prediction date back to decades ago, where the focus was evaluating effects of micro-architectural variations on pipeline and instruction level parallelism. More recently, in [13], Karkhanis *et al.* adpated and extended this approach to superscalar processors. It was not until recent years, however, that statistical learning and regression-based methodologies started to thrive. Lee and Brooks proposed a predictive modeling and spatial sampling method [16], [17] for efficient microarchitecture design space exploration. They employed linear regression models to characterize different microarchitectures, navigate a huge design space and identify all Pareto-optimal candidate architectures. In [12] and [11], Joseph *et al.* also utilized regression-based approaches to construct processor performance models, where an iterative error minization technique is applied to find the optimal fit. Similar ideas were also introduced by Ipek *et al.* [10] using artifical neural networks instead of regression. Lee *et al.* [18] and Khan *et al.* [14] extended and generalized the predictive modelinges approach from uni-processor to multi-processor and multi-core systems, respectively. Our objective is fundamentally different from all these existing approaches. Instead of trying to obtain statistical performance models for some target architecture of interest from measurements performed on the same base architecture, we aim to provide cross-platform performance prediction by establishing statistical models that correlate two distinct architectures.

In the simulation domain, approaches that estimate performance of a program running on slow but cycle-accurate ISS models [3], [19] of a target processor are widely used. Alternative approaches, such as source-level, host-compiled and transaction level modeling (TLM) solutions [4] have recently been proposed to improve the simulation speed while aiming to maintain accuracies close to an ISS. When including accurate cache simulation, such approaches can achieve 200-500 MIPS simulation throughput at more than 90% accuracy. However, they typically involve cumbersome static and dynamic analysis to back-annotate program code with abstracted target performance estimates. Often times, this requires a substantial amount of code injection and intrusion into the original source code (or some form of intermediate representation). In many cases, this is not a trivial task due to various resitrictions such as the effects of compiler optimizations, which inherently limits their accuracy. Furthermore, back-annotation is required for every new test program, which can be a lengthy process. Pure profiling approaches [6] use various information extracted directly from the source to predict performance of an application across platforms. However, they still require source code modifications and, owing to their high-level nature, are usually limited to simplistic very inaccurate target performance models. By contrast, our approach treats the code execution of a program as a "blackbox", and only requires a one-time training to construct a statistical model that can predict performance across arbitrary, unmodified test programs. Our approach is fast (820 - 1200 MIPS on average, depending on counter support in the host architecture), while providing similar accuracies as detailed, back-annotation based approaches. However, our training process has to be repeated for every change of the target platform. This inherently limits retargetability compared to other approaches [7].

VII. Conclusions and Future Work

This paper proposes a novel learning-based analytical cross-platform performance prediction methodology. In particular, we use hardware performance features obtained from a program executing on a host processor to predict the execution time of the same program running on a distinct target processor. We have shown that even when the host and the target are of vastly different microarchitectures and ISAs, the inherent relationship between executing the same program on one platform versus another can be accurately captured. The learning problem is formulated in a constrained optimization setting and we demonstrate its effectiveness on a set of 15 benchmark programs. The prediction achieves an average accuracy of 90% for test programs. Overall, results of this work motivate the use of learning based methods in computer system performance modeling and evalution. In future work, we aim to improve prediction accuracy by reducing model granularity, increasing the representativeness and completeness of the training set, and incorporating other workload and platform features that also aid in better retargetability.

Acknowledgments

This work has been supported by SRC grant 2012-HJ-2317. We would also like to thank the anonymous reviewers for their helpful suggestions to improve the paper.

References

[1] The ACM-ICPC International Collegiate Programming Contest. http://icpc.baylor.edu/.

[2] 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.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[4] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh. The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems. In *DATE*, 2015.

[5] K. P. Burnham and D. R. Anderson. *Model selection and multimodel inference: A practical information-theoretic approach*. Springer, 2002.

[6] L. Cai, A. Gerstlauer, and D. Gajski. Retargetable profiling for rapid, early system-level design space exploration. In *DAC*, 2004.

[7] S. Chakravarty, Z. Zhao, and A. Gerstlauer. Automated, retargetable back-annotation for host compiled performance and power modeling. In *CODES+ISSS*, 2013.

[8] P. G. Emma and E. S. Davidson. Characterization of branch and data dependencies on programs for evaluating pipeline performance. *IEEE Transactions on Computer*, 36(7):859–875, July 1987.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IISWC*, 2001.

[10] E. Ipek and S. A. Mckee. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, 2006.

[11] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. 2006.

[12] P. J. Joseph. A predictive performance model for superscalar processors. In *MICRO*, 2006.

[13] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, 2004.

[14] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. Using predictive modeling for cross-program design space exploration in multicore systems. In *PACT*, 2007.

[15] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, 1995.

[16] B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In *HPCA*, 2007.

[17] B. C. Lee and D. M. Brooks. A tutorial in spatial sampling and regression strategies for microarchitectural analysis, 2007.

[18] B. C. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models, 2008.

[19] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[20] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *DoD HPCMP*, 1999.

[21] Y. Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103:127–152, 2005.

[22] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *MICRO*, 1994.

[23] J. Shlens. A tutorial on principal component analysis. *CoRR*, abs/1404.1100, 2014.

[24] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *ISCA*, 1998.

[25] R. Tibshirani. Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[26] S. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. Taylor. SD-VBS: The San Diego vision benchmark suite. In *IISWC*, 2009.