P^4: Phase-Based Power/Performance Prediction of Heterogeneous Systems via Neural Networks

Yeseong Kim*, Pietro Mercati*, Ankit More**, Emily Shriver**, Tajana Rosing*

* *University of California San Diego* {yek048, pimercat, tajana}@ucsd.edu

** *Intel Corporation* {ankit.more, emily.shriver}@intel.com

***Abstract*—The emergence of Internet of Things increases the complexity and the heterogeneity of computing platforms. Migrating workload between various platforms is one way to improve both energy efficiency and performance. Effective migration decisions require accurate estimates of its costs and benefits. To date, these estimates were done by either instrumenting the source code/binaries, thus causing high overhead, or by using power estimates from hardware performance counters, which work well for individual machines, but until now have not been accurate for predicting across different architectures. In this paper, we propose P^4, a new Phase-based Power and Performance Prediction framework which identifies cross-platform application power and performance at runtime for heterogeneous computing systems. P^4 analyzes and detects machine-independent application *phases* by characterizing computing platforms offline with a set of benchmarks, and then builds neural network-based models to automatically identify and generalize the complex cross-platform relationships for each benchmark phase. It then leverages these models along with performance counter measurements collected at runtime to estimate performance and power consumption if it were running on a completely different computing platform, including a different CPU architecture, without ever having to run it on there. We evaluate the proposed framework on four commercial heterogeneous platforms, ranging from X86 servers to mobile ARM-based architecture, with 129 industry-standard benchmarks. Our experimental results show that P^4 can predict the power and performance changes with only 6.8% and 5.6% error, respectively, even for completely different architectures from the ones applications ran on.**

***Index Terms*—Power prediction, performance prediction, phase recognition, neural networks**

I. Introduction

With the emergence of the Internet of Things (IoT), application tasks can run on many different platforms (e.g., X86 Xeon server vs. ARM-based mobile device) and at varying operating conditions (e.g., CPU frequency, sleep states) [1]. In these computing ecosystems consisting of heterogeneous devices, dynamic management and task mapping to meet diverse objectives cannot be done without an accurate way to estimate the performance/power costs and benefits.

Much research has been conducted to build the power and performance models targeting a single machine [2, 3, 4]. A basic assumption of the existing modeling techniques is that the power level is proportional to the workload intensity, such as the amount of computation and memory accesses. Despite much research over the last decades, predicting power behavior across multiple heterogeneous machines still remains a difficult problem since application behavior significantly varies as a function of CPU architecture, platform design, and runtime conditions. In our analysis, power consumption to run the same application varies more than 100**x** between two CPU architectures. Power consumption of two different applications running on a single machine can be more than 3 times different. Thus, to perform the prediction tasks successfully, it is essential to profile machine-independent application behavior at a fine granularity and to identify the general cross-platform relationships given the particular workload's behavioral patterns. A recent work presented an accurate cross-platform prediction technique by identifying workload behavior at the level of basic blocks [5]. Their technique requires application source code to identify these basic blocks at compile time, which may not be always possible, thus limiting its applicability.

This work was supported by Intel Corporation and NSF grant 1527034.

In this work, we propose a novel Phase-based Power and Performance Prediction technique (P^4) for heterogeneous computing ecosystems. P^4 utilizes *application phases* to characterize fine-grained system behavior on different operating platforms. *Application phase* is defined as an execution period which homogeneous system usage behavior is observed. Unlike earlier work [5, 6] that depends on either source code or binary instrumentation, P^4 can automatically recognize the application phases in a non-intrusive way by using only Performance Monitoring Counters (PMC) which are readily available in the existing systems. P^4 characterizes each machine offline by running a small subset of benchmarks that allow us to develop power and performance models of each hardware (HW) platform, and to identify application phases in the benchmarks. With the application phases and the individual machine characterization, we train neural network models to generalize the cross-platform relationships of the workload characteristics for each phase. The neural network models accurately predict how much power and how fast an application will run on a platform that is completely different from the one it is currently running on. Online, when a machine is running an application that has not been characterized before, and a decision if to migrate to a different system needs to be made, our P^4 framework uses the models to estimate how much power and performance that application would have if it were migrated to a completely different system.

We evaluate the proposed technique on four completely different platforms/architectures (e.g., x86 Xeon E5440 vs. x86 Westmere vs. ARM Cortex A15) and scales (e.g., servers vs. mobiles), with 129 industry-standard benchmarks. The experimental results show that P^4 can automatically identify the workload phases, and predict time-variant power consumption and performance with only 6.8% and 5.6% error, on completely different architectures running different numbers of threads, such as Intel x86's eight to ARM's four threads.

II. Related Work

Most power models in literature assume linear relationship between power and performance monitoring counters [2], as summarized in the following surveys [3, 4]. A number of publications have explored estimates of power consumption when

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

a machine changes C and P power states, rarely change for different power states in a single machine, and developed a linear regression model to estimate power and performance. Similar techniques have been proposed for frequency changes [7] and for cores in heterogeneous multicore systems [8]. Due to increasing complexity and heterogeneity of architectures, the cross-platform workload behavior cannot be estimated by only relying on the assumption of the linearity between the PMC events to the power consumption, and thus it requires a more sophisticated approach to get the desired level of accuracy.

A promising way to better understand the workload behavior across architectures is to exploit fine-grained application phases. Different sections of a program execution show distinct power characteristics [6, 9]. The phase detection technique presented in [10] identified the application status by tracking function calls of Java mobile applications to design a phase-driven power management which applies different CPU frequencies to improve the energy efficiency. Work in [11] inferred the phases from system events of mobile device subcomponents, such as CPU and GPS to detect abnormal power behavior. Another work modeled the phases of HPC applications based on MPI-specific APIs to automatically generate parallel benchmarks [12]. Based on their phase identification algorithm, they developed phase-aware power management mechanism for multicore systems. Work in [13] proposed a thread scheduling technique which finds stable phases based on performance counters and migrates threads when a stable phase is finished to reduce long memory latencies.

Work in [5] recently showed that the phase information is useful to predict cross-platform power levels. They proposed a technique which identifies the phases during compile time at the level of basic blocks. In contrast, our technique is different from their cross-platform power prediction technique in that we extract the application phases by monitoring only HW performance counters, which are available on most existing processor architectures. This enables our prediction to be independent of a priori knowledge of applications such as program basic blocks, application source code or binaries.

III. Overview of P^4

Figure 1 shows an overview of our proposed P^4 framework. The framework characterizes power/performance tradeoffs of each machine of interest, and develops models to estimate power consumption of the machine and application phases observed in the benchmarks. It then formulates the model for prediction of power consumption and performance across different HW configurations. The offline characterization is done by running a set of benchmarks while collecting PMC data and measuring the power consumption on the multiple HW platforms of interest.

The first single-machine characterization step is performed by two modules, the *event-based power estimation* and *application phase extraction* module. The *event-based power estimation* module (Section IV-A) takes the measured power consumption and the performance counter events as inputs. The module automatically selects strongly power-related events among all collected events using *Lasso* analysis and builds a single-machine power estimation model for each platform. The selected events are used as key parameters for the further learning stages, i.e., *application phase extraction* and *cross-platform prediction*. The application phase extraction module identifies application phases from the selected events

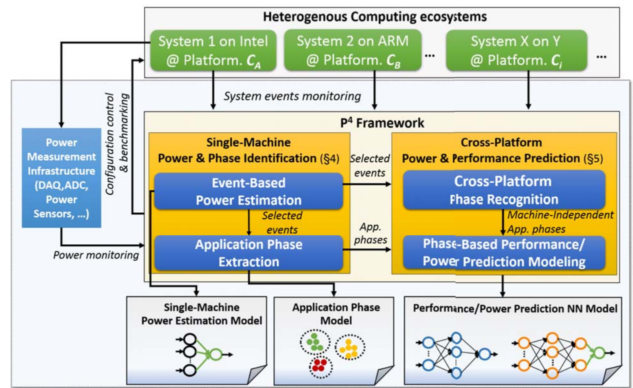


Fig. 1: An overview of the P^4 framework

by utilizing an automated unsupervised clustering procedure (Section IV-B). The goal of the cross-platform prediction is to train power/performance prediction models that are later used online. P^4 utilizes the phases as a basic unit to understand cross-platform relationships of application tasks. Since the application phases are identified for a single machine in the first step, we further identify the same phases across platforms through a phase matching procedure (Section V-A). Then, we can relate per-phase event behavior between platforms, and generalize the relationship by training two types of neural network (NN) models (Section V-B), which are designed to predict cross-platform power and performance behaviors. Combining the neural network models with the power estimation model, we can predict time-variant power consumption of arbitrary programs at runtime (Section V-B2).

IV. Single-Machine Characterization

A. Event-based Power Estimation

In modern computing systems, there are more than a hundred PMC events, but only a few can be collected at the same time due to monitoring overhead. Thus, a key challenge is how to select the minimum number of the power-related performance counters among all available. In the past, system engineers would select the right PMCs by leveraging domain knowledge. Increasing system heterogeneity makes this manual event selection difficult. We instead fully automate the PMC selection by using Lasso statistical analysis (Least Absolute Shrinkage and Selection Operator) [14]. The Lasso method exploits l_1-regularization to perform feature selection while building a regression model.

Let $V_{t_i}^{app_j} = \langle e_{1,t_i}^{app_j}, e_{2,t_i}^{app_j}, \ldots, e_{k,t_i}^{app_j} \rangle$ be a vector for k PMC events e at a time interval i for an application j, called an *event vector*. Then, the collected data for a computing platform $\boldsymbol{C}_A$ can be represented by a set of event vectors, $\mathbb{D}_{C_A} = \{V_{t_0}^{app_0}, V_{t_1}^{app_0}, \ldots, V_{t_L}^{app_N}\}$. For a general event vector $V_{t'} = \langle e_{1,t'}, e_{2,t'}, \ldots, e_{k,t'} \rangle$, a linear power model for $\boldsymbol{C}_A$ is represented by

$$P_{t'} = \sum_{i=1}^{k} \beta_i e_{i,t'} + \beta_0 \quad (1)$$

where β_i is the coefficient correspondent to each event and β_0 is the intercept. The linear regression finds the parameters using least square solutions. Unlike standard linear regression, the coefficients of less power-related events are set to zero by Lasso, and thus we can automatically exclude them and build the power model by only using the selected events.

TABLE I: Selected PMC events

		Availablity	
Event	**Description**	**x86**	**ARM**
CLOCK_CYCLES	Clock cycles	●	●
INSTRUCTIONS	Instructions	●	●
RS_UOPS_DISPATCHED	Micro operations dispatched	●	
FP_COMP_OPS_EXE	Floating point operations	●	
BR_INSTS	Branch instructions retired	●	●
BR_MISSS	Branch instructions missed	●	●
L1-DCACHE_LOADS	L1 data cache loaded	●	●
L1-DCACHE_STORES	L1 data cache stored	●	●
LLC_REFERENCES	Last level cache referenced	●	●
LLC_MISSES	Last level cache missed	●	●
BUS_CYCLE	Bus cycles	●	●
RESOURCE_STALLS	Resource stalls	●	
DP_SPEC	Speculative integer operations		●
UNALIGNED_LDST_SPEC	Speculative ld/st operations		●

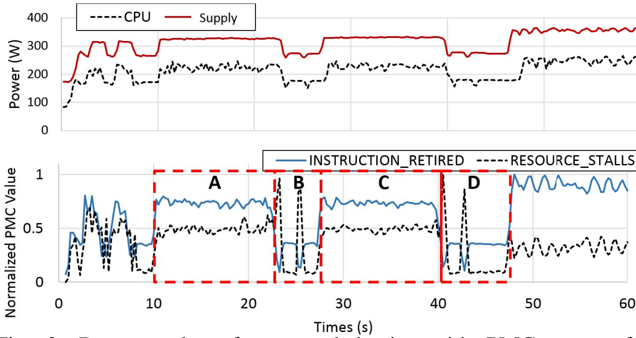


Fig. 2: Power and performance behavior with PMC events for Linpack benchmark on Intel SR1560SF server at maximum frequency

Table I shows the list of PMC events which are selected by the Lasso method in the P[4] framework. For three tested servers Lasso selected 12 PMCs, while for ARM it selected 11. We evaluate the accuracy of selected performance counters on the *event-based power estimation* in Section VI-B.

B. Application Phase Extraction

The *application phase extraction* module is responsible to identify *application phases*, i.e., clusters of homogeneous system usage behavior. To better explain the concept of the phases, Figure 2 shows power measurements and two representative PMC events for a *Linpack* benchmark [15] running on Intel SR1560SF server. As shown in the figure, the PMC events have similar patterns of changes over time which represent latent states related to power consumption, e.g., (A,C) and (B,D). In addition, similar performance characteristic, e.g., the number of instructions for an interval (INSTRUCTIONS), is observed for each labeled period.

Our solution automatically relates the similar event behaviors to different application phases with *k*-means clustering algorithm [16][1]. This algorithm requires two parameters, the number of clusters *k*, and the initial center of each cluster. The phase extraction procedure uses the set of event vectors $\mathbb{D}_{C_l}$, as the input data set of the *k*-means algorithm. Then, the algorithm assigns a cluster index $\rho_{i,j}$ to each vector $V_{t_i}^{app_j} \in \mathbb{D}_{C_l}$, where $0 \leq \rho_{i,j} < k$.

Automated parameter selection: The framework also automatically selects the two parameters: the number of clusters, *k*, and the initial centers of each cluster. First, *k* is selected

[1]We also tested other clustering algorithms such as DBSCAN and hierarchical clustering, and chose the *k*-Means since it identified the phases for most benchmarks sufficiently compared to the other algorithms.

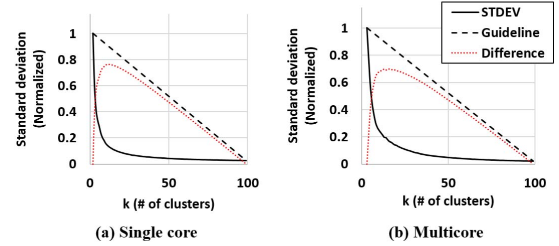


Fig. 3: Automated decision on the number of application phases, *k* (Left: single-core, Right: multicore on the Intel SR1560SF server)

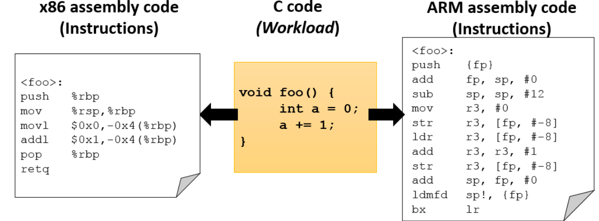


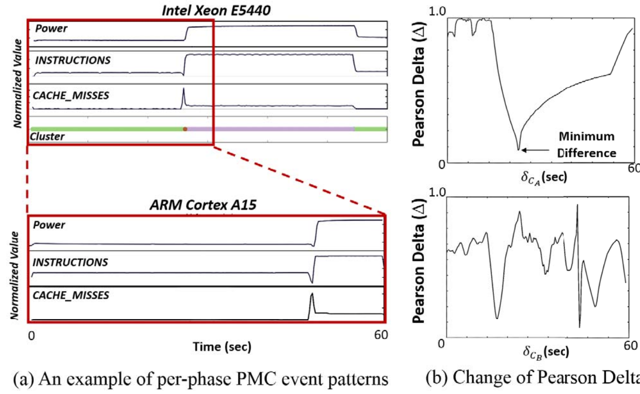
Fig. 4: Cross-architecture instruction changes

so that distinct system usage is sufficiently separated by the identified clusters. The similarity of event vectors in each cluster increases as *k* grows, and converges at value *k* for which event vectors of each cluster have enough similarity. The module automatically identifies the converged point using kNeedle algorithm [17]. Figure 3 illustrates the *k* decision procedure based on the kNeedle algorithm for a dataset collected on a Intel server. We first execute the *k*-means algorithm with $k = 2$, and calculate the standard deviation, denoted as STDEV, for the distances between each data point and its cluster center. Then, we increment the *k* values and compute the standard deviation again. This iteration is repeated up to $k_{\max}$. We empirically set $k_{\max}$ to 100 since it provides sufficient convergence of the standard deviation. The kNeedle algorithm then computes a guideline, depicted as the black dotted line, and determines the difference between the guideline and the standard deviation line, indicated as the 'difference' line. The parameter *k* is selected as the point which gives the maximum difference, that is, where the variance starts converging. Based on these results, we set *k* to 11 and 16 for the single- and multi-threaded application cases, respectively. Once *k* is found, we also determine the initial cluster centers using the *k*-means++ approach [16], which automatically finds appropriate cluster centers to improve algorithm performance and accuracy. Note that the identified phases are characterized for a dataset of a single machine. Thus, we need to identify the same phases across different machines. We present this procedure in Section V-A.

V. Cross-Platform Prediction

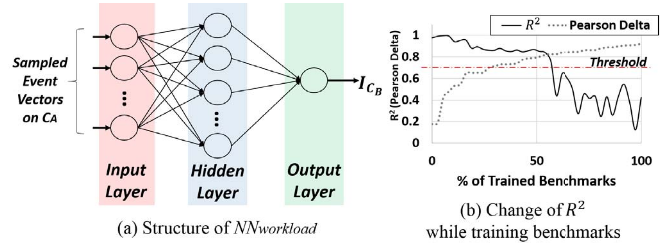
A. Cross-Platform Phase Recognition

To predict power and performance, we need to first understand how application behavior changes across different HW platforms. Intuitively, due to heterogeneity of architectures and operating condition differences, the events monitored on different platforms could vary even for the same application's task. As an example, Figure 4 shows how many instructions should be executed to perform an identical task on two different architectures. We compiled a code snippet of a simple function and got the assembly code by using *objdump*. As



(a) An example of per-phase PMC event patterns (b) Change of Pearson Delta

Fig. 5: PMC event pattern recognition



(a) Structure of $NN_{workload}$ (b) Change of R^2 while training benchmarks

Fig. 6: An NN model for workload amount identification

shown in the figure, due to the ISA differences between x86 (CISC) and ARM (RISC), the generated assembly code includes significantly different instructions. For example, since the function calls are handled differently, only 5 instructions are executed on x86, whereas 11 instructions are needed on ARM due to its smaller instruction set.

In order to capture cross-platform behavioral changes in a general way, we exploit the extracted application phases. Although applications have varying PMC patterns on different platforms, these patterns can be generalized patterns by how each phase utilizes the system. For example, a compute-intensive phase in a given platform is likely to be compute-intensive in another platform as well. We identify this relationship by associating the application phases extracted on a single machine with *machine-independent* phases on other platforms. The task is accomplished by two sub procedures, (i) event pattern recognition and (ii) cross-platform phase matching.

1) NN-Based Event Pattern Recognition: Figure 5a shows measurement results of a *blackscholes* benchmark for 60 seconds on two architectures, Intel Xeon E5440 and ARM Cortex A15 processor. Different clusters identified by the phase extraction procedure are denoted by different colors in the bar graph of the Intel processor. These results show that PMC event patterns are similar for the two platforms even though they are different architectures. The patterns are identified by the extracted phases on the Intel processor. For example, the green phase is characterized by a lower number of instructions and cache misses as compared to the purple phase. Since these patterns are observed on both platforms, we could estimate that the execution time of 60 seconds on the ARM is reduced to around 32 seconds on the x86 processor.

The event pattern recognition procedure of P^4 systematically relates the phases of the same workload running on two different platforms. Let $T_{C_A}^{app_p}$ and $T_{C_B}^{app_p}$ be the execution time of each of two computing platforms C_A and C_B for a benchmark. For the two platforms, we consider two event vectors, one for a sub-duration (e.g., $T_{C_A}^{app_p} - \delta_{C_A}$ where δ_{C_A} is a time duration less than $T_{C_A}^{app_p}$) and the other one for the full duration (e.g., $T_{C_B}^{app_p}$). Then, it compares the two event vectors with a custom difference metric, called *Pearson Delta* to find the sub-duration which exhibits the most similar PMC pattern. Pearson Delta exploits Pearson's correlation coefficient [18] to find the dependence of an event for the data collected on two different platforms per each time duration. If there is a strong positive linear relationship, the Pearson coefficient, r, has a high positive value close to 1.0 (where $r \leq 1.0$), while $r < 0$ when it has a negative relationship. Since we are only interested in a positive relationship, the Pearson Delta, in short Δ, is defined as follows:

$$\Delta = avg_i(\Delta_i) \quad \text{where}$$
$$\Delta_i = \begin{cases} 1 - r_i, & \text{if } r_i \geq 0 \\ 1, & \text{otherwise} \end{cases} \tag{2}$$

where r_i is the Pearson coefficient for an i-th PMC event. By changing δ_{C_A}, we calculate the Pearson Delta values for different sub-durations of the event vector of C_A to the full duration of C_B. We also consider the opposite case, say the subduration of $T_{C_B}^{app_p} - \delta_{C_B}$, i.e., when the performance of C_A is slower than that of C_B for the benchmark. Figure 5b shows how Δ changes while running *blackscholes*. The result shows that this procedure can find the valid sub-duration of 32 seconds in C_A which exhibits the most similar pattern to the full duration of C_B. We call the identified duration pair on the two platforms as the *best-pattern* duration pair.

In our experiments, this procedure could find execution time changes of many benchmarks, e.g., more than half of benchmarks across x86 and ARM. However, for some benchmarks there are no outstanding similar sub-durations or only monotonous PMC events keep occurring, so we get relatively high Pearson Delta values which result in inaccurate estimates. Thus, we build a neural network model which is capable of automatically identifying complex patterns from input data and capturing nonlinearity which are often observed in the cross-architecture event relationship. The NN model, called $NN_{workload}$, identifies how many instructions should be executed on a computing platform, say C_B, using PMC events observed on another platform, C_A, as the input. Figure 6a shows the neural network structure. The model consists of an input layer, a hidden layer, and an output layer. Each neuron of the input layer corresponds to each PMC event collected during the sampling interval on a platform C_A. The output layer produces a single output given the number of instructions on the predicted platform C_B. The hidden layer has 30 neurons, fully-connected to the two adjacent layers. The sigmoid function with bias is used for the activation function, and the back propagation method is used for training. To converge the NN weights, the model is trained 100 times for each training dataset[2].

We train the $NN_{workload}$ using a subset of benchmarks whose best-pattern duration pairs are correctly estimated. The training procedure starts with the best estimated benchmark which exhibits the minimum Pearson Delta. To generate the

[2]We also trained different NN structures, e.g., more neurons of the hidden layer (up to 100), more training iterations (up to 1000) and more hidden layers (up to 3), there was no meaningful improvement in the prediction results.

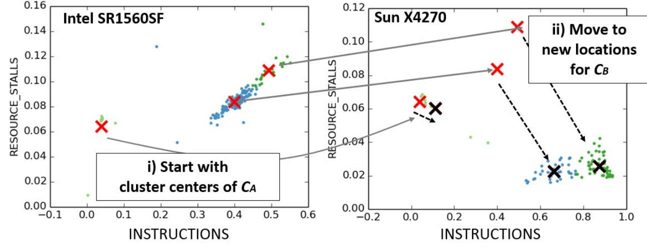
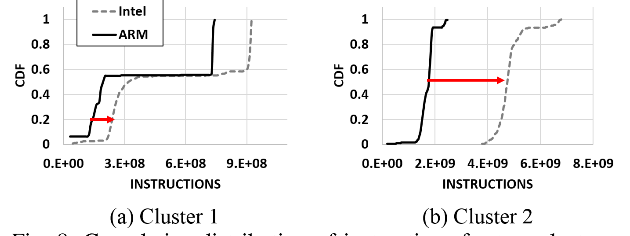


Fig. 7: Cross-platform phase matching (Intel SR1560SF to SUN X4270, *splash2x.lu_cb*)



(a) Cluster 1 (b) Cluster 2

Fig. 8: Cumulative distribution of instructions for two clusters

training data of the benchmark, we align the event vectors of C_A and C_B with the best-pattern duration pair. Then, the event vector of C_A and the number of instructions of C_B are sampled at the interval (I_{C_B}) that the PMC events are originally collected, as the NN training input and output respectively. Once the model is trained for the benchmark, we evaluate R^2 score of the model. This procedure is repeated by retraining the model with another benchmark in the order of Pearson Delta values. Figure 6b presents how the R^2 score changes through the iterations along with the Pearson Delta of the trained benchmarks. As shown in the results, the R^2 score suddenly drops after training the benchmark to 58%, since it has an inaccurate duration pair. Based on this observation, we choose the trained NN weights when $R^2 > 0.7$ as the final $NN_{workload}$ model.

Using the final model, P[4] re-calculates the best-pattern duration pairs for all benchmarks. For a benchmark, let $V_{t_i}^{C_A}$ be the event vector collected at t_i on C_A, and $Inst_{t_j}^{C_B}$ be the number of instructions collected at t_j on C_B. For $V_{t_i}^{C_A}$, the $NN_{workload}$ estimates the number of instructions on C_B, say $\widehat{Inst}_{t_i}^{C_B}$. Thus, we can estimate the best-pattern duration pair by identifying t_i and t_j so that

$$\sum_i \widehat{Inst}_{t_i}^{C_A} = \sum_j Inst_{t_j}^{C_B}.$$

P[4] exploits the event vectors within the identified durations to find the cross-platform phases.

2) Cross-Platform Phase Matching: In this step, P[4] identifies the machine-independent, cross-platform application phases using a modified *k*-means algorithm with the application phases identified while running on a single machine. Figure 7 illustrates the cross-platform phase matching procedure. Let $\mathbb{V}_{C_A}^{app_p} = \{\upsilon_{\rho_{1,p}}, \ldots, \upsilon_{\rho_{k,p}}\}$ be a set of cluster centers for the phases obtained for app_p on a platform C_A. Also, let $\mathbb{D}_{C_B}^{app_p} (\subset \mathbb{D}_{C_B})$ be the dataset of app_p executed on another platform C_B. To identify the clusters of $\mathbb{D}_{C_B}^{app_p}$ while keeping the identified phase indexes, we apply the *k*-means algorithm again with the initial cluster centers $\mathbb{V}_{C_A}^{app_p}$. Then, the k cluster centers are moved with the *k*-means procedure so that each phase is adjusted and fit into the new application dataset. The clusters newly identified for all benchmarks on C_B represent how each cluster on C_A behaves on a different platform C_B. We utilize the per-phase behavior to train the cross-platform prediction NN model described in the next section.

B. Performance and Power Prediction

1) Phase-Based Prediction Model Learning: Once the cross-platform phases are identified, we train a new NN model, called NN_{PMC}, which infers PMC events across platforms to predict the time-variant power levels. The NN_{PMC} has a similar structure to the $NN_{Workload}$, but has multiple neurons of the output layer which corresponds to the performance counters of the predicted platform. To generate the training dataset, we utilize the event distribution of each phase to capture the event changes between platforms. To better clarify, Figure 8 shows the cumulative distribution function (CDF) graphs of the `INSTRUCTIONS` event for two representative clusters, running on the Intel Xeon E5440 processor at 2.8 GHz frequency and the ARM Cortex A15 at 1.8 GHz frequency. The results show that different clusters for each platform could present very different trends, while the events in the same cluster behave very similarly across platforms.

The per-phase patterns are captured by training the neural network model. Let $V_t^{app_p,C_A}$ be an event vector in a computing platform C_A. In the cluster distribution of each event of C_A, we compute the percentile of $e_{i,t}^{app_p,C_A}$ of $V_t^{app_p,C_A}$, and find $\widehat{e_{i,t}}$ which is the event value at the same percentile of the distribution of the same event in C_B. For example, in Figure 8, the two arrows respectively describe the estimation of `INSTRUCTIONS` at the 0.2 percentile for Cluster 1 and the 0.5 percentile for Cluster 2. Then, we create an event vector $\widehat{V_t}^{app_p,C_B}$ in C_B where each element of the vector is $\widehat{e_{i,t}}$. Some events are available only for C_B, e.g., `UNALIGNED_LDST_SPEC` of ARM processor. In that case, $\widehat{e_{i,t}}$ is computed by selecting N event vectors of C_B which most similarly behaves in terms of each of N common events and averaging the event values of the selected vectors. Then, the model is trained with $V_t^{app_j,C_A}$ as the input and $\widehat{V_t}^{app_j,C_B}$ as the output.

2) Complete NN model for Online Prediction: Power consumption is computed at runtime using NN_{PMC}, $NN_{workload}$ and the Lasso power estimation model with PMCs. For the instantaneous power prediction, P[4] combines the NN_{PMC} and the single-machine model described in IV-A. The predicted event vector, $\widehat{V_t}^{app_j,C_B}$, is used as an input of the regression model since the NN_{PMC} predicts how PMC events will behave on a different platform for the same application. The execution time for a sampling period on a different platform, τ_{C_B}, can be predicted by using that on the current platform (τ_{C_A}), $NN_{workload}$ which estimates the number of instructions executed on C_B (I_{C_B}), and an output neuron of NN_{PMC} which produces the speed in Instructions Per Sampling interval (IPS) on C_B (IPS_{C_B}) with the following equation[3]:

$$\tau_{C_B} = \tau_{C_A} \times \frac{I_{C_B}}{IPS_{C_B}} \quad (3)$$

[3]Due to the space limitation, we do not include the detailed proof steps. It can be derived from $\tau_{C_i} \times IPS_{C_i} = I_{C_i}$ and $I_{C_A} = IPS_{C_A}$ in a straight-forward way.

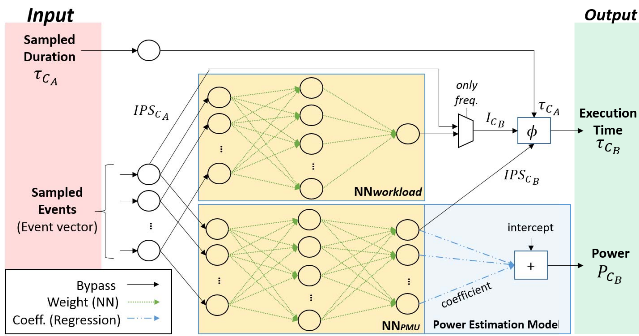


Fig. 9: A feed-forward neural network for online prediction

Figure 9 illustrates the online prediction procedure as a feed-forward network for a platform pair, C_A and C_B. Once the PMC events are collected as event vectors for an interval τ_{C_A} on C_A, the sampled event vector is input into the two neural networks. The number of instructions of C_B, i.e., I_{C_B}, identified by $NN_{workload}$, is delivered to the execution time conversion function, π, which computes τ_{C_B} based on Equation 3.

When P[4] predicts power consumption for different frequencies on the same platform, the $NN_{workload}$ is not activated since the number of instructions required to run the workload is the same regardless of frequency they are run at. Thus, the IPS of C_A is provided to the execution time conversion function. At the same time, the power is predicted using the regression-based power estimation model where the input of the regression model is given as the output of the NN_{PMC}. Note that all the computation of the complete NN model are performed only using the PMC events periodically monitored on C_A. Thus, P[4] can do cross-platform prediction online by just using collected PMC events.

VI. Evaluation

A. Experimental Setup

The proposed P[4] framework has been implemented using Python 2.7 with Scikit-learn 0.17.1 library for the statistical analysis [19]. We conducted the measurements on four computing platforms: Intel SR1560SF, Sun X4270, Dell PowerEdge R810, and Odroid XU3. Table II summarizes the specifications of each platform. For the server systems, we measure the supply power using the HIOKI 3334 power meter, while the power consumption of Odroid XU3 is measured by reading the embedded sensors on each core. The Intel SR1560SF has also been instrumented to measure CPU power consumption by reading voltage drop of two 0.1Ω shunt resistors which are installed in the cables providing power to the two quad cores, e.g. CPU 0-3 and 4-7 respectively. All the power measurements are sampled at a rate of 100 ms. For each platform, we execute the benchmarks at three processor frequency levels, i.e., lowest (L), medium (M) and highest (H).

Benchmarks: We select industry-standard benchmarks which represent a wide range of workloads. Benchmarks are executed on each platform with varying number of threads and at various processor frequencies. The benchmark set includes SPEC2006 [20], PARSEC/SPLASH2x [21] with native inputs, NERSC datacenter benchmarks [22], and Linpack [15], which has been used for TOP500 runs [23]. A few of PARSEC benchmarks (e.g., raytrace), Intel Linpack, and NERSC could not be ported to the Odroid ARM processor due to the ISA

TABLE II: Evaluated heterogeneous platforms

Type	Model	Hardware component description
Mobile	Odroid XU3 (ARM)	Exynos5422 ARM big.LITTLE Cortex-A15 big: 1.4 GHz ∼ 1.0 GHz, Cortex-A7 LITTLE: 2.0 GHz ∼ 1.2 GHz L1 cache sizes: 32KB, DRAM size: 2GB
Server	Intel SR1560SF	Intel Xeon E5440 1.99GHz ∼ 2.83GHz L1 cache sizes: 64KB, DRAM size: 8GB
Server	Sun Fire X4270	Intel Xeon E5500 1.6GHz ∼ 2.93 GHz L1 cache sizes: 64KB, DRAM size: 24GB
Server	Dell PowerEdge R810	Intel E7 4870 Westmere 2.39 GHz ∼ 1.06 GHz L1 cache sizes: 32KB, DRAM size: 128GB

TABLE III: Overhead of P[4] modules

	Power Estimation	Phase model	Perf. prediction model	Power prediction model
Online	1.8 μs	N/A	34 μs	88 μs
Offline	4.6 s	161 s	34 min.	68.4 s

difference and library dependencies. In total, we could execute 129 benchmarks (50 single-core and 79 multicore) on the three server platforms, and 88 benchmarks on the ARM platform. We run the workloads for more than 2 hours, and collect the performance counter events using *perf* tool every 250 ms while measuring power consumption.

The experimental results for power estimation and prediction are cross-validated using the "leave-one-out" strategy to evaluate each benchmark by separating the tested program from the training set. We pick a benchmark for testing the online identification stage while all other benchmarks are used to build the models in the offline learning stage. This cross-validation was performed for all benchmarks. The accuracy of the estimation and the prediction is evaluated using Mean Absolute Percentage Error (MAPE).

Overhead: Table III shows the overhead of P[4] modules evaluated on Intel i7-6700k quad-core CPU. We report the average process time of each event vector for the online stage and the running time of each platform pair for the offline stage. In the online stage, P[4] computes the feed forward network and only requires selected event counters. The runtime overhead to process each event vector is less than 124 μs. Compared to the PMC sampling rate of 250 ms, the runtime overhead is negligible. Most overhead of the offline learning stage arose from the benchmark execution. In our evaluation, the offline learning stage was performed for 200 minutes, including the benchmark execution of 165 minutes. Since the offline learning happens only once for each machine, the overhead of the offline stage is negligible.

B. Event-Based Power Estimation

To evaluate the event selection of the Lasso method, we compare it against linear regression. The linear regression method uses 10 additional performance counters, including TLB misses, thermal trip, SSE execution, and snoop-related events, on top of the event counters that P[4] selected. We also compare the results with two state-of-the-art processor models published in Su et al. [24] and Lee et al. [7]. The models exploit 8 and 3 events, respectively, selected based on the domain knowledge of their architecture. Figure 10 shows the comparison of estimation accuracy for processor power

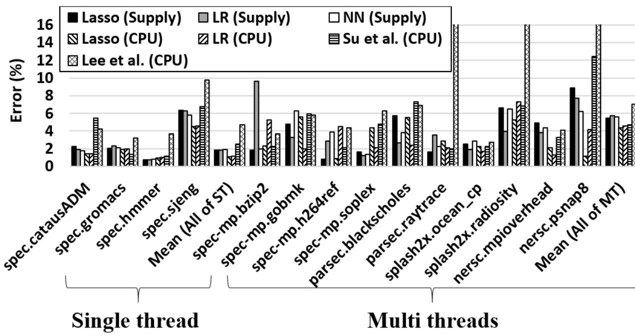


Fig. 10: Single-machine power estimation

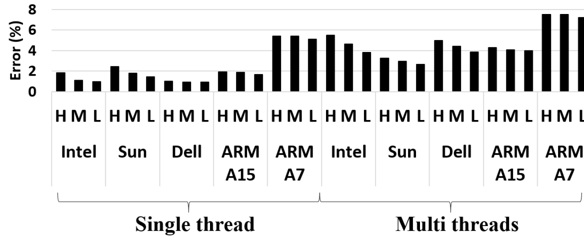


Fig. 11: Power estimation error per platform

and supply power of the Intel server for the single- and the multi-threaded test cases running at the highest frequency. We also tested an NN model which estimates the power level with 30 neurons to account for non-linearity. Because of the limited space, we show 14 representative benchmarks out of 129 and the average error over all tested benchmarks. The results show that Lasso estimates processor and supply power accurately, even though Lasso is using only a subset of available performance counters. The Lasso estimation error is similar to the linear regression (LR) model, which uses 22 PMC events, with only 0.2% difference on average. Lasso model has better accuracy than the two published models, since P^4 automatically selects strongly power-related events for the given target platforms. In addition, it shows comparable results to the NN-based model. Thus, we conclude that the events statistically selected by P^4 provide very accurate power estimates without the need for domain knowledge as was done by Su et al. [24] and Lee et al. [7].

Figure 11 summarizes the power estimation error of P^4 which uses the selected events for other platforms. The result shows that the power estimates of multicore and higher frequency cases are more challenging due to the larger fluctuations in power levels. Nevertheless, P^4 estimates power with 5.4%, 3.23%, 4.98%, 4.28%, and 7.5% of average error for the Intel, Sun and Dell servers, Cortex A15, and Cortex A7 respectively. The error on Cortex A7 is a bit higher than others, since the processor has relatively low static power, making it highly sensitive even to small errors.Even for the worst case benchmark, the model can estimate within 13% of error.

C. Application Phases Identification

Figure 12 presents a qualitative comparison of the cross-platform phase behavior for four benchmarks with different colors denoting different clusters (phases). The IntelH (Intel server running at the highest frequency) is used as the reference platform to detect the baseline phases. The phases of

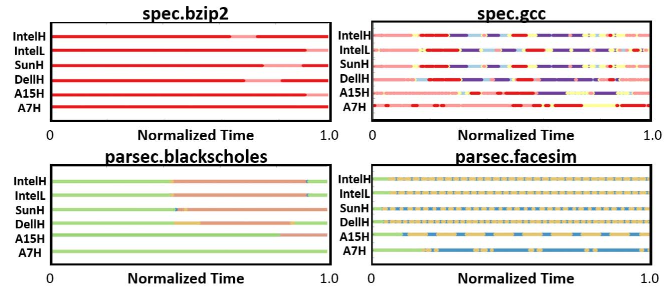


Fig. 12: Identified phases for four benchmarks

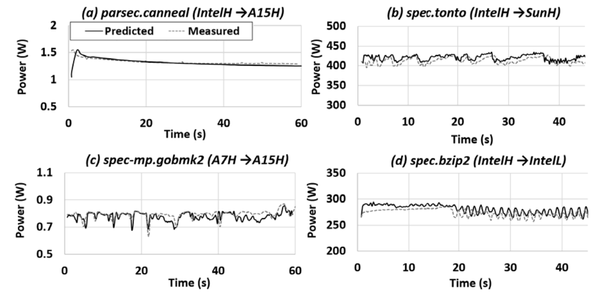


Fig. 13: Performance-aware power prediction for four heterogeneous platform combinations

the top two benchmarks, *spec.bzip2* and *spec.gcc*, are from single-threaded benchmarks, while the bottom are of multi-threaded ones. The comparison includes different frequencies, i.e. IntelH vs. IntelL, various platforms, i.e. IntelH vs. SunH, and completely different CPU architectures, i.e. IntelH vs. A15H. The result shows that the phase recognition and matching techniques accurately recognize the phases across different platform configurations. For example, the *spec.bzip2* benchmark has a dominant phase, denoted with red color, and an intermediate phase of pink color. The pink color is not visible on A7H, since the benchmark terminates before executing this phase. Similar findings are also observed for *parsec.blackscholes* in the multi-threaded case. P^4 identifies the cross-platform phases accurately for more complex benchmarks, e.g., *spec.gcc* and *parsec.facesim*.

D. Cross-Platform Prediction

Our key contribution is the generalized power prediction capability across heterogeneous platforms. Figure 13 shows how the proposed P^4 predicts power consumption using the online prediction network described in Section V-B2. The results show four heterogeneous platform combinations, (a) two different-architecture, Intel x86 to ARM Cortex A15, (b) a cross-server case from IntelH to SunH, (c) a big-to-LITTLE example in moving from A7 to A15, and (d) a frequency change from IntelH to IntelL, for four representative multi-threaded benchmarks. The results show that based on the trained neural networks, P^4 can accurately predict power changes over time for all the heterogeneous platform combinations. The execution time for each benchmark is also predicted for each of the platforms. For example, the prediction of *parsec.canneal* of the 60 seconds on Cortex A15 is made with the PMC events for 18.5 seconds observed on IntelH. This means that P^4 can predict both instantaneous power and performance changes using monitored events.

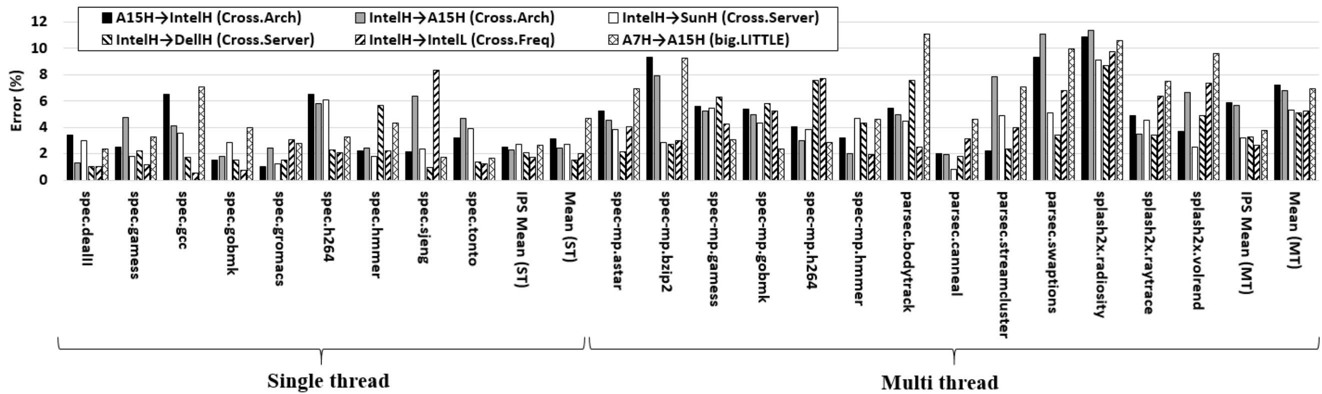


Fig. 14: Summary of cross-platform time-variant prediction accuracy

Figure 14 reports the prediction results for the six configuration change cases. In the evaluation, we observed that our methodology can accurately predict performance and power consumption. P^4 gets 5.2% error on average for predicting time-varying power consumption on servers for multi-threading benchmarks. Similarly, when comparing completely different architectures for multi-threading benchmarks, P^4 gets 7.2% and 6.8% error on average, for A15H-to-IntelH and IntelH-to-A15H cases respectively. Note that, for in this case the number of threads is also different, i.e., 8 on Intel vs. 4 on ARM A15, as well as their frequency levels. When predicting power consumption for the big-LITTLE example (A7-to-A15), the error is 6.9%. We also compute the performance prediction error with the IPS metric. The results show that the average error is less than 6% for even the most challenging cross-architectural prediction cases such as A15H-to-IntelH and IntelH-to-A15H. Thus, P^4 can accurately predict the power and performance for the complex combinations, including changes in the number of threads, CPU frequencies, platforms (mobile to server), and CPU architectures (x86 to ARM).

VII. Conclusion

In this paper, we propose P^4, which characterizes the diverse workload for heterogeneous computing ecosystems and accurately predicts power and performance across different CPU architectures and computing platform configurations. Our technique automatically selects the event counters strongly related to the power consumption and extracts application phases which represent the groups of similar system usage behavior without a priori knowledge. Then, it automatically trains neural networks to predict power and performance across different platforms at runtime with negligible overhead. In our evaluation conducted on four heterogeneous computing platforms, we showed that our framework successfully recognizes the distinct application power states, and accurately predicts power consumption with less than 7.2% of error for all diverse configuration changes, including frequency levels, platforms, and CPU architectures.

VIII. References

[1] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[2] Ramon Bertran, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010.

[3] Ramon Bertran, Marc Gonzàlez, Xavier Martorell, Nacho Navarro, and Eduard Ayguadé. Counter-based power modeling methods: Top-down vs. bottom-up. *The Computer Journal*, 2013.

[4] Sherief Reda and Abdullah N Nowroz. Power modeling and characterization of computing devices: a survey. *Foundations and Trends in Electronic Design Automation*, 2012.

[5] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. Accurate phase-level cross-platform power and performance estimation. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016.

[6] Erez Perelman, Marzia Polito, J-Y Bouguet, Jack Sampson, Brad Calder, and Carole Dulong. Detecting phases in parallel applications on shared memory architectures. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006.

[7] Young-Ho Lee and Jihong Kim. Fast and accurate on-line prediction of performance and power consumption in multicore-based systems. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013.

[8] Vinivius Petrucci, Orlando Loques, and Daniel Mosse. Lucky scheduling for energy-efficient heterogeneous multi-core systems. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems (HotPower)*, 2012.

[9] Ashutosh S Dhodapkar and James E Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003.

[10] Yeseong Kim, Francesco Parterna, Sameer Tilak, and Tajana S Rosing. Smartphone analysis and optimization based on user activity recognition. In *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*. IEEE, 2015.

[11] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K Saul, and Geoffrey M Voelker. edoctor: automatically diagnosing abnormal battery drain issues on smartphones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[12] Ye Jin, Xiaosong Ma, Mingliang Liu, Qing Liu, Jeremy Logan, Norbert Podhorszki, Jong Youl Choi, and Scott Klasky. Combining phase identification and statistic modeling for automated parallel benchmark generation. *ACM SIGMETRICS Performance Evaluation Review*, 2015.

[13] Arunachalam Annamalai, Rance Rodrigues, Israel Koren, and Sandip Kundu. An opportunistic prediction-based thread scheduling to maximize throughput/watt in amps. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013.

[14] John C McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C Snoeren, and Rajesh K Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conf*, 2011.

[15] Linpack benchmarks. https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite.

[16] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007.

[17] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a kneedle in a haystack: Detecting knee points in system behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*. IEEE, 2011.

[18] Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient.

[19] Scikit-learn machine learning in python. http://scikit-learn.org/stable/.

[20] Spec2006. https://www.spec.org/cpu2006/.

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.

[22] Nersc benchmarks. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/.

[23] Top 500 supercomputer sites. https://www.top500.org/.

[24] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L Greathouse, and Zhiying Wang. Ppep: Online performance, power, and energy prediction framework and dvfs space exploration. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014.