A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints

Nikita Mishra Huazhe Zhang John D. Lafferty Henry Hoffmann

University of Chicago

{nmishra,huazhe,hankhoffmann}@cs.uchicago.edu, lafferty@galton.uchicago.edu

Abstract

In many deployments, computer systems are underutilized – meaning that applications have performance requirements that demand less than full system capacity. Ideally, we would take advantage of this under-utilization by allocating system resources so that the performance requirements are met and energy is minimized. This optimization problem is complicated by the fact that the performance and power consumption of various system configurations are often application – or even input – dependent. Thus, practically, minimizing energy for a performance constraint requires fast, accurate estimations of application-dependent performance and power tradeoffs.

This paper investigates machine learning techniques that enable energy savings by learning Pareto-optimal power and performance tradeoffs. Specifically, we propose LEO, a probabilistic graphical model-based learning system that provides accurate online estimates of an application's power and performance as a function of system configuration. We compare LEO to (1) offline learning, (2) online learning, (3) a heuristic approach, and (4) the true optimal solution. We find that LEO produces the most accurate estimates and near optimal energy savings.

This work was funded by the U.S. Government under the DARPA PERFECT program, the Dept. of Energy under DOE DE-AC02-06CH11357, NSF grant IIS-1116730 and ONR grant N000141210762. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

ASPLOS '15, March 14-18, 2015, Istanbul, Turkey..

http://dx.doi.org/10.1145/2694344.2694373

Categories and Subject Descriptors

C.4 [*Modeling Techniques*]: Design Techniques; I.6.5 [*Computing Methodologies*]: Model Development—Modeling methodologies

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Adaptation; Dynamic systems; Statistics; Probabilistic Graphical models; Energy minimization.

1. Introduction

This paper addresses two trends in modern computing systems. First, energy is increasingly important; reducing energy consumption reduces operating costs in datacenters and increases battery life in mobile devices. Second, computer systems are often underutilized, meaning there are significant portions of time where application performance demands do not require the full system capacity [3, 41].

These two trends raise the problem of allocating available resources to meet the current performance demand while minimizing energy consumption. This problem is a *constrained optimization* problem. The current utilization level represents a performance constraint (*i.e.,*an amount of work that must be completed in a given time); system energy consumption represents the objective function to be minimized.

This problem is challenging because it requires a great deal of knowledge to solve. More than knowledge of the single fastest, or most energy efficient system configuration solving this problem requires knowledge of the power and performance available in all configurations and the extraction of those configurations that represent Pareto-optimal tradeoffs. Acquiring this knowledge is additionally complicated by the fact that these power/performance tradeoffs are often application – or even input – dependent. Thus, there is a need for techniques that accurately estimate these application-dependent parameters during run-time.

Machine learning techniques represent a promising approach to addressing this estimation problem. *Offline learning* approaches collect profiling data for known applications

and use that to predict optimal behavior for unseen applications (examples include [10, 33, 35, 50, 59]). *Online learning* approaches use information collected while an application is running to quickly estimate the optimal configuration (examples include [1, 34, 37, 40, 44, 45, 52]). Offline methods require minimal runtime overhead, but suffer because they estimate only trends and cannot adapt to particulars of the current application. Online methods customize to the current application, but cannot leverage experience from other applications. In a sense, offline approaches are dependent on a rich training set that represents all possible behavior, while the online approaches generate a statistically weak – *i.e.,*inaccurate – estimator due to small sample size.

In this paper, we present LEO (Learning for Energy Optimization), a learning framework that combines the best of both worlds, *i.e.,*the statistical properties both offline and online estimation. We assume that there is some set of applications for which the power and performance tradeoffs are gathered offline. LEO uses a graphical model to integrate a small number of observations of the current application with knowledge of the previously observed applications to produce accurate estimations of power and performance tradeoffs for the current application in all configurations. LEO's strength is that it quickly matches the behavior of the current application to a subset of the previously observed applications. For example, if LEO has previously seen an application that only scales to 8 cores, it can use that information to quickly determine if the current application will be limited in its scaling.

LEO is a fairly general approach in that it supports many types of applications with different resource needs. It is not, however, appropriate for all computer systems, especially ones, which run many small, unique jobs. Instead, it focuses on supporting systems that 1) execute longer running jobs (in the 10s of seconds) or many repeated instances of short jobs, 2) run at a wide range of utilizations, and 3) might have phases where optimal tradeoffs may change online. For systems that meet these criteria, LEO provides a powerful ability to reduce the energy consumption. For systems that service short (< 1 second), largely unique jobs, LEO will work, but other approaches are probably better matched to those specific needs.

We have implemented LEO on a Linux x86 server and tested its ability to minimize energy for 25 different applications from a variety of different benchmark suites. We first compare LEO's performance and power prediction accuracy to (1) the true value, (2) an offline approach, and (3) an online approach (See Section 6.2). On average, LEO is within 97% of the true value while the offline and online approaches only achieve 79% and 86% accuracy, respectively. We then use LEO to minimize energy for various performance requirements (or system utilizations) (See Section 6.4). Overall we find that our approach is within 6% of the true optimal energy, while the offline approach exceeds optimal energy consumption by 29% and online approach by 24%. Finally, we show that LEO provides near optimal energy savings when adapting to phases within an application.

This paper makes the following contributions:

- To the best of our knowledge, this is the first application of probabilistic graphical models for solving crucial system optimization problems such as energy minimization.
- It presents a graphical model capable of accurately estimating the application-specific performance and power of computer system configurations without prior knowledge of the application. (See Section 5).
- It makes the source code for this learning system available in both Matlab and C++[1].
- It evaluates LEO on a real system. (See Section 6).
- It compares the accuracy of LEO's estimations to both the truth and to offline and online learning approaches (See Section 6.3).
- It integrates LEO into a runtime for energy optimization and finds this learning framework achieves near-optimal energy savings. Furthermore, LEO significantly reduces energy compared to both offline and online approaches as well as the popular race-to-idle heuristic. (See Section 6.4).

The rest of the paper is organized as follows. Section 2 provides a motivational example to build intuition. Section 3 presents notation. Section 4 formalizes the energy minimization problem as a linear program and discusses the application-specific parameters of this problem. Section 5 elaborates our probabilistic graphical model and describes LEO in full detail. Section 6 presents empirical studies on LEO. Related work is discussed in Section 7 and the paper concludes in Section 8.

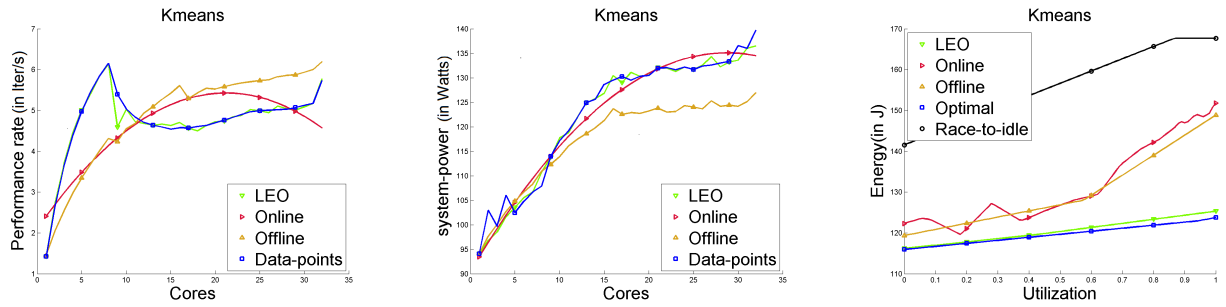
2. Motivational Example

This section presents an example to motivate LEO and build intuition for the formal models presented subsequently. We consider energy optimization of the `Kmeans` benchmark from Minebench [43]. Kmeans is a clustering algorithm used to analyze large data sets. For this example, we run on a 16-core Linux x86 server with hyperthreading (allowing up to 32 cores to be allocated)[2]. We assume that `Kmeans` may be run with different performance demands and we would like to minimize energy for any given performance demand. To do so for `Kmeans` on our 32-core system we must estimate its performance and power as a function of the number of cores allocated to the application. Given this information, we can easily select the most energy efficient number of cores to use for any performance demand.

To illustrate the benefits of LEO, we will compare it with three other approaches: a heuristic, offline learning, and online learning. The heuristic uses the well know *race-to-idle*

[1] leo.cs.uchicago.edu

[2] Our full system evaluation tests more parameters than simply core allocation. See Section 6 for details.



(a) (b) (c)

Figure 1. Power estimation for `Kmeans` clustering application using LEO, *Online* and *Offline* algorithms. The estimations are made using only 6 observed values (Cores) out of 32.

strategy – simply allocating all resources (cores, clockspeed, etc.) to `Kmeans` and then idling the system once the application completes. The offline learning approach builds a statistical model of performance and power for each configuration based on prior measurements of other applications. The online approach uses polynomial regression to learn the tradeoffs for each configuration while `Kmeans` is running. (More details on the specifics of these approaches can be found in Section 6.2).

Each of these three approaches has their limitations. The heuristic approach simply assumes that the most energy efficient configuration is the one where all the system resources are in use, but that has been shown to be a poor assumption for this type of application [21, 41]. The offline approach predicts average behavior for a range of applications, but it may be a poor predictor of specific applications (`Kmeans`, in this case). The online approach will produce a good prediction if it takes a sufficient number of samples, but the required number of samples may be prohibitive.

LEO combines the best features of both the offline and online approaches. At runtime, it changes core allocation (using process affinity masks), observes the power and performance, and combines this data with that from previously seen applications to obtain the most probable estimates for other unobserved cores. The key advantage of LEO's graphical model approach is that it quickly finds similarities between `Kmeans` and previously observed applications. It builds its estimation not from every previous application, but only those that exhibit similar performance and power responses to core usage. This exploitation of similarity is the key to quickly producing a more accurate estimate than either strictly online or offline approaches.

Figure 1 shows the results for this example. Figure 1a shows each approach's performance estimates as a function of cores, while Figure 1b shows the estimate of power consumption. These runtime estimates are then used to determine the minimal energy configuration for various system utilizations. Figure 1c shows the energy consumption data where higher utilizations mean more demanding performance requirements. As can be seen in the figures, LEO is the only estimation method that captures the true behavior of the application and this results in significant energy savings across the full range of utilizations.

Learning the performance for `Kmeans` is hard because the application scales well to 8 cores, but its performance degrades sharply with more. It is quite challenging to find the peak without exploring every possible number of cores. We observe the power and performance at 6 uniformly distributed values (5, 10, $\cdots$, 30 cores). The offline learning method predicts the highest performance at 32 cores because that is the general trend over all applications. The online method predicts peak performance at 24 cores, so it learns that performance degrades, but requires many more samples to correctly place the peak. LEO – in contrast – leverages its prior knowledge of an application whose performance peaks with 8 cores. Because LEO has previously seen an application with similar behavior, it is able to quickly realize that `Kmeans` follows this pattern and LEO produces accurate estimates with just a small number of observations.

The next three sections formalize this example. Section 3 describes the notation we will use. Section 4 presents a general formalization of this energy minimization problem for any configurable system (not just cores). Section 5 presents the technical description of how LEO incorporates online and offline approaches to find similar applications and produce accurate runtime estimates of power and performance.

3. Notations

The set of real numbers is denoted by $\mathbb{R}$. $\mathbb{R}^d$ denotes the set of d-dimensional vectors of real numbers; $\mathbb{R}^{d\times n}$ denotes the set of real $d \times n$ dimensional matrices. We denote the vectors by lower-case and matrices with upper-case boldfaced letters. The transpose of a vector $\mathbf{x}$ (or matrix $\mathbf{X}$) is denoted by $\mathbf{x}^T$ or just $\mathbf{x}'$. $\|\mathbf{x}\|_2$ is the $\mathcal{L}_2$ norm of vector $\mathbf{x}$, i.e. $\mathbf{x} = \sqrt{\sum_{i=1}^{d} x^2[i]}$. $\|\mathbf{X}\|_F$ is the Frobenius norm of matrix $\mathbf{X}$; *i.e.*, $\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{d}\sum_{i=1}^{n} X^2[i][j]}$. Let $\mathbf{A} \in \mathbb{R}^{d\times d}$ denote a d-dimensional square matrix. $\mathtt{tr}(\mathbf{A})$ is the trace of

the matrix $\mathbf{A}$ and is given as, $\mathsf{tr}(\mathbf{A}) = \sum_{i=1}^{d} \mathbf{A}[i][i]$. And, $\mathrm{diag}(\mathbf{x})$ is a d-dimensional diagonal matrix $\mathbf{B}$ with the diagonal elements given as, $\mathbf{B}[i][i] = x[i]$ and off-diagonal elements being 0.

We now review the standard statistical notation used below. Let $\mathbf{x}, \mathbf{y}$ denote any random variables in $\mathbb{R}^d$. The notation $\mathbf{x} \sim \mathcal{D}$ represents that $\mathbf{x}$ is drawn from the distribution $\mathcal{D}$. Similarly, the notation $\mathbf{x}, \mathbf{y} \sim \mathcal{D}$ represents that $\mathbf{x}$ and $\mathbf{y}$ are jointly drawn from the distribution $\mathcal{D}$, and finally $\mathbf{x}|\mathbf{y} \sim \mathcal{D}$ represents that $\mathbf{x}$ is drawn from the distribution after observing (or conditioned on) the random variable $\mathbf{y}$. The following are the operators on $\mathbf{x}$: $\mathbb{E}[\mathbf{x}]$: expected value of $\mathbf{x}$, $\mathrm{var}[\mathbf{x}]$: variance of $\mathbf{x}$, $\mathrm{Cov}[\mathbf{x}, \mathbf{y}]$: covariance of $\mathbf{x}$ and $\mathbf{y}$. $\hat{\mathbf{x}}$ denotes the estimated value for the random variable $\mathbf{x}$.

4. Energy Minimization

This section formalizes the problem of minimizing an application's energy consumption for some *performance constraint*; *i.e.,*work that should be accomplished by a particular deadline. We assume a configurable system where each configuration has different application-specific performance and power characteristics. Our aim is to select the configuration that finishes the work by the deadline while minimizing the energy consumption.

Formally, the application must accomplish W work units in time T. The system has a set of configurations (*e.g.,*combinations of cores and clockspeeds) denoted by $\mathcal{C}$. Assuming that each configuration $c \in \mathcal{C}$ has an application-specific performance (or work rate) r_c and power consumption p_c, then we formulate the energy minimization problem as a linear program in Equation (1):

$$\begin{aligned} \min_{\mathbf{t} \geq \mathbf{0}} \quad & \sum_{c \in \mathcal{C}} p_c t_c, \\ \text{subject to} \quad & \sum_{c \in \mathcal{C}} r_c t_c = W, \\ & \sum_{c \in \mathcal{C}} t_c \leq T. \end{aligned} \tag{1}$$

where p_c: Power consumed when running on c^{th} configuration; r_c: Performance rate when running on c^{th} configuration; W: Work that needs to be done by the application; t_c: Time spent by the application in c^{th} configuration; T: Total run time of the application. The linear program above finds the times t_c during which the application runs in the c^{th} configuration so as to minimize the total energy consumption and ensure all work is completed by the deadline. The values p_c and r_c are the key to solving this problem. If they are known, the structure of this linear program allows the minimal energy schedule to be found using convex optimization techniques [6].

This formulation is abstract so that it can be applied to many applications and systems. To help build intuition, we relate it to our `Kmeans` example. For `Kmeans` the workload is the number of samples to cluster. The deadline T is the time by which the clustering must be completed. Configurations represent assigning `Kmeans` different resources. In Section 2, we restricted configurations to be assignment of cores. In Section 6, we will expand configurations to include assignment of cores, clockspeed, memory controllers, and hyperthreads. For `Kmeans`, each assignment of resources results in a different rate of computation (points clustered per time) and power consumption.

Unfortunately, power and performance are entirely application dependent. For many applications, these values also vary with varying inputs. Hence, for any new application in use we do not know the values of these coefficients. One way to solve the problem would be run this new application on each configuration in a brute force manner. But, as we pointed out earlier, we might have very large number of configurations and the brute force approach may not be tractable. Alternatively, we can just run the application in small subset of configurations and use these measurements to estimate the behavior of unmeasured configurations. We might also consider using the data from other applications from the same system to estimate these parameters (we can collect this data offline). The question now is how do we utilize this data to find our estimates. One simple yet clever thing would be to simply take a mean of p_c (similarly for r_c) across all the applications. Now, this offline method will work well for any application that follows the general trend exhibited by all prior applications. Another quick solution could be to just use the small subset of collected sample and run a multivariate polynomial regression on the configuration parameters vs p_c (or r_c) to predict power (performance) in all other configurations. This online method might not work for all the applications s because they might have local minima or maxima that are not captured by a small sample. In the next section, Section 5 we give details of LEO, our solution to this problem which uses both the data from the current application and previously seen applications for fast, accurate estimates.

5. Modeling Power and Performance

5.1 Introduction to Probabilistic Graphical Models

We present an introduction to graphical models in general before we delve into the details of LEO specifically. Directed graphical models (or Bayesian networks) are a type of graphical model capturing dependence between random variables. Each node in these models denotes a random variable and the edges denote a conditional dependence among the connecting nodes. The nodes which have no edges connecting them are conditionally independent. By convention shaded nodes denote an observed variable (*i.e.,*one whose value is known), whereas the unshaded ones denote an unobserved variable. In Figure 2a, variables A and B are dependent on C. If C is observed, A and B would be independent in Figure 2b.

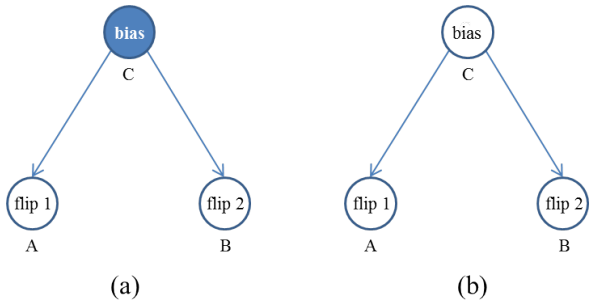


Figure 2. Conditional dependence in Bayesian Model.

The dependence structure in Bayesian networks can be understood using a coin flipping example with a biased coin. Suppose A represents the outcome of the first coin flip, B represents that of second coin flip and C represents the coin's bias. Suppose we know this bias is $P(Heads) = 0.7$, then both the flips are independent — irrespective of the first flip the second flip gives *heads* with probability 0.7. If the bias is unknown, however, then the value of B is conditionally dependent on A. Thus, knowing that $A = Heads$ increases belief that the bias is towards *Heads* – that $C > 0.5$. Therefore, the probability that the second coin flip gives *Heads* (*i.e.,*$B = Heads$) increases.

LEO exploits this conditional dependence in the presence of hidden, or unobserved, random variables. LEO models the performance and power consumption of every system configuration as a random variable drawn from a Gaussian probability distribution with unknown mean and standard deviation. *Therefore, previously observed applications will condition LEO's estimations of the performance and power for new, unobserved applications.*

5.2 Hierarchical Bayesian Model

Hierarchical Bayesian Models are slightly more complex Bayesian networks, usually with more than one layer of hidden nodes representing unobserved variables. LEO utilizes these hidden nodes to improve its estimates for a new application using prior observations from other applications. The intuition is that *knowing about one application should help in producing better predictors for other applications*. In our examples, learning about one biased coin flip should tell us something about another. Similarly, learning about another application that scales up to 8 cores should tell us something about Kmeans. LEO utilizes this conditional dependence in the problem of performance and power prediction for an application using other applications. LEO's model is explained in the figure Figure 3,

Suppose we have $n = |\mathcal{C}|$ configurations in our system. We have a *target application* whose energy we wish to minimize, while meeting a performance requirement (as in (1)). Additionally, we have a set of $M-1$ applications whose performance and power are known (as they have been measured offline).

We will illustrate how LEO estimates power as a function of system configuration. The identical process is used to estimate performance. Let the vector $\mathbf{y}_i \in \mathbb{R}^n$ represent the power estimate of application i in all n configurations of the system; *i.e.,*the cth component of $\mathbf{y}_i$ is the power for application i in configuration c (or $\mathbf{y}_i[c] = p_c$). Also, let $\{\mathbf{y}_i\}_{i=1}^{M}$ be the shorthand for the power estimates for all applications. Without loss of generality, we assume that the first $M-1$ columns, *i.e.,*$\{\mathbf{y}_i\}_{i=1}^{M-1}$ represent the data for those applications whose power consumption is known (this data is collected offline). The Mth column, $\mathbf{y}_M$ represents the power consumption for the new, unknown application. We have some small number of observations for this application. Specifically, for the Mth application we have observed configurations belonging to the set Ω_M where $|\Omega_M| \ll n$; *i.e.,*we have a very small number of observations for this application. Our objective is to estimate the power for application M for all configurations that we have not observed. The model is described in terms of statistical equations below,

$$\begin{aligned} \mathbf{y}_i|\mathbf{z}_i &\sim N(\mathbf{z}_i, \sigma^2\mathbb{I}), \\ \mathbf{z}_i|\mu, \Sigma &\sim N(\mu, \Sigma), \\ \mu, \Sigma &\sim N(\mu_0, \Sigma/\pi)IW(\Sigma|\nu, \Psi), \end{aligned} \tag{2}$$

where $\mathbf{y}_i \in \mathbb{R}^n$, $\mathbf{z}_i \in \mathbb{R}^n$, $\mu \in \mathbb{R}^n$, $\Sigma \in \mathbb{R}^{n\times n}$. It describes that the power (denoted by $\mathbf{y}_i$) for each of the i^{th} application, is drawn from multivariate-Gaussian distribution with mean $\mathbf{z}_i$ and a diagonal covariance matrix $\sigma^2\mathbb{I}$. Similarly, $\mathbf{z}_i$ is from multivariate-Gaussian distribution with mean μ and covariance Σ. And, μ and Σ are jointly drawn from normal-inverse-Wishart distribution with parameters μ_0, π, Ψ, ν. The parameters for our model are μ, Σ, whereas, μ_0, π, Ψ, ν are the hyper-parameters, which we set as $\mu_0 = 0, \pi = 1, \Psi = \mathbb{I}, \nu = 1$.

The first layer in this model as described in Figure 3, is the filtration layer and accounts for the measurement error for each application. Interestingly, even if we have just a single measurement of each configuration for each application, this layer plays a crucial role as it creates a shrinkage effect. The shrinkage effect penalizes large variations in the application and essentially help in reducing the risk of the model (See [15] for shrinkage effect and [42] for shrinkage in hierarchical models). The second layer on the other hand binds the variable $\mathbf{z}_i$ for each application and enforces that they are drawn from the same distribution with unknown mean and covariance. We work with a normal-inverse-Wishart distribution as described in [19] as our hyper prior on μ, Σ, since this distribution is the conjugate prior for a multivariate Gaussian distribution. Thus, we essentially have a normal means model at the first level of our hierarchy for each of the different apps and we have a Gaussian prior on the parameter of this model. Now, if the mean μ, covariance Σ and noise σ were known, $\mathbf{y}_i$ are conditionally independent given these

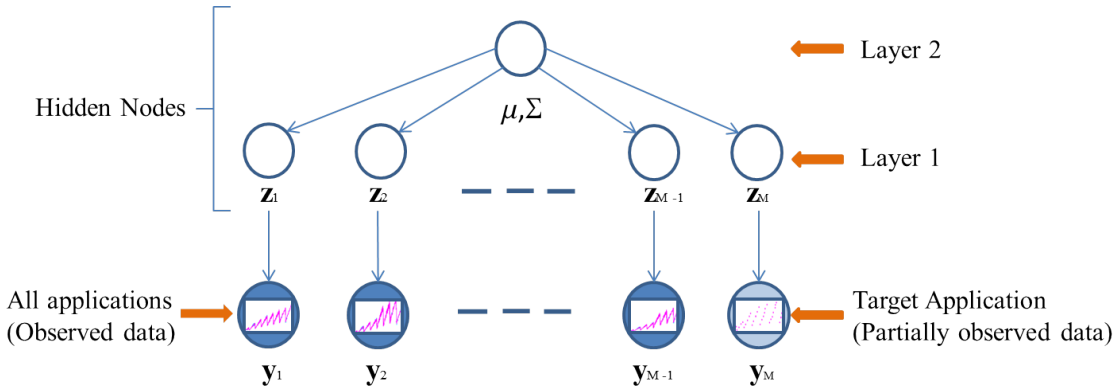


Figure 3. Hierarchical Bayesian Model.

parameters. Since they are unknown we have introduced a dependence amongst all the $\mathbf{y}_i$s. This is a similar situation to our coin flipping example in Figure 2, where the value of one coin influences our prediction about the other coin. Σ captures the correlation between different configurations as depicted in Figure 4.

We use $\theta = \{\mu, \Sigma, \sigma\}$ to denote the unknown parameters in the model. It can be shown that $\mathbf{y}_M$ is Gaussian given θ (See [60]). Thus, the problem boils down to estimating θ. Maximum-likelihood estimators are the set of values of the model parameters that maximizes the likelihood function (or the probability function of the observed outcomes given the parameter values). Essentially, the maximum-likelihood estimates of parameters are those values which most agree with the model. Suppose $\phi(\mathbf{y})$ is the set of the observed entries in vector $\mathbf{y}$. Ideally, we would like to find the maximum likelihood estimate of the parameter θ by maximizing the probability of $\mathbf{y}_M$ conditioned on $\phi(\mathbf{y}_i)_{i=1}^M$ and then use the expectation of $\mathbf{y}_M$ given $\phi(\mathbf{y}_i)_{i=1}^M$ and $\hat{\theta}$ and as our estimator for $\mathbf{y}_M$. Due to the presence of latent variables (layer 1 and layer 2 in Figure 3) , we do not have a closed form for $\Pr(\mathbf{y}_M|\{\phi(\mathbf{y}_i)\}_{i=1}^M, \theta)$ and we have to resort to the iterative algorithms like Expectation Maximization algorithm to solve this problem.

5.3 Expectation Maximization Algorithm

The EM (Expectation Maximization) algorithm is a popular approach in statistics for optimizing over analytically intractable problems. The EM algorithm switches between two steps: *expectation* (E) and *maximization* (M) until convergence. During the E step, a function for the expectation of the log of the likelihood is found using the current estimate for the parameters. In the M step, we compute parameters maximizing the expected log-likelihood found on the E step. These parameter estimates are then used to determine the distribution of the latent variables in the next E step. We have left out some details of the algebra here, but a more detailed proof on similar lines can be found here [60].

As described earlier, Ω_i is the set of observed indices for i^{th} application. Let L denote the indicator matrix with $L(i,j) = 1$ if $j \in \Omega_i$ and 0 otherwise. That is, $L(i,j) = 1$ if we have observed application i in system configuration j. We are using L_i for $L(:,i)$ for i^{th} application for shorter notation. We can write the expectation and covariance for $\mathbf{z}_i$ given θ as following,

$$\begin{aligned} \text{Cov}(\mathbf{z}_i) &= \left(\frac{\text{diag}(L_i)}{\sigma^2} + \Sigma^{-1}\right)^{-1} \quad \text{and} \\ \mathbb{E}(\mathbf{z}_i) &= \hat{\mathbf{C}}_i \left(\frac{\text{diag}(L_i)\mathbf{y}_i}{\sigma^2} + \Sigma^{-1}\mu\right). \end{aligned} \tag{3}$$

We use $\hat{\mathbf{C}}_i$ as shorthand for $\text{Cov}(\mathbf{z}_i)$ and $\hat{\mathbf{z}}_i$ denotes $\mathbb{E}(\mathbf{z}_i)$. Later, we maximize log-likelihood w.r.t. θ and taking derivative w.r.t. Σ, σ and μ and setting them to 0 gives,

$$\begin{aligned} \mu &= \frac{1}{M+\pi}\sum_{i=1}^{M}\hat{\mathbf{z}}_i, \\ \Sigma &= \frac{1}{M+1}\left(\sum_{i=1}^{M}\hat{\mathbf{C}}_i + (\hat{\mathbf{z}}_i - \mu)(\hat{\mathbf{z}}_i - \mu)'\right) + \pi\mu\mu' + \mathbb{I}, \\ \sigma^2 &= \frac{1}{\|L\|_F^2}\sum_{i=1}^{M}\text{tr}\left(\text{diag}(L_i)(\hat{\mathbf{C}}_i' + (\hat{\mathbf{z}}_i - \mathbf{y}_i)(\hat{\mathbf{z}}_i - \mathbf{y}_i)')\right), \end{aligned} \tag{4}$$

LEO iterates over the E step (equation (3)) and M step (equation (4)) until convergence to obtain the estimated parameters θ. Then, conditioned on those values of the parameters, LEO sets $\mathbf{y}_M$ as $\mathbb{E}(\mathbf{z}_M|\theta)$ given by (3). LEO uses the same algorithm to estimate performance as well.

Given performance and power estimates, the energy minimization problem can be solved using existing convex optimization techniques [24, 27, 36, 61]. LEO simply first take the estimates, then finds the set of configurations that represent Pareto-optimal performance and power tradeoffs, and

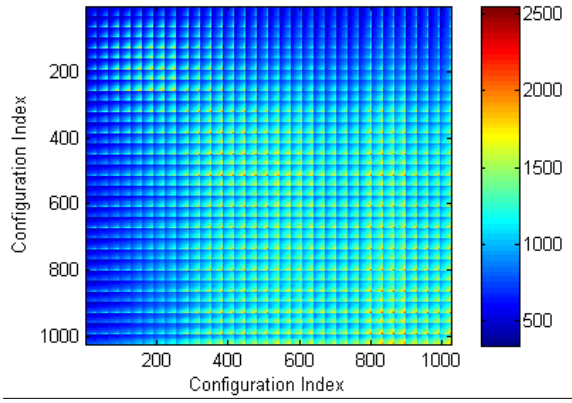


Figure 4. A illustrative example of covariance Σ between different configurations, in equation (2)

finally walks along the convex hull of this optimal tradeoff space until the performance goal is reached. The configuration representing this this point in the Pareto-optimal space is the desired tradeoff.

5.4 Example

We illustrate how LEO can be applied to our running example of `Kmeans` from Section 2. We have 32 configurations (hence $n = 32$) corresponding to cores. We also have 24 other applications (hence $M = 25$) with all the data for different configurations collected offline and denoted by $\{\mathbf{y}_i\}_{i=1}^{M-1}$; $\mathbf{y}_M$ denotes the power data for `Kmeans`. Referring to Figure 3, `Kmeans` is the final node, labeled "Target Application," whereas the rest of the applications would be the remaining nodes in any order. LEO estimates $\mathbf{z}_M$, the node above $\mathbf{y}_M$ in Figure 3, which is an unbiased estimator for $\mathbf{y}_M$. LEO collects data $\mathbf{y}_M$ for 6 different configurations (5, 10, $\cdots$, 30 cores). Hence, $\Omega_M = \{5, 10, \cdots, 30\}$ and $\mathbf{y}_M[j]$ is known iff $j \in \Omega_M$. Also, L_i or $L(:, i)$ is an all one vector of length n if $i \neq M$ and $L(j, i) = 1$ if $j \in \Omega_M$ and $L(j, i) = 0$ otherwise.

Now, we describe the main steps of LEO. The algorithm starts by setting some initialization for the parameter $\theta = \{\mu, \Sigma, \sigma\}$ and then evaluates equation (3) for each ith and later uses these values of $\hat{\mathbf{z}}_i$ and $\hat{C}_i$ to evaluate equation (4), which is fed back to expectation (3) and so on. This alternating step between equation (3) and (4) runs until the algorithm converges. The algorithm uses $\hat{\mathbf{z}}_M$ as the estimate of `Kmeans` power (*i.e.*,$p_c = \hat{\mathbf{z}}_M[c], \forall c \in \mathcal{C}$ in equation (1)). Similarly, LEO estimates the performance r_c. After the estimation step the linear program in equation (1) is solved to obtain the best configuration.

5.5 Discussion

The key to LEO is that it does not assume any parametric function to describe how the power varies along the underlying configuration knobs such as cores, memory controllers or speed settings. The upside of this representation is that LEO captures a much wider variety of applications, whereas the downside is a higher computational load. LEO finds covariance in the configurations and exploits these relationships to estimate the data for each of the configurations (See Figure 4). We want to again point out how our modeling of the problem is markedly different from some of the previous approaches (such as [12]), which assume that power and performance are convex functions of the configuration knobs and employ algorithms similar to gradient descent to find the optimal configuration. While such methods work well for most applications, it may not be suitable for more complicated applications. *In contrast, LEO assumes that there will be many local minima and maxima in the functions mapping system configuration to power and performance – LEO is designed to be robust to the presence of local extrema, but this property is achieved at a cost of higher computational complexity.*

We describe some of the properties of LEO. The EM algorithm's convergence is dependent on the initial model [56]. We can initialize the algorithm randomly. Empirically, however, we observe that the initialization of μ with the estimates from the online or offline approaches (given in Section 6.2) improves LEO's accuracy. Experimentally we have observed that the algorithm converges quickly for our benchmark sets, generally requiring 3-4 iterations to reach the desired accuracy. We discuss the overhead of LEO further in Section 6.7.

6. Experimental Results

This section evaluates LEO's performance and power estimates, and its ability to use those estimates to minimize energy across a range of performance requirements. We begin by describing our experimental setup and the approaches to which we compare LEO. We discuss LEO's accuracy for performance and power estimates. We then show that LEO provides near optimal energy savings using these estimates. We conclude the evaluation with a sensitivity analysis showing how LEO performs with respect to different sample sizes and a measurement of LEO's overhead.

6.1 Experimental Setup

Our test platform is a dual-socket Linux 3.2.0 system with a SuperMICRO X9DRL-iF motherboard and two Intel Xeon E5-2690 processors. We use the `cpufrequtils` package to set the processor's clock speed. These processors have eight cores, fifteen DVFS settings (from 1.2 – 2.9 GHz), hyper-threading, and TurboBoost. In addition, each chip has its own memory controller, and we use the `numactl` library to control access to memory controllers. In total, the system supports 1024 user-accessible configurations, each with its own power/performance tradeoffs[3]. According to Intel's documentation, the thermal design power for these proces-

[3] 16 cores, 2 hyperthreads, 2 memory controllers, and 16 speed settings (15 DVFS settings plus TurboBoost)

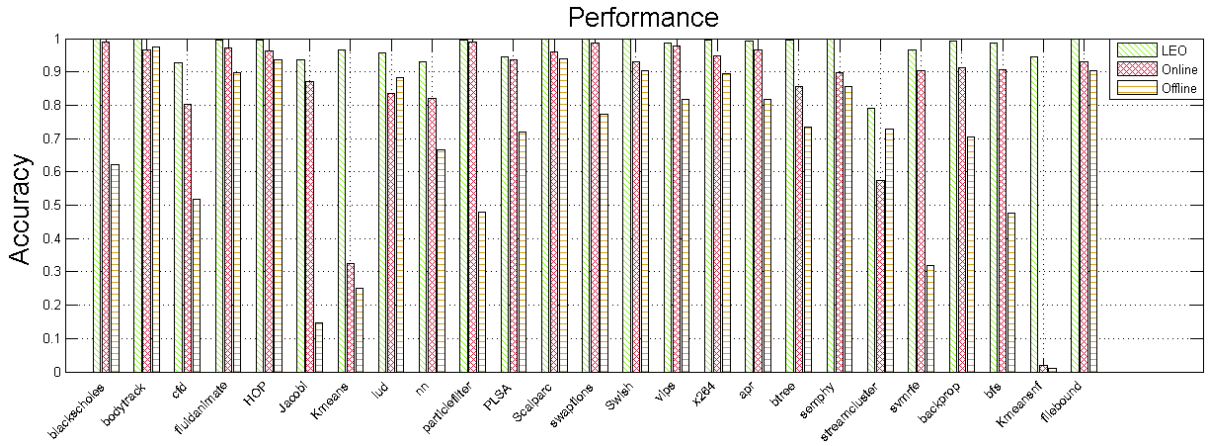


Figure 5. Comparison of performance (measured as speedup) estimation by different techniques for various benchmarks. On an average (over all benchmarks), *LEO*'s accuracy is 0.97 compared to 0.87 and 0.68 for *Online* and *Offline* respectively. The results are normalized with respect to the *Exhaustive search* method.

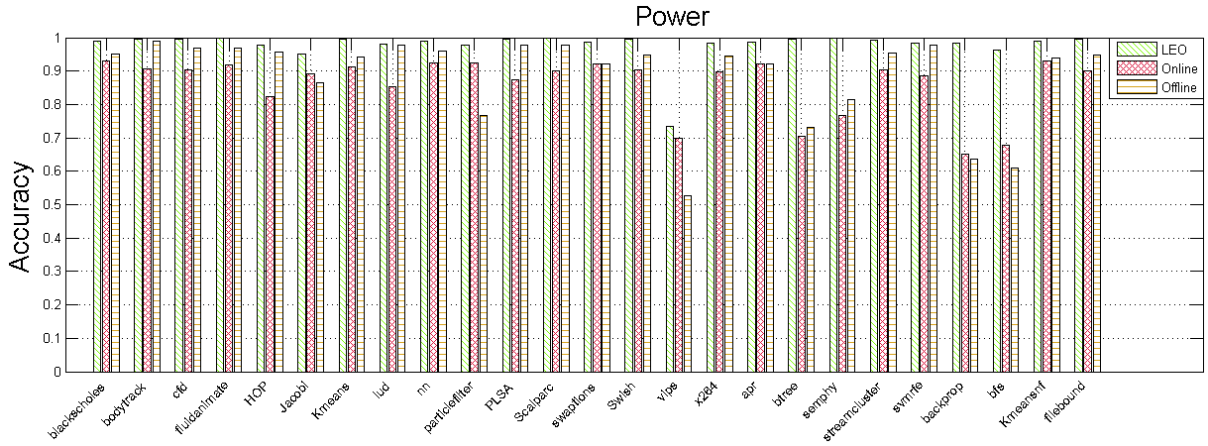
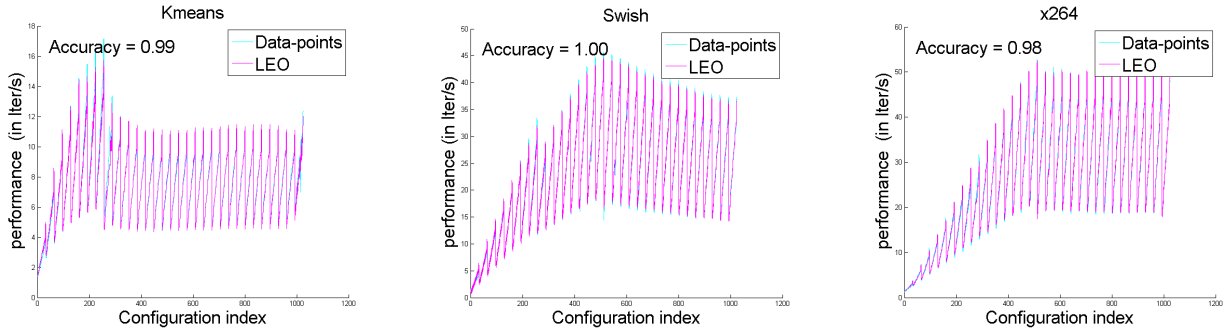


Figure 6. Comparison of power (measured in Watts) estimation by different techniques for various benchmarks. On an average (over all benchmarks), *LEO*'s accuracy is 0.98 compared to 0.85 and 0.89 for *Online* approach and *Offline* approach respectively. Again, the results are normalized with respect to the *Exhaustive search* method.

sors is 135 Watts. The system is connected to a WattsUp meter which provides total system power measurements at 1s intervals. In addition, we use Intel's RAPL power monitor to measure chip power for both sockets at finer-grain intervals. We use 25 benchmarks from three different suites including PARSEC (`blackscholes`, `bodytrack`, `fluidanimate`, `swaptions`, `x264`) [4], Minebench (`ScalParC`, `apr`, `semphy`, `svmrfe`, `Kmeans`, `HOP`, `PLSA`, `non fuzzy kmeans (Kmeansnf)`) [43], and Rodinia (`cfd`, `nn`, `lud`, `particlefilter`, `vips`, `btree`, `streamcluster`, `backprop`, `bfs`) [8]. We also use a partial differential equation solver (`jacobi`), a file intensive benchmark (`filebound` and the `swish++` search web-server [25]. These benchmarks test a range of important multi-core applications with both compute-intensive and i/o-intensive workloads. All the applications run with up to 32 threads (the maximum supported in hardware on our test machine). In addition, all workloads are long running, taking at least 10 seconds to complete. This duration gives sufficient time to measure system behavior. All applications are instrumented with the Application Heartbeats library which provides application specific performance feedback to LEO [22, 27]. Thus LEO is ensured of optimizing the performance that matters to the application. All performance results are then estimated and measured in terms of heartbeats/s. In the `Kmeans` example, this metric would represent the samples clustered per second.

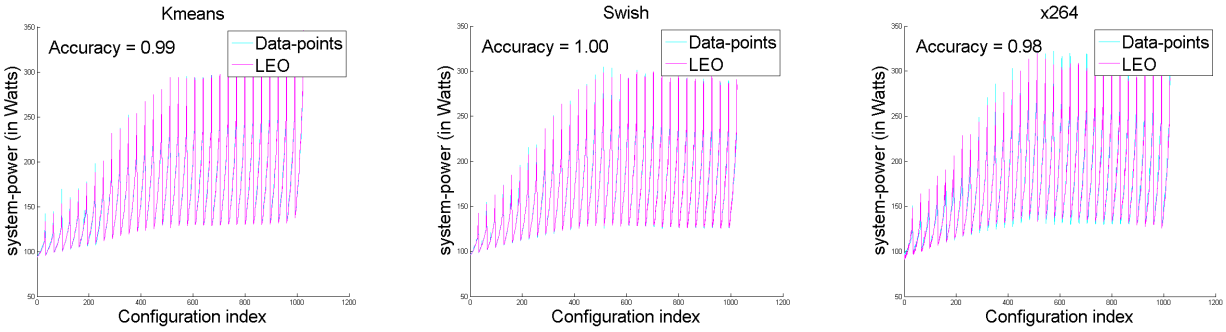
To evaluate LEO quantitatively, we measure the *accuracy* of the predicted performance and power values $\hat{\mathbf{y}}$ with respect to the true data $\mathbf{y}$ is measured as,

$$\text{accuracy}(\hat{\mathbf{y}}, \mathbf{y}) = \max\left(1 - \frac{\|\hat{\mathbf{y}} - \mathbf{y}\|_2^2}{\|\mathbf{y} - \bar{\mathbf{y}}\|_2^2}, 0\right). \quad (5)$$



(a) (b) (c)

Figure 7. Examples of performance estimation using LEO. Performance is measured as application iterations (or heartbeats) per second. (See Section 6.1).



(a) (b) (c)

Figure 8. Examples of power estimation using LEO. Power is measured as total system power.

6.2 Points of Comparison

We evaluate LEO in comparison to four baselines:

1. *Race-to-idle* – This approach allocates all resources to the application and once it is finished the system goes to idle. This strategy incurs almost no runtime overhead, but may be suboptimal in terms of energy, since maximum resource allocation is not always the best solution to the energy minimization equation (1) [7, 21, 32].
2. *Online* – This strategy carries out polynomial multivariate regression on the observed dataset using configuration values (the number of cores, memory control and speed-settings) as predictors, and estimates the rest of the data-points based the same model. Then it solves the linear program given by (1). This method uses only the observations and not the prior data.
3. *Offline* – This method takes the mean over the rest of the applications to estimate the power and performance of the given application and uses these predictions to solve for minimal energy. This strategy only uses prior information and we does not update based on runtime observations.
4. *Exhaustive search* – This brute-force approach searches every possible configuration to determine the true performance, power, and optimal energy for all applications.

6.3 Power and Performance using LEO

We compare LEO's estimates to the *online*, *offline*, and exhaustive search methods described in Section 6.2. We deploy each of our 25 applications on our test system and estimate performance and power. We allow LEO and the online method to sample randomly select 20 configurations each. Unlike online method, which only uses these 20 samples, LEO utilizes these 20 samples along with all the data from the other applications for the estimation purpose. For both LEO and the online approach, we take the average estimates produced over 10 separate trials to account for random variations. The offline approach does no sampling. The exhaustive approach samples all 1024 configurations.

The performance and power estimation accuracies are shown in Figure 5 and Figure 6, respectively. Each chart shows the benchmarks on the x-axis and estimation accuracy (computed using Equation 5) on the y-axis. Unity represents perfect accuracy. As seen in these charts, LEO produces significantly higher accuracy for both performance and power. On average – across all benchmarks and all configurations – LEO's estimations achieve 0.97 accuracy for performance and 0.98 for power. In contrast, the online approach achieves accuracies of 0.87 and 0.85, while the offline approach's accuracies are 0.68 and 0.89. Even for difficult benchmarks

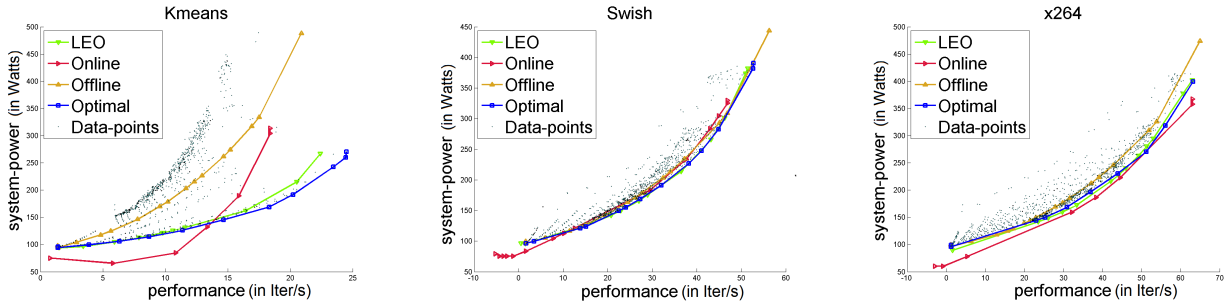


Figure 9. Pareto frontier for power and performance estimation using different estimation algorithms. We compare estimated Pareto-optimal frontiers to the true frontier found with exhaustive search, providing insight into how LEO solves equation (1). When the estimated curves are below optimal plots, it represents worse performance i.e. missed deadlines, whereas the estimations above the optimal waste energy.

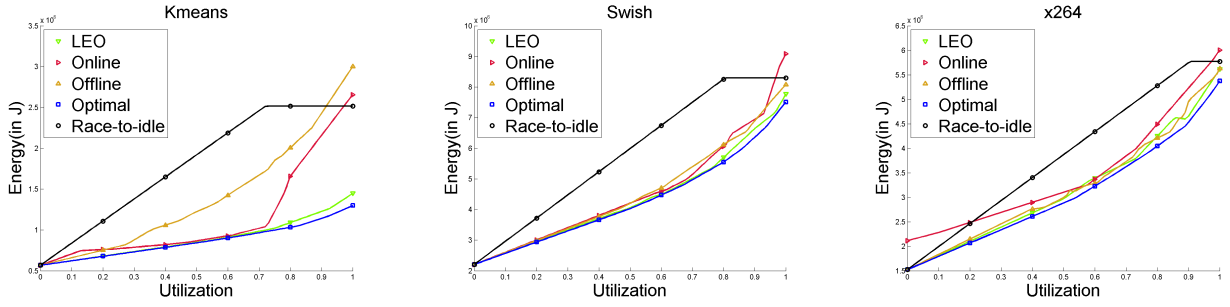


Figure 10. Energy consumption vs utilization for different estimation algorithms.

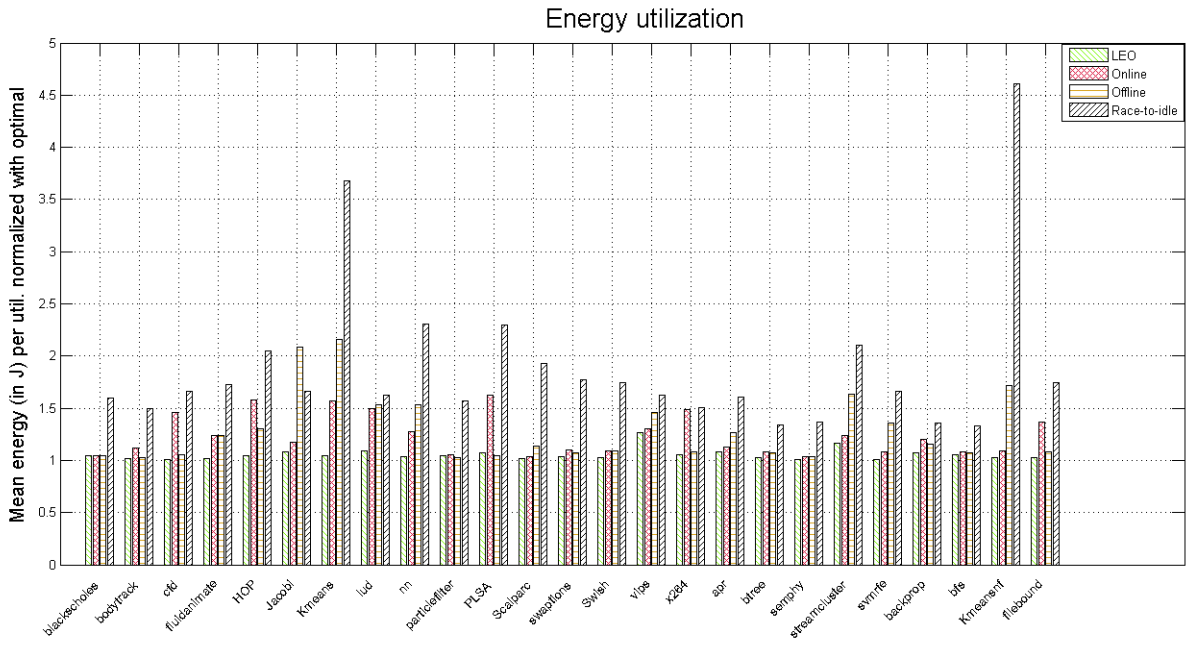


Figure 11. Comparison of average energy (normalized to optimal) by different estimation techniques for various benchmarks. On an average (taken over all the benchmarks); *LEO* consumes 6% over optimal, as compared to the *Online*, *Offline*, and *Race-to-idle* approaches, which respectively consume 24%, 29% and 90% more energy than optimal.

(like Kmeans), LEO produces accurate estimations despite sampling less than 2% of the possible configuration space.

To further illustrate LEO, we include some individual estimations for three representative applications: Kmeans, Swish, and x264. Kmeans is the same application used in our example (Section 2), now extended to consider all 1024 configurations of the test system. Swish is an open source search web server. x264 is a video encoder. All three are representative of our target applications: they are long running and they may be launched with different performance demands. Furthermore, all three represent some unusual trends: performance for Kmeans peaks at 8 cores, for Swish it peaks at 16 cores, and for x264 it is (essentially) constant after 16 cores.

Despite this behavior, LEO produces highly accurate estimates of performance (Figure 7) and power (Figure 8). Each figure shows the configuration index on the x-axis and the predicted performance (or power) on the y-axis. Each chart shows both the estimated values and the measured data points, but LEO is so accurate that it is hard to distinguish the two. The figures (Figure 7 and Figure 8) are saw-tooth in appearance since the speed settings vary from low to high along the *Configuration index* multiple times. The saw-tooth nature of the curves arises from two sources: (1) the extrema that naturally arise (*e.g.,*response to cores) and (2) we have flattened a multi-dimensional configuration space into the configuration index. The number of memory controllers is the fastest changing component of configuration, followed by clockspeed, followed by number of cores. LEO captures the peak performance configuration for all three applications and it captures local minima and maxima. These accurate estimates of unusual behavior make LEO well-suited for use in energy minimization problems.

6.4 Minimizing Energy

Our original goal, of course, is not just to estimate performance and power, but to minimize energy for a performance (or utilization) target. As described in Section 5.3, LEO uses its estimates to form the Pareto-optimal frontier of performance and power tradeoffs. Figure 9 shows the true convex hull and those estimated by the LEO, Offline and Online approaches. Due to space limitations, we show only the hulls for our three representative applications: Kmeans, Swish, and x264. In these figures performance (measured as speedup) is shown on the x-axis and system wide power consumption (in Watts) on the y-axis. These figures clearly show that LEO's more accurate estimates of power and performance produce more accurate estimates of Pareto-optimal tradeoffs.

To evaluate energy savings, we deploy each application with varying performance demands. Technically, we fix the deadline and vary the workload W from Equation 1 so that $W \in [minPerformance, maxPerformance]$ for each application. We test 100 different values for W – each representing a different utilization demand from 1 to 100% – for each application. We then use each approach to estimate power and performance and form the estimated convex hull and select the minimal energy configuration.

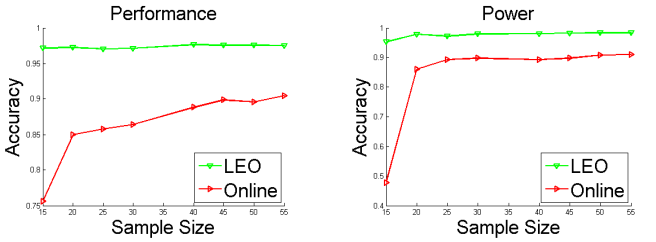


Figure 12. Sensitivity analysis of LEO and Online estimation. Our baseline method (online regression) cannot perform below 15 samples because the design matrix of regression model would be rank deficient – effectively 0 accuracy. On the other hand, with 0 samples, LEO behaves as the offline method and its accuracy increases with the sample size until it quickly reaches near optimal accuracy.

Figure 10 shows the results for our three representative benchmarks. Each chart shows the utilization demand on the x-axis and the measured energy (in Joules) on the y-axis. Each chart shows the results for the LEO, Online, and Offline estimators as well as the race-to-idle approach and the true optimal energy. As shown in these figures, LEO produces the lowest energy results across the full range of different utilization targets. LEO is always close to optimal and outperforms the other estimators. Note that all approaches do significantly better than race-to-idle. We repeat the above experiment for all applications, then average the energy consumption for each application across all utilization levels. These results are shown in Figure 11, which displays the benchmark on the x-axis and the average energy (normalized to optimal) on the y-axis. On an average across all the applications, LEO does only 6% worse than optimal. In contrast, Online, Offline and race-to-idle methods are 24% , 29% and 90% worse respectively. These results demonstrate that LEO not only produces more accurate estimates of performance and power, but that these estimates produce significant – near optimal – energy savings.

6.5 Sensitivity to Measured Samples

One of the key parameters of LEO is the number of samples it must measure to produce an accurate estimate. All of the above measurements were taken with the system configured to sample 20 configurations. In this section, we investigate the effect of the number of configurations on the accuracy of performance and power estimation. In Figure 12, we show the accuracy (averaged over all benchmarks) for performance (a) and power (b) estimation as a function of sample size. We observe that LEO performs well with an even smaller sample size, whereas the Online approach does poorly for very small sample sizes.

6.6 Reacting to Dynamic Changes

This section shows that LEO can quickly react to changes in application workload. In this section we run `fluidanimate`, which renders frames, with an input that has two distinct phases. Both phases must be completed in the same time, but the second phase requires significantly less work. In particular, the second phase requires $2/3$ the resources of the first phase. Our goal is to demonstrate that LEO can quickly react to phase changes and maintain near optimal energy consumption.

Table 1. Relative energy consumption by various algorithms with respect to optimal.

Algorithm	Phase#1	Phase#2	Overall
LEO	1.045	1.005	1.028
Offline	1.169	1.275	1.216
Online	1.325	1.248	1.291

The results of this experiment are shown in Figure 13. Each chart shows time (measured in frames) on the x-axis. Figure 13a shows performance normalized to real-time on the x-axis, while Figure 13b shows power in Watts (subtracting out idle power) on the y-axis. The dashed vertical line shows where the phase change occurs. Each chart shows the behavior for LEO, Offline, Online, and optimal approaches.

All approaches are able to meet the performance goal in both phases. This fact is not surprising as all use gradient ascent to increase performance until the demand is met. The real difference comes when looking at power consumption, however. Here we see that LEO again produces near optimal power consumption despite the presence of phases. Furthermore, this power consumption results in near optimal energy consumption as well, as shown in Table 1. These results indicate that LEO produces accurate results even in dynamically changing environments.

6.7 Overhead

The runtime takes several measurements, incurring minuscule sampling overhead. After collecting these samples, it incurs a one-time cost of executing LEO. After executing this algorithm, the models are sufficient for making predictions and LEO does not need to be executed again for the life of the application under control. This is the reason we believe LEO is best suited for long running applications which may operate at a range of different utilizations. The one-time estimation process is sufficient to provide accurate estimates for the full range of utilizations (see Section 6.4).

Therefore, we measure overhead in two ways. First, we measure the average time required to execute LEO on our system. The average execution time is 0.8 seconds across each benchmarks for each power and performance. Second, we measure the average total system energy consumption while executing the runtime, obtaining an energy overhead of 178.5 Joules. These overheads are not trivial, and they indicate (as stated in the introduction) that LEO is not appropriate for all deployments. For applications that run in the 10s of seconds to minutes or more, however, LEO's overheads are easily amortized by the large energy savings it enables. For comparison, the exhaustive search approach takes more than 5 days to produce the estimates for `semphy`. For the fastest application in our suite, `HOP`, exhaustive search takes at least 3 hours.

7. Related Work

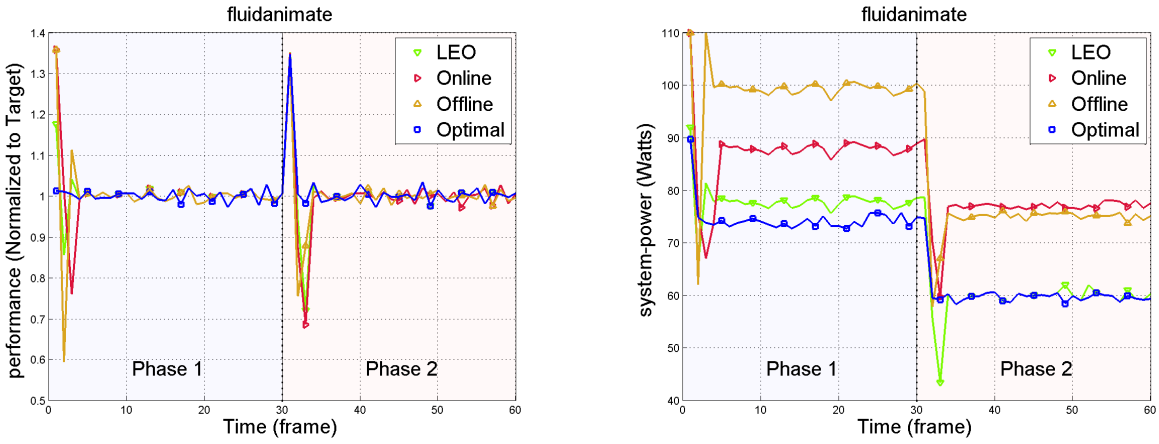
We discuss related work on energy and power optimization. Offline optimization techniques have been proposed (*e.g.,*[2, 10, 33, 35, 59], but they are limited by reliance on a robust training phase. If behavior occurs online that was not represented in the training data, then these approaches may produce suboptimal results.

Several approaches augment offline model building with online measurement. For example, many systems employ control theoretic designs which couple offline model building with online feedback control [9, 24, 27, 36, 39, 46, 47, 51, 57, 61]. Over a narrow range of applications the combination of offline learning and control works well, as the offline models capture the general behavior of the entire class of application and require negligible online overhead. This focused approach is extremely effective for multimedia applications [17, 18, 30, 39, 54] and web-servers [26, 38, 53]. The goal of LEO, however, is to build a more general framework applicable to a broad range of applications. LEO's approach is complementary to control based approaches. For example, incorporating LEO into control-based approaches might extend them to other domains even when the application characteristics are not known ahead of time.

Some approaches have combined offline predictive models with online adaptation [11, 14, 48, 50, 55, 58, 62]. For example, Dubach et al. propose such a combo for optimizing the microarchitecture of a single core [14]. Such predictive models have also been employed at the OS level to manage system energy consumption [48, 50]. [58].

Other approaches adopt an almost completely online model, optimizing based only on dynamic runtime feedback [1, 34, 37, 44, 45, 52]. For example, Flicker is a configurable architecture and optimization framework that uses only online models to maximize performance under a power limitation [44]. Another example, ParallelismDial, uses online adaptation to tailor parallelism to application workload.

Perhaps the most similar approaches to LEO are others that combine offline modeling with online model updates [5, 16, 28]. For example, Bitirgen et al use an artificial neural network to allocate resources to multiple applications in a multicore [5]. The neural network is trained offline and then adapted online using measured feedback. This approach optimizes performance but does not consider power or energy minimization.



(a) (b)

Figure 13. Power and performance for `fluidanimate` transitioning through phases with different computational demands.

Like these approaches, LEO combines offline model building and with online model updates. *Unlike prior approaches, LEO learns not a single best state, but rather all Pareto-optimal tradeoffs in the power/performance space (like those illustrated in Figure 9).* These tradeoffs can be used to maximize performance or to minimize energy across an application's entire range of possible utilization. There is a cost for this added benefit: LEO's online phase is likely higher overhead than these prior approaches that focus only on maximizing performance. In that sense, however, these approaches complement each other. If fastest performance is the goal, then prior approaches are likely the best option. If the goal is to minimize energy for a range of possible performance, then LEO produces near optimal energy.

8. Conclusion

This paper has presented LEO, a system capable of learning Pareto-optimal power and performance tradeoffs for an application running on a configurable system. LEO combines some of the best features of both online and offline learning approaches. Offline, LEO acquires knowledge about a range of application behaviors. Online, LEO quickly matches the observed behavior of a new application to previously seen behavior from other applications to produce highly accurate estimates of performance and power. We have implemented LEO, made the source code available, and tested it on a real system with 25 different applications exhibiting a range of behaviors. Across all applications, LEO achieves greater than 97% accuracy in its performance and power estimations despite only sampling less than 2% of the possible configuration space for an application it has never seen before. These estimations are then used to allocate resources and save energy. LEO produces energy savings within 6% of optimal while purely Offline or Online approaches are both over 24% of optimal. LEO's learning framework represents a promising approach to help generalize resource allocation in energy limited computing environments and could be used in conjunction with other control techniques to help develop a *self-aware* computing system [13, 20, 23, 29, 31, 49].

References

[1] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *CASES*, 2012.

[2] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *CGO*, 2011.

[3] L.A Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec 2007.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[5] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[6] S.P. Bradley, A.C. Hax, and T.L. Magnanti. *Applied mathematical programming*. Addison-Wesley Pub. Co., 1977.

[7] Aaron Carroll and Gernot Heiser. Mobile multicores: Use them or waste them. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 12:1–12:5, New York, NY, USA, 2013. ACM.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[9] Jian Chen and Lizy Kurian John. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *ICS*, 2011.

[10] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. Modeling program resource demand using inherent program characteristics. *SIGMETRICS Perform. Eval. Rev.*, 39(1):1–12, June 2011.

[11] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *MICRO*, 2011.

[12] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F Wenisch, and Ricardo Bianchini. Coscale: Coordinating cpu and memory system dvfs in server systems. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 143–154. IEEE, 2012.

[13] Petre Dini, Wolfgang Gentzsch, Mark Potts, Alexander Clemm, Mazin Yousif, and Andreas Polze. Internet, GRID, self-adaptability and beyond: Are we ready? Aug 2004.

[14] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *MICRO*, 2010.

[15] Bradley Efron and Carl Morris. Data analysis using stein's estimator and its generalizations. *Journal of the American Statistical Association*, 70(350):311–319, 1975.

[16] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *ICSE*, 2014.

[17] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP*, 1999.

[18] Jason Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Trans. Comp. Syst.*, 22(2), May 2004.

[19] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis*. CRC press, 2013.

[20] W. Gentzsch, K. Iwano, D. Johnston-Watt, M.A. Minhas, and M. Yousif. Self-adaptable autonomic computing systems: An industry view. In *Proceedings of the 16th International Workshop on Database and Expert Systems Applications*, pages 201–205, Aug 2005.

[21] Henry Hoffmann. Racing vs. pacing to idle: A comparison of heuristics for energy-aware resource allocation. In *HotPower*, 2013.

[22] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, 2010.

[23] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. Self-aware computing in the angstrom processor. In *DAC*, 2012.

[24] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. A generalized software framework for accurate and efficient managment of performance goals. In *EMSOFT*, 2013.

[25] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[26] T. Horvath, T. Abdelzaher, K. Skadron, and Xue Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *Computers, IEEE Transactions on*, 56(4), 2007.

[27] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. Poet: A portable approach to minimizing energy under soft real-time constraints. In *RTAS*, 2015.

[28] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[29] J.O. Kephart. Research challenges of autonomic computing. In *ICSE*, 2005.

[30] Minyoung Kim, Mark-Oliver Stehr, Carolyn Talcott, Nikil Dutt, and Nalini Venkatasubramanian. xtune: A formal methodology for cross-layer tuning of mobile embedded systems. *ACM Trans. Embed. Comput. Syst.*, 11(4), January 2013.

[31] Robert Laddaga. Guest editor's introduction: Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14, 1999.

[32] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *Proceedings of the 2011 USENIX Annual Technical Conference*, Portland, OR, USA, June 2011.

[33] B.C. Lee, J. Collins, Hong Wang, and D. Brooks. Cpr: Composable performance regression for scalable multiprocessor models. In *MICRO*, 2008.

[34] Benjamin C. Lee and David Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *ASPLOS*, 2008.

[35] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS*, 2006.

[36] Baochun Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9), 1999.

[37] J. Li and J.F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *HPCA*, 2006.

[38] C. Lu, Y. Lu, T.F. Abdelzaher, J.A. Stankovic, and S.H. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE TPDS*, 17(9):1014–1027, September 2006.

[39] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio an d Anant Agarwal, and Alberto Leva. Power optimization in embedded systems via feedback control of resource allocation. *IEEE Transactions on Control Systems Technology (to appear)*.

[40] Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 7(4):36:1–36:32, December 2012.

[41] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. *ISCA*, 2011.

[42] Carl N Morris. Parametric empirical bayes inference: theory and applications. *Journal of the American Statistical Association*, 78(381):47–55, 1983.

[43] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *IISWC*, 2006.

[44] Paula Petrica, Adam M. Izraelevitz, David H. Albonesi, and Christine A. Shoemaker. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *ISCA*, 2013.

[45] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *MICRO*, 2001.

[46] R. Raghavendra, P. Ranganathan, V Talwar, Z. Wang, and X. Zhu. No "power" struggles: coordinated multi-level power management for the data center. In *ASPLOS*, 2008.

[47] R. Rajkumar, C. Lee, J. Lehoczky, and Dan Siewiorek. A resource allocation model for qos management. In *RTSS*, 1997.

[48] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In *EuroSys*, 2011.

[49] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[50] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: A platform for os-level power management. In *EuroSys*, 2009.

[51] Michal Sojka, Pavel Písa, Dario Faggioli, Tommaso Cucinotta, Fabio Checconi, Zdenek Hanzálek, and Giuseppe Lipari. Modular software architecture for flexible reservation mechanisms on heterogeneous resources. *Journal of Systems Architecture*, 57(4), 2011.

[52] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *ICS*, 2013.

[53] Q. Sun, G. Dai, and W. Pan. LPV model and its application in web server performance control. In *ICCSSE*, 2008.

[54] Vibhore Vardhan, Wanghong Yuan, Albert F. Harris III, Sarita V. Adve, Robin Kravets, Klara Nahrstedt, Daniel Grobe Sachs, and Douglas L. Jones. Grace-2: integrating fine-grained application adaptation with global adaptation for saving energy. *IJES*, 4(2), 2009.

[55] Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *PACT*, 2010.

[56] CF Jeff Wu. On the convergence properties of the em algorithm. *The Annals of statistics*, pages 95–103, 1983.

[57] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *ASPLOS*, 2004.

[58] Weidan Wu and Benjamin C Lee. Inferred models for dynamic and sparse hardware-software spaces. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 413–424. IEEE, 2012.

[59] Joshua J. Yi, David J. Lilja, and Douglas M. Hawkins. A statistically rigorous approach for improving simulation methodology. In *HPCA*, 2003.

[60] Kai Yu, Volker Tresp, and Anton Schwaighofer. Learning gaussian processes from multiple tasks. In *Proceedings of the 22nd international conference on Machine learning*, pages 1012–1019. ACM, 2005.

[61] R. Zhang, C. Lu, T.F. Abdelzaher, and J.A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS*, 2002.

[62] Xiao Zhang, Rongrong Zhong, Sandhya Dwarkadas, and Kai Shen. A flexible framework for throttling-enabled multicore management (temm). In *ICPP*, 2012.