**Noname manuscript No.**
(will be inserted by the editor)

# LACross: Learning-based Analytical Cross-Platform Performance and Power Prediction

**Xinnian Zheng · Lizy K. John · Andreas Gerstlauer**

the date of receipt and acceptance should be inserted later

**Abstract** Fast and accurate performance and power prediction is a key challenge in pre-silicon design evaluations during the early phases of hardware and software co-development. Performance evaluation using full-system simulation is prohibitively slow, especially with real world applications. By contrast, 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 present LACross, a learning-based cross-platform prediction technique aimed at predicting the time-varying performance and power of a benchmark on a target platform using hardware counter statistics obtained while running natively on a host platform. We employ a fine-grained phase-based approach, where the learning algorithm synthesizes analytical proxy models that predict the performance and power of the workload in each program phase from performance statistics obtained on the host. Our learning approach relies on a one-time training phase using a target reference model or real hardware. We train our models on less than 160 programs from the ACM ICPC database, and demonstrate prediction accuracy and speed on 35 programs from SPEC CPU2006, MiBench and SD-VBS benchmark suites. Results show that with careful choice of phase granularity, we can achieve on average over 97% performance and power prediction accuracy at simulation speeds of over 500 MIPS.

## 1 Introduction

Fast and accurate performance and power prediction at early design stages is one of the core challenges in computer system design. Being able to predict performance and power consumption of real-world application software and benchmarks running on a target processor that does not yet physically exist is

Xinnian Zheng · Lizy K. John · Andreas Gerstlauer
The University of Texas at Austin, Austin, TX 78712, USA
E-mail: xzheng1@utexas.edu, ljohn@ece.utexas.edu, gerstl@ece.utexas.edu

an essential part of agile hardware/software co-design flows. As the complexities of both software and hardware systems continue to grow, such prediction becomes increasingly difficult. At the same time, applications often exhibit significant power and performance variations, where estimation of time-varying program behavior can provide crucial information for optimization of bottlenecks in both hardware and software. The need for such fine-grain estimation further increases prediction challenges.

Cycle-accurate, cycle-approximate or functional virtual platforms and instruction set simulators (ISSs) [12, 17, 36] are widely used today to estimate performance and power consumption of application software executing on a hardware system. Such models can be very accurate, but simulation speeds are often prohibitively slow, especially with real-world applications. This severely limits the amount of exploration that software or hardware developers can perform. Analytical system models have been proposed as a fast alternative for modeling software performance and power consumption. However, models constructed via analytical techniques are often inaccurate and exclusively targeted at design space exploration for specific hardware and a given set of benchmarks, which inherently limits their usefulness. Moreover, such models still require execution traces or statistics obtained by running the actual workload in question on a partial ISS model [35] or a physical realization of a close micro-architecture variant [32, 33].

By contrast, real-world applications can be run on real machines to understand their performance characteristics. At the same time, it may be possible to run a few smaller benchmarks on a slow simulator or other reference model of a studied system. From basic intuition, we know that there exists some latent relationship between the execution of a program on two different platforms. Given a program $A$ that takes $t$ seconds to execute on a particular machine, we 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. An interesting question is thus whether a small number of example runs on a slow, detailed simulator or reference model and the corresponding runs on some other real hardware can give insight into the correlation between the two, and whether such correlation can be exploited by a machine learning framework to predict the performance and power of a platform $X$ using runs on another platform, say platform $Y$, without actually going through the slow, detailed simulations of $X$ itself. Platform $X$ can be a detailed full-system simulator, which is prohibitively slow, and platform $Y$ can be a fast machine, such as a state-of-the-art x86 workstation. Our goal is to provide a systematic way of extracting such underlying correlation and use it for novel *cross-platform prediction* methods that bridge the gap between traditional analytical modeling and simulation-based techniques.

Towards this goal, we present *LACross*, a learning-based, analytical cross-platform prediction framework that is capable of fast and accurate performance and power estimation at fine temporal granularities. Such accurate, fine-grained prediction can provide crucial hints for hardware and software developers in localizing potential performance and power hotspots of constantly

evolving applications executing on platform hardware that is also under development or not otherwise easily accessible (e.g., due to being proprietary or still being evaluated). Instead of using standard benchmark suites with limited significance leading to sub-optimal design or purchasing decisions, developers can run real applications on fast and accurate cross-platform prediction models that serve as *architecture proxies*. Such proxies trained on real implementations or pre-silicon reference models can be given to software developers by hardware manufacturers without needing to provide access to or exposing internals of hardware still under development. At the same time, architecture proxies trained on small micro-benchmarks allow hardware developers to evaluate time-varying behavior of large, real-world applications that may otherwise be too slow to run when only detailed pre-silicon simulators are available in early development stages.

In early work [48], we introduced a basic learning-based cross-platform prediction formulation that leverages hardware support for collecting performance counters on modern platforms to predict performance of programs running on a different platform. This prior work was limited to predicting performance of whole programs only. Temporal variations of program behavior were not captured and errors of more than 40% were shown for small embedded benchmarks. The main issue was lack of training set coverage, where prediction accuracies suffered for programs with performance patterns not covered by the training set. In [47], we extended our prior approach to predict both power and performance at the granularity of program phases. It is known that programs tend to exhibit more homogeneous behavior at the individual phase level [41]. With training and prediction at finer granularity and proper choice of phases, temporal variations in large-scale program behavior can be accurately captured and overall accuracy is significantly improved. The benefit of a phase-based approach is thereby not only in providing more data, but also in the fact that the quality of the data is better.

In this paper, we make the following additional contributions over our prior work: (1) we demonstrate phase-level power and performance prediction for additional host architectures, specifically showing that comparable prediction accuracy is achieved for an AMD Phenom II host with a smaller set of performance counters as supported on our original Intel machine; (2) we perform additional predictions from Intel to AMD and from AMD to Intel machines, showing that high accuracy is maintained even when predicting between different generations of x86 architectures with similarly high complexity; and (3) we extend the analysis of training set coverage using a technique known as latent semantic indexing to show the effects of using different phase granularities on training coverage versus homogeneity and their implications on accuracy.

The remainder of the paper is organized as follows: after an overview of our phase-based approach in Section 2, Section 3 surveys the related work. Section 4 describes the formulation of our performance and power prediction problem. Section 5 then discusses our experimental setup, and Section 6 presents empirical results of our cross-platform performance and power pre-

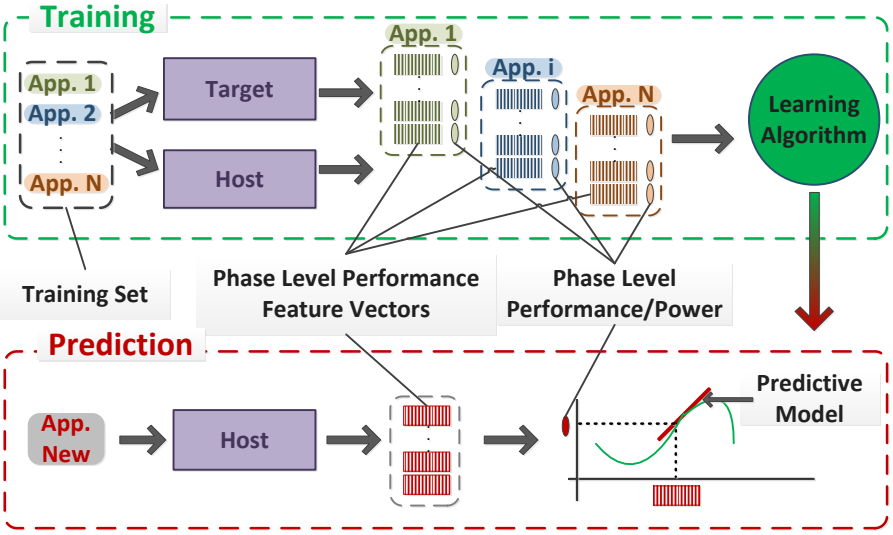



Fig. 1: LACross framework.

diction framework. Finally, Section 7 concludes with a summary of the key contributions and results of this work.

## 2 Overview of LACross

An overview of LACross is shown in Figure 1. The learning-based formulation of the performance and power prediction problem consists of two stages: a training stage and a prediction stage. During the training stage, a set of sample programs ("training set") are executed both on the host machine ("host") and a reference target model ("target"). The reference model could be either a simulator or real physical hardware, such as a development board. The target and host do not necessarily have to be of similar architectures. In fact, as our results will show, it is possible to achieve accurate prediction between targets and hosts that are of vastly different micro-architectures and instruction set architectures (ISAs).

For each workload we obtain, at phase level, various hardware performance features from the host as well as reference performance and power from the target. Execution statistics on the host are obtained non-intrusively using built-in hardware counters. Our goal is to extract the latent relationship between the host and target. We formulate this problem into a statistical learning setting, and derive prediction models for both performance and power on the target. During the prediction stage, a new application is executed only on the host. A set of performance features is obtained at phase level and used as inputs to the prediction model in order to produce an estimate of the performance and power on the target.

## 3 Related Work

Simulation-based and analytical models are two main techniques for performance and power prediction. Traditional simulation approaches estimate performance of a program by executing it on cycle-accurate or cycle-approximate ISSs [12, 17, 36]. Such approaches tend to be accurate but slow, as throughput of most ISSs is on the order of several hundred KIPS to several MIPS. FPGA-based acceleration [19] or source-level, host-compiled and transaction level modeling (TLM) techniques [14] have recently been proposed for improving simulation speed while trying to maintain accuracy close to an ISS. Throughput of these higher-level approaches is often around 200-500 MIPS including cache simulation, while accuracy is often above 90%. However, they typically involve cumbersome static and dynamic analysis to back-annotate source-level code or hardware models with simulated target performance estimates. For example, for each new application, the work in [18] requires pre-characterizing each possible pair of basic code blocks on a cycle-accurate reference simulator with subsequent back-annotation of path-tracking timing and energy models at the intermediate representation level. This introduces a substantial amount of code intrusion and development effort. Furthermore, with the effect of compiler optimizations and out-of-order hardware execution, such steps become difficult to statically track, limiting their accuracy. Pure source-level profiling approaches [16] utilize performance information from the source to predict performance of an application across platforms. However, this still requires source code modifications and, since only limited information is available at the source level, prediction models are often overly simplistic and inaccurate. By contrast, our approach treats the code execution of a program as a black box, and only requires a one-time training to construct a statistical model that can predict performance across arbitrary, unmodified test programs. With the proper choice of phase granularity, our approach is fast on average while providing better accuracies than detailed, back-annotation based approaches. Our training process has to be repeated for every change of the target platform, however. This limits retargetability compared to other approaches [18].

Analytical processor models [20, 40, 44] date back decades ago, with their main focus being on the evaluation and study of micro-architectural variations on pipeline and instruction level parallelism. More recently, Karkhanis *et al.* [29] extend prior works and applied those techniques to study superscalar processors. In recent years, statistical and regression-based methodologies started to thrive. Linear regressions are simple and widely used, but can suffer from overfitting and other problems [8]. More recently, advances in linear and nonlinear regression techniques have tried to address such problems, e.g. by adding a least absolute shrinkage and selection operator (LASSO) to linear regression [45] or by using general, non-linear but computationally expensive neural networks [23] or support vector machine regressions (SVRs) [43]. McCullough *et al.* [37] evaluated various regression methods for processor power characterization, showing that linear regression models can perform poorly under certain conditions. Bircher *et al.* introduced techniques using linear re-

gression for predicting power consumption from performance counters with good results [13]. Lee and Brooks proposed a predictive modeling and spatial sampling method [32, 33] for efficient micro-architecture design space exploration. They employed linear regression models to characterize different micro-architectures, navigate a large design space and identify all Pareto-optimal candidate architectures. Joseph *et al.* [27,28] also utilized regression-based approaches to construct processor performance models, where an iterative error minimization technique is applied to find the optimal fit. Similar ideas were also introduced by Ipek *et al.* [26] using artificial neural networks instead of regression models. Lee *et al.* [34] and Khan *et al.* [30] extended and generalized the predictive modeling 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 and power prediction by establishing analytical models that correlate two distinct architectures.

## 4 Learning Formulation

Many possible granularities for characterizing program phases have been proposed. Huang *et al.* [25] identify program phases at the granularity of subroutines and functions. Balasubramonian *et al.* [10] use conditional branch counts, whereas Sherwood *et al.* [41] use a granularity of 100 million instructions throughout the execution of a program to characterize program phases. Conceptually speaking, at the finest granularity, each instruction can be considered a separate program phase. By contrast, and at the most coarse granularity, an entire program can be considered as a single phase. Choosing the correct granularity in many cases depends on the use case and application. In our approach, we define the program phases in units of compiler basic blocks [9], and we study the effect of different granularities on prediction.

We apply a variant of a LASSO linear regression [45] to our performance and power prediction problem with two key differences: We impose extra constraints on the model parameters, and we perform linear regression on a phase-specific basis. We formulate our performance and power prediction problem as a piecewise constrained, locally sparse linear regression (PCLSLR), which extends the constrained locally sparse linear regression (CLSLR) proposed in [48]. For each workload, we obtain feature vectors consisting of selected hardware counter measurements for every program phase. We then apply a CLSLR to obtain a local linear prediction model specific to each program phase that correlates the host performance features and target timing and average power. The joint PCLSLR model for the whole program is simply the superposition of the models in each program phase.

Formally, for each program phase $j$, let $\hat{x}_j \in \mathbb{R}^d$ denote the performance feature vector obtained from the host, and $\hat{y}_j \in \mathbb{R}$ denote the cycles or power

from the reference simulator or hardware. The goal is then to extrapolate a mapping $\mathfrak{F} : \mathbb{R}^d \to \mathbb{R}$ such that for all $j$,

$$\mathfrak{F}(\hat{x}_j) \approx \hat{y}_j.$$

A popular approach is to assume that $\mathfrak{F}$ is a globally linear function, and thus formulate the problem as a standard linear regression. However, linear regression is known to suffer from problems like overfitting [8]. More importantly, due to the linearity assumption on the target function, this approach performs poorly when the underlying function is non-linear. In reality, and as will be shown in Section 6.1, performance features on one platform and performance/power on another platform follow an inherently non-linear relationship [37]. Instead of assuming that $\mathfrak{F}$ is globally linear, we only impose a differentiability assumption, i.e. that $\mathfrak{F}$ is differentiable everywhere in its domain. Although the function $\mathfrak{F}$ can no longer be expressed explicitly in closed form as in the ordinary linear regression case, this still allows us to approximate $\mathfrak{F}$ point-wise via a first-order locally linear approximation, which we denote as $\widehat{\mathfrak{F}}$.

Formally, given the feature vector $\hat{x}_j$ of a program phase $j$, let $\{(x_i, y_i), i = 1, \ldots, m\}$ be the set of $m$ performance feature vectors and reference performance or power pairs in the training set that are close to $\hat{x}_j$ based upon the following distance criteria,

$$\|\hat{x}_j - x_i\|_2 \leq \epsilon,$$

where $\epsilon$ is a parameter for determining the size of the local neighborhood of interest. Let $X \in \mathbb{R}^{m \times d}$ be the matrix that contains all the $x_i^T$ as its row vectors, and $Y \in \mathbb{R}^m$ be the column vector that contains all the $y_i$ as its elements. The CLSLR then solves the following optimization problem,

$$\begin{aligned} \underset{\theta_j}{\text{minimize}} \quad & \frac{1}{2m}\|X\theta_j - Y\|_2^2 + \lambda\|\theta_j\|_1 \\ \text{subject to} \quad & \theta_j \geq 0. \end{aligned} \tag{1}$$

The solution $\theta_j$ is then used as the parameters to the local linear approximation $\widehat{\mathfrak{F}}_j$ at $\hat{x}_j$, i.e., $\mathfrak{F}(\hat{x}_j) \approx \widehat{\mathfrak{F}}_j(\hat{x}_j) = \theta_j^T \hat{x}_j$.

Intuitively, the quadratic objective function in (1) aims to minimize the prediction error in each program phase by considering only the feature vectors in the training set that are close to $\hat{x}_j$ as determined by a $l_2$-distance threshold of $\epsilon$. Since feature vectors that are close to each other are more likely to exhibit similar performance patterns across different architectures, we impose the distance constraint such that only relevant feature vectors in the training set are considered when forming the prediction model for each phase. The $l_1$-norm constraint on the hyperplane parameter $\theta_j$ restricts solutions to be small in order to avoid overfitting, and the positivity constraint states that all the performance features on the host should contribute positively to the performance or power on the target.

Note that the optimization problem in (1) does not have an analytical solution. In fact, it belongs to a particular type of convex optimization problems

for which 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 methods [11, 39].

By solving (1), we obtain a 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 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. $\widehat{\mathfrak{F}}_j(\hat{x}_j) \not\approx \mathfrak{F}(\hat{x}_j)$) and the CLSLR is likely to give erroneous predictions.

In the CLSLR, we need to choose two tuning parameters, the sparsity penalty $\lambda$ and the parameter $\epsilon$ for controlling the size of the neighborhood. We employ a standard technique known as cross-validation [31] to determine their values. In particular, we randomly chose a subset of the original training dataset and divide it into a training subset and a test subset. We train using only data from the training subset, and we use data from the test subset to compute an average prediction error percentage. We iteratively repeat this process applying different values for $\lambda$ and $\epsilon$ until the cross-validation error is less than a threshold of 5%.

During prediction, the constructed models for all the unique per-phase feature vectors are cached, such that (1) does not need to be solved repeatedly for the same phase. Two feature vectors $\hat{x}_j$ and $\hat{x}_k$ are defined to be unique iff

$$\|\hat{x}_j - \hat{x}_k\|_\infty \geq T,$$

where the threshold $T$ is empirically chosen to be 200 with respect to the training set. A threshold of 200 is found to be enough for filtering out the inherent noise in processor performance counter based phase measurements.

## 5 Prediction Infrastructure

In order to verify the effectiveness of our cross-platform prediction framework, we base our experiments on measurements obtained on real hardware systems and software workloads. In the following, we describe our experimental setup, including the software workloads used as training set, as well as the host and target hardware platforms used for prediction, including data acquisition frameworks used to obtain hardware performance counters on the hosts as well as reference performance and power measurements on the targets.

### 5.1 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

Table 1: Breakdown of the training set.

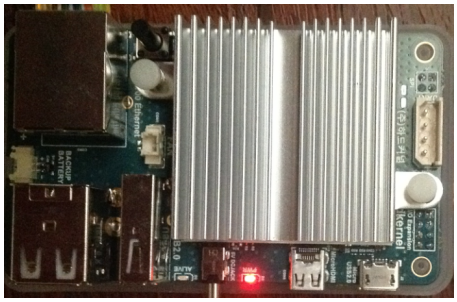
| **Application Domains** | **Number of Programs** |
|---|---|
| Simulation | 14 |
| Enumeration | 16 |
| String Manipulation | 30 |
| Graph Algorithm | 26 |
| Dynamic Programming | 21 |
| Geometry | 25 |
| Recursion | 13 |
| Miscellaneous | 12 |

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:

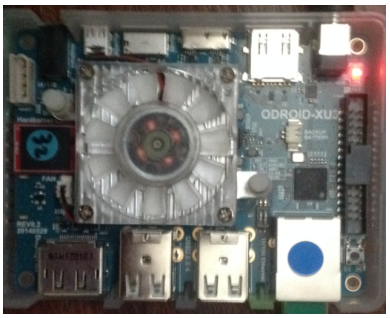
- Each workload inside the training set should individually be a good representative of the programs encountered during the later prediction phase.
- The variety of the workloads in the training set should be sufficiently large to cover the application space of interest.
- The overall number of program 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 157 diverse and representative programs from the ACM International Collegiate Programming Contest (ICPC) [6] database. The ACM-ICPC is the largest and most prestigious programming contest, where hundreds of programming problems each year are created to test participant knowledge on algorithms and programming as well as the ability to create new software applications. As such, the ACM-ICPC database provides a great resource for large-scale program mining. Table 1 shows the breakdown of the programs in our training set with respect to their related application domains. For programs in the simulation domain, they often involve step-wise replay of the given input sequence and produce outputs based upon some predefined rules. Emulating a play in a board game or tracing through a discrete event simulation are typical problems encountered in this domain. These programs are often computationally intensive. The enumeration domain contains programs that solve some form of combinatorial problem, where enumeration of all possible candidate solutions is required. As these problems often require large amount of storage for maintaining the search space, they are likely both computation and memory intensive. The string manipulation domain contains problems such as parsing, translation, encryption/decryption and other general text processing. The graph algorithm category includes programs that require manipulation of graph data structures. For instance, variations of computing the shortest path, graph search,



(a) ODROID-U3.



(b) ODROID-XU3.

Fig. 2: Reference target platforms.

connected components, and network flow problems are common representatives of programs in this domain. Many problems from the dynamic programming domain require manipulation of large multidimensional arrays to store partial solutions in order to compute future solutions. Both graph algorithms and dynamic programming problems are therefore memory intensive. In many cases, they also result in irregular memory access patterns due to pointer chasing code. Recursion problems often incur high branch miss rates, due to the frequent non-predictable call and return structures underlying the recursive algorithm. Finally, geometry and miscellaneous programs that solve numerical problems complete the training set. Programs in these domains often consist of large amounts of floating-point ALU operations and are thus compute bound.

The large variety of programs in the ACM-ICPC database resolves the representativeness and diversity requirements of the training set. We use original programs and inputs. In our earlier work [48], the size of the training set was artificially increased to improve coverage by creating 100 random inputs for each program. This is not necessary in our case. Profiling programs at phase granularity provides sufficient amount of training data, and no addition of artificial and possibly unrepresentative data is necessary. With our new phase-based approach, a small training set with low training overhead is sufficient to achieve high accuracy. Since training and prediction are performed at program phase level, the diversity of program phases is our primary concern. The program phases in the training set should ideally provide a good coverage of the program phases encountered during prediction. We study training set coverage and diversity in Sections 6.4 and 6.5.

### 5.2 Profiling and Measurement Infrastructure

We perform profiling of the training set on two different host platforms: an Intel Core i7-920 processor [2] with 24 GB of memory, and an AMD Phenom II X6 1055T processor [1] with 8 GB of memory.

To demonstrate effectiveness of our approach on state-of-the-art mobile and embedded target platforms, we employ physical hardware references as

| Intel Core i7-920 | AMD Phenom II X6 |
|---|---|
| L1 Total Cache Misses | L1 Total Cache Misses |
| L2 Total Cache Misses | L2 Total Cache Misses |
| L3 Total Cache Misses | Branch Misses |
| TLB Loads | Instructions |
| Unconditional Branches | Cycle Stalled |
| Conditional Branches | Cycles |
| Branch Misses | L1 Total Cache Accesses |
| Instructions | Floating Point Operations |
| Cycle Stalled | |
| Cycles | |
| L1 Total Cache Accesses | |
| L2 Total Cache Accesses | |
| L3 Total Cache Accesses | |
| Floating Point Operations | |

Table 2: Hardware performance counters profiled on the hosts.

targets for training and prediction. We use the ODROID-U3 (U3) development board [3] (Fig. 2a) to obtain reference performance measurements (cycle counts). The U3 board uses the Samsung Exynos 4412 SoC as its hardware platform. The Exynos 4412 SoC contains a homogeneous quad-core ARM Cortex-A9 processor with 32 KB L1 instruction and data cache. For reference power measurements, we use the ODROID-XU3 (XU3) development board [4] (Fig. 2b). It uses the Samsung Exynos 5422 SoC as the hardware platform. The Exynos 5422 SoC consists of a heterogeneous big.LITTLE CPU arrangement, which combines a Cortex-A15 and a Cortex-A7 processor cluster. The XU3 development board integrates an on-chip TI INA231 current sensor for power measurements of all eight cores, but the CPU hardware is restricted to not allow any performance counter measurements. Thus, two different boards serve as performance and power references due to hardware limitations associated with each board.

For our study, we are mainly interested in predicting performance and power for single-core workloads. Thus, all programs are restricted to run on one core till completion, which minimizes measurement noise due to core migration. On the XU3 board, we restrict the workloads to run solely on one of the A15 processors with DVFS disabled.

We use the PAPI toolset [38] for collecting various hardware performance counters on both the hosts and the targets. For power measurement, we developed a custom API that interacts directly with the onboard TI INA231 power sensor. As shown in Table 2, for each phase of the training programs we profile 14 and 8 hardware performance counters on the Intel and AMD host platforms, respectively.

Figure 3 shows the 14 hardware performance events that we collect on the Intel host machine using a granularity of 5,000 basic blocks, and the correlation coefficients between each individual event measured on the host and the performance of the A9 as well as the power of the A15 target. The number of CPU cycles, together with the number of instructions, the total number

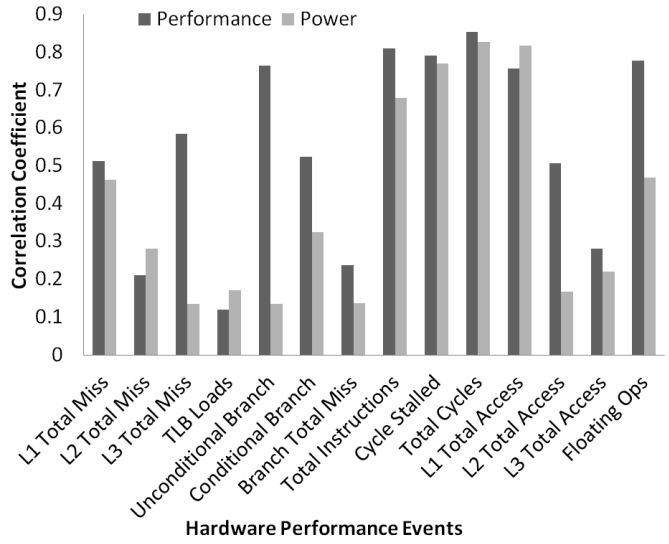



Fig. 3: Correlation of hardware performance events with target performance and power (Intel Core i7 920).

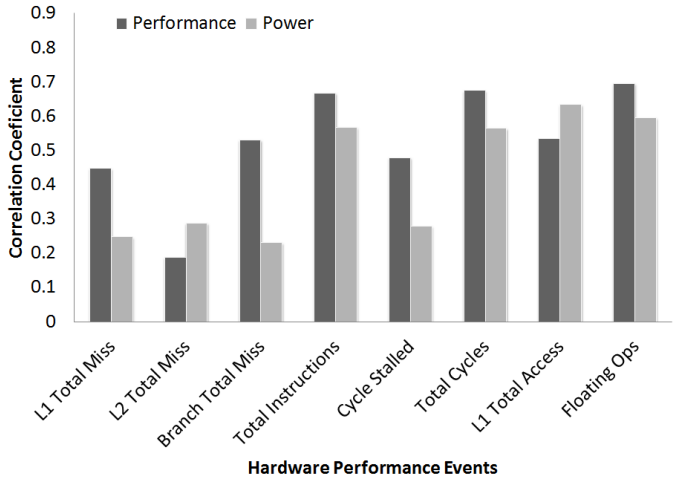



Fig. 4: Correlation of hardware performance events with target performance and power (AMD Phenom II X6 1055T).

of L1 cache accesses and the number of floating point operations appears to be highly correlated with the target timing. Other events (i.e, the 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. Nevertheless, the individual contributions of different counters can vary widely from application to application, and we include all 14 counters in our prediction to cover a wide range of potential behavior.

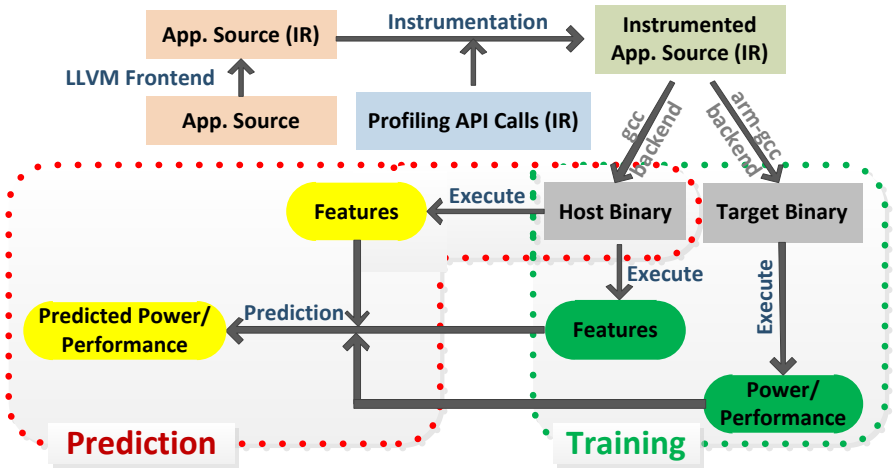



Fig. 5: Prediction framework.

Similarly, Figure 4 shows the 8 hardware performance events we collected on the AMD host machine at a granularity of 5,000 basic blocks 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 4, the majority of the 8 hardware events continue to show strong correlation with the target performance and power. Note that these are the performance feature vectors we denoted as $\hat{x}_j$ in our problem formulation (Section 4) for any given program phase $j$. The disparity in performance feature vectors between the two host machines will be discussed further in Section 6 to study its effect on prediction accuracy and speed.

### 5.3 Prediction Framework

Figure 5 shows the tool flow used in LACross. for fine-grain, phase-level profiling and prediction. We utilize the LLVM compiler framework [7] to instrument profiling API calls at intermediate representation (IR) basic block level of each program during compilation. The application sources are first compiled into LLVM IR and then instrumented and linked against the profiling API. The instrumented LLVM IR tracks the number of dynamic basic blocks executed by the program in order to log various counter and power measurements at the end of each program phase. Instrumentation at the IR level thereby guarantees that features and reference performance and power obtained for each phase correspond to the same execution on both the host and the target. During training, the instrumented IR is cross-compiled into host and target binaries. During prediction, we obtain the performance features on the host for each program phase, and we use previously collected training data to solve (1) and

predict per-phase performance and power. We use MATLAB 2013a as the main computation environment. The optimization problem (1) can be transformed into a quadratic program (QP) and solved using the built-in **quadprog()** function in MATLAB [5].

## 6 Experiments and Results

To show the effectiveness of LACross, we test it with 35 selected benchmark programs from three standard benchmark suites that are not encountered in the training set. We use 7 programs (**aes, crc, dijkstra, fft, patricia, qsort, sha**) from the MiBench suite [22], 9 programs (**disparity, localization, mser, multi_ncut, sift, stitch, svm, texture_synthesis, tracking**) from the San Diego Vision Benchmark Suite (SD-BVS) [46], and 19 programs (**perlbench, bzip2, gcc, mcf, milc, namd, gobmk, dealII, soplex, povray, hmmer, sjeng, libquantum, h264ref, lbm, omnetpp, astar, sphinx3, xalancbmk**) from SPEC CPU 2006 [24]. We chose the 19 programs from SPEC implemented in C/C++ as we use the C/C++ interface provided in PAPI to instrument counter profiling calls. We use the "large", the "fullhd" and the "ref" input set for programs from MiBench, SD-VBS and SPEC CPU 2006, respectively.

These 35 programs are first profiled on the Intel and AMD hosts to obtain 14-dimensional and 8-dimensional hardware performance feature vectors at program phase level using the PAPI toolset. We then conduct performance and power predictions using previously trained models for the U3 and XU3 boards, respectively. In order to study the effect of phase granularity on the prediction accuracy and speed, we perform our experiments at different phase sizes. Note that for power prediction, due to hardware constraints on the sampling rate of the TI INA 231 current sensor on the XU3 development board, the smallest phase granularity used is 20,000 basic blocks. The sampling speed is inherently limited by the ADC conversion speed of the sensor. From our experiments, we found a sampling period of approximately 20,000 basic blocks to be the fastest the sensor hardware could support.

### 6.1 Cross-Validation Error

We first evaluate the accuracy of the statistical models in terms of their cross-validation error. We employ 10-fold cross-validation [31] of the training set as an estimate of the generalization error of different regression techniques. The comparison is shown in Figure 6. This experiment is performed at a phase granularity of 20,000 basic blocks.

A simple LASSO linear regression [45], which assumes global linearity of the underlying data results in more than 20% cross-validation error for both performance and power prediction. By contrast, the PCLSLR yields only a 2% average prediction error. The fact that the non-linear PCLSLR technique

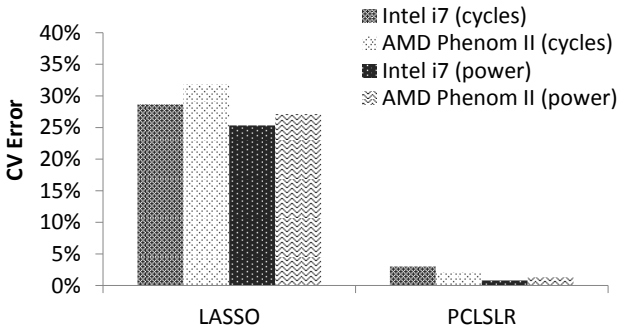



Fig. 6: Cross-validation error for phase-level performance and power prediction (phase granularity = 20,000 blocks).

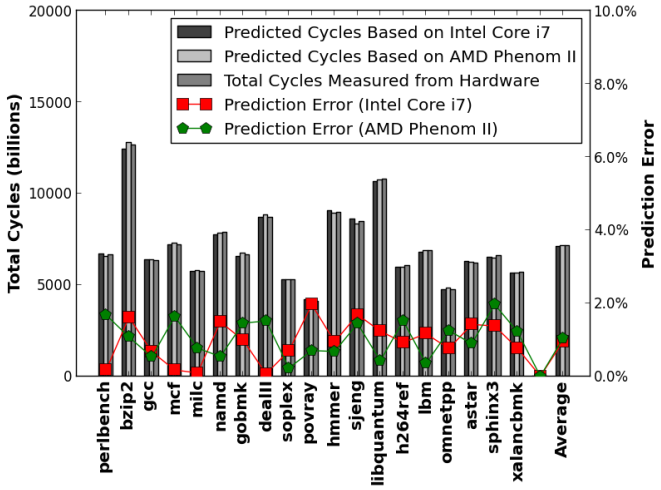
performs an order of magnitude better than the linear model strongly suggests that the underlying relationship $\mathfrak{F}$ between the hardware performance counters on the hosts and performance and power on the ARM target is inherently non-linear. Our prior work [48] demonstrated similar results at whole program granularity. The accurate prediction of PCLSLR also provides us with insights into the inherent nature of the data set. It reassures us that the assumption about the smoothness of the target function $\mathfrak{F}$ is indeed valid empirically.

### 6.2 Overall Prediction Accuracy and Speed

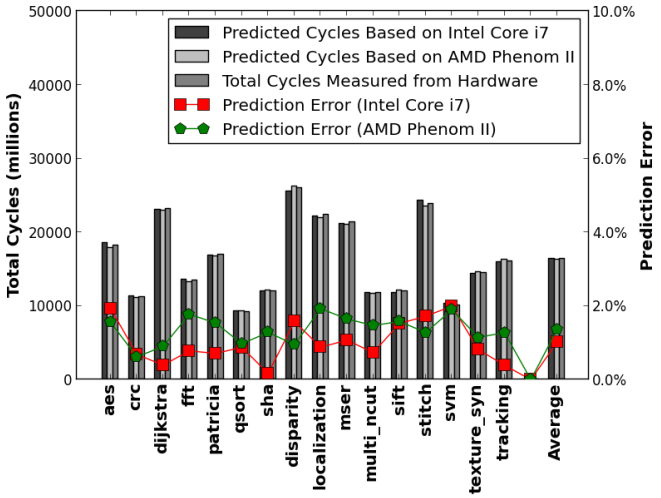
Figure 7 shows the accuracy of predicting whole program performance for the 35 test programs profiled at a phase granularity of 5,000 basic blocks. The predicted cycles of the 35 benchmark programs are very close to the actual cycle measurements obtained on physical hardware. The worst-case prediction error is around 2% for both the Intel and the AMD hosts, with average errors less than 1%. Note that the accuracy we refer to here is the percentage prediction accuracy of whole program performance.

Figure 8 similarly shows the overall program-wise power prediction accuracy of the 35 programs from the test set, profiled at a phase granularity of 20,000 basic blocks. The average error for predicting average power over whole programs is about 2.5%, while the worst-case prediction error is less than 10%.

The total runtime of each test program, as shown in Figure 9 consists of the profiling time and the prediction time. The profiling time is the time it takes to collect counters on the host. Due to hardware limitations on the simultaneous number of counters that can be collected, this requires 5 separate runs of each program to collect 14 features on the Intel host and 3 runs each on the AMD to collect 8 counters. The prediction time measures the total duration of solving the optimization problem (1) for each phase of the program. Solving time is governed by the dimension of data matrix $X$ and of the neighborhood



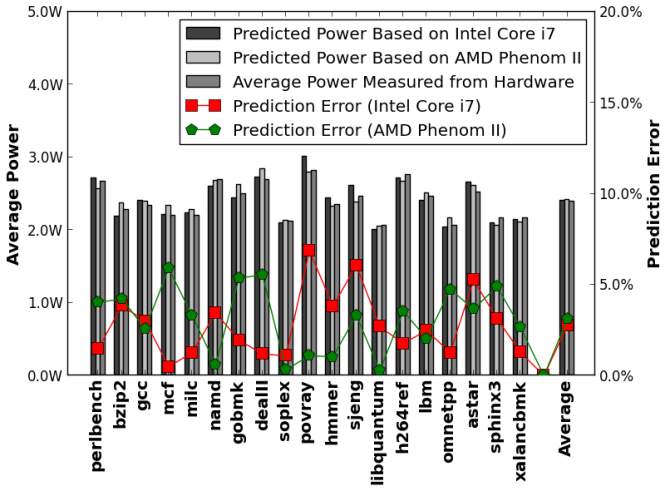

(a) 19 SPEC CPU programs



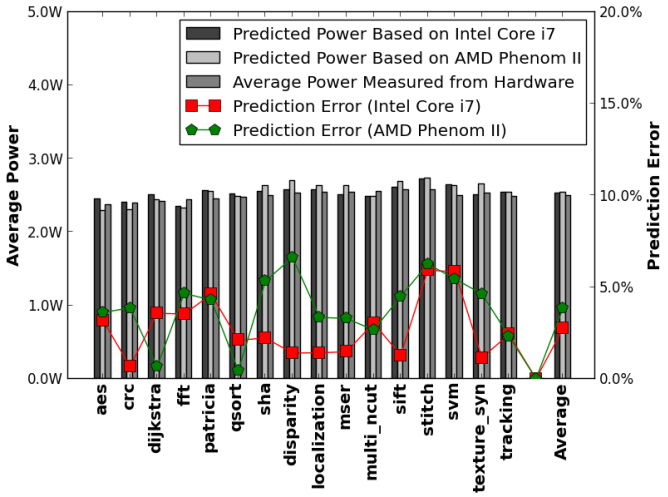

(b) 16 MiBench and SD-VBS programs

Fig. 7: Predicted target cycles and prediction accuracy of 35 benchmarks (phase granularity = 5,000 blocks).

defined by distance threshold $\epsilon$. For the same phase granularity, comparing runtimes of the SPEC programs (Fig. 9b) with the MiBench and SD-VBS programs (Fig. 9a), we see that as programs execute longer, the number of dynamic phases grows proportionally, which results in more time spent in solving the optimization problem (1). As phase granularity increases from 5,000 blocks to 20,000 blocks, the average total runtime generally decreases due to a decrease in both the profiling and prediction time. As the sampling



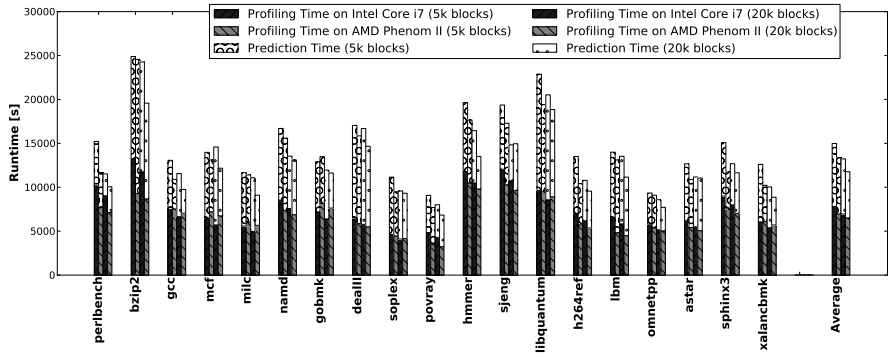

(a) 19 SPEC CPU programs



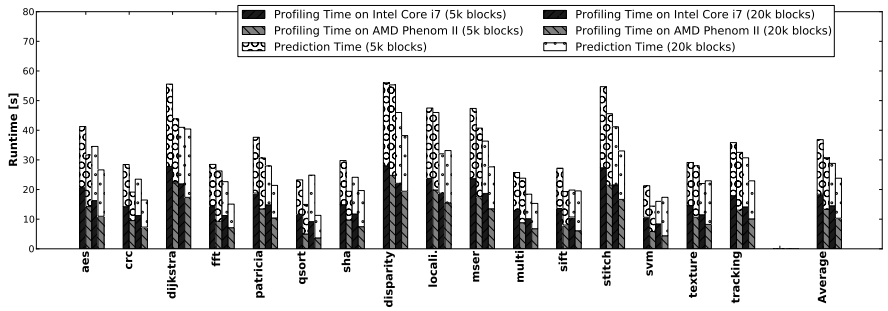

(b) 16 MiBench and SD-VBS programs

Fig. 8: Predicted target power and prediction accuracy of 35 benchmarks (phase granularity = 20,000 blocks).

granularity of program phases becomes larger, the total number of dynamic phases decreases accordingly, and thus fewer instances of the CLSLR problem need to be solved for each program. Note that prediction complexities vary across applications and hosts due to differences in the convergence speed of solving (1) numerically. At the same time, the sampling of the performance counters via the PAPI toolset also incurs performance overheads [15]. Hence, as the number of dynamic phases decreases, the overhead of profiling also




(a) 19 SPEC CPU programs.




(b) 16 MiBench and SD-VBS programs.

Fig. 9: Runtime of 35 benchmarks

becomes smaller. Profiling times vary across hosts due to differences in host performance and the number of runs required.

We further demonstrate an example of fine-grained dynamic power and performance tracing. Figure 10 and 11, show the dynamic behavior of executing the **dealII** benchmark on the predicted and real targets. Here, we use a phase granularity of 20,000 basic blocks, and results show that the prediction tracks accurately against performance and power measurements obtained from the hardware.

In addition, Figure 12 demonstrates the accuracy of performance prediction across the two x86 host machines for the 16 MiBench and SD-VBS programs. The predicted cycles of the test programs at whole program level remain very close to the actual cycles measured on the hardware, even when predicting between different micro-architectures of similar complexity.

### 6.3 Phase Granularity Tradeoffs

As indicated above, the choice of phase granularity will influence prediction accuracy and speed. A finer phase granularity can potentially increase training

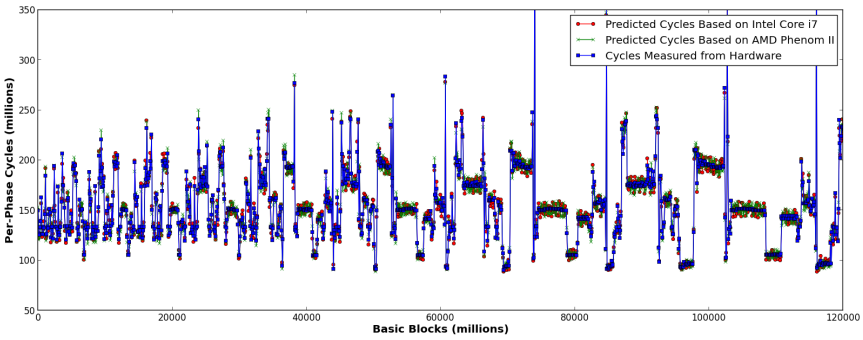



Fig. 10: Fine-grained performance behavior of **dealII** on U3 target.

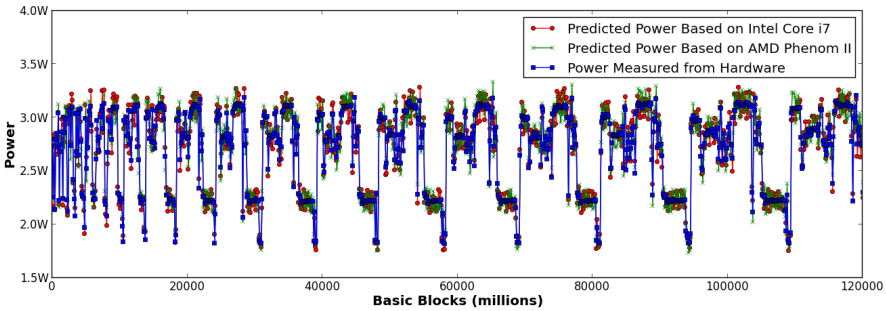



Fig. 11: Fine-grained power behavior of **dealII** on XU3 target.

set coverage and thus improve accuracy. Finer phase granularity, however, also requires more frequent profiling and prediction. We further study the tradeoff between prediction accuracy and speed with respect to different choices of phase granularity.

To measure simulation speed, we use the total number of dynamic instructions in a target program divided by the total time it takes to obtain the predicted performance and power on the host (i.e, profiling of the performance features vectors plus the time spent on solving the CLSLR (1) for all phases of the program). To measure accuracy, we use the mean absolute percentage error (MAPE) between the prediction and the actual measurement across all phases and programs.

As shown in Figure 13, the overall accuracy and speed for predicting workload performance varies significantly with respect to the choice of program phase granularity. At a granularity of 5,000 basic blocks, the per-phase performance prediction accuracy is about 92% on both hosts. Note that this is worse than the prediction accuracy across whole programs we have seen in Section 6.2 and Figure 7b. Due to averaging effects when aggregating all the phases of an entire program, overall performance accuracy is considerably higher.

At a granularity of 500 blocks, phase-level predictions are about 95% accurate as compared to real hardware measurements. As the phase granularity

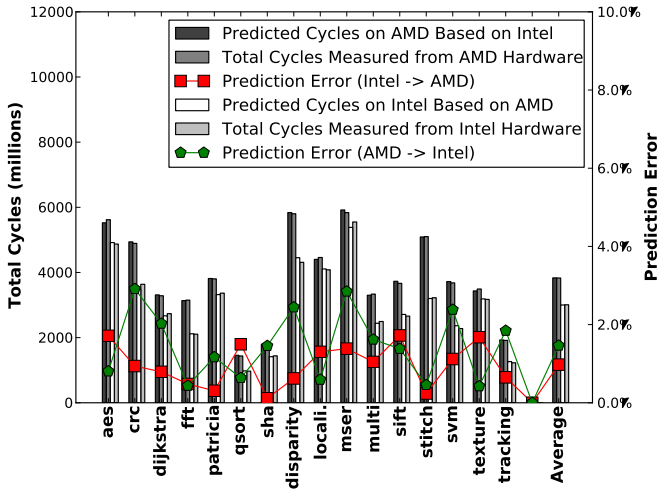
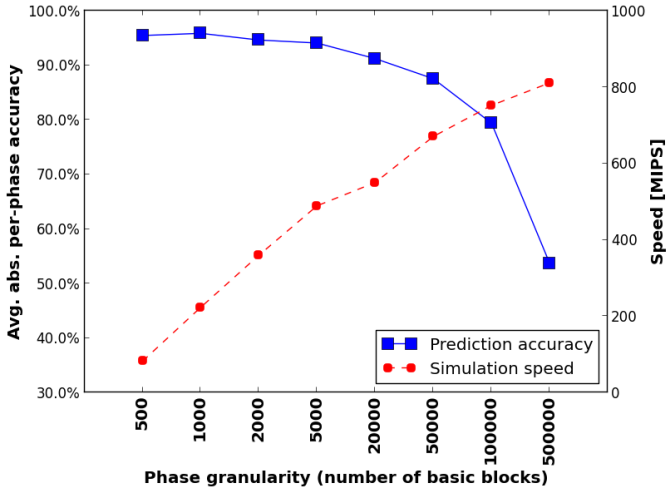



Fig. 12: Predicted target cycles and prediction accuracy across the host machines (phase granularity = 5,000 blocks)

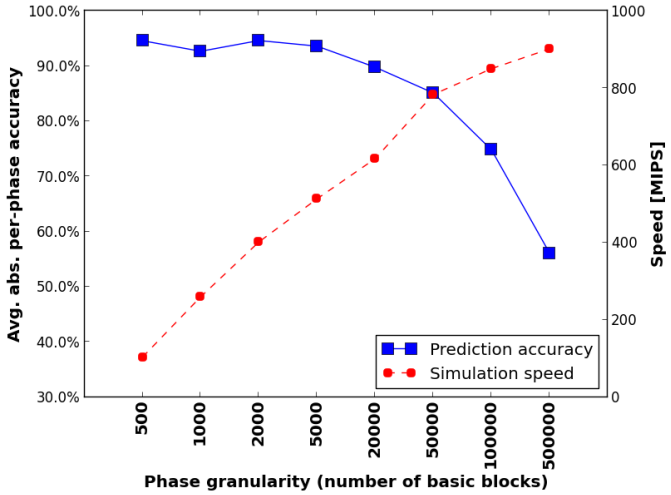
gradually increases from 500 to 20,000 basic blocks, the prediction accuracy only experiences a minor decreasing trend. The diminishing returns at finer and finer granularities are likely due to the inherent noise of performance counter measurements at very small sampling periods. Such noisy measurements can deviate the PCLSLR from the nominal target function. When the phase granularity grows beyond 50,000 basic blocks, however, the prediction accuracy drops drastically due to a lack of coverage in the training data. This is consistent with our earlier work [48], where large errors were seen when performing predictions at whole program level despite a much larger training set.

At the same time, the simulation speed improves proportionally with a decrease in phase granularity. As such, there is an optimal tradeoff between speed and accuracy at medium granularities. Note that for the same phase granularity, the average prediction speed obtained from the AMD host is observed to be slightly faster than the Intel host. This is due to the fact that only 3 profiling runs are required as opposed to 5 on the Intel host.

For power prediction on the two hosts (Figure 14), a similar tradeoff is observed. Our method, however, is still accurate (over 90% phase-wise accuracy compared to real hardware measurements) and fast (over 500 MIPS) on both hosts. Again, due to error cancellation effects, at the same phase granularity, the accuracy of predicting average power for a whole program is significantly better than the average absolute per-phase prediction accuracy.




(a) Intel host to U3 target.

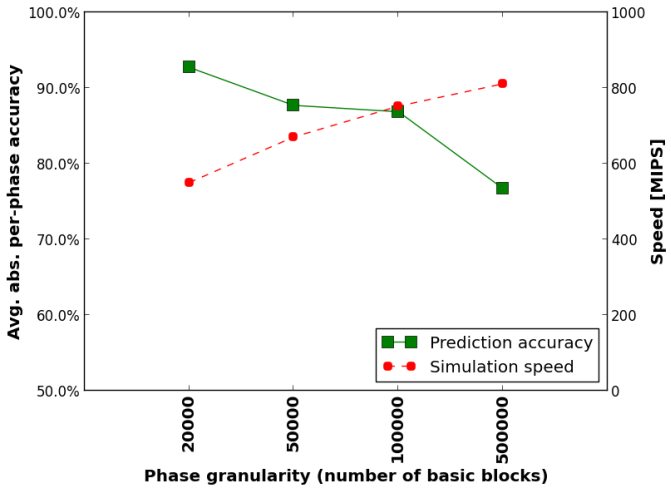



(b) AMD host to U3 target.

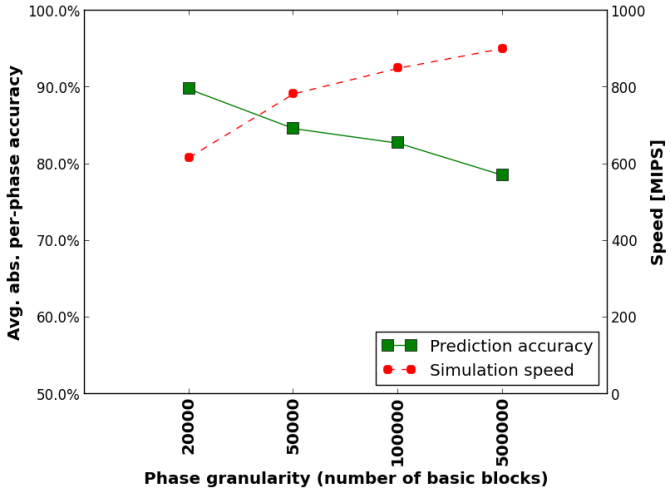
Fig. 13: Speed and accuracy tradeoff (performance prediction).

## 6.4 Training Set Coverage

In the following, we further study coverage of the training set and its effect on prediction accuracy with respect to the space of feature vectors. We use a dimensionality reduction technique known as principal component analysis (PCA) [42] based on Singular Value Decomposition (SVD) to extract and visualize the latent semantics in the data and to project the 14- and 8-dimensional feature space into a lower-dimensional and thus easier to comprehend representation. We choose to keep so-called principle components (PCs) representing



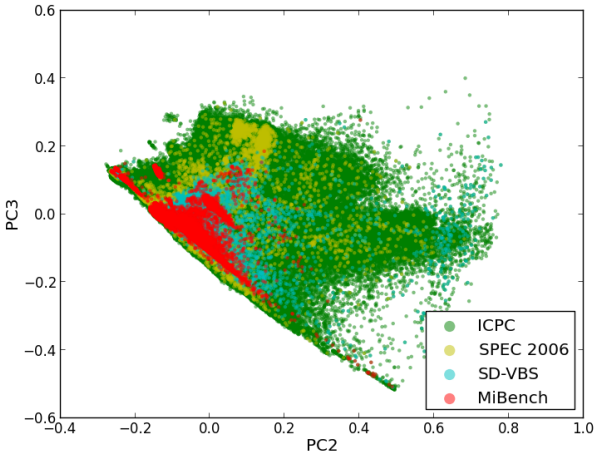

(a) Intel host to XU3 target.



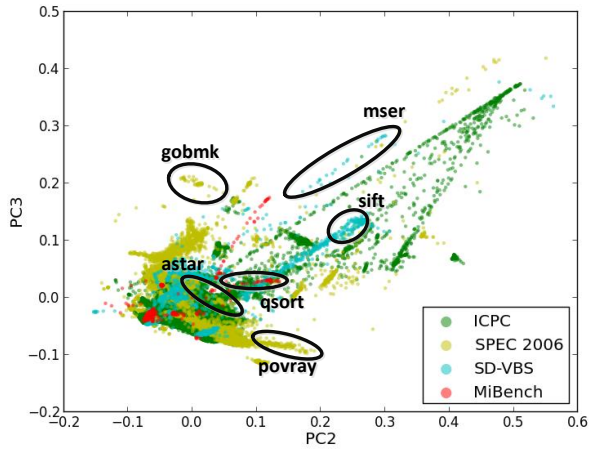

(b) AMD host to XU3 target.

Fig. 14: Speed and accuracy tradeoff (power prediction).

the 3 most dominant linear combinations of features, which in all cases cover more than 90% of the sum of all singular values. For the purpose of demonstration, we only plot the 3D-projected data onto the plane spanned by the second and third principal component, as they are visually more informative. Plotting the projected data over the first principal component results in a thin elliptic band that does not allow useful conclusions to be drawn. This effect is common when the data is well correlated and properly normalized [21].



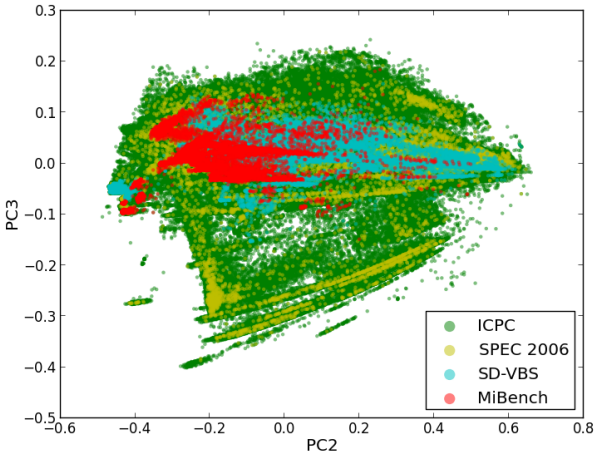

(a) Phase granularity = 500 basic blocks



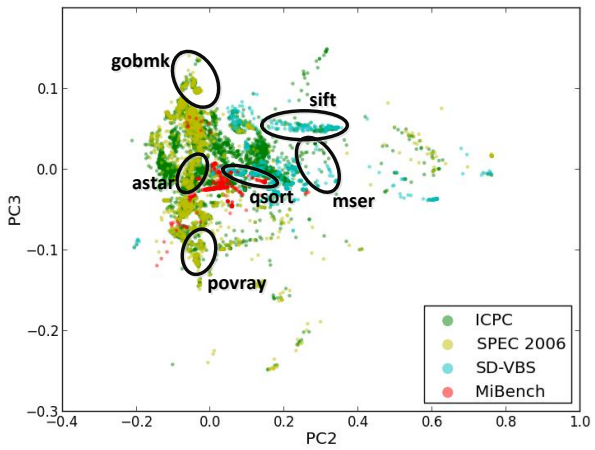

(b) Phase granularity = 500,000 basic blocks

Fig. 15: 2D feature space projection into principle components (Intel).

Figure 15 shows the 2D projection of the phase-by-phase feature vectors into their PC2 and PC3 components obtained by executing all programs in the training set as well as the 35 test programs on the Intel host. Closeness of the projected features points preserves the closeness of the original feature vectors in higher dimensions, which implies a similarity in the performance patterns of program phases. At a small phase granularity (500 blocks, Figure 15a), performance patterns obtained from the program phases in the training set provide a strong coverage of the program phases observed in the test programs.




(a) Phase granularity = 500 basic blocks




(b) Phase granularity = 500,000 basic blocks

Fig. 16: 2D feature space projection into principle components (AMD).

This assures that the local linear models we derive retain a good approximation of the target function $\mathcal{F}$ at those feature vectors that are seen in the test programs. At more coarse phase granularity (500,000 blocks, Figure 15b), the coverage of the feature space is no longer guaranteed. Many feature vectors from the test programs (such as **mser, sift, gobmk** or **povray**) start to have fewer close neighbors. As such, the local linearity assumption in the PCLSLR formulation begins to break down. However, not all programs suffer from the coverage problem. As shown in Figure 15b, test programs such as **qsort** or

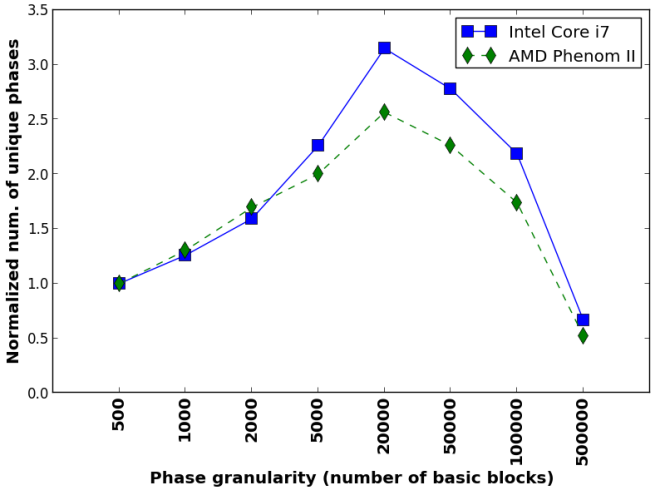



Fig. 17: Total number of unique phases.

**astar** still have a sufficient amount of nearby training data despite the sparse feature space.

Similar results are seen for the 8-dimensional feature vector projections obtained by executing the training set on the AMD Phenom II machine (Figure 16). The coverage decreases as the phase granularity becomes coarser. Prediction accuracy at large phase granularities also suffers due to the breakdown of the local linearity assumption of the PCLSLR.

Overall, an effect of training set coverage on accuracy can be observed. When there is a sufficient amount of data in the feature space that lies close the input points of interest, the CLSLR produces accurate predictions. Conversely, if existing data lies far away from the inputs, the locally linear model may not be able to capture the underlying behavior of the non-linear target function, which leads to inaccuracies in the prediction. Results confirm that by performing more fine-grained profiling at the level of program phases, we effectively mitigate the issue of lack of coverage in the training, and thereby are able to achieve significantly better prediction accuracy overall.

### 6.5 Program Phase Homogeneity

Results above have shown that prediction accuracy correlates closely with training set coverage. However, it is not clear whether the training set coverage is simply from having more training data when sampling at a finer granularity or as a result of the homogeneous nature of performance patterns in smaller program phases. To help clarify this question, we show in Figure 17 the total number of unique dynamic program phases obtained on both hosts across all training set programs as a function of phase granularities. As the phase granularity increases, the total number of unique phases increases. This indicates

that more diverse performance patterns emerge as increasingly larger chunks of a program are encapsulated in one phase. Conversely, as program phases become more fine-grained, even though the total number of phases encountered increases, the total number of unique program phases decreases. This indicates that the improvement in the feature space coverage of the training set at smaller granularities is due to an increase in the homogeneity of performance patterns as phases becomes smaller. In other words, there are fewer distinct program patterns out of which programs are composed at smaller phases.

Note that as granularity continues to increase, the total number of unique phases would also be expected to continue to increase. However, since all programs complete their execution in a finite amount of time, the number of dynamic phases is finite. Thus, the total number of unique phases eventually decreases when the phase granularity becomes too large. This agrees with the downward trend observed in Figure 17 for phase granularities greater than 20,000.

## 7 Summary and Conclusions

This paper proposes LACross, a learning-based framework for fast and accurate phase-level cross-platform prediction of performance and power of a workload running on a target machine. The key idea behind LACross is the simple observation that performance and power consumption of an application running on two different platforms are correlated, even if the two platforms are of vastly different architectures. We employ a unified learning-based formulation for the problem of both cross-platform performance and power prediction. We further show that constructing prediction models at program phase level helps to resolve training set coverage issues and thus increases accuracy. With proper choice of phase granularity, the prediction achieves over 97% accuracy at speeds over 500 MIPS for both performance and power. This is orders of magnitude faster than traditional cycle-accurate simulation, with the drawback that it requires application source code to be available. Predictions based on an x86 host can thereby run at almost the same speeds as executions on physical ARM hardware. With better counter support in the hosts, prediction speeds could be even higher. Overall, results of this work demonstrate the use of advanced machine learning methods for fast and highly accurate computer system performance modeling and evaluation. Resulting prediction models can serve as fast and accurate architecture proxies that enable exploration of pre-silicon hardware designs using large-scale real-world applications while simultaneously supporting independent software development for such evolving architectures.

## Acknowledgments

This work has been supported by Semiconductor Research Corporation (SRC) grant 2012-HJ-2317.

## References


1. AMD Phenom II Processor. http://www.amd.com/en-us/products/processors/desktop/phenom-ii.
2. Intel Core i7-920 Processor. http://ark.intel.com/products/37147/Intel-Core-i7-920-Processor-8M-Cache-2_66-GHz-4_80-GTs-Intel-QPI.
3. ODROID U3 Development Board. http://www.hardkernel.com/main/products/prdt_info.php?g_code=g138745696275.
4. ODROID XU3 Development Board. http://www.amd.com/en-us/products/processors/desktop/phenom-ii.
5. Quadratic programming - MATLAB quadprog . http://www.mathworks.com/help/optim/ug/quadprog.html.
6. The ACM-ICPC International Collegiate Programming Contest. http://icpc.baylor.edu/.
7. The LLVM Compiler Infrastructure. http://llvm.org/.
8. Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin. *Learning From Data*. AMLBook, 2012.
9. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
10. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.
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. 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.
13. W. Bircher, M. Valluri, J. Law, and L. John. Runtime identification of microprocessor energy saving opportunities. In *ISLPED*, 2005.
14. 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.
15. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.
16. L. Cai, A. Gerstlauer, and D. Gajski. Retargetable profiling for rapid, early system-level design space exploration. In *DAC*, 2004.
17. T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.
18. S. Chakravarty, Z. Zhao, and A. Gerstlauer. Automated, retargetable back-annotation for host compiled performance and power modeling. In *CODES+ISSS*, 2013.
19. D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *MICRO*, 2007.
20. 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.
21. G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, 1996.
22. 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.
23. S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 2nd edition, 1998.
24. J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

25. M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A framework for dynamic energy efficiency and temperature management. In *MICRO*, 2000.
26. E. Ipek and S. A. Mckee. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, 2006.
27. P. Joseph, K. Vaswani, and M. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. 2006.
28. P. J. Joseph. A predictive performance model for superscalar processors. In *MICRO*, 2006.
29. T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, 2004.
30. S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. Using predictive modeling for cross-program design space exploration in multicore systems. In *PACT*, 2007.
31. R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, 1995.
32. B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In *HPCA*, 2007.
33. B. C. Lee and D. M. Brooks. A tutorial in spatial sampling and regression strategies for microarchitectural analysis, 2007.
34. B. C. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models, 2008.
35. S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*, 2009.
36. 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.
37. J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX*, 2011.
38. P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *DoD HPCMP*, 1999.
39. Y. Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103:127–152, 2005.
40. D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *MICRO*, 1994.
41. T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, 2001.
42. J. Shlens. A tutorial on principal component analysis. *CoRR*, abs/1404.1100, 2014.
43. A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222.
44. 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.
45. R. Tibshirani. Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.
46. 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.
47. X. Zheng, L. K. John, and A. Gerstlauer. Accurate phase-level cross-platform power and performance estimation. In *DAC*, 2016.
48. X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer. Learning-based analytical cross-platform performance prediction. In *SAMOS*, 2015.