A STATISTICAL APPROACH FOR MINIMIZING ENERGY UNDER PERFORMANCE CONSTRAINTS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE DIVISION
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
NIKITA MISHRA

CHICAGO, ILLINOIS
MARCH 22, 2015
Dedicated to Science
“If I have seen further it is by standing on the shoulders of Giants.”

- Isaac Newton
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 RELATED WORK</td>
<td>5</td>
</tr>
<tr>
<td>3 ENERGY MINIMIZATION</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Motivational Example</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Notations</td>
<td>9</td>
</tr>
<tr>
<td>4 MODELLING POWER AND PERFORMANCE</td>
<td>13</td>
</tr>
<tr>
<td>4.1 Introduction to Probabilistic Graphical Models</td>
<td>13</td>
</tr>
<tr>
<td>4.2 Hierarchical Bayesian Model</td>
<td>14</td>
</tr>
<tr>
<td>4.3 Expectation Maximization Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>4.4 Example</td>
<td>19</td>
</tr>
<tr>
<td>4.5 Discussion</td>
<td>20</td>
</tr>
<tr>
<td>5 EXPERIMENTAL RESULTS</td>
<td>22</td>
</tr>
<tr>
<td>5.1 Experimental Setup</td>
<td>22</td>
</tr>
<tr>
<td>5.2 Points of Comparison</td>
<td>25</td>
</tr>
<tr>
<td>5.3 Power and Performance using LEO</td>
<td>28</td>
</tr>
<tr>
<td>5.4 Minimizing Energy</td>
<td>29</td>
</tr>
<tr>
<td>5.5 Sensitivity to Measured Samples</td>
<td>30</td>
</tr>
<tr>
<td>5.6 Reacting to Dynamic Changes</td>
<td>31</td>
</tr>
<tr>
<td>5.7 Overhead</td>
<td>32</td>
</tr>
<tr>
<td>6 CONCLUSION AND FUTURE WORK</td>
<td>34</td>
</tr>
<tr>
<td>6.1 Future work</td>
<td>34</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

3.1 Power estimation for \texttt{kmeans} clustering application using LEO, \textit{Online} and \textit{Offline} algorithms. The estimations are made using only 6 observed values (Cores) out of 32. ................................................................. 8

4.1 Conditional dependence in Bayesian Model. ........................................... 13
4.2 Hierarchical Bayesian Model. .............................................................. 14
4.3 A illustrative example of covariance $\Sigma$ between different configurations, in equation (4.1) .......................................................... 21

5.1 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 \textit{Online} and \textit{Offline} respectively. The results are normalized with respect to the \textit{Exhaustive search} method. .... 23
5.2 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 \textit{Online} approach and \textit{Offline} approach respectively. Again, the results are normalized with respect to the \textit{Exhaustive search} method. 23
5.3 Examples of performance estimation using LEO. Performance is measured as application iterations (or heartbeats) per second. (See Section 5.1). ............ 24
5.4 Examples of power estimation using LEO. Power is measured as total system power. ................................................................. 24
5.5 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 (3.1). When the estimated curves are below optimal plots, it represents worse performance i.e. missed deadlines, whereas the estimations above the optimal waste energy. ................................................................. 26
5.6 Energy consumption vs utilization for different estimation algorithms. ........ 26
5.7 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 \textit{Online}, \textit{Offline}, and \textit{Race-to-idle} approaches, which respectively consume 24%, 29% and 90% more energy than optimal. ............................................................. 27
5.8 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. ............ 31
5.9 Power and performance for \texttt{fluidanimate} transitioning through phases with different computational demands. .................................................. 32

6.1 A sequence of applications processing a stream of data. ......................... 35
LIST OF TABLES

5.1 Relative energy consumption by various algorithms with respect to optimal. . . 31
ACKNOWLEDGMENTS

I would like to take this as an opportunity to express my profound sense of gratitude to Prof. John Lafferty and Prof. Hank Hoffmann for their valuable inspiration and guidance throughout this project. Thanks John for being such a wonderful guide and showing so much patience during all these years. I have realized that all machine learning questions have multiple solutions, none of which are absolutely right. And as researchers we have to find the least wrong model for the problem. I am grateful to John since he taught me how get that intuition.

Finding the right problem to work on is always the biggest hurdle in any project and I must thank Hank for suggesting this unique problem in computer architecture. Thanks for always being around and providing every possible help and teaching me the scientific temperament.

I am grateful to all my friends who were involved directly or indirectly during the project. Last but not the least I would like to thank my family members for their support, inspiration and good wishes.
ABSTRACT

This thesis deals with the application of statistical techniques particularly graphical models to optimize the energy utilization of applications subject to certain performance constraint. 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.

We explore the problem of minimizing the energy consumed by a new application subject to some power constraint; e.g., real-time or quality-of-service requirement. Specifically, we study the problem of online identification of power and performance tradeoffs. We compare our results to the previous works which involves the convex optimization techniques and the optimal offline approach which does an offline profiling of the applications. In this work, we present an online adaptive learning algorithm for the identification of the Pareto optimal frontier of power and performance. We use a hierarchical Bayesian network to model this problem which essentially strengthens the estimators of power/performance of other application by using the data from applications.

Specifically, in thesis 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.
CHAPTER 1
INTRODUCTION

This work 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 [59, 50, 35, 33, 10]). Online learning approaches use information collected while an application is running to quickly estimate the optimal configuration (examples include [37, 44, 52, 45, 1, 34, 40]). 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 work, 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 of-
offline approach, and (3) an online approach (See Section 5.2). 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 5.4). 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 work 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 Chapter 4).
- 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 5.1).
- It compares the accuracy of LEO’s estimations to both the truth and to offline and online learning approaches (See Section 5.3).
- 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 5.4).

The rest of the thesis is organized as follows. Chapter 3 provides a motivational example to build intuition and formalizes the energy minimization problem as a linear program and discusses the application-specific parameters of this problem. Chapter 4 elaborates our probabilistic graphical model and describes LEO in full detail. Chapter 5 presents empirical

1. leo.cs.uchicago.edu
studies on LEO. Related work is discussed in Chapter 2 and concludes in Chapter 6.
CHAPTER 2
RELATED WORK

We discuss related work on energy and power optimization. Offline optimization techniques have been proposed (e.g.,[59, 35, 33, 10, 2], 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 [57, 39, 9, 24, 27, 61, 36, 47, 51, 46]. 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 [54, 17, 18, 30, 39] 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 [62, 11, 55, 14, 50, 48, 58]. 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 [50, 48]. [58].

Other approaches adopt an almost completely online model, optimizing based only on dynamic runtime feedback [37, 44, 52, 45, 1, 34]. 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 [16, 5, 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.

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 5.5). 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.
CHAPTER 3

ENERGY MINIMIZATION

3.1 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)

1. 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 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 5.2).

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

1. Our full system evaluation tests more parameters than simply core allocation. See Section 5 for details.
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 3.1 shows the results for this example. Figure 3.1a shows each approach’s performance estimates as a function of cores, while Figure 3.1b shows the estimate of power consumption. These runtime estimates are then used to determine the minimal energy configuration for various system utilizations. Figure 3.1c 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 \texttt{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 \texttt{Kmeans} follows this pattern and LEO produces accurate estimates with just a small number of observations.

The next three sections formalize this example. Section 3.2 describes the notation we will use. Section 3 presents a general formalization of this energy minimization problem for any configurable system (not just cores). Section 4 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.

\section{3.2 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 $L_2$ 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., $\| \mathbf{X} \|_F = \sqrt{\sum_{i=1}^{d} \sum_{j=1}^{n} X^2[i][j]}$. Let $\mathbf{A} \in \mathbb{R}^{d \times d}$ denote a d-dimensional square matrix. $\text{tr}(\mathbf{A})$ is the trace of the matrix $\mathbf{A}$ and
is given as, \( \text{tr}(A) = \sum_{i=1}^{d} A[i][i] \). And, \( \text{diag}(x) \) is a \( d \)-dimensional diagonal matrix \( B \) with the diagonal elements given as, \( B[i][i] = x[i] \) and off-diagonal elements being 0.

We now review the standard statistical notation used below. Let \( x, y \) denote any random variables in \( \mathbb{R}^d \). The notation \( x \sim D \) represents that \( x \) is drawn from the distribution \( D \). Similarly, the notation \( x, y \sim D \) represents that \( x \) and \( y \) are jointly drawn from the distribution \( D \), and finally \( x|y \sim D \) represents that \( x \) is drawn from the distribution after observing (or conditioned on) the random variable \( y \). The following are the operators on \( x \): \( \mathbb{E}[x] \) : expected value of \( x \), \( \text{var}[x] \) : variance of \( x \), \( \text{Cov}[x, y] \) : covariance of \( x \) and \( y \). \( \hat{x} \) denotes the estimated value for the random variable \( x \). 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 \( C \). Assuming that each configuration \( c \in C \) 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 (3.1):

\[
\begin{align*}
\min_{t \geq 0} & \quad \sum_{c \in C} p_c t_c, \\
\text{subject to} & \quad \sum_{c \in C} r_c t_c = W, \\
& \quad \sum_{c \in C} t_c \leq T.
\end{align*}
\]

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 Chapter 3, we restricted configurations to be assignment of cores. In Chapter 5, 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 because they might have local minima or maxima that are not captured by a small sample. In the next section, Chapter 4 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.
CHAPTER 4
MODELLING POWER AND PERFORMANCE

4.1 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 4.1a, variables A and B are dependent on C. If C is observed, A and B would be independent in Figure 4.1b.

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

![Figure 4.1: Conditional dependence in Bayesian Model.](image-url)
that the bias is towards Heads – that $C > 0.5$. Therefore, the probability that the second coin flip gives Heads (i.e., $B = \text{Heads}$) 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.

### 4.2 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 4.2,

Suppose we have $n = |C|$ configurations in our system. We have a target application whose energy we wish to minimize, while meeting a performance requirement (as in (3.1)). 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 $c$th 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 $M$th 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 $M$th application we have observed configurations belonging to the set $\Omega_M$ 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,

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

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 $\sigma^2 I$. Similarly, $z_i$ is from multivariate-Gaussian distribution with mean $\mu$ and covariance $\Sigma$. And, $\mu$ and $\Sigma$ are jointly drawn from normal-inverse-Wishart distribution with parameters $\mu_0, \pi, \Psi, \nu$. The parameters for our model are $\mu, \Sigma$, whereas, $\mu_0, \pi, \Psi, \nu$ are the hyper-parameters, which we set as $\mu_0 = 0, \pi = 1, \Psi = I, \nu = 1$.

The first layer in this model as described in Figure 4.2, 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 $z_i$ 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 $\mu, \Sigma$, 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 $\mu$, covariance $\Sigma$ and noise $\sigma$ were known, $y_i$ are conditionally independent given these parameters. Since they are unknown we have introduced a dependence amongst all the $y_i$s. This is a similar situation to our coin flipping example in Figure 4.1, where the value of one coin influences our prediction about the other coin. $\Sigma$ captures the correlation between different configurations as depicted in Figure 4.3.

We use $\theta = \{\mu, \Sigma, \sigma\}$ to denote the unknown parameters in the model. It can be shown that $y_M$ is Gaussian given $\theta$ (See [60]). Thus, the problem boils down to estimating $\theta$. 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 $\phi(y)$ is the set of the observed entries in vector $y$. Ideally, we would like to find the maximum likelihood estimate of the
Algorithm 1 LEO algorithm for power estimation

Input: M ← Number of applications, n ← Number of configurations. \{y_i\}_{i=1}^M ← Power measurements for M applications. \Omega_i ← Known indices in y_i. \Omega ← \{\Omega_i\}_{i=1}^M, \epsilon ← Tolerance to control convergence.

1: Construct indicator matrix \(L \in \mathbb{R}^{nxM}\) from \(\Omega\). \(L_i(j) = 1\) if \(j \in \Omega_i\), \(L_i(j) = 0\) otherwise.
2: Initialize \(\hat{\theta} = \mu, \Sigma, \sigma\) and likelihood \(\hat{L}_0, \epsilon\).
3: Set \(\hat{L} = 2\hat{L}_0\).
4: repeat
5: Expectation step: Compute \(C_i\) and \(\hat{z}_i\) using (4.2),
6: Maximization step: Compute \(\hat{\theta} = (\mu, \Sigma, \sigma)\) using (4.5),
7: Calculate likelihood \(\hat{L} = \mathcal{L}\left(\theta|\{\phi(y_i)\}_i^M, \{\hat{z}_i\}_i^M\right)\) using Equation (4.3).
8: Set \(\hat{L} = \hat{L}_0\).
9: until \(\|\hat{L} - \hat{L}_0\| / \hat{L}_0 > \epsilon\)
10: \(\hat{y}_M = \hat{z}_M\)
11: return \(\hat{y}_M\).

parameter \(\theta\) by maximizing the probability of \(y_M\) conditioned on \(\phi(y_i)_{i=1}^M\) and then use the expectation of \(y_M\) given \(\phi(y_i)_{i=1}^M\) and \(\hat{\theta}\) and as our estimator for \(y_M\). Due to the presence of latent variables (layer 1 and layer 2 in Figure 4.2), we do not have a closed form for \(\Pr(y_M|\{\phi(y_i)\}_{i=1}^M, \theta)\) and we have to resort to the iterative algorithms like Expectation Maximization algorithm to solve this problem.

4.3 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, $\Omega_i$ 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 $L(\cdot, i)$ for $i^{th}$ application for shorter notation. We can write the expectation and covariance for $z_i$ given $\theta$ as following,

$$
\text{Cov}(z_i) = \left(\frac{\text{diag}(L_i)}{\sigma^2} + \Sigma^{-1}\right)^{-1} \text{and}
\mathbb{E}(z_i) = \hat{C}_i \left(\frac{\text{diag}(L_i)y_i}{\sigma^2} + \Sigma^{-1}\mu\right).
$$

E step: We calculate the expected log-likelihood as,

$$
Q(\theta) = \mathbb{E}_{z_i^M|\{\phi(y_i)\}^M, \mu, \Sigma}(-\log(f(\{\phi(y_i)\}^M, \{z_i\}^M|\theta) f(\mu, \Sigma)),
$$

The log of the likelihood function is given by,

$$
L(\theta|\{\phi(y_i)\}^M, \{z_i\}^M) = \log(f(\{\phi(y_i)\}^M, \{z_i\}^M|\theta)),
$$

$$
= -\frac{1}{2} \sum_{i=1}^{M} \left[ y_i' \left(\frac{\text{diag}(L_i)}{\sigma^2} + \Sigma^{-1}\right) z_i - 2 \left(\frac{y_i'\text{diag}(L_i)}{\sigma^2} + \mu'\Sigma^{-1}\right) \hat{z}_i \right. \\
\left. + \frac{y_i'\text{diag}(L_i) y_i}{\sigma^2} + \mu'\Sigma^{-1}\mu\right] - \log \left(\frac{(\sqrt{2\pi\sigma})}{\sqrt{2\pi\sigma}} \frac{||L||^2_F}{(2\pi \text{det}(\Sigma)) \frac{M}{2}}\right),
$$

Thus after some algebraic manipulation we have,

$$
Q(\theta) = \text{Const} + \sum_{i=1}^{M} \left[ \text{tr} \left(\frac{\text{diag}(L_i)}{\sigma^2} + \Sigma^{-1}\right) \left(\hat{C}_i + \hat{z}_i \hat{z}_i'\right) \right. \\
- 2 \left(\frac{y_i'\text{diag}(L_i)}{\sigma^2} + \mu'\Sigma^{-1}\right) \hat{z}_i + \frac{y_i'\text{diag}(L_i) y_i}{\sigma^2} \\
+ \mu'\Sigma^{-1}\mu + \frac{||L||^2_F}{M} \log(\sigma^2) + \log(\text{det}(\Sigma))] \\
+ \log(\text{det}(\Sigma)) + \pi \mu'\Sigma^{-1}\mu + \text{tr}(\Sigma^{-1}),
$$
We use $\hat{C}_i$ as shorthand for $\text{Cov}(z_i)$ and $\hat{z}_i$ denotes $\mathbb{E}(z_i)$. Later, we maximize log-likelihood w.r.t. $\theta$ and taking derivative w.r.t. $\Sigma, \sigma$ and $\mu$ and setting them to 0 gives,

$$\mu = \frac{1}{M + \pi} \sum_{i=1}^{M} \hat{z}_i,$$

$$\Sigma = \frac{1}{M + 1} \left( \sum_{i=1}^{M} \hat{C}_i + (\hat{z}_i - \mu)(\hat{z}_i - \mu)' \right) + \pi \mu \mu' + I,$$

$$\sigma^2 = \frac{1}{\|L\|_F^2} \sum_{i=1}^{M} \text{tr} \left( \text{diag}(L_i)(\hat{C}_i' + (\hat{z}_i - y_i)(\hat{z}_i - y_i)') \right), \quad (4.5)$$

Algorithm 1 summarizes how LEO applies the E and M steps to use prior observations to estimate values for unseen configurations.

LEO iterates over the E step (equation (4.2)) and M step (equation (4.5)) until convergence and the convergence criterion checks if the relative likelihood (measured using Equation (4.3)) is sufficiently small. Then, conditioned on those values of the parameters, LEO sets $y_M$ as $\mathbb{E}(z_M|\theta)$ given by (4.2). 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, 61, 36, 27]. LEO simply first take the estimates, then finds the set of configurations that represent Pareto-optimal performance and power tradeoffs, and 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.

### 4.4 Example

We illustrate how LEO can be applied to our running example of Kmeans from Chapter 3.1. 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 $\{y_i\}_{i=1}^{M-1}$; $y_M$ denotes the power data for Kmeans. Referring to Figure 4.2, \texttt{Kmeans} is the final node, labeled “Target Application,” whereas the rest of the applications would be the remaining nodes in any order. LEO estimates $z_M$, the node above $y_M$ in Figure 4.2, which is an unbiased estimator for $y_M$. LEO collects data $y_M$ for 6 different configurations (5, 10, · · · , 30 cores). Hence, $\Omega_M = \{5, 10, \cdots , 30\}$ and $y_M[j]$ 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 (4.2) for each $i$th and later uses these values of $\hat{z}_i$ and $\hat{C}_i$ to evaluate equation (4.5), which is fed back to expectation (4.2) and so on. This alternating step between equation (4.2) and (4.5) runs until the algorithm converges. The algorithm uses $\hat{z}_M$ as the estimate of Kmeans power (i.e., $p_c = \hat{z}_M[c], \forall c \in C$ in equation (3.1)). Similarly, LEO estimates the performance $r_c$. After the estimation step the linear program in equation (3.1) is solved to obtain the best configuration.

4.5 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 4.3). 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 $\mu$ with the estimates from the online or offline approaches (given in Section 5.2) 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 5.7.
CHAPTER 5

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.

5.1 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\(^1\). According to Intel’s documentation, the thermal design power for these processors 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, 

\(^1\) 16 cores, 2 hyperthreads, 2 memory controllers, and 16 speed settings (15 DVFS settings plus Turbo-Boost)
Figure 5.1: 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 5.2: 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.
Figure 5.3: Examples of performance estimation using LEO. Performance is measured as application iterations (or heartbeats) per second. (See Section 5.1).

Figure 5.4: Examples of power estimation using LEO. Power is measured as total system power.
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 $\hat{y}$ with respect to the true data $y$ is measured as,

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

(5.1)

### 5.2 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 (3.1) [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 (3.1). This method uses only the observa-
Figure 5.5: 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 (3.1). 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 5.6: Energy consumption vs utilization for different estimation algorithms.

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 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.
Figure 5.7: 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.
5.3 Power and Performance using LEO

We compare LEO’s estimates to the online, offline, and exhaustive search methods described in Section 5.2. 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 5.1 and Figure 5.2, respectively. Each chart shows the benchmarks on the x-axis and estimation accuracy (computed using Equation 5.1) 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 (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 3.1), 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 5.3) and power (Figure 5.4). 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 5.3 and Figure 5.4) 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.

### 5.4 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 4.3, LEO uses its estimates to form the Pareto-optimal frontier of performance and power tradeoffs. Figure 5.5 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 3.1 so
that $W \in [\text{minPerformance}, \text{maxPerformance}]$ 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 5.6 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 5.7, 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.

### 5.5 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 5.8, 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
5.6 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 5.1: Relative energy consumption by various algorithms with respect to optimal.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Phase#1</th>
<th>Phase#2</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEO</td>
<td>1.045</td>
<td>1.005</td>
<td>1.028</td>
</tr>
<tr>
<td>Offline</td>
<td>1.169</td>
<td>1.275</td>
<td>1.216</td>
</tr>
<tr>
<td>Online</td>
<td>1.325</td>
<td>1.248</td>
<td>1.291</td>
</tr>
</tbody>
</table>

The results of this experiment are shown in Figure 5.9. Each chart shows time (measured in frames) on the x-axis. Figure 5.9a shows performance normalized to real-time on the x-axis, while Figure 5.9b shows power in Watts (subtracting out idle power) on the y-axis. The
Figure 5.9: Power and performance for fluidanimate transitioning through phases with different computational demands.

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 5.1. These results indicate that LEO produces accurate results even in dynamically changing environments.

5.7 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 5.4).

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.
CHAPTER 6
CONCLUSION AND FUTURE WORK

This work 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 [23, 20, 13, 49, 29, 31].

6.1 Future work

We would like to discuss the main limitation of our techniques here. Some of the limitations are due to the model assumptions, but some crop due to the way we have set up our problem itself. First, LEO is a computationally expensive algorithm since it is cubic in the size of configuration space it is not most suitable for fast running applications. Plus the algorithm itself expends certain amount of energy, so the application which is getting optimized must be large enough so that running LEO gives substantial energy savings. Second, LEO must
be well initialized to reach the optimal solution, since EM algorithm is susceptible to local minima. Third, we have not addressed the problems where multiple applications might be running. We think multiple application scenario is another interesting problem which we would like to solve as a future work and this work is just one step towards the ultimate goal.

Now, we would like to discuss the possibilities that might violate our model assumptions. Our energy optimization equation (3.1) requires the knowledge of the coefficients that is provided by LEO by sampling. The sampling itself is sometimes a difficult problem. In many problems, we may not get accurate samples of $p_c$ and $r_c$. Sometimes when multiple applications are running the samples of power estimations i.e. $p_c$ are not reliable. Similarly for the performance estimation. Let's see some concrete example where the estimation of $r_c$ (performance parameter, e.g. rate) is difficult. Suppose we have a sequence of applications which have to process the stream of data (for example in Figure 6.1 the radar data has to pass through 3 applications) and the measurement of $r_c$ would depend on the rate at which it receives frames from the previous source. Also there is another constraint in this problem that we want each individual process to run as fast as possible so that the least number of frames are dropped. Hence, for the future work it would be interesting to see how such problems can be tackled.

Figure 6.1: A sequence of applications processing a stream of data.
REFERENCES


