THE UNIVERSITY OF CHICAGO

STATISTICAL METHODS FOR IMPROVING DYNAMIC SCHEDULING AND RESOURCE USAGE IN COMPUTING SYSTEMS

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

DEPARTMENT OF COMPUTER SCIENCE

BY
NIKITA MISHRA

CHICAGO, ILLINOIS
JUNE 10, 2017

Dedication Text

Epigraph Text

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

ABSTRACT

This thesis is about using statistical methods for performance and power estimation which would allow us to develop better scheduling algorithms and also more energy efficient systems. In many deployments, computer systems are underutilized – meaning that applications have perfor- mance 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 per- formance 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 propose a set of algorithms for different scenarios to tackle this problem. First, 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. This work mostly focused on the performance estimation for single applications. As the second part of our work, we look into the estimation for application's performance when they are co-scheduled with other applications. Applications co-scheduled on the same phys- ical hardware interfere with one another by contending for shared resources [10, 19, 40, 30]. We quantify interference as slowdown, or the performance loss one application experiences in the pres- ence of co-scheduled applications. Predicting this interference ahead of time would be particularly valuable for job scheduling. Given an accurate interference prediction, a scheduler can determine optimal assignments of applications to physical machines, leading to higher throughput in batch systems and better quality-of-service for latency-sensitive applications. In data centers and super computers schedulers often have a great deal of accumulated data about past jobs and their interference, yet turning this data into effective interference predictors is difficult [19]. Fundamentally, two basic decisions must be made: (1) what features should be measured and (2) what model will map these features into an accurate prediction. The smaller the feature space, the more computationally efficient the model, but smaller feature

spaces may also miss key data and produce inaccurate models. The art to regression modeling is to manage the tradeoffs between the feature-space and the accuracy of the regression model. Using state-of-the- art statistical methods for large feature sets, such as regularized regression, the feature space and the model are determined simultaneously [13, 36, 44]. We explore such state-of-the-art regularized regression models for estimating application in- terference. We find that regularized linear regression methods require a relatively small number of features, but produce inaccurate models. In contrast, non-linear models that include interac- tion terms – i.e., permit features to be multiplied together – are more accurate, but are extremely inefficient and not practical for online scheduling. We therefore propose an efficient technique for estimating application interference based on sparse regression. We call our approach ESP for Estimating co-Scheduled Performance. The key insight in ESP is to split regression modeling into two parts: feature selection and model building. ESP uses linear techniques to perform feature selection, but uses quadratic techniques for model building. The result is a highly accurate predictor that is still practical and can be integrated into a real application scheduler. ESP assumes that there is a known (possibly very large) set of applications that may be run on the system and some offline measurements have been taken of these individual applications. Specifically, ESP measures low-level hardware features like cache misses and instructions per clock during a training phase. At runtime, applications from this set may be launched in any arbitrary combination. The goal is to efficiently predict the interference (i.e., slowdown) that ap- plications incur when co-scheduled. As, the third part of our work, we design a system called CALOREE which allows the learnt models to be combined witha controller so that the system is robust to dynamic situations with changing resource requirement. Two central challenges arise when allocating system resources to meet these conflicting goals: (1) complexity—modern hardware exposes diverse resources with com- plicated interactions—and (2) dynamics—performance must be maintained despite unpredictable changes in operating environment or input. Machine learning accurately predicts the performance of complex, interacting resources, but does not address system dynamics; control theory adjusts resource usage dynamically, but

struggles with complex resource interaction. We therefore propose CALOREE, a combination of learning and control that automatically adjusts resource usage to meet performance requirements with minimal en- ergy in complex, dynamic environments. CALOREE breaks resource allocation into two sub-tasks: learning speedup as a function of resource usage, and controlling speedup to meet performance requirements. CALOREE also defines a general interface allowing different learners to be combined with a controller while maintaining control's formal guarantees that performance will converge to the goal. We implement CALO- REE and test its ability to deliver reliable performance on heterogeneous ARM big.LITTLE architectures in both single and multi-application scenarios. Compared to state-of-the- art learning and control solutions, we find that CALOREE reduces deadline misses by 2–6x while reducing energy consumption by 7–10%.

CHAPTER 1

INTRODUCTION

1.1 Introduction

Large classes of computing systems—from embedded to cloud—must deliver reliable performance to users while minimizing energy to increase battery life or decrease operating costs. Moreover, energy is increasingly important; reducing energy consumption reduces operating costs in datacenters and increases battery life in mobile devices. Computer systems are often underutilized, meaning there are significant portions of time where application performance demands do not require the full system capacity. To address these conflicting requirements, hardware architects expose diverse, heterogeneous resources with a wide array of performance and energy tradeoffs. Software must allocate these resources to guarantee performance requirements are met with minimal energy. Barroso and Holzle [2007], Meisner et al. [2011]. The softwares must allocate 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. There are three primary difficulties in determining how to allocate heterogeneous resources.

The first is *complexity*: resources interact in complicated ways, leading to non-convex optimization spaces. 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.

The second is *dynamics*: performance requirements must be met despite unpredictable disturbances; *e.g.*, phases in input or changes in operating environment. Prior work addresses each of these difficulties individually. We want the benefits of both learning and control to ensure performance requirements are met with minimal energy in complex and dynamic environments.

The third is *application interference*: Applications co-scheduled on the same physical hardware *interfere* with one another by contending for shared resources Dwyer et al. [2012], Kambadur et al. [2012], Yang et al. [2013], Merkel et al. [2010]. By accurately predicting this interference, a scheduler can determine optimal assignments of applications to physical machines, leading to higher throughput in batch systems and better quality-of-service for latency-sensitive applications. Data center and super computer operators often have a great deal of accumulated data about past jobs and their interference, yet turning this data into effective interference predictors is difficult Kambadur et al. [2012].

In this thesis we present new statistical methods for performance and power estimation which would allow us to develop better scheduling algorithms and also more energy efficient systems.

1.2 Contributions

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.
- CALOREE creates a general control system implementation that factors out the parameters that must be learned.
- ESP (**E**stimating co-**S**cheduled **P**erformance) for application interference estimation.

1.2.1 LEO

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 Yi et al. [2003], Snowdon et al. [2009], Lee and Brooks [2006], Lee et al. [2008], Chen et al. [2011]). *Online learning* approaches use information collected while an application is running to quickly estimate the optimal configuration (examples include Li and Martinez [2006], Petrica et al. [2013], Sridharan et al. [2013], Ponomarev et al. [2001], Ansel et al. [2012], Lee and Brooks [2008], Maggio et al. [2012]). 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.

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 3.5.2). 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 3.5.4). 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 Section 3.4).
- 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 3.5).
- It compares the accuracy of LEO's estimations to both the truth and to offline and online learning approaches (See Section 3.5.3).

1. leo.cs.uchicago.edu

- 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 3.5.4).

1.2.2 CALOREE

Many machine learning approaches model the complex performance/power tradeoff spaces inherent to heterogeneous computing Zhu and Reddi [2013], Dubach et al. [2010], Bitirgen et al. [2008], Snowdon et al. [2009], Mishra et al. [2015], Petrica et al. [2013], Ponomarev et al. [2001], Delimitrou and Kozyrakis [2013]. Learning handles non-convexity, identifying local optima and moving to globally optimal solutions. These techniques are computationally expensive and lack support for dynamics; *i.e.*, when the environment changes the expensive model building process must be restarted. Control theoretic solutions provide formal guarantees that they will converge to the desired performance despite system dynamics Hellerstein et al. [2004], Chen and John [2011], Imes et al. [2015a], Zhang et al. [2002], Li and Nahrstedt [1999], Vardhan et al. [2009], Hoffmann [2015]. Control provides formal guarantees of convergence to the desired performance, but these guarantees require *ground truth* models and control will not converge if the models do not accurately capture system complexity—including local optima and non-linearities.

CALOREE creates a general control system implementation that factors out the parameters that must be learned. While traditional control designs assume all combinations of resource configurations have been measured accurately (requiring 100s or 1000s of samples Ljung [1999], Filieri et al. [2015]), CALOREE tunes its internal models and parameters while sampling only a small subset of possible configurations. Among these, the controller's *pole* is a key parameter. CALOREE automatically tunes the pole, providing formal guarantees the controller converges to the desired performance—even when behavior is estimated by a noisy learner rather than directly measured. CALOREE implements modular abstractions and self-tuning mechanisms such that the

expensive learning can be offloaded to another system and the controller itself runs in constant—$O(1)$—time. Thus, learning's high cost can be amortized as CALOREE uses *transfer learning* to apply knowledge from multiple devices and applications Pan and Yang [2010].

CALOREE's generality allows a wide range of learning techniques to be paired with its controller. So, CALOREE not only provides an advantage over existing individual learning and control techniques, it allows us to explore different combinations of learning and control to find the best.

We evaluate CALOREE by implementing the learner on an x86 server and the controller on heterogeneous ARM big.LITTLE devices. We compare CALOREE to existing, state-of-the-art learning (including polynomial regression Snowdon et al. [2009], Dubach et al. [2010], the Netflix algorithm Bell et al. [2008], Delimitrou and Kozyrakis [2013], and a hierarchical Bayesian model Mishra et al. [2015]) and control (including proportional-integral-derivative Hellerstein et al. [2004] and adaptive, or self-tuning Levine [2005]) techniques. We set performance goals—as latency requirements—for a set of benchmark applications and then measure both the percentage of time the requirements are violated and the energy for each application. We test both *single-app*, where an application runs alone, and *multi-app* environments, where other applications enter the system and compete for resources. CALOREE achieves the:

- *Most reliable performance:*
  - In the *single-app* case, the best prior technique misses 12% of deadlines on average, while CALOREE misses only 5% on average—reducing deadline misses by more than 58% compared to prior approaches.
  - In the *multi-app* case, the best prior approach averages 30% deadline misses, but CALOREE misses just 5.6% of deadlines—a huge improvement over prior work.
- *Best energy savings:* We compare to an *oracle* with a perfect model of the application, system, and future events.
  - In the *single-app* case, the best prior approach averages 12% more energy consumption than

the oracle, but CALOREE consumes only 5% more.

– In the *multi-app* case, the best prior approach averages 18% more energy than the oracle, while CALOREE consumes just 8% more.

In summary, control approaches are well suited to dynamic environments and learning techniques accurately model complex, heterogeneous processors. *To the best of our knowledge, CALOREE is the first work to combine the two to ensure application performance without prior knowledge of the controlled application.* Our formal analysis of CALOREE's convergence despite noisy inputs shows how to incorporate learned variance into control theoretic guarantees. We demonstrate the practical benefits of these contributions by implementing CALOREE for mobile/embedded processors to show it provides more reliable performance and lower energy than individual, state-of-the-art learning and control solutions.

1.2.3 ESP

To apply machine learning to build an accurate predictor from this data, two fundamental decisions must be made: (1) what *features* should be measured and (2) what *model* maps these features into an accurate prediction. Smaller feature spaces provide more computationally efficient models, but may miss key data and reduce prediction accuracy. The art to modeling is managing the tradeoffs between feature set size and the model accuracy. One family of machine learning techniques—*regularization*— addresses the particular problem where the number of features is much larger than the number of samples; *i.e.*, the problem is *ill-posed* and unsolvable with standard regression analysis. Regularization methods solve such ill-posed problems by simultaneously selecting both the features and the model Hoerl and Kennard [1988], Tibshirani [1996], Zou and Hastie [2005].

To apply machine learning to build an accurate predictor from this data, two fundamental decisions must be made: (1) what *features* should be measured and (2) what *model* maps these features into an accurate prediction. Smaller feature spaces provide more computationally efficient models, but may miss key data and reduce prediction accuracy. The art to modeling is managing

the tradeoffs between feature set size and the model accuracy. One family of machine learning techniques—*regularization*— addresses the particular problem where the number of features is much larger than the number of samples; *i.e.*, the problem is *ill-posed* and unsolvable with standard regression analysis. Regularization methods solve such ill-posed problems by simultaneously selecting both the features and the model Hoerl and Kennard [1988], Tibshirani [1996], Zou and Hastie [2005].

This work explores such state-of-the-art regularization models for predicting application interference. We find that regularized linear regression methods require a relatively small number of features, but produce inaccurate models. In contrast, non-linear models that include *interaction terms—i.e.*, permit features to be multiplied together—are more accurate, but are extremely inefficient and not practical for online scheduling.

We therefore combine linear and non-linear approaches to produce accurate and practical predictions. We call our approach ESP for **E**stimating co-**S**cheduled **P**erformance. ESP's key insight is to split regularization modeling into two parts: *feature selection* and *model building*. ESP uses linear techniques to perform feature selection, but uses quadratic techniques for model building. The result is a highly accurate predictor that is still practical and can be integrated into real application schedulers.

ESP assumes there is a known (possibly very large) set of applications that may be run on the system and some offline measurements have been taken of these individual applications. Specifically, ESP measures low-level hardware features like cache misses and instructions retired during a training phase. At runtime, applications from this set may be launched in any arbitrary combination. The goal is to efficiently predict the interference (*i.e.*, slowdown) of co-scheduled applications. We evaluate ESP by integrating it into both single and multi-node schedulers running on Linux/x86 servers. In the single-node case, we construct a batch scheduler that orders application execution to minimize the total completion time. In the multi-node case, we build a first-come-first-serve scheduler that assigns applications to nodes as they arrive to minimize ap-

plication slowdown. We compare ESP-based schedulers to prior scheduling techniques that use contention-aware heuristics to avoid interference Dey et al. [2013], Merkel et al. [2010], Tembey et al. [2014]. We also compare ESP's accuracy to predictors built with a number of cutting-edge regularization methods. We find:

- The single-node ESP schedules are, on average, 27% faster than techniques based on heuristics. Even with its runtime overhead, the ESP results are only 5% worse (on average) than an oracle that has perfect knowledge of interference and no overhead. See Section 5.3.2.

- The multi-node ESP schedules are 60% faster than activity vector based schedules and only 5% to 13% worse than an oracle. See Section 5.3.3.

- Critically, ESP produces better results as more applications are scheduled. ESP produces quantifiable performance predictions while heuristic techniques simply produce a binary decision: co-schedule or not. ESP's quantifiable predictions allow schedulers to make optimal decisions even when interference cannot be avoided. In contrast, heuristic techniques do not quantify interference and thus cannot rank decisions. As the number of applications increases, the chance of heuristics making a very poor choice in the face of unavoidable contention also increases. See Section 5.3.4.

- ESP is more accurate than existing linear regression techniques in most cases. When considering two applications, ESP is similar to existing regularized linear regression techniques. Considering more than two applications, however, ESP is uniformly more accurate than standard linear regression techniques. See Section 5.3.5. For reasons explained in Section 5.1.2, existing regularized regression methods with interaction terms cannot be evaluated for accuracy because their models are too complex to be implemented in practice.

CHAPTER 2

BACKGROUND AND FUTURE WORK

We discuss related work on energy and power optimization. Previously, some offline optimization techniques have been proposed (*e.g.*, Yi et al. [2003], Lee and Brooks [2006], Lee et al. [2008], Chen et al. [2011], Ansel et al. [2011], 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 Wu et al. [2004], Maggio et al., Chen and John [2011], Hoffmann et al. [2013], Imes et al. [2015b], Zhang et al. [2002], Li and Nahrstedt [1999], Rajkumar et al. [1997], Sojka et al. [2011], Raghavendra et al. [2008]. 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 Vardhan et al. [2009], Flinn and Satyanarayanan [1999, 2004], Kim et al. [2013], Maggio et al. and web-servers Horvath et al. [2007], Lu et al. [2006], Sun et al. [2008]. 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 Zhang et al. [2012], Cochran et al. [2011], Winter et al. [2010], Dubach et al. [2010], Snowdon et al. [2009], Roy et al. [2011], Wu and Lee [2012]. For example, Dubach et al. propose such a combo for optimizing the microarchitecture of a single core Dubach et al. [2010]. Such predictive models have also been employed at the OS level to manage system energy consumption Snowdon et al. [2009], Roy et al. [2011]. Wu and Lee [2012].

Other approaches adopt an almost completely online model, optimizing based only on dynamic

runtime feedback Li and Martinez [2006], Petrica et al. [2013], Sridharan et al. [2013], Ponomarev et al. [2001], Ansel et al. [2012], Lee and Brooks [2008]. For example, Flicker is a configurable architecture and optimization framework that uses only online models to maximize performance under a power limitation Petrica et al. [2013]. 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 Filieri et al. [2014], Bitirgen et al. [2008]. For example, Bitirgen et al use an artificial neural network to allocate resources to multiple applications in a multicore Bitirgen et al. [2008]. 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 3.9).* 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.

We discuss related work in managing resources to meet performance goals and reduce energy.

2.1 Machine Learning

We break learning for resource management into 3 categories: offline, online, and hybrid approaches.

Offline Learning These approaches build models before deployment and then use those fixed models to allocate resources Yi et al. [2003], Lee and Brooks [2006], Lee et al. [2008], Chen et al.

[2011], Ansel et al. [2011]. The model-building phase is expensive, requiring both a large number of samples and substantial computation. Applying the model online, however, is low overhead. The main drawback is that the models are not updated as the system runs: a problem for adapting workloads. Carat is a good example of an offline learner that aggregates data across multiple devices to generate a report for human users about how to configure their device to increase battery life Oliner et al. [2013]. While both Carat and CALOREE learn across devices, they have very different goals. Carat returns very high-level information to human users; *e.g.*, update a driver to extend battery life. CALOREE automatically builds and applies low-level models to save energy.

Online Learning Online techniques observe the current application to tune system resource usage for that application Li and Martinez [2006], Petrica et al. [2013], Sridharan et al. [2013], Ponomarev et al. [2001], Ansel et al. [2012], Lee and Brooks [2008]. For example, Flicker is a configurable architecture and optimization framework that uses online models to maximize performance under a power limitation Petrica et al. [2013]. Another example, ParallelismDial, uses online adaptation to tailor parallelism to application workload Sridharan et al. [2013].

Hybrid Approaches Some approaches combine offline predictive models with online adaptation Zhang et al. [2012], Cochran et al. [2011], Winter et al. [2010], Dubach et al. [2010], Snowdon et al. [2009], Roy et al. [2011], Wu and Lee [2012]. For example, Dubach et al. use hybrid models to optimize the microarchitecture of a single core Dubach et al. [2010]. Such predictive models have also been employed at the operating system level to manage system energy consumption Snowdon et al. [2009], Roy et al. [2011], Wu and Lee [2012]. Other approaches combine offline modeling with online updates Hoffmann [2015], Bitirgen et al. [2008]. For example, Bitirgen et al use an artificial neural network to allocate resources to multiple applications in a multicore Bitirgen et al. [2008]. The neural network is trained offline and then adapted online using measured feedback. This approach maximizes performance but does not consider power or energy minimization.

2.2 Control

Almost all control solutions can be thought of as a combination of offline model building with online adaptation. The model building involves substantial empirical measurement that is used to synthesize a control system Wu et al. [2004], Maggio et al., Chen and John [2011], Hoffmann et al. [2013], Imes et al. [2015a], Zhang et al. [2002], Li and Nahrstedt [1999], Rajkumar et al. [1997], Sojka et al. [2011], Raghavendra et al. [2008]. The combination of offline learning and control works well over a narrow range of applications, as the offline models capture the general behavior of a class of application and require negligible online overhead. This focused approach is extremely effective for multimedia applications Vardhan et al. [2009], Flinn and Satyanarayanan [1999, 2004], Kim et al. [2013], Maggio et al. and web-servers Horvath et al. [2007], Lu et al. [2006], Sun et al. [2008] because the workloads can be characterized ahead of time so that the models produce sound control.

Indeed, the need for good models is the central tension in developing control for computing systems. It is always possible to build a controller for a specific application and system by extensively modeling that pair. More general controllers, which work with a range of applications, have addressed the need for models in various ways. Some provide libraries that encapsulate control functionality and require user-specified models Zhang et al. [2002], Sojka et al. [2011], Rajkumar et al. [1997], Imes et al. [2015a], Goel et al. [1998]. Others automatically synthesize both a model and a controller for either hardware Pothukuchi et al. [2016] or software Filieri et al. [2014, 2015]. JouleGuard combines learning for energy efficiency with control for managing application parameters Hoffmann [2015]. In JouleGuard, a learner adapts the controller's coefficients to model uncertainty, but JouleGuard's learner does not produce a new model for the controller. Because JouleGuard's learner runs on the same device as the controlled application, it must be computationally efficient and thus it cannot identify correlations across applications or even different resource configurations. CALOREE is unique in that a separate learner generates an application-specific model automatically. By offloading the learning task, CALOREE (1) combines data from many

applications and systems and (2) applies computationally expensive, but highly accurate learning techniques.

2.3 Application interference

Accurate performance estimates are essential for solving scheduling and resource allocation problems Chiang et al. [2002]. Accurate estimates are difficult to obtain due to the complexity and diversity of large-scale systems Kanev et al. [2015]. A particular challenge is modeling performance loss due to contention among applications Kambadur et al. [2012]. Better contention models could improve system utilization while ensuring quality-of-service in latency sensitive applications Yang et al. [2013].

Several approaches investigate statistical and machine learning techniques for estimating power, performance, and energy of a single application running on a single system. Many of these approaches improve the time of developing new hardware designs, but are not suitable for online resource management Yi et al. [2003], Lee and Brooks [2006], Lee et al. [2008]. Other approaches can predict the power and performance of various resource allocations to single applications Snowdon et al. [2009] or optimize energy efficiency under latency constraints Mishra et al. [2015]. A recent approach combines machine learning with control theory to guarantee energy consumption, but only for a single application Hoffmann [2015]. Perhaps most similar to ESP is the Mantis project which also uses higher-order regularized regression models (based on Lasso) to predict smartphone app performance Kwon et al. [2013]. Mantis, however, does not predict contention among multiple applications, which is ESP's focus.

Other approaches predict and mitigate contention in single-node systems. Many decide to co-schedule or not, but they do not produce quantitative slowdown estimates. For example, Dwyer et al. propose a classifier that predicts whether contention will be high or low, but this approach does not produce a numerical estimate Dwyer et al. [2012]. Similarly, ReSense detects highly contended resources and reacts to reduce that contention, but it never estimates contention Dey et al. [2013].

Another approach estimates throughput (total system performance), but does not produce estimates of individual application performance Xu et al. [2010], Chen et al. [2010]. Subramanian et al. propose an approach that does produce accurate estimates of performance (within 9.9%) based on only last-level cache access rate and memory bandwidth Subramanian et al. [2015]. These results are achieved on a simulator rather than a real system, however, and on our real system these two features are not sufficient to predict contention with any level of accuracy. Another single-node system, D-Factor, uses non-linear models to predict the slowdown of a new application given current resource usage Lim et al. [2012]. Unlike ESP, D-Factor is not capable of predicting how two applications will interfere with each other if neither is currently running.

Several approaches estimate and mitigate contention to schedule for multi-node systems. The activity vectors approach works on single or multi-node systems by maximizing the variance among resource usage in co-scheduled applications Merkel et al. [2010]. This heuristic makes intuitive sense – applications with very different resource needs are less likely to interfere with each other – but this approach does not produce quantitative estimates and therefore can make bad decisions when contention is unavoidable. Merlin, is somewhat similar, in that it tries to estimate contended resources and migrate virtual machines to areas of lower contention, but it also does not produce slowdown estimates Tembey et al. [2014]. DejaVu is a machine learning approach that classifies application workloads and then schedules according to known good schedules for the classification Vasić et al. [2012]. DejaVu creates an *interference index* which can be used to rank slowdowns and migrate VMs or reallocate resources, but it does not produce accurate estimates of the actual slowdowns incurred. Similarly, Quasar Delimitrou and Kozyrakis [2014] and Stay-Away Rameshan et al. [2014] use classification schemes to predict and mitigate interference, but neither produces performance estimates in the face of interference. Bubble-flux Yang et al. [2013], an improvement over the earlier Bubble-up Mars et al. [2011] does produce slowdown estimates and, like ESP, it is efficient enough to consider interference among more than two applications. The main difference between Bubble-flux and ESP is that Bubble-flux uses no offline prior information

and must dynamically probe the system. This lack of prior information means that Bubble-flux must suffer either poor utilization or inaccurate schedules during the probing phase. ESP uses a highly accurate offline model and combines that with online updates to its predictions to further improve accuracy while making use of the vast amount of data available to inform offline model building.

We model interference estimation as a *high-dimensional* regression problem with prohibitively many dimensions. Many statistical methods address high-dimensionality (*e.g.*, SURE Fan and Lv [2008]). In computer system performance, however, measurable features are highly correlated and existing methods do not provide high accuracy. Other statistical models emit more accurate predictors given correlated features (*e.g.*, Yuan and Lin [2006] and Bien et al. [2013]), but they do not scale to our problem size. Even though we make a strong assumption that the interaction terms are present only if their individual linear terms are significant, the heuristic has very high accuracy and produces good schedules in practice.

CHAPTER 3

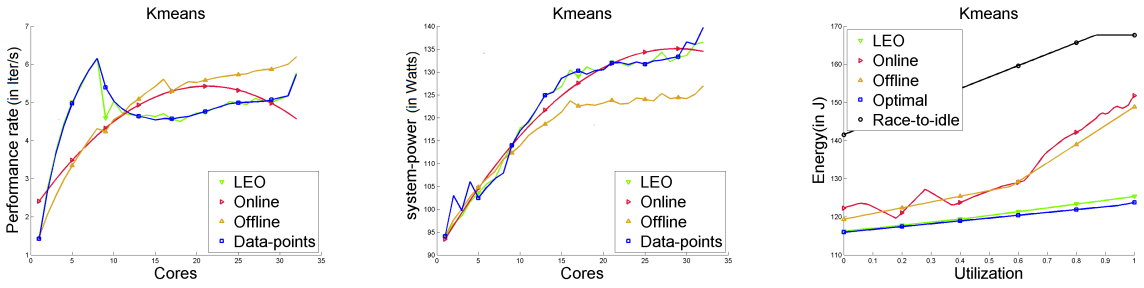
LEO: ESTIMATION FOR SINGLE APPLICATION

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 Narayanan et al. [2006]. 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

1. Our full system evaluation tests more parameters than simply core allocation. See Section 3.5 for details.



(a) (b) (c)

Figure 3.1: 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.Add performance description

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 3.5.2).

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 Hoffmann [2013], Meisner et al. [2011]. 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 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 `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 3.2 describes the notation we will use. Section 3.3 presents a general formalization of this energy minimization problem for any configurable system (not just cores). Section 3.4 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.

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 $\mathscr{L}_2$ 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. $\mathsf{tr}(\mathbf{A})$ 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 vari-

ables in $\mathbb{R}^d$. The notation $\mathbf{x} \sim \mathscr{D}$ represents that $\mathbf{x}$ is drawn from the distribution $\mathscr{D}$. Similarly, the notation $\mathbf{x}, \mathbf{y} \sim \mathscr{D}$ represents that $\mathbf{x}$ and $\mathbf{y}$ are jointly drawn from the distribution $\mathscr{D}$, and finally $\mathbf{x}|\mathbf{y} \sim \mathscr{D}$ 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 $\mathbf{x}$, var$[\mathbf{x}]$: variance of $\mathbf{x}$, Cov$[\mathbf{x}, \mathbf{y}]$: covariance of $\mathbf{x}$ and $\mathbf{y}$. $\hat{\mathbf{x}}$ denotes the estimated value for the random variable $\mathbf{x}$.

3.3 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 $\mathscr{C}$. Assuming that each configuration $c \in \mathscr{C}$ 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):

$$\begin{aligned} \min_{\mathbf{t} \geq \mathbf{0}} \quad & \sum_{c \in \mathscr{C}} p_c t_c, \\ \text{subject to} \quad & \sum_{c \in \mathscr{C}} r_c t_c = W, \\ & \sum_{c \in \mathscr{C}} t_c \leq T. \end{aligned} \tag{3.1}$$

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 Bradley et al. [1977].

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 4.1, we restricted configurations to be assignment of cores. In Section 3.5, 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

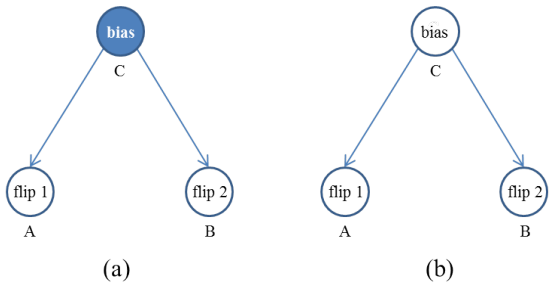


Figure 3.2: Conditional dependence in Bayesian Model.

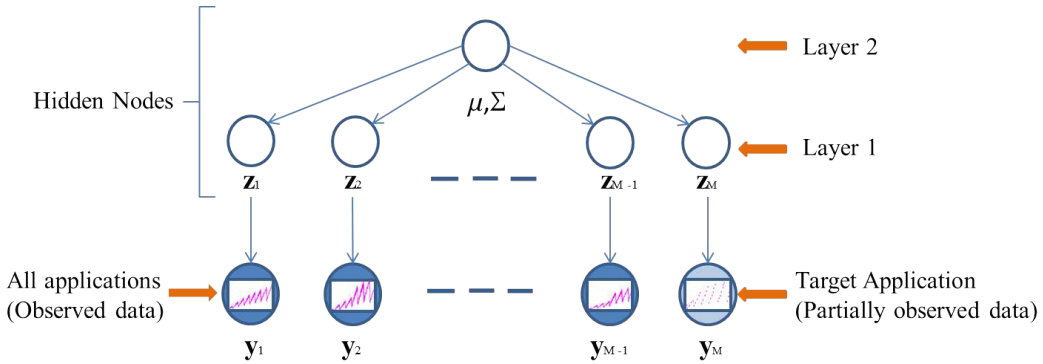
all the applications s because they might have local minima or maxima that are not captured by a small sample. In the next section, Section 3.4 we give details of CALOREE, our solution to this problem which uses both the data from the current application and previously seen applications for fast, accurate estimates.

3.4 Modeling Power and Performance

3.4.1 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 3.2a, variables A and B are dependent on C. If C is observed, A and B would be independent in Figure 3.2b.

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 $P(Heads) = 0.7$,



$\mathbf{y}_i$: Vector of power consumption by the i^{th} application for different configurations.

Figure 3.3: Hierarchical Bayesian Model.

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.*

3.4.2 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 3.3,

Suppose we have $n = |\mathscr{C}|$ 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 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,

$$\begin{aligned} \mathbf{y}_i|\mathbf{z}_i &\sim N(\mathbf{z}_i, \sigma^2\mathbb{I}), \\ \mathbf{z}_i|\mu,\Sigma &\sim N(\mu,\Sigma), \\ \mu,\Sigma &\sim N(\mu_0,\Sigma/\pi)IW(\Sigma|\nu,\Psi), \end{aligned} \tag{3.2}$$

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 3.3, 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 Efron and Morris [1975] for shrinkage effect and Morris [1983] 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 Gelman et al. [2013] 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 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 3.2, where the value of one coin influences our prediction about the other coin. Σ captures the correlation between different configurations as depicted in Figure 3.4.

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 Yu et al. [2005]). 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 3.3) , 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.

3.4.3 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 Yu et al. [2005].

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,

$$\begin{aligned} \mathrm{Cov}(\mathbf{z}_i) &= \left(\frac{\mathrm{diag}(L_i)}{\sigma^2} + \Sigma^{-1}\right)^{-1} \text{ and} \\ \mathbb{E}(\mathbf{z}_i) &= \hat{\mathbf{C}}_i \left(\frac{\mathrm{diag}(L_i)\mathbf{y}_i}{\sigma^2} + \Sigma^{-1}\mu\right). \end{aligned} \tag{3.3}$$

We use $\hat{\mathbf{C}}_i$ as shorthand for $\mathrm{Cov}(\mathbf{z}_i)$ 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,

$$\begin{aligned}
\mu &= \frac{1}{M+\pi} \sum_{i=1}^{M} \hat{\mathbf{z}}_i, \\
\Sigma &= \frac{1}{M+1} \left(\sum_{i=1}^{M} \hat{\mathbf{C}}_i + (\hat{\mathbf{z}}_i - \mu)(\hat{\mathbf{z}}_i - \mu)' \right) + \pi \mu \mu' + \mathbb{I}, \\
\sigma^2 &= \frac{1}{\|L\|_F^2} \sum_{i=1}^{M} \mathrm{tr}\left(\mathrm{diag}(L_i)(\hat{\mathbf{C}}_i' + (\hat{\mathbf{z}}_i - \mathbf{y}_i)(\hat{\mathbf{z}}_i - \mathbf{y}_i)') \right),
\end{aligned} \tag{3.4}$$

LEO iterates over the E step (equation (3.3)) and M step (equation (3.4)) 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 (3.3). 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 Hoffmann et al. [2013], Zhang et al. [2002], Li and Nahrstedt [1999], Imes et al. [2015b]. 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.

3.4.4 Example

We illustrate how LEO can be applied to our running example of `Kmeans` from Section 4.1. 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 $\{\mathbf{y}_i\}_{i=1}^{M-1}$; $\mathbf{y}_M$ denotes the power data for `Kmeans`. Referring to Figure 3.3, `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 3.3, 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.

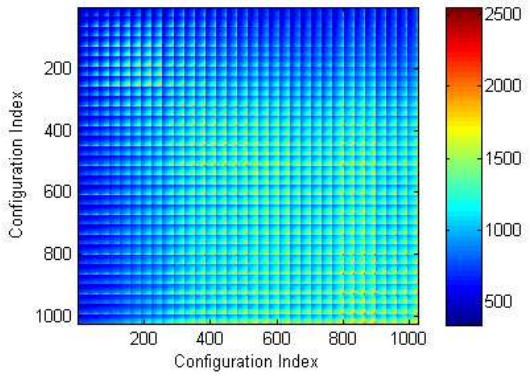


Figure 3.4: A illustrative example of covariance Σ between different configurations, in equation (3.2)

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 (3.3) for each ith and later uses these values of $\hat{\mathbf{z}}_i$ and $\hat{C}_i$ to evaluate equation (3.4), which is fed back to expectation (3.3) and so on. This alternating step between equation (3.3) and (3.4) runs until the algorithm converges. The algorithm uses $\hat{\mathbf{z}}_M$ as the estimate of `Kmeans` power (*i.e.*, $p_c = \hat{\mathbf{z}}_M[c], \forall c \in \mathscr{C}$ 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.

3.4.5 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 3.4). We want to again point out how our modeling of the problem is markedly different from some of the previous approaches (such as Deng et al. [2012]), which assume that power and performance are convex functions of the configuration knobs and employ algorithms similar to gradient descent to find the

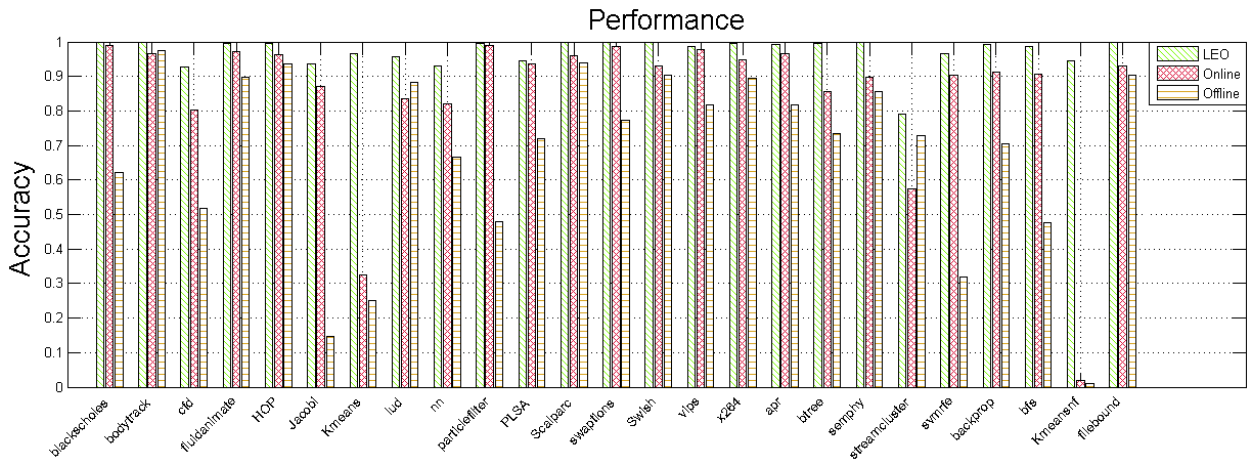


Figure 3.5: Comparison of performance (measured as speedup) estimation by different techniques for various benchmarks. On an average (over all benchmarks), *LEO*'s accuracy is 0.97 compared to 0.87 and 0.68 for *Online* and *Offline* respectively. The results are normalized with respect to the *Exhaustive search* method.

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 Wu [1983]. 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 3.5.2) 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 3.5.7.

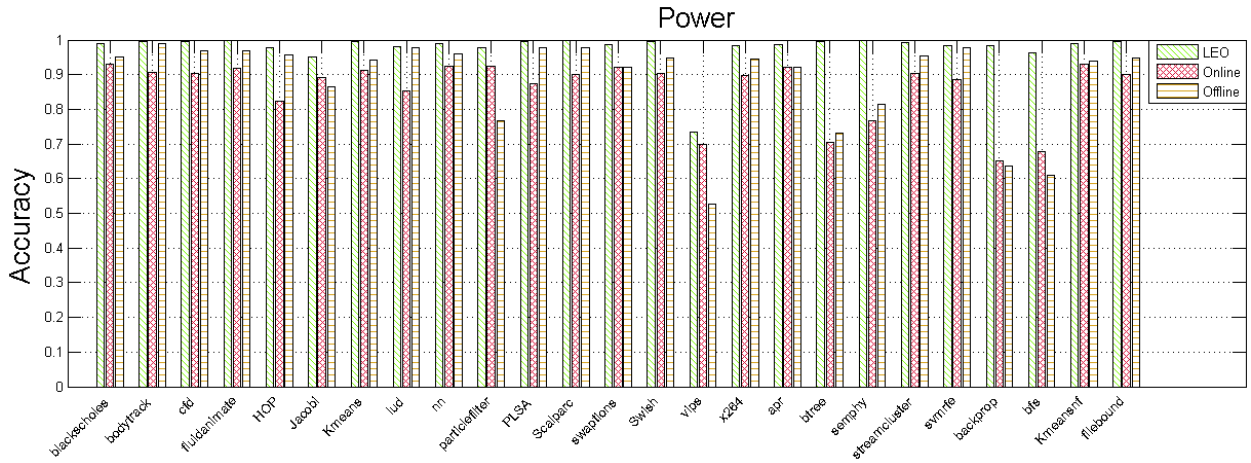


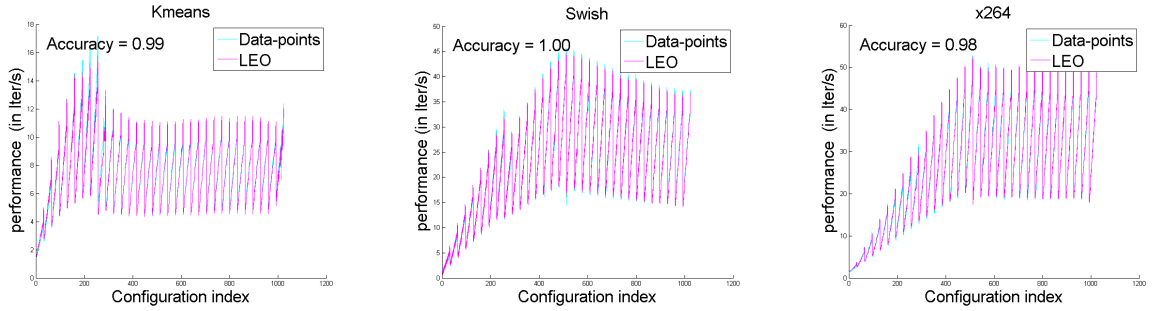
Figure 3.6: Comparison of power (measured in Watts) estimation by different techniques for various benchmarks. On an average (over all benchmarks), *LEO*'s accuracy is 0.98 compared to 0.85 and 0.89 for *Online* approach and *Offline* approach respectively. Again, the results are normalized with respect to the *Exhaustive search* method.

3.5 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.

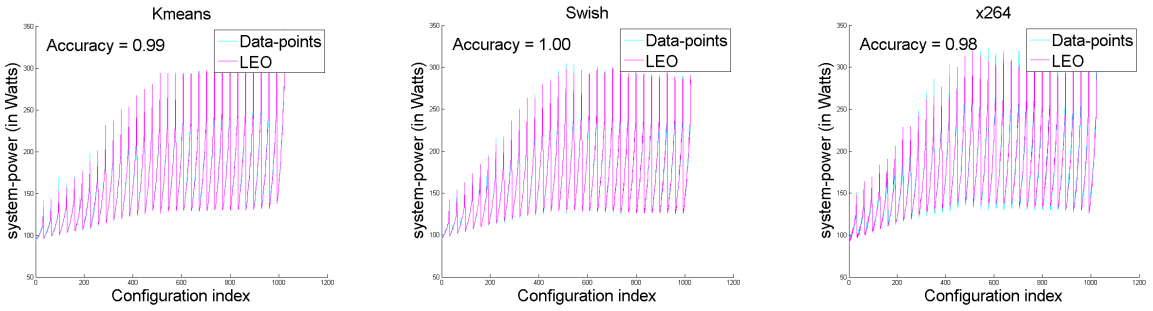
3.5.1 Experimental Setup

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



(a) (b) (c)

Figure 3.7: Examples of performance estimation using LEO. Performance is measured as application iterations (or heartbeats) per second. (See Section 5.3.1).



(a) (b) (c)

Figure 3.8: Examples of power estimation using LEO. Power is measured as total system power.

system supports 1024 user-accessible configurations, each with its own power/performance tradeoffs[2]. 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`) Bienia et al. [2008], Minebench (`ScalParC`, `apr`, `semphy`, `svmrfe`, `Kmeans`, `HOP`, `PLSA`, `non fuzzy kmeans (Kmeansnf)`) Narayanan et al. [2006], and Rodinia (`cfd`, `nn`, `lud`, `particlefilter`, `vips`, `btree`, `streamcluster`, `backprop`, `bfs`) Che et al. [2009].

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

We also use a partial differential equation solver (`jacobi`), a file intensive benchmark (`filebound` and the `swish++` search web-server Hoffmann et al. [2011a]. 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 Imes et al. [2015b]. 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,

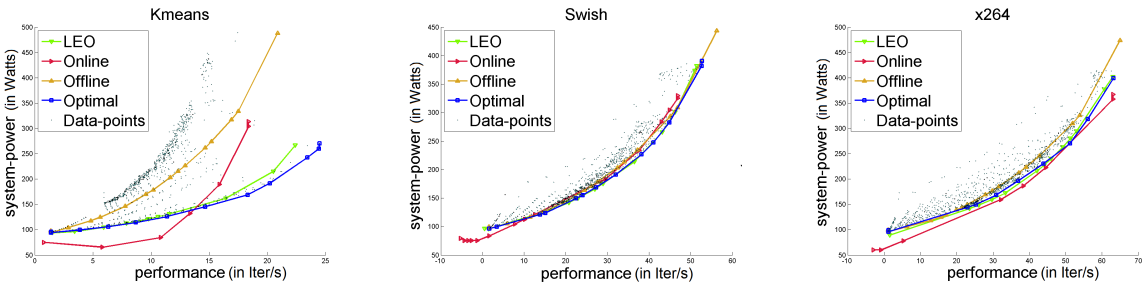
$$\text{accuracy}(\hat{\mathbf{y}},\mathbf{y}) = \max\left(1 - \frac{\|\hat{\mathbf{y}}-\mathbf{y}\|_2^2}{\|\mathbf{y}-\bar{\mathbf{y}}\|_2^2}, 0\right). \tag{3.5}$$

3.5.2 Points of Comparison

We evaluate LEO in comparison to four baselines:

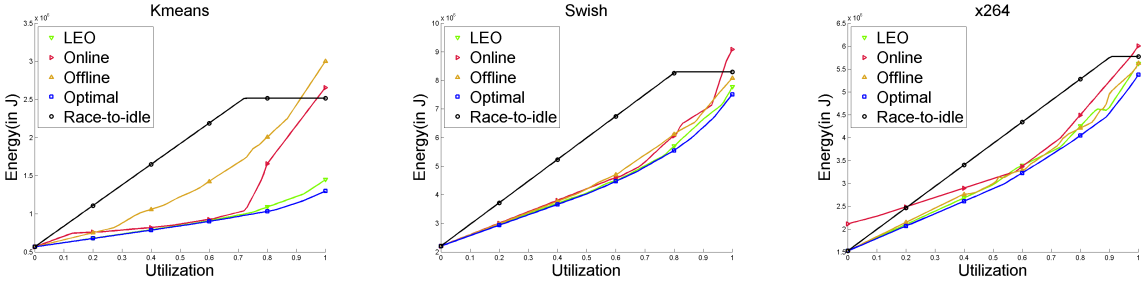
1. *Race-to-idle* – This approach allocates all resources to the application and once it is finished the system goes to idle. This strategy incurs almost no runtime overhead, but may be suboptimal in terms of energy, since maximum resource allocation is not always the best solution to the energy minimization equation (3.1) Carroll and Heiser [2013], Hoffmann [2013], Le Sueur and Heiser [2011].

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 observations and not the prior



(a) (b) (c)

Figure 3.9: 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 (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.



(a) (b) (c)

Figure 3.10: Energy consumption vs utilization for different estimation algorithms.

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.

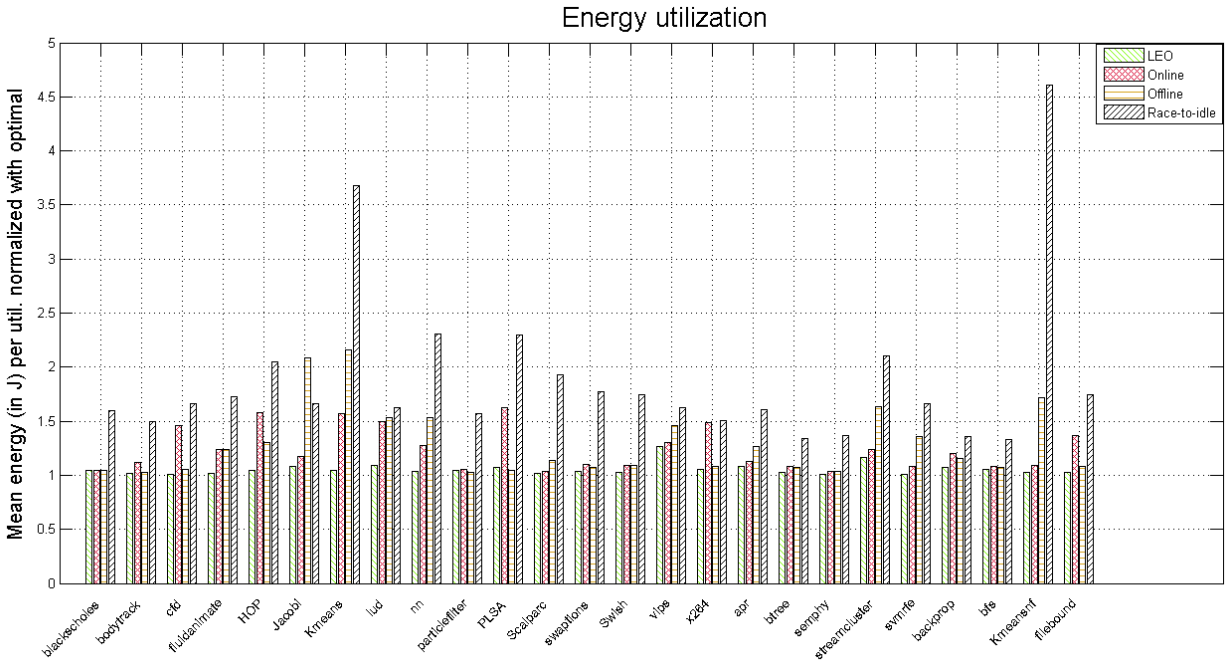


Figure 3.11: Comparison of average energy (normalized to optimal) by different estimation techniques for various benchmarks. On an average (taken over all the benchmarks); *LEO* consumes 6% over optimal, as compared to the *Online*, *Offline*, and *Race-to-idle* approaches, which respectively consume 24%, 29% and 90% more energy than optimal.

3.5.3 Power and Performance using LEO

We compare LEO's estimates to the *online*, *offline*, and exhaustive search methods described in Section 3.5.2. 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 3.5 and Figure 3.6, respectively. Each chart shows the benchmarks on the x-axis and estimation accuracy (computed using Equation 5.11) 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 4.1), 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 3.7)

and power (Figure 3.8). 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 3.7 and Figure 3.8) 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.

3.5.4 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 3.4.3, LEO uses its estimates to form the Pareto-optimal frontier of performance and power tradeoffs. Figure 3.9 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.

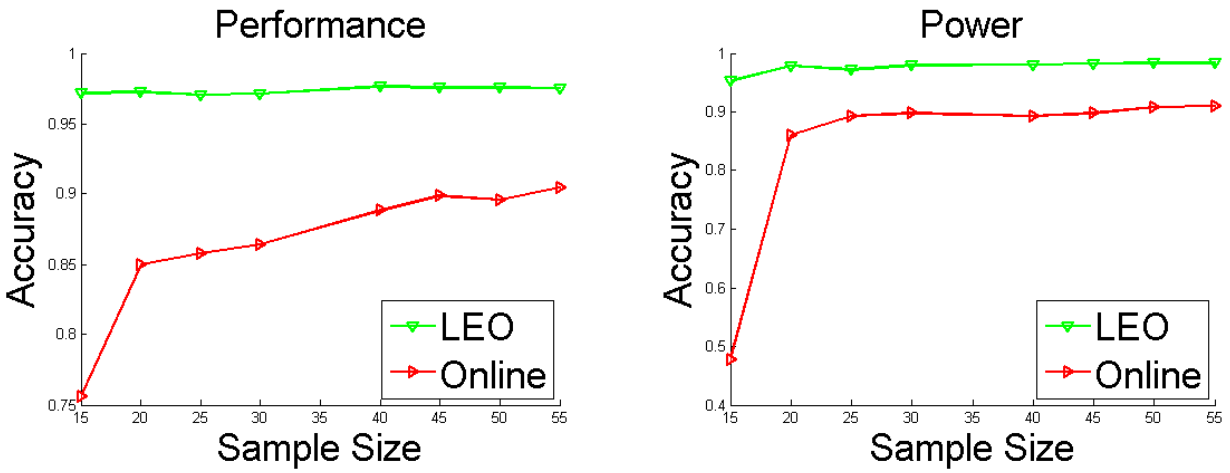
To evaluate energy savings, we deploy each application with varying performance demands. Technically, we fix the deadline and vary the workload W from Equation 3.1 so that $W \in [minPerformance, maxPe$ for each application. We test 100 different values for W – each representing a different utilization 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 3.10 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 3.11, 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.

3.5.5 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 3.12, 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.



(a) (b)

Figure 3.12: 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.

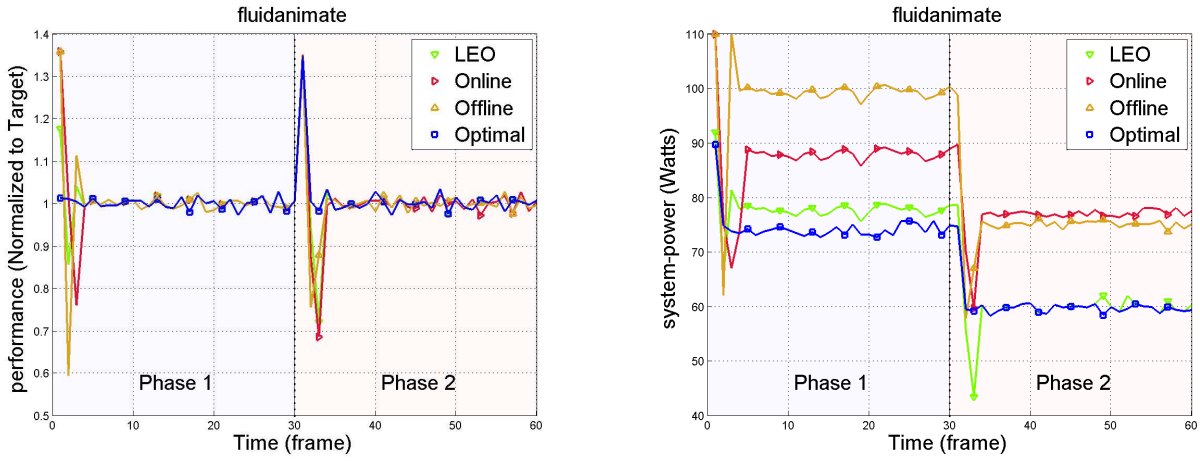
3.5.6 Reacting to Dynamic Changes

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 3.1: 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 3.13. Each chart shows time (measured



(a) (b)

Figure 3.13: Power and performance for `fluidanimate` transitioning through phases with different computational demands.

in frames) on the x-axis. Figure 3.13a shows performance normalized to real-time on the x-axis, while Figure 3.13b 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 3.1. These results indicate that LEO produces accurate results even in dynamically changing environments.

3.5.7 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 3.5.4).

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.

3.6 Summary

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 Hoffmann et al. [2012], Gentzsch et al. [2005], Dini et al. [2004], Salehie and Tahvildari [2009], Kephart [2005], Laddaga [1999].

CHAPTER 4

CALOREE: CONTROLLING APPLICATIONS UNDER SYSTEM DYNAMICS

4.1 Motivation

Many learning approaches estimate an application's most energy efficient resource allocation. These include *offline* techniques that train models and then apply those models to new applications Yi et al. [2003], Lee and Brooks [2006], Lee et al. [2008], Zhu and Reddi [2013], Zhang and Hoffmann [2016], Delimitrou and Kozyrakis [2014]. *Online* techniques construct models dynamically as an application runs Li and Martinez [2006], Petrica et al. [2013], Sridharan et al. [2013], Ponomarev et al. [2001], Lee and Brooks [2008]. *Hybrid* techniques combine offline modeling with online model updates Cochran et al. [2011], Winter et al. [2010], Dubach et al. [2010], Snowdon et al. [2009], Roy et al. [2011], Wu and Lee [2012], Mishra et al. [2015].

Control theory provides techniques for maintaining desired behavior in dynamic systems Hellerstein et al. [2004]. *Adaptive controllers* or *self-tuning regulators* adjust their internal models in response to dynamic changes Levine [2005]. They are especially useful in webservers with fluctuating request rates Horvath et al. [2007], Lu et al. [2006], Sun et al. [2008] and multimedia applications with dynamically varying inputs Maggio et al., Li and Nahrstedt [1999], Vardhan et al. [2009]. Prior work has generalized adaptive control design by exposing key model parameters to users who customize control to their needs Zhang et al. [2002], Imes et al. [2015a]. User customization provides greater flexibility, but the controller will not converge to the desired performance if the custom models do not accurately capture the relationship between resources and performance. This practice means users must not only be experts in their application domain, but must also have sufficient control knowledge to specify robust models.

This section illustrates how learning handles complexity, how control handles dynamics, and then describes a key difficulty that must be overcome to combine learning and control.

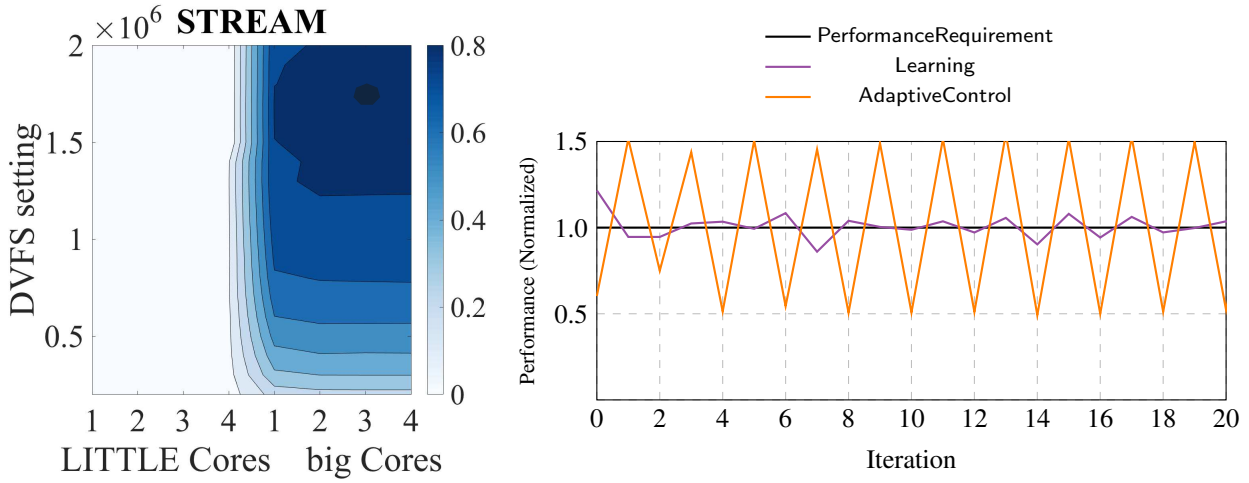


Figure 4.1: (a) `STREAM` performance as a function of configuration. (b) Managing `STREAM`'s performance: *Learning* handles the complex configuration space, but *control* oscillates.

4.1.1 Learning *Complexity*

We demonstrate how well learning handles complex resource interaction for `STREAM` on an ARM big.LITTLE processor with four big, high-performance cores and four LITTLE, energy efficient cores. The big cores support 19 clock speeds, while the LITTLE cores support 14.

Figure 4.1a shows `STREAM`'s performance for different resource configurations. This memory-bound application has complicated behavior: the LITTLE cores' memory hierarchy cannot deliver the required performance. The big cores' more powerful memory system delivers much greater performance, but the peak occurs with 3 big cores. Furthermore, at low clockspeeds, these 3 big cores cannot saturate the memory bandwidth, while at high clockspeeds the performance drops as the processor overheats, triggering thermal management. For `STREAM`, the peak speed occurs with 3 big cores at 1.2 GHz, and it is not efficient to spend any time on the LITTLE cores. `STREAM`, however, does not have distinct phases, so once a resource allocator finds the most energy efficient configuration, it simply needs to maintain it.

Figure 4.1b shows 20 iterations of both learning Mishra et al. [2015] and adaptive control Imes et al. [2015a] allocating resources to `STREAM`. The x-axis shows iteration and the y-axis shows

performance normalized to the requirement. The *learning* approach estimates `STREAM`'s performance and power for all configurations and uses the lowest energy configuration that delivers the required performance. The *adaptive controller* begins with a generic model of power/performance tradeoffs. As the controller runs, it measures performance and adjusts both the allocated resources and its own parameters using online measurements. The adaptive controller dynamically adjusts to non-linearities with a series of linear approximations; however, it is sensitive to model inaccuracies, which cause the oscillations that lead to performance violations. This behavior occurs because the controller's adaptive mechanisms cannot handle the STREAM's complexity, a known limitation of adaptive control systems Zhang et al. [2002], Imes et al. [2015a], Filieri et al. [2014]. Hence, the *learner*'s ability to model complex behavior is crucial.

4.1.2 Controlling *Dynamics*

We now consider a dynamic environment. We begin with `bodytrack` running alone on the system. Halfway through its execution, we launch a second application—`STREAM`—on a single big core, dynamically changing available resources. Figure 4.2a shows `bodytrack`'s behavior. It achieves the best performance on 4 big cores at the highest clockspeed; the 4 LITTLE are more energy-efficient but slower. For `bodytrack`, the challenge is determining how to split time between the LITTLE and big cores to conserve energy while still meeting the performance requirements.

Figure 4.2b shows the results of this experiment. The vertical dashed line—at frame 99—represents when the second application begins. The figure clearly shows adaptive control's benefits in this dynamic scenario. When the second application starts, the controller detects `bodytrack`'s performance dip–rather than detecting the new application specifically—and it changes resource allocation (increasing clockspeed and moving bodytrack from 4 to 3 big cores). The learning system however, does not have any inherent mechanism to measure the change or adapt to the altered performance. While we could theoretically add feedback to the learner and re-estimate

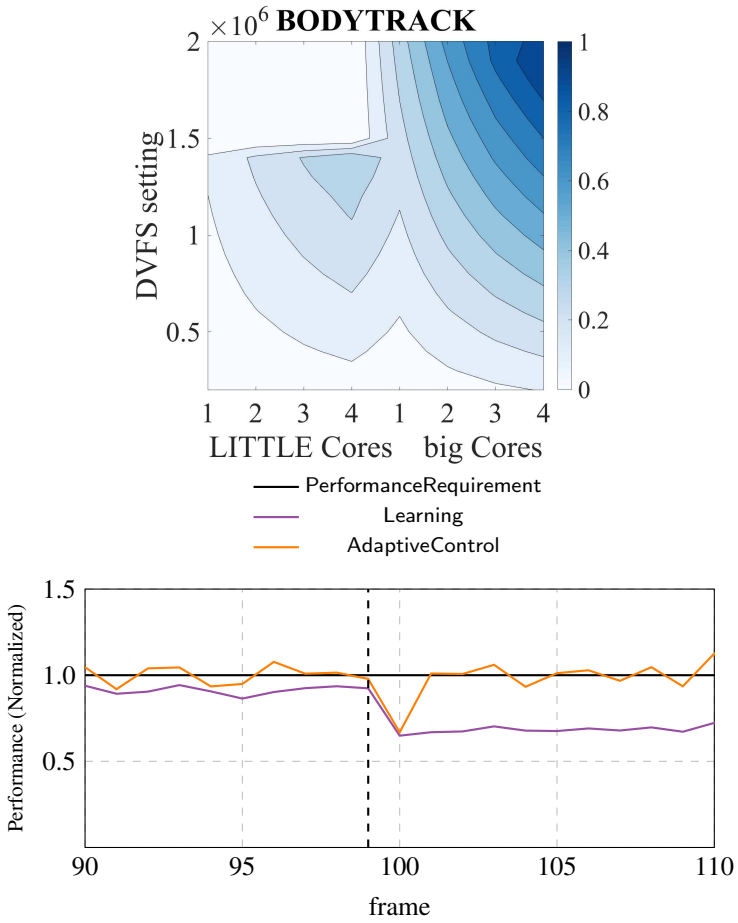


Figure 4.2: (a) `bodytrack` performance as a function of configuration. (b) Managing `bodytrack`'s performance with another application: *control* detects the change (at the vertical dashed line) and adjusts, but *learning* cannot.

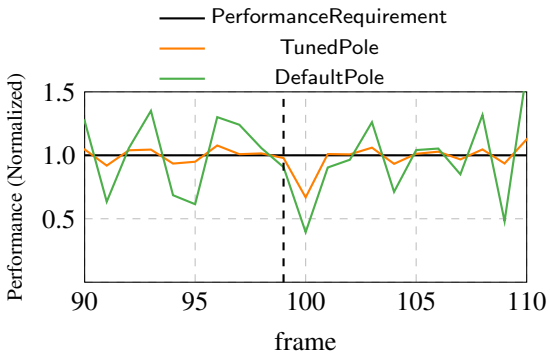


Figure 4.3: Comparison of carefully tuned and default poles.

the configuration space whenever the environment changes, doing so is impractical due to high overhead for learners capable of handling this complexity Delimitrou and Kozyrakis [2013, 2014], Mishra et al. [2015].

4.1.3 Challenges of Parameter-free Control

The adaptive controller requires user-specified parameters and Figure **??** shows the consequences of getting those parameters wrong. The controller's *pole* is a particularly important parameter. Control engineers tune the pole with a model to trade response time for noise sensitivity. This model may be an abstraction but is considered *ground truth* Hellerstein et al. [2004], meaning all possible resource configurations the controller might select have been directly measured for some input. CALOREE, however, tunes the pole based on an estimated model, which may have noise and/or errors.

To demonstrate the pole's importance when using a learned model, we again control `bodytrack`, this time using the adaptive controller from the previous subsection with a model produced by the learner from the first subsection. We compare the results with a carefully tuned pole to those using the default pole provided by the controller developers Imes et al. [2015a].

Figure 4.3 shows the results. The carefully tuned pole converges because the pole accounts for the learned model's error. The default pole, however, oscillates around the performance target, resulting in a number of missed deadlines. Additionally, the frames that exceed the desired

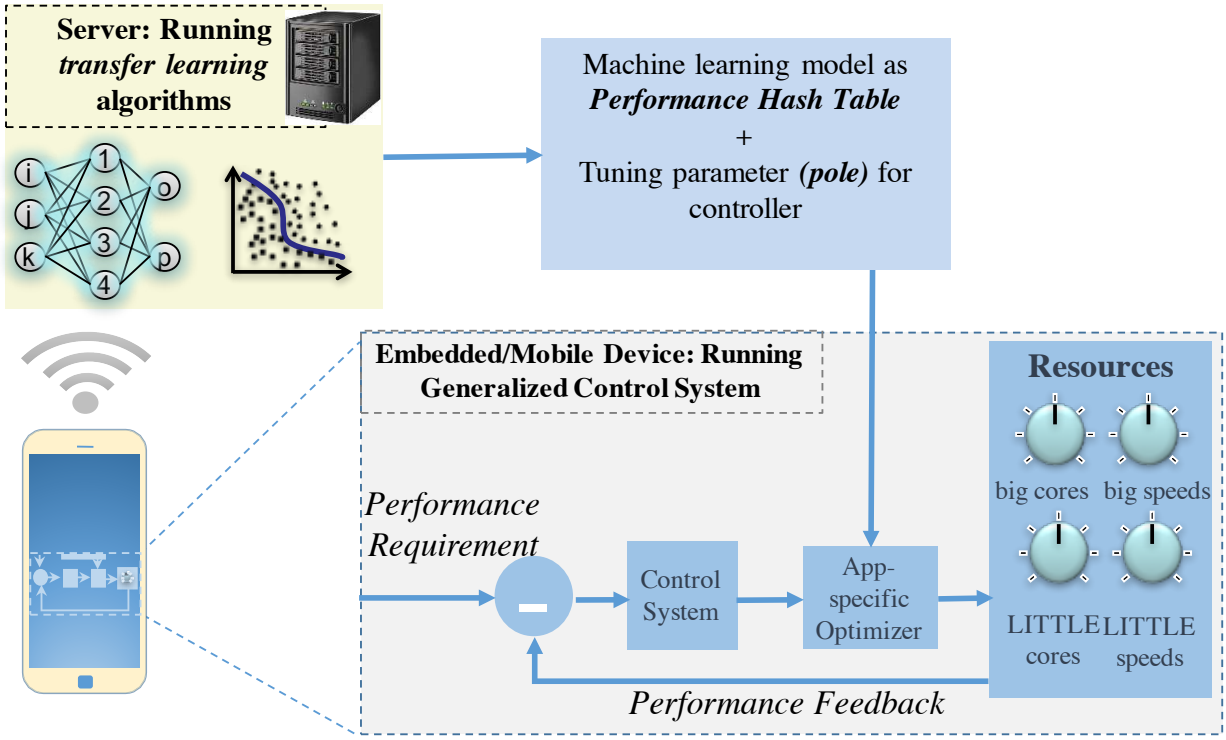


Figure 4.4: CALOREE overview.

performance waste energy because they spend more time on the big, inefficient cores. The pole parameterizes the system's *inertia*—dictating how fast it should react to environmental changes. If the estimated model is noisy, the controller should trust it less and move slowly. Rather than require users with both computing and control knowledge to tune the pole, *CALOREE incorporates the learner's confidence interval and estimated variance to compute a pole that provides probabilistic convergence guarantees.*

4.2 CALOREE: Learning Control

We describe CALOREE's parameter-free resource management system, which assumes no prior knowledge of the application and requires no user-specified parameters. Figure 4.4 shows CALOREE's approach. A new application enters the system with a performance requirement. A generalized adaptive control system allocates resources according to a generic model and records per-

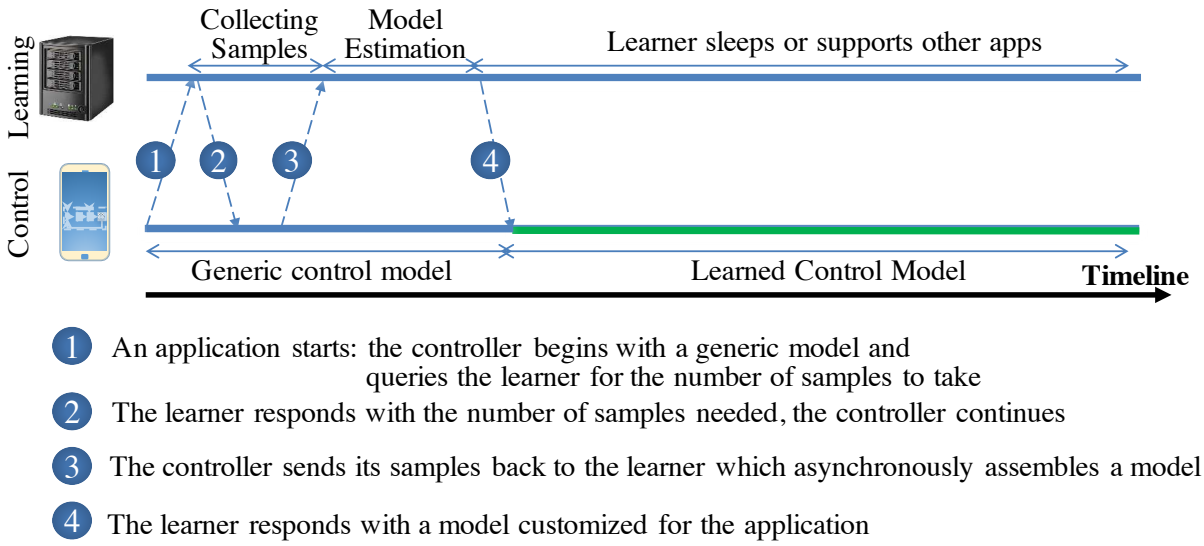


Figure 4.5: Temporal relationship of learning and control.

formance and power. The recorded values are sent to a learner, which estimates the application's performance and power in all other configurations and extracts those that represent Pareto-optimal tradeoffs. These configurations are packaged in a special data structure—the performance hash table (PHT). The learner sends the PHT and the estimated variance to the controller. Using these values, the controller selects an energy minimal resource configuration in constant time with formal guarantees of convergence to the desired performance.

Figure 4.5 illustrates the asynchronous interaction between CALOREE's learner and controller over time. The controller starts when a new application launches. It has no prior knowledge of this application, so it begins using a generic model. At start, the controller sends the learner the application name and device type (message 1, Figure 4.5). The learner determines how many samples are needed for an accurate estimate and sends this number back to the controller (message 2). The controller takes these samples while using the generic model and sends the learner the performance and power of each measured configuration (message 3). Computationally expensive learners may require time to build a model. Thus, the controller does not wait for the learner, but continues with the generic model. Once the learner has a model and variance estimate, it sends that data to the controller (message 4), which uses the learned model from then on.

Figure 4.5 shows several key points about the relationship between learning and control. First, the controller never waits for the learner—it uses a generic model to provide less efficient control until the learner can produce a customized model. Second, the controller does not continuously communicate with the learner, this interaction happens once at application launch. Third, if the learner crashed, the controller would just default to the behavior of a generic adaptive control system. If the learner crashed after producing a model, control might not even need to know. Finally, because the learner and controller have a clearly defined interface, they can be run in separate processes or physically separate devices.

This section first describes a traditional adaptive control system. We then generalize this approach and separate out parameters to be learned. Next, we discuss the general class of learning systems that work with CALOREE. Then, we present an encoding for the learned model that can be accessed by the controller in constant time. Finally, we formally analyze CALOREE's guarantees.

4.2.1 Traditional Control for Computing

Several researchers have proposed controlling computing systems. A controller that can manage multiple resources to meet multiple goals is a multiple-input, multiple-output (MIMO) controller. The inputs are measurements, *e.g.*, performance. The outputs are the resource settings to be used at a particular time, *e.g.*, an allocation of big and LITTLE cores and a clockspeed for each.

These difference equations describe a generic MIMO controller for allocating n resources to meet m goals at time t:[1]

$$\begin{aligned} \mathbf{x}(t+1) &= \mathbf{A}\cdot\mathbf{x}(t) &+\mathbf{B}\cdot\mathbf{c}(t) \\ \mathbf{y}(t) &= \mathbf{C}\cdot\mathbf{x}(t) & , \end{aligned} \tag{4.1}$$

where $\mathbf{x} \in \mathbb{R}^q$ is the controller's *state*, an abstract representation of the relationship between resources and goals and q is the controller's *degree*, or complexity of its internal state. $\mathbf{c}(t) \in \mathbb{R}^n$ is

1. We assume discrete time, and thus, use difference equations rather than differential equations that would be used for continuous systems.

a vector representing the current *configuration* of resources; *i.e.*, the ith vector element represents the amount of resource i to be allocated at time t. $\mathbf{y}(t) \in \mathbb{R}^m$ represents the current value of the goal dimensions at time t. The matrices $\mathbf{A} \in \mathbb{R}^{q \times q}$ and $\mathbf{B} \in \mathbb{R}^{q \times n}$ relate the resource configuration to the controller state. The matrix $\mathbf{C} \in \mathbb{R}^{m \times q}$ relates the controller state to the expected behavior. This model does not assume the states or the resources are independent, but it does assume that their relationship is linear.

For example, to allocate resources to meet performance goals in our ARM big.LITTLE system—from Section 4.1—there are four resources: the number of big cores, the number of LITTLE cores, and the speeds for each of the big and LITTLE cores. There is also a single goal: performance. Thus, in this example, $n = 4$ and $m = 1$. The vector $\mathbf{c}(t)$ has four elements representing the resource allocation at time t. q is the number of variables in the controller's state which can vary between, we can choose q to be between 1 to n. The matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ capture the linear relationship between the control state $\mathbf{x}$, the resource usage $\mathbf{c}$, and the measured behavior. In this example, we know there is a non-linear relationship between the resources. We overcome this difficulty by tuning the identified matrices at each time step—approximating the non-linear system through a series of changing linear formulations. This approximation is a form of *adaptive* or *self-tuning* control Levine [2005]. Such adaptive controllers provide formal guarantees that they will converge to the desired performance even in the face of non-linearities, but they still assume convexity.

This control formulation has two major drawbacks. First, because it requires matrix computation, its overhead scales linearly in the number of resources and in the number of goals Hellerstein et al. [2004], Sharifi et al. [2011]. Second, the adaptive mechanisms are not parameter-free—they require users to specify starting values of the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ and the method for updating these matrices to account for any non-convexity in the relationship between resources and performance Imes et al. [2015a], Sharifi et al. [2011], Zhang et al. [2002], Levine [2005]. Therefore, typically 100s to 1000s of samples are taken at design time to ensure that the starting matrices are sufficient to ensure convergence Filieri et al. [2015], Ljung [1999], Pothukuchi et al. [2016].

4.2.2 CALOREE Control System

To overcome the above issues, CALOREE abstracts the controller of Eqn. 4.1 and factors out those parameter to be learned. Specifically, CALOREE takes three steps to transform a standard control system into one that works without prior knowledge of the application to be controlled:

1. controlling *speedup* (which is an abstraction on performance) rather than resources,
2. translating speedup into an energy minimal *resource schedule* in a separate step, and
3. exploiting the *problem structure* to solve this scheduling problem in constant time.

These steps assume a separate learner has produced a model of resource performance and power. The result is that CALOREE's controller runs in constant time without requiring any user-specified parameters.

Controlling Speedup

Instead of the matrix equations of Eqn. 4.1, CALOREE uses a scalar difference model relating speedup to performance:

$$perf(t) = b(t) \cdot speedup(t-1) + \delta \tag{4.2}$$

where $b(t)$ is the application's *base speed*: defined as the speed when all resources are available. While $b(t)$ is application specific, it is easy to measure online, by simply allocating all resources. Such a configuration should not violate any performance constraints (although it is unlikely to be energy efficient) so it is safe to take this measurement. CALOREE assumes base speed is time-variant as applications will transition through phases.

With this model, CALOREE's control law is simply:

$$error(t) = goal - perf(t) \tag{4.3}$$

$$speedup(t) = speedup(t-1) - \frac{1-\rho(t)}{b(t)}.error(t) \tag{4.4}$$

which states that the speedup at time t is a function of the previous speedup, the error at time t, the base speed $b(t)$, and the controller's *pole*, $\rho(t)$. Standard control techniques statically determine the pole and the base speed, but CALOREE *dynamically sets the pole and base speed to account for error in the learned models—an essential modification for providing formal guarantees of the combined control and learning systems.* For stable control, CALOREE ensures $0 \leq \rho(t) < 1$. Small values of $\rho(t)$ eliminate error quickly, but make the controller more sensitive to model inaccuracies. Larger $\rho(t)$ makes the system more robust at the cost of reduced convergence time. CALOREE sets $b(t)$ using the standard technique of Kalman filter estimation Welch and Bishop. Section 4.2.5 describes how CALOREE automatically sets the pole; however, we first address converting an abstract speedup into a resource allocation.

Converting Speedup to Resource Schedules

CALOREE must map Eqn. 4.4's speedup into a resource allocation. On our example ARM big.LITTLE architecture that means mapping speedup into an allocation of big and LITTLE cores as well as a speed for both (big and LITTLE cores are in separate clock domains).

The primary challenge is that speedups in real systems are discrete non-linear functions of resource usage, while Eqn. 4.4 is a continuous linear function. We bridge this divide by assigning time to resource allocations such that the average speedup over a control interval is that produced by Eqn. 4.4.

The assignment of time to resource configurations is a *schedule*; *e.g.*, spending 10 ms on the LITTLE cores at 0.6 GHz and then 15 ms on the big cores at 1 GHz. Typically many schedules can deliver a particular speedup and CALOREE must find one with minimal energy. Given a time interval T, the $speedup(t)$ from Eqn. 4.4, and C different resource configurations, CALOREE

solves:

$$\min_{\tau \in \mathbb{R}^C} \quad \sum_{c=0}^{C-1} \tau_c \cdot p_c \tag{4.5}$$

$$s.t. \quad \sum_{c=0}^{C-1} \tau_c \cdot s_c = speedup(t)T \tag{4.6}$$

$$\sum_{c=0}^{C-1} \tau_c = T \tag{4.7}$$

$$0 \leq \tau_c \leq T, \qquad \forall c \in \{0, \ldots, C-1\} \tag{4.8}$$

where p_c and s_c are configuration c's estimated *powerup*—analogous to speedup—and speedup; τ_c is the time to spend in configuration c. Eqn. 4.5 is the objective: minimizing energy (power times time). Eqn. 4.6 states that the average speedup must be maintained, while Eqn. 4.7 requires the time to be fully utilized. Eqn. 4.8 simply avoids negative time.

4.2.3 Exploiting Structure for Fast Solutions

By exploiting the problem structure and encoding the learned model in the performance hash table (PHT), CALOREE solves Eqns. 4.5–4.8 in constant ($O(1)$) time.

Kim et al. analyze the problem of minimizing energy while meeting a performance constraint and observe that there must be an optimal solution with the following properties Kim et al. [2015]:

- At most two of τ_c are non-zero, meaning that at most two configurations will be used in any time interval.
- If you chart the configurations in the power and performance tradeoff space (*e.g.*, the top half of Figure 4.6) the two configurations with non-zero τ_c lie on the lower convex hull of the power performance tradeoff space.
- The two configurations with non-zero τ_c are adjacent on the convex hull: one above the constraint and one below.

The PHT (shown in Figure 4.6) provides constant time access to points on the lower convex

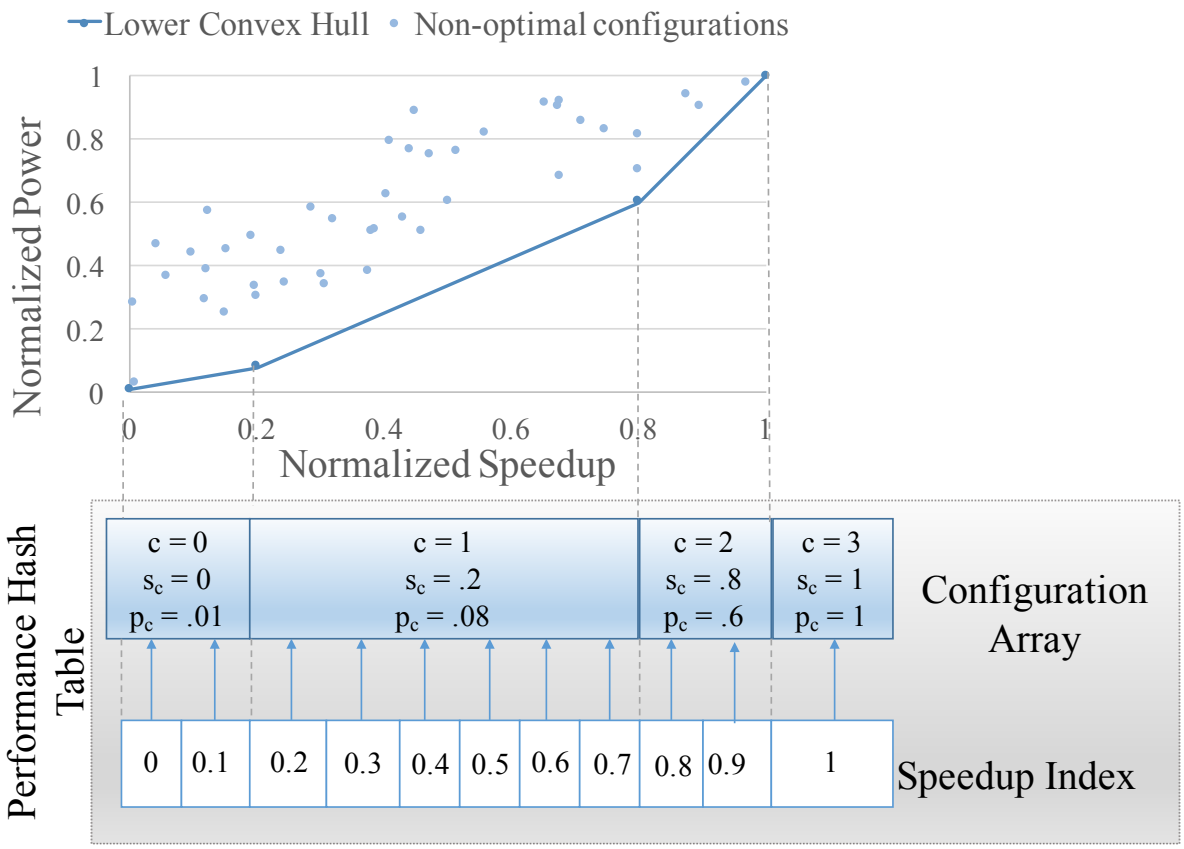


Figure 4.6: Data structure to efficiently convert required speedup into a resource configuration.

hull. It consists of two arrays: the first being pointers into the second, which stores resource configurations on the lower convex hull sorted by speedup. Recall speedups are computed relative to the base speed, which uses all resources. The largest estimated speedup is therefore 1, so CALOREE needs only consider speedups between 0 and 1. The first array of pointers has a *resolution* indicating how many decimal points of precision it captures and it is indexed by speedup. The example in Figure 4.6 has a resolution of 0.1. Each pointer in the first array points to the configuration in the second array that has the largest speedup less than or equal to the index.

CALOREE computes $speedup(t)$ and uses the PHT to convert speedup into two configurations: *hi* and *lo*. To find the *hi* configuration, CALOREE clamps the desired speedup to the largest index lower than $speedup(t)$ and then walks forward until it finds the first configuration with a speedup higher than $speedup(t)$. To find the *lo* configuration, it clamps the desired speedup to the smallest index higher than $speedup(t)$ and then walks backwards until it finds the configuration with the largest speedup less than $speedup(t)$.

For example, consider the PHT in Figure 4.6 and an optimizer meeting $speedup(t) = .65$. To find *hi*, the optimizer indexes at .6 and walks up to find $c = 2$ with $s_c = .8$, setting $hi = 2$. To find *lo*, the optimizer indexes the table at .7 and walks backward to find $c = 1$ with $s_c = .2$, setting $lo = 1$.

CALOREE sets τ_{hi} and τ_{lo} by solving:

$$T = \tau_{hi} + \tau_{lo} \tag{4.9}$$

$$speedup(t) = \frac{s_{hi} \cdot \tau_{hi} + s_{lo} \cdot \tau_{lo}}{T} \tag{4.10}$$

where $speedup(t)$ is given by the controller and s_c are speedups estimated by the learner. By solving Eqns. 4.9 and 4.10, CALOREE has turned the controller's speedup into a schedule of resource allocations using learned models stored in the PHT.

4.2.4 CALOREE Learning Algorithms

The previous subsection describes a general control system, which can be customized with a number of different learning methods. The requirements on the learner are that it must produce 1) estimates of the speedup and powerup values for each resource configuration and 2) an estimate of its own inaccuracy. This section describes the general class of learning mechanisms that meet these requirements.

When selecting a learning framework we must find a tradeoff between the specific and the general; *i.e.*, between frameworks that build application-specific models and frameworks that combine observations across applications. For example, the key to energy efficiency on heterogeneous mobile systems is knowing when to make use of the smaller, low-power cores Kim et al. [2015], Carroll and Heiser [2013], Le Sueur and Heiser [2011]. An application-specific model will capture that precisely, but may require many observations before producing the correct model. A more general model will capture the trend, *e.g.*, when most applications should transition, but this general model might miss the key inflection point for some applications. We refer to application-specific models as *online* because they build models for the current application and do not incorporate knowledge of other applications. We refer to general models as *offline* as they use prior observations of other applications to predict the behavior of a new application. A third class of *transfer learning* models combines information from the previously seen applications and current application to make the predictions on the application in hand Pan and Yang [2010] . These models allow us to obtain higher prediction accuracies since we are able to augment our data with extra information from other applications.

Offline models

These models utilize data from previously seen applications. If the model is for applications with very similar behavior the prediction accuracy might be high but the model will fail if the application's behaviors diverge **??**. For example, these techniques will predict the performance of

applications based on prior applications, meaning they will over-allocate speed to memory-bound applications and under-allocate to compute-bound ones.

Online models

In general, all machine learning techniques take observations of some phenomena and produce a model to estimate future outcomes. In our specific case, we want to take observations about application's performance and power given a resource allocation and predict future applications' behavior. The most straightforward approach is to create a linear regression model based on the configuration laid out as the input features Tibshirani [1996], Yuan and Lin [2006], Lee et al. [2008]. Learning in this model requires samples of the performance and power for different resource configurations. Since the systems are usually non-linear, polynomial regression models often give better results, but increases the number of samples required to fit the model Lee and Brooks [2008]. Regression models also provide a confidence interval for their estimates which is important for adaptively tuning the pole.

Transfer learning

Transfer learning refers to the process of applying knowledge gained from one scenario to a new, similar one Pan and Yang [2010]. To provide intuition we offer a simple example. Suppose we have observed many prior applications, all of which are either completely compute-bound or completely memory-bound, and we have an equal number of both. The only resource we can allocate is clockspeed, which will increase the performance of compute-bound applications, but not memory-bound ones. When we encounter new application, we must estimate its response to clockspeed. The online model will not use prior knowledge, but will observe many different clock speeds for the new application, leading to high overhead. The offline model will predict the mean response of prior applications, meaning it will over-allocate speed to memory-bound applications and under-allocate to compute-bound ones. The transfer learning approach takes a small number of samples

and combines those with prior knowledge: if the new observations show that clockspeed has no effect on performance these models will use only the prior memory bound applications, otherwise, it will use the compute-bound applications.

Netflix Algorithm: The Netflix problem is a famous challenge posted by Netflix to find estimate users' movie preferences. The challenge was one by realizing that if 2 users both like some movies they might have similar taste in other movies. This approach allows allows models to borrow large amounts of data from other users to answer some questions about a new user. Delimitriou and Kozyrakis use this algorithm to predict application response to heterogeneous resources in data centers Delimitrou and Kozyrakis [2013, 2014].

Bayesian Models: A hierarchical Bayesian model (HBM) provides a statistically sound framework for learning across applications and devices Mishra et al. [2015]. In the HBM, each application has its own model, allowing specificity, but these models are conditionally dependent on some underlying probability distribution with a hidden mean and co-variance. In practice, an HBM estimates a model for a new application using a small number of observations and combining those observations with the large number of previous observations of other applications. Rather than over-generalizing, the HBM uses only similar applications to learn new models. The HBM's accuracy increases as more applications are observed because increasingly diverse behaviors are represented in the pool of prior knowledge. Of course, the computational complexity of learning also increases with increasing applications.

4.2.5 Formal Analysis

Control System Complexity

CALOREE's control system (see Algorithm 1) runs on the local device along with the application under control, so its overhead must be minimal. In fact, each controller invocation is $O(1)$. The only parts that are not obviously constant time are the PHT lookups. Provided the PHT resolution

Algorithm 1 CALOREE control.

Input: Initialize the controller with a general model of speedup and powerup. Send power and performance samples to learner and receive a PHT.

```
while True do
    Measure streaming application performance
    Compute required speedup (Equation (4.2))
    Lookup s_hi and s_lo with PHT
    Compute τ_hi and τ_lo (Equations 4.9 & 4.10)
    Configure to system to hi & sleep τ_hi.
    Configure to lo & sleep τ_lo.
end while
```

is sufficiently high to avoid collisions, then each PHT lookup requires constant time.

Control Theoretic Formal Guarantees

The controller's pole $\rho(t)$ is critical to providing control theoretic guarantees in the presence of learned models. CALOREE requires any learner estimate not only speedup and powerup, but also the variance σ. CALOREE uses this information to derive a lower bound for the pole which guarantees probabilistic convergence to the desired performance. Specifically, we prove that with probability 99.7% CALOREE converges to the desired performance if the pole is

$$\lfloor 1 - \lfloor max(\hat{s})/(min(\hat{s}) - 3\sigma) \rfloor_0 \rfloor_0 \leq \rho(t) < 1,$$

where $\lfloor x \rfloor_0 = \max(x, 0)$ and $\hat{s}$ is the estimated speedup. The appendix contains the proof. Users who need higher confidence can set the scalar multiplier on σ higher; *e.g.*, using 6 provides a 99.99966% probability of convergence.

Thus we provide a lower-bound on the value of $\rho(t)$ required for a user to be confident that CALOREE will converge to the desired performance. This pole value only considers performance, and not energy efficiency. In practice, we find that it better to use a higher pole based on the *uncertainty* between the controller's observed energy efficiency and that predicted by the learned

model. We follow prior work in quantifying uncertainty as $\rho(t)$ Tokic [2010]:

$$\begin{aligned} \beta(t) &= \exp\left(-\left(\left|\frac{\bar{s}(t)}{\bar{p}(t)} - \frac{\hat{s}(t)}{\hat{p}(t)}\right|\right)/5\right) \\ \rho(t) &= \frac{1-\beta(t)}{1+\beta(t)} \end{aligned} \tag{4.11}$$

where $\bar{s}$ and $\bar{p}$ are the measured values of speedup and power up and $\hat{s}$ and $\hat{p}$ are the estimated values from the learner. This measure of uncertainty captures both power and performance. We find that it is generally higher than the pole value given by our lower bound, so in practice CALOREE sets the pole dynamically to be the higher of the two values and CALOREE makes spot updates to the estimated speedup and power based on its observations.

Probabilistic Convergence Guarantees

Theorem 1. *Let* $\mathbf{s}$ *and* $\hat{\mathbf{s}}$ *denote the true and estimated speedups of various configurations in set* C *as* $\mathbf{s} \in \mathbb{R}^{|C|}$. *Let* σ *denote the estimation error for speedups such that,* $\hat{s}_i \sim N(s_i, \sigma^2)\ \forall\ i$. *We can show that with probability greater than 99.7%, the pole* $\rho(t)$ *can be chosen to lie in the range,* $[\lfloor 1 - \lfloor max(\hat{s})/(min(\hat{s}) - 3\sigma)\rfloor_0\rfloor_0, 1)$, *where* $\lfloor x\rfloor_0 = \max(x, 0)$.

Proof. Let Δ denote the multiplicative error over speedups, such that $\widehat{speedup}(t)\Delta = speedup(t)$. To guarantee convergence the value of pole, $\rho(t)$ can vary in the range $\lfloor 1 - \frac{2}{\Delta})\rfloor_0, 1)$Filieri et al. [2014]. The lowest value of $\rho(t)$ offer the fastest convergence. We have described in Equations 4.9 & 4.10 that any speedup can be written as a linear combination of two configuration speedups as,

$$speedup(t) = \hat{s}_{hi} \cdot \tau_{hi} + \hat{s}_{lo} \cdot (T - \tau_{hi}) \tag{4.12}$$

$$\widehat{speedup}(t) = s_{hi} \cdot \tau_{hi} + s_{lo} \cdot (T - \tau_{hi}) \tag{4.13}$$

We can upper bound and lower bound each of these terms,

$$speedup(t) \leq T\hat{s}_{hi} \text{ and } \widehat{speedup}(t) \geq Ts_{lo} \tag{4.14}$$

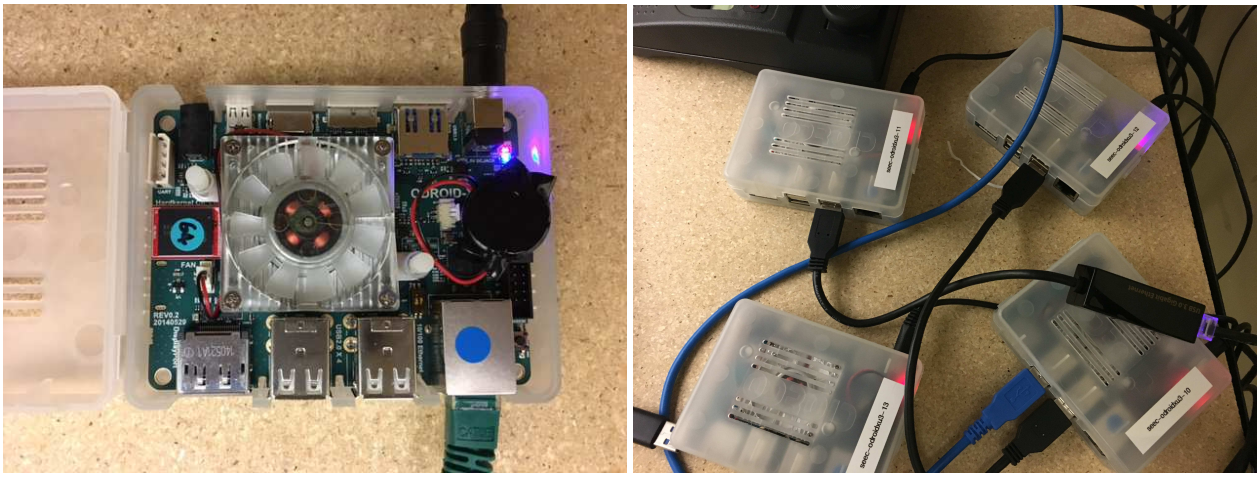


Figure 4.7: ODROID-XU3 boards used in the evaluation.

The estimates of speedups are close to the actual speedups since $\hat{s} \sim N(s, \sigma^2)$, therefore with probability greater than 99.7% and the speedups can be given by, $s_{lo} \geq \hat{s}_{lo} - 3\sigma$. Hence, $\widehat{speedup}(t) \geq T(\hat{s}_{lo} - 3\sigma)$. Since, over all configurations, $\Delta \leq \lfloor max(\hat{s})/(min(\hat{s}) - 3\sigma) \rfloor_0$, we can choose the pole from the range, $([\lfloor 1 - \lfloor max(\hat{s})/(min(\hat{s}) - 3\sigma) \rfloor_0 \rfloor_0, 1)$.

□

4.3 Experimental Setup

4.3.1 Platform and Benchmarks

We run streaming applications on four ODROID-XU3 devices running Ubuntu 14.04, as shown in Figure 4.14. The ODROIDs have Samsung Exynos 5 Octa processors using the ARM big.LITTLE architecture. Each has 19 speed settings for the 4 big cores and 13 for 4 LITTLE cores. Each board has an on-board power meter updated at 1/4 msintervals, this meter captures core, GPU, memory, and flash drive power. Each resource configuration (combination of big/LITTLE cores and clock-speeds) has a different performance and power, which is application-dependent.

We use 20 benchmarks from different suites including PARSEC Bienia et al. [2008], Minebench

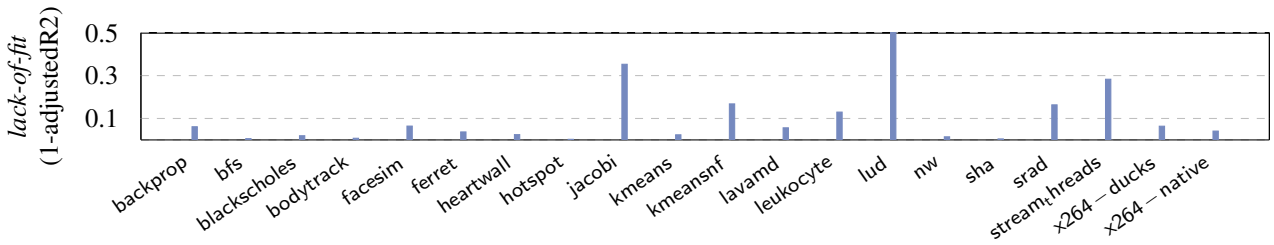


Figure 4.8: *Lack-of-fit* for performance vs clock-speed. Lower lack-of-fit indicates a more compute-bound application, higher values indicate a memory-bound one.

Narayanan et al. [2006], Rodinia Che et al. [2009], and STREAM McCalpin [1995]. Figure 4.8 shows the variety of workloads indicated by the *lack-of-fit* or the absence of correlation between frequency and performance. Applications with high lack-of-fit do not speed up with increasing frequency—typical of memory bound applications. Applications with low lack-of-fit do see increasing performance with increasing clock-speed Miyoshi et al. [2002]. Applications with intermediate lack-of-fit tend to improve with increasing clock speed up to a point and then see no further improvement. Each application has an outer loop which processes one unit in a data stream (*e.g.*, a point for `kmeans` or a frame for `x264`). The application signals the completion of processing a single stream element using a standard API Hoffmann et al. [2010]. Performance targets are specified as application-specific latencies for these stream elements.

4.3.2 Evaluation Metrics

The latency targets represent performance requirements. To test a variety of requirements, we run our applications with targets that represent 50-90% of the maximum speed and evaluate CALOREE under each constraint. We quantify the performance reliability by measuring the number of deadlines that were missed for each application and performance target. If the application processes n elements total and m of those elements took longer than the target latency we compute deadline misses as:

$$deadline\,misses = 100\% \cdot \frac{m}{n}. \tag{4.15}$$

We evaluate energy savings by constructing an oracle. We run every application in every resource configuration and record performance and power for every stream element. By post-processing this data we determine the optimal resource configuration for each stream element and performance target. To compare across applications, we normalize energy:

$$normalized\,energy = 100\%.\left(\frac{e_{measured}}{e_{optimal}} - 1\right) \tag{4.16}$$

where $e_{measured}$ is measured energy and $e_{optimal}$ is the optimal energy produced by our oracle. We subtract 1, so that this metric shows the percentage of energy over optimal.

4.3.3 Points of Comparison

We compare CALOREE to a number of existing learning and control approaches:

1. *Race-to-idle*: This well known heuristic allocates all resources to the application to complete each stream element as fast as possible, then idles until the next element is available Kim et al. [2015], Miyoshi et al. [2002], **?**. This heuristic requires no knowledge of the application and never misses deadlines in a single-application scenario.

2. *PID-Control*: This is a standard single-input (performance), multiple-output (big/LITTLE core counts and speeds) proportional-integral-controller representative of several that have been proposed for computer resource management Hellerstein et al. [2004], Sharifi et al. [2011]. This controller is tuned to provide the best average case behavior across all applications and targets.

3. *Online*: observes the new application in small number of configurations then performs polynomial multivariate regression to estimate unobserved configurations' behavior Mishra et al. [2015], Li and Martinez [2006], Ponomarev et al. [2001].

4. *Offline*: does not observe any values for the current application—instead using previously observed applications to estimate power and performance as a linear regression. This base-

line represents the class learners that build models with a training set and apply them without new observations Zhang and Hoffmann [2016], Lee and Brooks [2006], Lee et al. [2008].

5. *Netflix*: is a matrix completion algorithm for the Netflix challenge. Variations of this approach allocate heterogeneous resources in data centers Delimitrou and Kozyrakis [2013, 2014].

6. *HBM*: is a hierarchical Bayesian learner previously used to allocate resources to meet performance goals with minimal energy in server systems Mishra et al. [2015].

7. *Adaptive-Control*: is a state-of-the-art, adaptive controller that meets application performance with minimal energy Imes et al. [2015a]. This approach requires a user-specified starting model, we use the model produced by the *Offline* learner. A recent study comparing ML and control techniques for resource allocation showed that there was no single best approach, but adaptive control had the best average behavior Maggio et al. [2012].

CALOREE is a framework for combining different learners with the CALOREE control system. We compare the above baselines to four different versions of CALOREE:

1. *CALOREE-NoPole*: uses the HBM learner, but sets the pole set to 0. This baseline shows the importance of incorporating the confidence interval and model variance into control. All other versions of CALOREE set the pole according to Section 4.2.5.

2. *CALOREE-online*: uses the online learner.

3. *CALOREE-nuclear*: uses the Nuclear learner.

4. *CALOREE-HBM*: uses the HBM learner.

In all cases that require prior knowledge, we ensure that knowledge of the application under test is never included in that set of prior knowledge. Specifically, we use leave-one-out cross validation: to test application x, we form a set of all other applications, train the models, and then test on x.

4.4 Experimental Evaluation

4.4.1 Performance and Energy for Single App

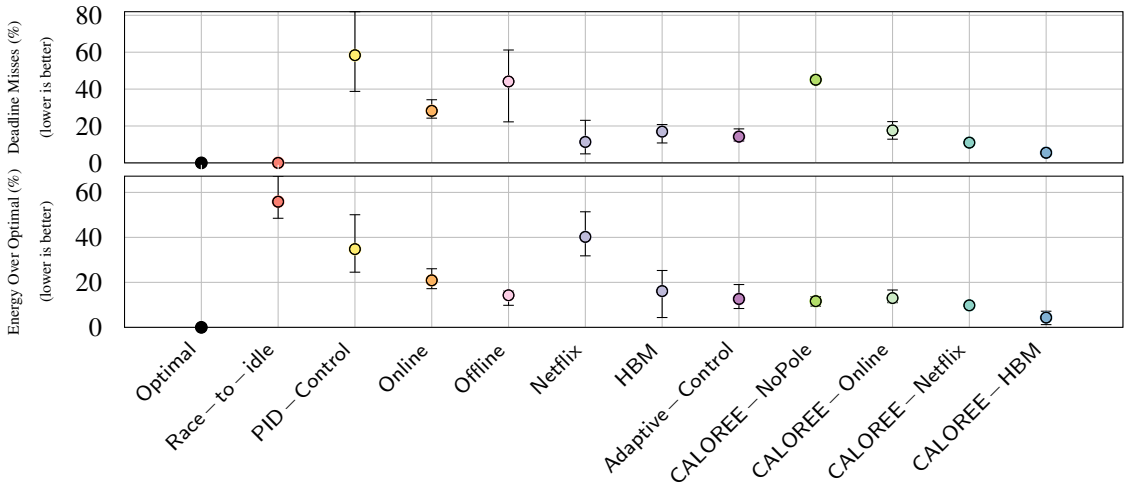


Figure 4.9: Summary data for single-app scenario.

We set a range of performance targets from 50-90% of the maximum achievable performance and measure the deadline misses and energy over optimal for all points of comparison. Figure 4.9 represents the summary results as an average error across all targets for the single application scenario. This figure shows two charts with the percentage of deadline misses in the top chart and the energy over optimal in the bottom. The dots show the average for each technique, while the error bars show the minimum and maximum values.

Not surprisingly, race-to-idle meets all deadlines, but its conservative resource allocation has the highest average energy consumption. Among the prior learning approaches Nuclear has the lowest average deadline misses (11%), but with high energy (40% more than optimal), while the HBM has higher deadline misses (17%) but with significantly lower energy consumption (16%). Adaptive control achieves similar deadline misses (14%) with lower average energy than any of the prior learning approaches (12%). CALOREE with no pole misses 45% of all deadlines, which is clearly unacceptable.

When we allow CALOREE to adaptively tune its pole, however, we see greatly improved

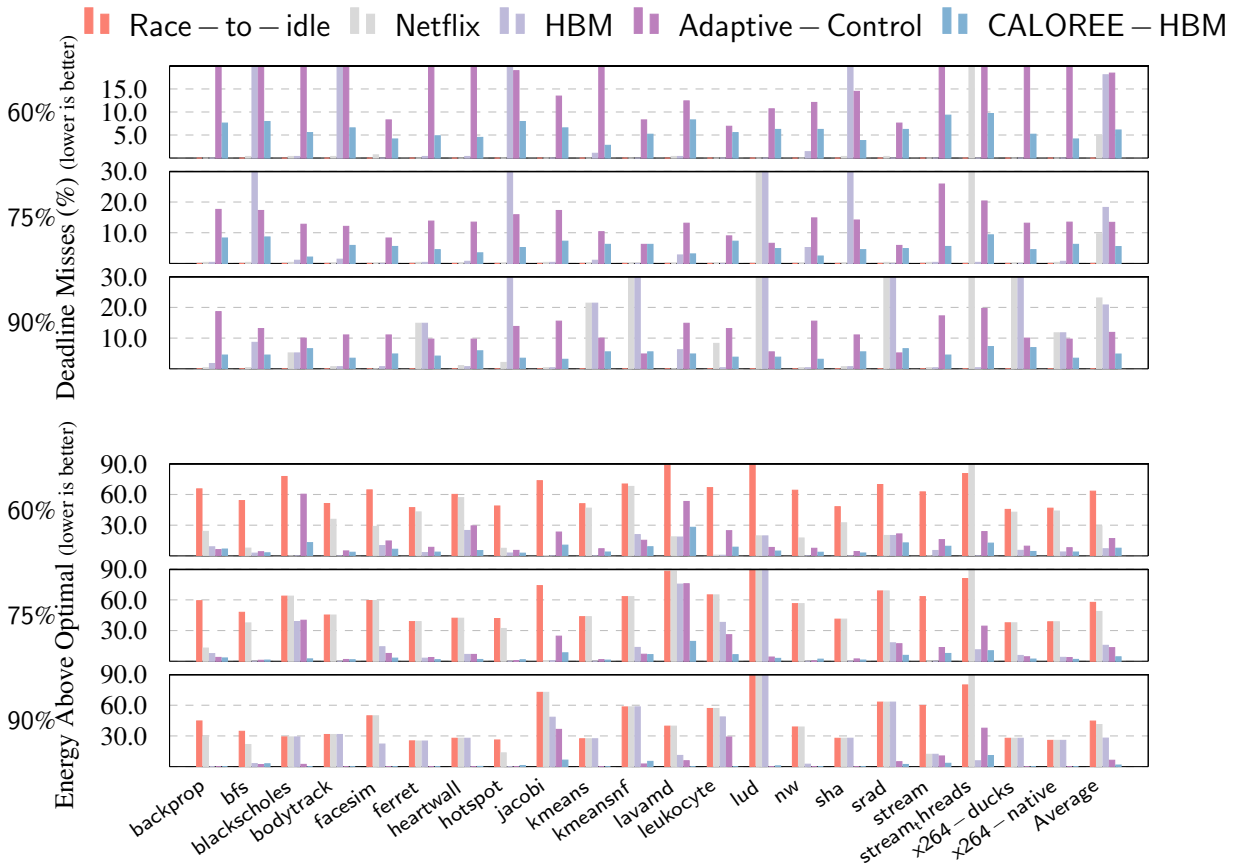


Figure 4.10: Comparison of application performance error and energy for single application scenario.

results. The best combination is CALOREE with the HBM, which misses only 5.5% of deadlines on average, while consuming just 4.4% more energy than optimal. These numbers represent large improvements in both performance reliability and energy efficiency compared to prior approaches. The other learners paired with CALOREE achieve similar results to the prior adaptive control approach.

The summary data shows us that the best prior approaches are race-to-idle, HBM, and adaptive control. Figures **??** and **??** show the detailed results for the 60, 75, and 90% targets comparing the best of these prior approaches to CALOREE with no pole and CALOREE coupled with the HBM—other data has been omitted for space. The benchmarks are shown on the x-axis; the y-axis shows the number of deadline misses and the normalized energy, respectively.

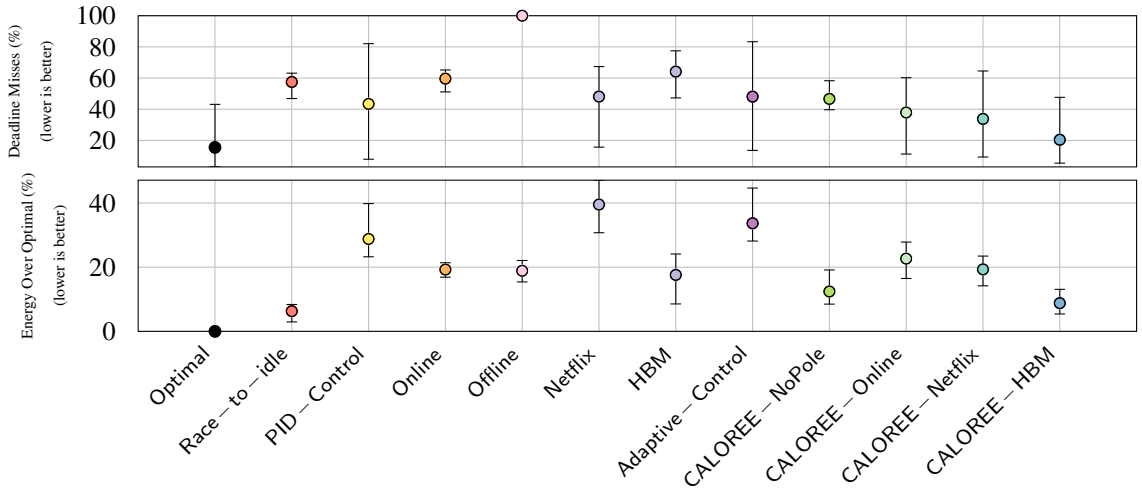


Figure 4.11: Summary data for multi-app scenario.

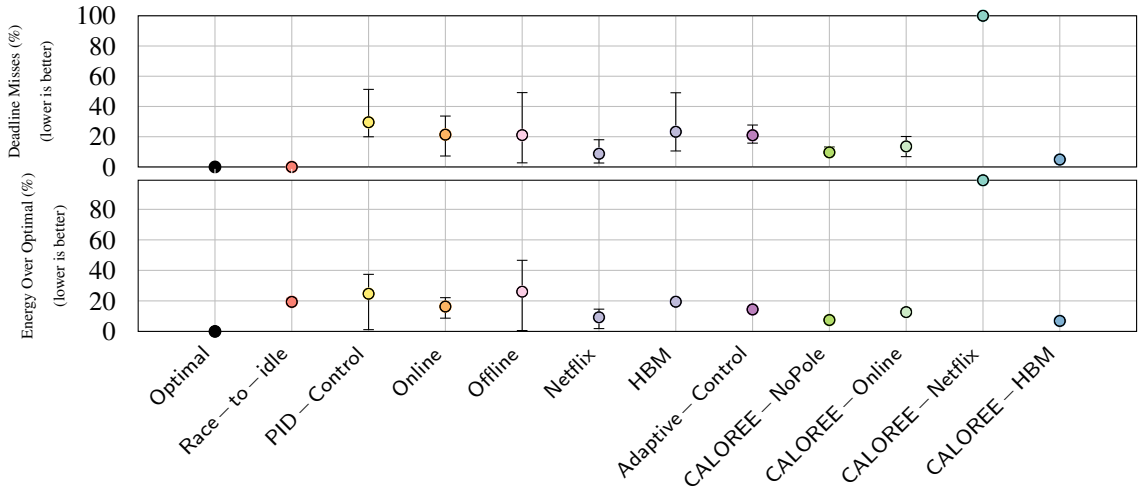


Figure 4.12: Clover Summary data for single-app scenario.

4.4.2 Performance and Energy for Multiple Apps

We again launch each benchmark with a performance target (using the same targets as the prior study). Halfway through execution, we start another application randomly drawn from our benchmark set, which we bind to one big core, which interferes with the application CALOREE is controlling. Delivering performance to the original application in this dynamic scenario tests CALOREE's ability to react to environmental changes.

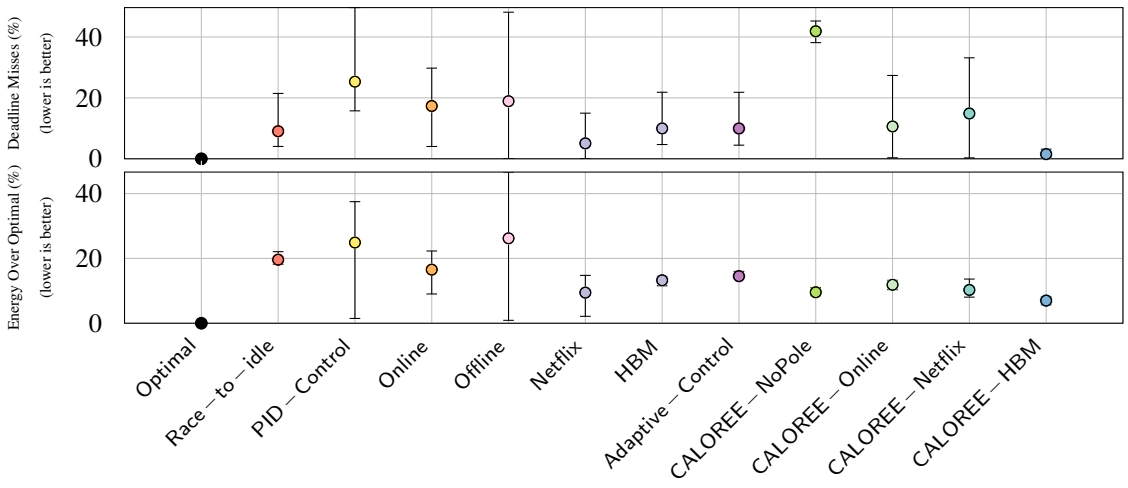


Figure 4.13: Clover Summary data for multi-app scenario.

Figure 4.13 summarizes the results as the average number of deadline misses and energy over optimal for all approaches. We note that some targets are unachievable for some applications. Due to these unachievable targets, both optimal and race-to-idle show some deadline misses. Race-to-idle misses more deadlines than optimal because it cannot make use of LITTLE cores to do some work, it simply continues using all big cores despite the degraded performance due to the second application. In fact, most approaches do badly in this scenario—even adaptive control misses 50% of the deadlines. CALOREE with the HBM produces the lowest deadline misses with an average of 20%, which is only 5% more than optimal. It also produces the second lowest energy, using slightly more than race-to-idle because it uses LITTLE cores to make up for some work that cannot be done on a big core due to the second application. Figures **??** and **??** shows the detailed results. The 90% target is generally not reachable in this scenario as it would require the controlled application to have exclusive use of all big cores. Hank: I am thinking that we subtract out the optimal number of deadline misses here.

4.4.3 Adapting to Phase Changes

We compare CALOREE and Adaptive-Control reacting to input variations. Figure 4.15 shows the `x264` video encoder application with 2 different phases caused by a scene change in the input that

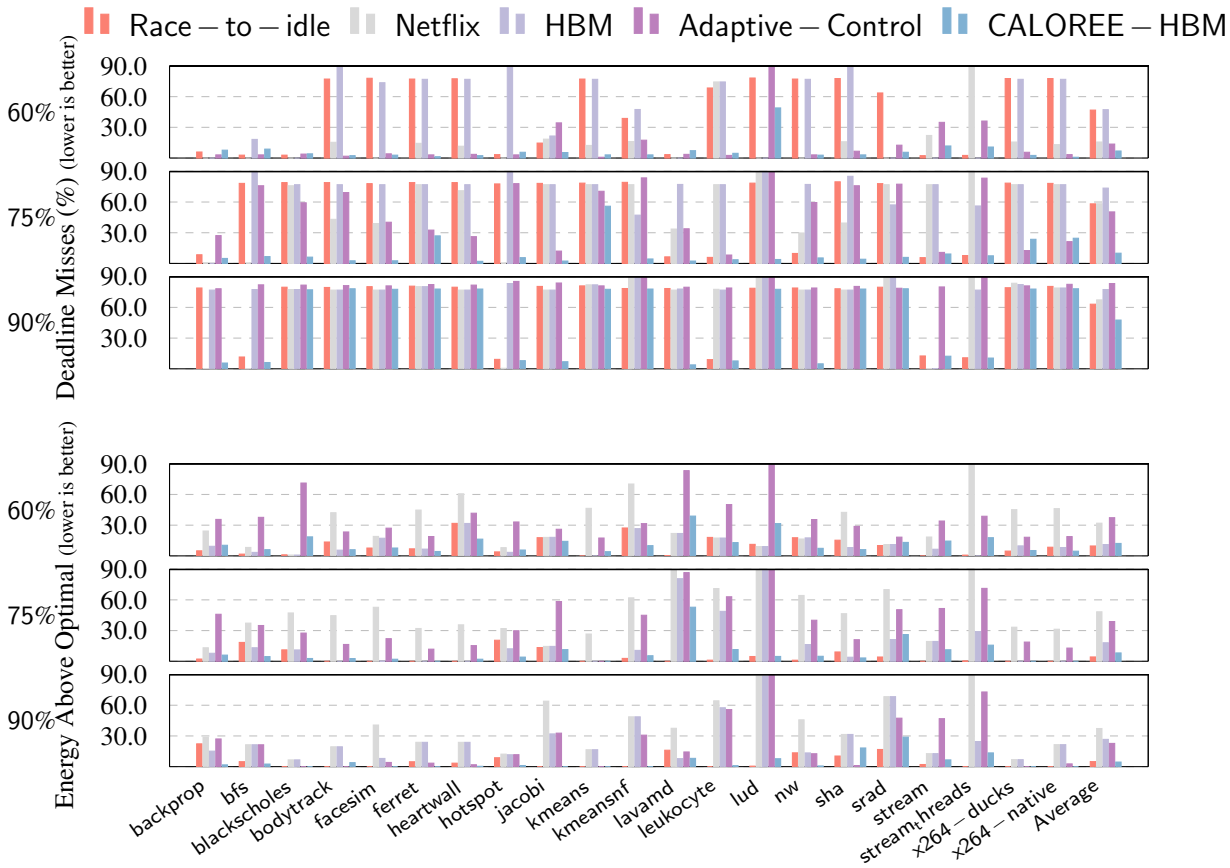


Figure 4.14: Comparison of application performance error and energy for multiple application scenario.

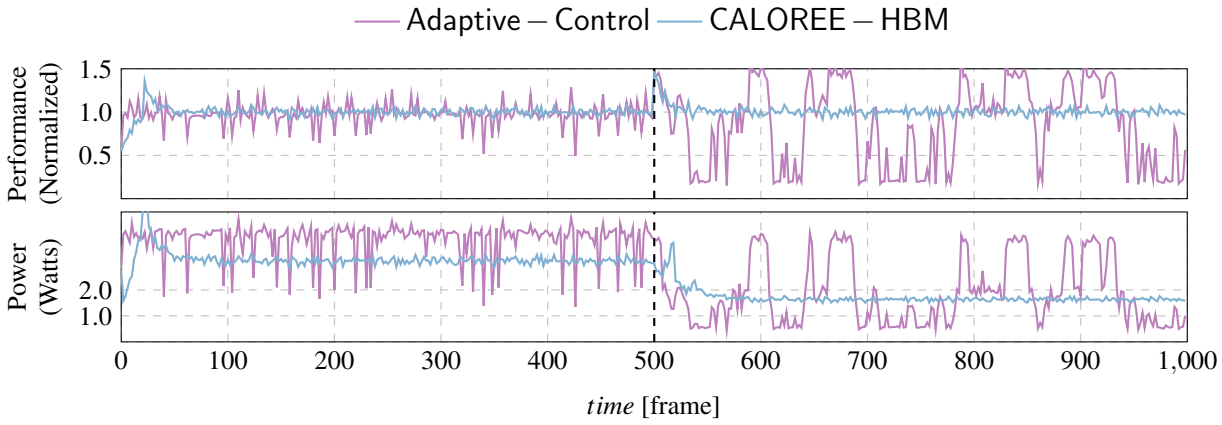


Figure 4.15: Controlling `x264` through scene changes.

occurs at the 500^{th} frame. The first scene is difficult and the second one is significantly easier. In the first scene, CALOREE is closer to the desired performance (1 in the figure) and operates at a lower power state compared to Adaptive-Control. Adaptive-Control is operating at a configuration

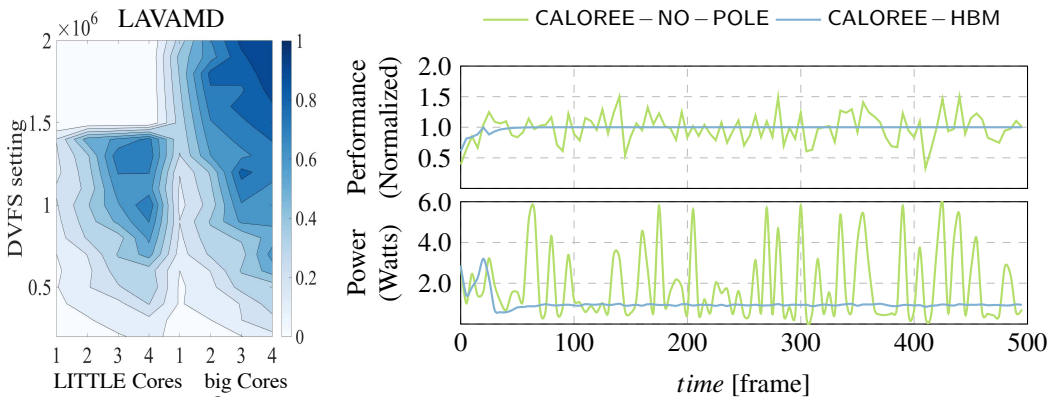


Figure 4.16: (a) LAVAMD's performance with different resources. (b) The pole's effects on LAVAMD's behaviors.

not on the Pareto frontier of power and performance. When the input changes, CALOREE is still meets the performance target with far less fluctuations compared to Adaptive-Control.

4.4.4 The Pole's Importance

LAVAMD has one of the most complicated responses to resource usage on our system with multiple local optima, as shown in Figure **??**. Section 4.2.5 presents an analytical argument that tuning the controller to model variance and confidence interval prevents oscillation and provide probabilistic control theoretic guarantees despite using noisy, learned models to control such complicated behavior. We now demonstrate this empirically by showing LAVAMD's single-app behavior controlled by both CALOREE-NoPole and CALOREE-HBM to meet the 80% target.

Figure **??** shows the results, with time on the x-axis and normalized performance and power on the respective y-axes. CALOREE-NoPole oscillates around the desired performance and causes wide fluctuations in power consumption. In contrast, after receiving the model from its learner, CALOREE provides reliable performance right at the target value (normalized to 1 in this case). CALOREE also saves tremendous energy because it does not oscillate but uses a mixture of big and LITTLE cores to keep energy near minimal.

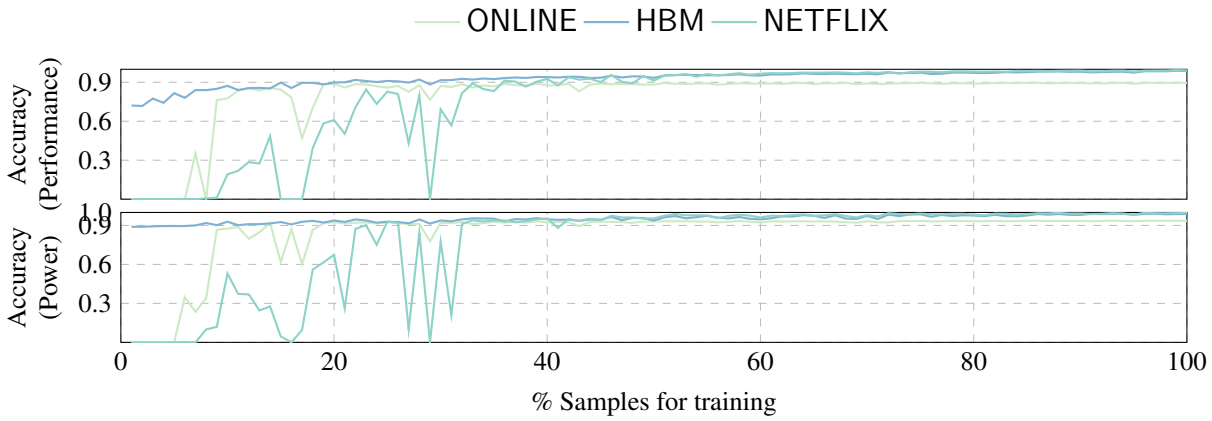


Figure 4.17: Estimation accuracy versus sample size.

4.4.5 Sensitivity to the Measured Samples

We vary the number of samples taken and show how it affects model accuracy for the Online, Netflix, and HBM learners. Accuracy is how close the model is to ground truth (found through exhaustive exploration), with 1 meaning the model perfectly reproduced the real performance or power. Accuracy is significant, because the smaller the number of samples, the faster the controller can switch from a general model to the learner's application-specific model.

Figure 4.17 compares the Online, HBM, and Netflix learners for both performance (top) and power (bottom). The figure shows sample size on the x-axis and accuracy on the y-axis. The HBM initially performs as well as Offline and as sample size increases, the accuracy uniformly improves, exceeding 90% after 20 samples. The Online approach needs at least 7 samples before it can even generate a model. As Online receives more samples, its accuracy improves but never exceeds HBM's for the same number of samples. Netflix is very noisy for sample sizes, but once the number of samples reaches about 50, it is competitive with HBM. These results not only demonstrate the sensitivity to sample size, they show why CALOREE-HBM achieves better results than the other learning mechanisms.

4.4.6 Overhead

CALOREE's main source of overhead is sampling where the applications need to run through a few configurations before CALOREE can reliably estimate the entire power and performance frontier. We argue that the sampling cost can be distributed across devices by asking each of them to contribute samples for estimation. Once the sampling phase is over, the HBM is quite fast and can generate an estimate as fast as 500 ms which is significantly smaller than the time required to run any of our applications. Additionally, CALOREE's asynchronous communication means that the controller never waits for the learner. Using the four ODROIDs in our experimental system, each board only needs to contribute 4 samples to achieve 90% accuracy. In the worst case (`facesim`), this sampling overhead is less than 2%. For all other benchmarks it is lower, and for most it is negligible.

The controller requires only a few floating point operations to execute, plus the table lookups in the PHT. To evaluate its overhead, we time 1000 iterations. We find that it is under 2 microseconds, which is significantly faster than we can change any resource allocation on our system. We conclude that the controller has negligible impact on performance and energy consumption of the controlled device.

4.5 Summary

While much recent work has built systems to support learning and big data, in this work we use learning and big data to build better systems. Specifically by proposing CALOREE, a combination of machine learning and control for managing resources to meet performance requirements with minimal energy. CALOREE's unique contribution is showing how machine learning and control theory can be combined at runtime to provide more reliable performance and lower energy than either in isolation. Furthermore, this combination is not just practical, it provides formal guarantees that the system will converge to the desired performance.

CHAPTER 5

ESP: APPLICATION INTERFERENCE ESTIMATION

5.1 Motivation

This section discusses current state-of-the-art regularization methods and how they could be used to predict application interference. A complete coverage of these techniques is beyond the scope of this work—our goal is to present enough background for readers to understand ESP's unique approach to interference prediction.

We use a running example to build intuition. This example is simplified, but our hope is that it provides sufficient intuition for readers to understand the full results.

Example *Suppose we have 4 applications given as,* {`bfs, cfd, jacobi, kmeans`}. *For each application, we measure 4 features of its execution when run by itself: instructions per clock* (`IPC`), *L2 Access Rate* (`L2`), *L3 Access Rate* (`L3`) *and its memory bandwidth* (`MEM`). *The goal is to predict the slowdown that each of these applications will experience when run together in any combination knowing only these 4 features for each application run individually.*

5.1.1 Regularization Overview

We are interested in predicting applications' slowdown when co-scheduled with other applications as a function of individual applications' features. The following is a general formulation of this problem:

$$\mathbf{z} \sim f(\mathbf{X}), \tag{5.1}$$

where $\mathbf{z}$ is a vector each element is one application's slowdown when run with the other applications. $\mathbf{X}$ is a matrix containing the measured features of applications when run individually. $f(.)$ is the*prediction function* that maps the measured features into predicted slowdown. The $\sim$ sign indicates that $\mathbf{z}$ is a random variable drawn from a distribution given by $f(.)$.

To go from Equation (5.1) to a useful model two issues must be addressed: (1) which features we choose to include in $\mathbf{X}$ and (2) what function $f(.)$ best represents this problem. In fact, these issues are intertwined; depending on the features we include we can choose different functions. It is important to choose a function that captures the data without overfitting so that the model's generalization error is small. The model also must be computationally fast so that it is practical.

Example *Suppose we want to predict the interference of* `bfs` *and* `cfd` *when co-scheduled. The vector z represents slowdown, with the first element* `bfs`*'s slowdown and the second is* `cfd`*'s. X in Equation* (5.1) *is constructed using* `IPC`*,* `L2`*,* `L3` *and* `MEM` *of the* `bfs` *and* `cfd` *when they run in isolation. The first four columns of X are the features for* `bfs` *and the last four columns are* `cfd`*'s features. The goal is to find the* $f(.)$ *that produces the best prediction of slowdown.*

5.1.2 Linear Regression

Regression models are the most commonly used statistical models for predictions. A linear model predicts outcomes as a linear combination of input features:

$$\mathbf{z} = \mathbf{X}\beta + \varepsilon, \tag{5.2}$$

$\mathbf{z} \in \mathbb{R}^n$ is the dependent variable to be predicted. $\mathbf{X} \in \mathbb{R}^{n \times p}$ is the measured independent features. $\beta \in \mathbb{R}^p$ is the coefficient vector that combines the features to produce the prediction. $\varepsilon \in \mathbb{R}^n$ is a vector representing inherent noise.

Building a linear model is the process of determining β by running experiments and measuring both the features and the corresponding outcomes. These measurements are samples of $\mathbf{z}$ and $\mathbf{X}$, written as $(s(\mathbf{z}), s(\mathbf{X}))$. After sampling, $s(\mathbf{z})$ and $s(\mathbf{X})$ are known and we can solve for β. Then we can use β and new features ($\mathbf{X}$) to predict $\mathbf{z}$ for instances that have not been measured.

5.1.3 Regularized Linear Regression

When determining β, if the number of samples in $\mathbf{z}$ (or length of $s(\mathbf{z})$) is less than the number of features p the problem is ill-posed; *i.e.*, $\mathbf{X}$ is not invertible and there are infinitely many solutions. This particular case where $p > n$ is called the *high-dimensional* setting. Machine learning researchers have developed several *regularization* methods which add structure to high-dimensional problems to make them solvable.

Example *To apply linear regression, $f(.)$ is chosen to be $f(.) = \mathbf{X}\beta$ and β is the parameter that we would like to learn from the data. For example, we could observe 3 pairs from the set of 6 pairs (for 4 applications in our example) and predict the performance for the other 3 pairs. $\mathbf{z}$ denotes the performance vector for our applications, thus for 3 pairs, $\mathbf{z} \in \mathbb{R}^6$. The matrix $\mathbf{X}$ in Equation* (5.1) *is $\mathbf{X} \in \mathbb{R}^{6\times 8}$ and $\beta \in \mathbb{R}^8$. Now, the system of equations $\mathbf{z} = \mathbf{X}\beta$ has infinitely many solutions so we must* regularize*—i.e., add more structure to—the problem.*

There are several methods to add structure to a regression problem. In general, none are uniformly better than the others and their performance is data-dependent. Hence, standard practice is to use the model with the highest out-of-sample predictive power; *i.e.*, to separate samples into training and test data, build multiple models with the training data, and use the model that most accurately predicts the test data.

Ridge regularization

One way to add structure to the problem is to include additional constraints. *Ridge* regression penalizes large coefficients in β Hoerl and Kennard [1988] by requiring that $\|\beta\|_2^2 \leq t$, where t is a threshold:

$$\min_{\beta} \|\mathbf{z} - \mathbf{X}\beta\|_2^2 \quad \text{s.t. } \|\beta\|_2^2 \leq t \tag{5.3}$$

Example *When we add the ridge regularizer to our example, the modified problem is $\|\mathbf{z} - \mathbf{X}\beta\|_2^2$ and $\sum_{i=1}^{8} \beta^2[i] < t$. This problem has a unique solution for a given t. This regularization shrinks*

large β values but does not make them 0. Hence, even if a feature i is not important it receives a positive coefficient β_i. The downside of this approach, then, is that it may incorporate irrelevant features into the model. In our example, it is unlikely that L2 Access Rate is a valuable feature for predicting application interference, because L2 caches are private in most server class processors and therefore not a source of contention.

Lasso regularization

feature selection is another regularization method equivalent to setting some elements of β to 0, so those features cannot influence $\mathbf{z}$. Formally, feature selection can be achieved by adding $\mathscr{L}_0$ constraint—$\|\beta\|_0 \leq t$—to Equation (5.2), but doing so makes the problem non-convex and NP-hard. $\mathscr{L}_1$ regularization—known as *lasso*—is the best convex relaxation for $\mathscr{L}_0$ regularization:

$$\min_{\beta} \|\mathbf{z} - \mathbf{X}\beta\|_2^2 \quad \text{s.t. } \|\beta\|_1 \leq t \tag{5.4}$$

Using lasso, the number of selected features will be smaller than the number of samples. A potential drawback, however, is that it cannot capture models where there are more relevant features than samples.

Example *The modified example looks like, $\|\mathbf{z} - \mathbf{X}\beta\|_2^2$ and $\sum_{i=1}^{8} |\beta[i]| < t$. Lasso regularization will set $n - p$ features to 0. In our example, that means three features will be zeroed out. This could be a problem because it is likely that IPC, L3 accesses, and Memory Bandwidth for both applications (i.e., a total of 6 features for the two applications) will be necessary to predict interference as all three of these features correspond to shared hardware structures. Specifically, lasso regularization will probably zero-out the L2 Accesses for both applications, but by construction, it must also zero-out at least one of the relevant features, likely producing lower accuracy estimates.*

Elastic-net regularization

Elastic-net addresses the drawbacks of the two previous techniques Zou and Hastie [2005]:

$$\min_{\beta} \|\mathbf{z} - \mathbf{X}\beta\|_2^2 \quad \text{s.t. } \alpha\|\beta\|_2^2 + (1-\alpha)\|\beta\|_1 \leq t \tag{5.5}$$

where $\alpha = \lambda_1/(\lambda_1 + \lambda_2)$ and λ_1 and λ_2 are the regularization parameters. In practice, setting $\alpha = 0.5$ is common and λ_1 is set during model training using cross-validation. Thus under these settings elastic-net is,

$$\text{EN}(\mathbf{z}, \mathbf{X}) = \underset{\|\beta\|_2^2 + \|\beta\|_1 \leq t}{\arg\min} \|\mathbf{z} - \mathbf{X}\beta\|_2^2 \tag{5.6}$$

Elastic-net *groups* variables so that strongly correlated variables tend to be selected or rejected together.

Example *Adding elastic-net to our example, the modified problem looks like,* $\|\mathbf{z} - \mathbf{X}\beta\|_2^2$ *and* $\sum_{i=1}^{8} \alpha|\beta[i]| + (1-\alpha)\|\beta[i]\|^2 < t$. *The regularization in this case would shrink large values for* β *and some coefficients would shrink to 0. It would zero-out unimportant feature like* `L2` *for both applications, yet capture all the important features (IPC, L3, MEM) for both applications even when the number of samples is smaller than the number of features.*

5.1.4 Higher-order Models

In some cases, linear models do not accurately predict the problem. One higher-order approach is to add *interaction* terms to the model, meaning that the dependent variable is possibly a multiplicative combination of some features.

A model with interaction terms can be found by adding additional terms to $\mathbf{X}$ and β. Thus, a high dimensional problem becomes even higher dimensional, increasing model complexity and overhead. For a matrix $\mathbf{X}$ with dimensions $n \times p$, $n \ll p$ *elastic-net* is $\mathcal{O}(p^3)$ Zou and Hastie [2005], hence for a quadratic model the computational complexity blows up to $\mathcal{O}(p^6)$. Even though this model is richer and captures complex interactions among features, it is prohibitively expensive. ESP's motivation is to achieve the prediction accuracy of interaction terms while maintaining the

practicality of linear models.

Example *We believe the interaction between our features captures the interference more accurately than a linear model. However, we do not know which of these feature interactions are important in advance—e.g., is it IPC and L3, L3 and MEM, the L3 features for both applications, or something else? Clearly, even in our simple example, the design space has become much more complex. Specifically, we capture the new interaction terms by extending the feature matrix* $\mathbf{X}$ *to* $\tilde{\mathbf{X}}$*, which has 8 linear terms plus* $\binom{8}{2}$ *higher order terms. The new design matrix* $\tilde{\mathbf{X}}$ *in Equation* (5.1) *is* $\tilde{\mathbf{X}} \in \mathbb{R}^{6\times 36}$ *and* $\beta \in \mathbb{R}^{36}$*. We see—even for our simple example—the model complexity has greatly increased.*

5.1.5 A New Regularization Method

To summarize, regression models map features into predictions. When the feature space is large, regularization adds structure to make the problem well-formed. Higher-order models may provide more accurate predictions, but increase the cost of both training and applying the model.

Inspired by these observations, we present ESP, which splits regression modeling into two parts: (1) feature selection and (2) model building with interactive terms. First, ESP builds a linear model with elastic-net, which greatly reduces feature size without capturing interaction terms. Second, ESP builds a higher-order model with interaction terms using just those features selecting in the first step. By reducing the feature size in the first step, the complexity of the higher-order model remains tractable and we get the benefits of both approaches: *highly accurate predictions with manageable complexity.*

Given a set of m applications k applications to be co-scheduled, there are $n = \binom{m}{k}$ possible sets of k applications. We use p to denote the number of features. Let $f^{(k)}(.)$ be the prediction function; *i.e.*, $f^{(k)}(.)$ predicts the slowdown of each of the k co-scheduled applications using features measured when each application runs individually:

$$\mathbf{z}^{(k)} = f^{(k)}(\mathbf{X}^{(k)}) + \varepsilon, \tag{5.7}$$

Algorithm 2 CALOREE

Input: Training samples: $\left(s(\mathbf{z}^{(k)}),\ \mathbf{x}^{(k)} = s(\mathbf{X}^{(k)})\right)$; Feature matrix: $\mathbf{X}^{(k)}$,

1: Variable selection using elastic-net on linear model: $\mathscr{S} : \{i \in \mathscr{S}$ if $\beta^{(k)}[i] \neq 0\}$, where $\beta^{(k)} = \text{EN}(s(\mathbf{z}^{(k)}), \mathbf{x}^{(k)})$.
2: New feature matrix with higher order terms: $\tilde{\mathbf{X}} = [X_{\mathscr{S}}, \text{Int}(X_{\mathscr{S}}))]$.
3: Estimating performance: $\hat{\mathbf{z}}^{(k)} = \tilde{\mathbf{X}}^{(k)}\beta^{(k)}$, where $\beta^{(k)} = \text{EN}(s(\mathbf{z}^{(k)}), s(\tilde{\mathbf{X}}^{(k)})$
4: **return** Slowdown: $\hat{\mathbf{z}}^{(k)}$.

where $\mathbf{z}^{(k)} \in \mathbb{R}^{nk}$ is the slowdown vector, $\mathbf{X}^{(k)} \in \mathbb{R}^{nk \times 2p}$ is feature matrix and $\varepsilon \in \mathbb{R}^{nk}$ is the Gaussian error vector. For any index $j = \text{index}(i,S)$ of vector $\mathbf{z}$, z_j is the slowdown of application-i when co-scheduled with applications from set S. $X_{(j,:)}$ = [features of application-i, $\sum_{s \in S}$(features of application-s)]. In practice, the set of possible features includes everything that can be measured with the Intel performance counter monitor tool Willhalm [2012] on Linux x86 systems (see Table 5.1). Taking these measurements for each processor core we get $p = 409$ unique features per individual application.

Table 5.1: Category of low level features using Intel Performance Counter Monitor.

Code	Low level feature description
IPC	Instructions per CPU cycle
AFREQ	Relation to nominal CPU frequency while in active state
L2MISS	L2 cache misses (including other core's L2 cache *hits*)
L2HIT	L2 cache hit ratio (0.00-1.00)
L2CLK	Ratio of CPU cycles lost due to missing L2 cache but still hitting L3 cache (0.00-1.00)
L3HIT	L2 cache hit ratio (0.00-1.00)
L3CLK	Ratio of CPU cycles lost due to L3 cache misses (0.00-1.00)
READ	Bytes read from memory controller (in GBytes)
WRITE	Bytes written to memory controller (in GBytes)
COR-RES	Core residency
PKG-RES	Cackage residency
TEMP	Temperature reading in 1 degree Celsius relative to the TjMax temperature (thermal headroom)
EXEC	Instructions per nominal CPU cycle
FREQ	Relation to nominal CPU frequency='unhalted clock ticks'/'invariant timer ticks' (includes Intel Turbo Boost)
L3MISS	L3 cache misses
ENERGY	Energy consumed

Algorithm 2 summarizes ESP, which predicts slowdown when k applications are co-scheduled. The algorithm takes a few samples of random combinations of applications running together, denoted by $\left(s(\mathbf{z}^{(k)}),\ \mathbf{x}^{(k)} = s(\mathbf{X}^{(k)})\right)$, a design matrix containing the low-level-features for all $\binom{n}{k}$

combinations (denoted by $\mathbf{X}^{(k)}$) and outputs a slowdown vector for all combinations of applications. The first step does a feature selection on the linear model using Elastic-net as $\beta^{(k)} = \text{EN}(y^{(k)}, \mathbf{x}^{(k)})$. The non-zero coefficients of $\beta^{(k)}$ indicate the selected features, denoted by $\mathscr{S}$. Then, we construct a higher-order feature matrix $\tilde{\mathbf{X}}$ for those selected features only. We run elastic net again for the new feature space to obtain the final model which makes the performance prediction, denoted by $\hat{\mathbf{z}}^{(k)}$.

Algorithm 2 is run entirely offline. It produces the slowdown estimates for all applications (even those not sampled) in up to k co-scheduling groups. These slowdown estimates can then be used to predict performance for new combinations of applications in online schedulers. We show examples in the next section. We also show how the estimates can be trivially updated as new measurements become available online.

5.2 Scheduling with ESP

We examine two use cases for ESP: (1) batch scheduling applications on a single processor, (2) and scheduling dynamically arriving applications on multiple processors.

5.2.1 Single-node Scheduling

We assume a set S of applications with $|S| = m$. Each of the applications have work $w_1, w_2, ...w_m$. They can be scheduled to run alone or with other applications. Our goal is to compute the schedule that completes all applications' work in the minimal total time. The optimal schedule can be described as a solution to a linear program if the performance of every application was known ahead of time. Since we do not know the exact performance, our algorithm uses ESP to predict the performance and then generate near-optimal schedules.

We assume that at most k-tuples of applications can run together at a time, meaning we must predict the performance for $2\binom{m}{2} + 3\binom{m}{3} + \ldots + k\binom{m}{k}$ application combinations. The intuition behind the restriction on k is that the system will become saturated at some point and it is no

longer beneficial to continue to add applications to a saturated node.

We first develop a linear program for optimal scheduling assuming known performance. We will relax that assumption momentarily. We also assume that pre-emption is allowed. Then, an optimal schedule is given by:

$$\begin{aligned} \mathbf{y} &= \arg\min_{\mathbf{y} \geq 0} \ \|\mathbf{y}\|_1 \\ &\text{subject to} \quad \mathbf{A}\mathbf{y} = \mathbf{w} \end{aligned} \tag{5.8}$$

This equation is quite simple, but the complexity is in the structure of the $\mathbf{y}$ vector and $\mathbf{A}$ matrix. Each element of $\mathbf{y}$ is the time for a specific set of applications to be co-scheduled, while the columns of $\mathbf{A}$ represent the each application's performance when co-scheduled in that set. For example, if $m = 3$, the superset of all possible sets of applications is $\{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{1,3\},\{1,2,3\}\}$. If we order these sets, then $\mathbf{y}_j$ in Equation (5.8) is the time spent when all the applications in set j run together and $\mathbf{A}_{ij}$ is the speed of application i when co-scheduled with applications from set j. This linear program has a sparse solution with at most $2m$ non-zero solutions, hence the context switching cost is not very high and is bounded by the total number of applications.

Equation (5.8) produces a minimal time schedule but is not practical. It requires that all application interference is known *a priori*. Further, it assumes deterministic application performance, but in real systems, performance is stochastic and subject to inherent noise. We therefore extend Equation (5.8) by assuming that the observed performance $\hat{\mathbf{A}}$ is drawn from a Gaussian distribution:

$$\hat{A}_{ij} \sim \mathcal{N}(A_{ij}, \Sigma_{ij}). \tag{5.9}$$

So, instead of using A_{ij}, we sample $\hat{A}_{ij}$ to predict A_{ij} and Σ_{ij}. Suppose $\tilde{\mathbf{A}}$ is our prediction for $\mathbf{A}$, then instead of solving Equation (5.8) we solve:

$$\begin{aligned} \mathbf{y} &= \arg\min_{\mathbf{y} \geq 0} \ \|\mathbf{y}\|_1 \\ &\text{subject to} \quad \tilde{\mathbf{A}}\mathbf{y} = \mathbf{w} \end{aligned} \tag{5.10}$$

This equation is a proxy for Equation (5.9), but it cannot guarantee that the required work is

Algorithm 3 Iterative Scheduling Algorithm

Input: Tolerance: TOL $= 10^{-6}$
Performance prediction matrix from ESP: $\tilde{\mathbf{A}}$
Total work: $\mathbf{w}$
Exploration-exploitation parameter: η ($\eta \geq 1$)

1: Initialize scheduling matrix $\mathbf{y}^{(0)} = 0$, work remaining: $\mathbf{w}_{rem} = \mathbf{w}$, total scheduling time: $s = 0$.
2: **while** $\|\mathbf{w}_{rem}\|_2 \geq$ TOL **do**
3: Work to be processed, $\mathbf{w}_{sent} = \mathbf{w}_{rem}/\eta$.
4: Solve linear program based on the prediction,
$\mathbf{y} = \arg\min_{\mathbf{y} \in \mathscr{C}} \|\mathbf{y}\|_1$,
where $\mathscr{C} = \{\mathbf{y} : \tilde{\mathbf{A}}\mathbf{y} = \mathbf{w}_{sent}, \mathbf{y} \geq 0\}$.
5: Schedule/run applications according to time slices given by $\mathbf{y}$. Update scheduling time as, $s = s + \|A\mathbf{y}\|_1$.
6: Update predicted performance $\tilde{\mathbf{A}}$ based on true performance observed $\hat{\mathbf{A}}$ and update remaining work, $\mathbf{w}_{rem} = (\mathbf{w}_{sent} - \hat{\mathbf{A}}\mathbf{y})_+$.
7: **end while**
8: **return** Total scheduling time: s.

finished because it uses predictions rather than true performance. To deal with this uncertainty we design an iterative algorithm which repeatedly solves the approximate linear program in Equation (5.10) until all work is finished. This approach accounts for inherent noise due to error in both performance measurement and prediction.

Algorithm 3 takes four parameters: an error tolerance (defaulted to 10^{-6}), CALOREE's predicted performance matrix, a vector representing the total work for each application, and a scalar η controlling the trade-off between exploration-exploitation. If we believe that ESP's performance prediction is very good, we can set $\eta = 1$ to run fewer iterations, saving computation. If we have less confidence, then we could send a smaller amount of work to the linear program thereby reducing the error at each iteration and learning more about the particular application mix scheduled that iteration.

This algorithm loops until all work is complete. For each loop iteration, the algorithm takes a *step* to complete some work, with the step size inversely proportional to η. It then solves Equation (5.10) using the predicted performance and schedules the applications according to this solution. After running the schedule for the specified time, the algorithm updates the performance predic-

tion using its latest observations and updates the work vector with the work accomplished in this step. Given perfect knowledge of the application interference and no system noise, the algorithm requires only a single step and would be optimal.

5.2.2 Multi-node Scheduling

We now consider scheduling dynamically arriving applications on multiple processors. Applications arrive in a stream and a centralized scheduler assigns an application to a processor (which may already have applications running). We assign jobs in a first-come-first-serve order; a new job is immediately assigned a processor with the goal of minimizing job completion time.

The multi-node scheduler (see Algorithm 4) takes as input: (1) an error tolerance (again defaulted to 10^{-6}), (2) ESP's performance prediction $\tilde{A}$, (3) a job sequence (we do not look ahead, so in practice the sequence does not need to be known in advance), (4) the number of nodes q, and the exploration-exploitation trade-off η. Each job is denoted as $J_i = (a_i, v_i)$, where a_i is the application index and v_i is that application's work. The algorithm loops until all jobs are completed. Each iteration takes the next job and determines the expected work and time if assigned to each processor (lines 4–7). It chooses the processor that has the fastest predicted completion time (line 8) and schedules that job on the processor using Algorithm 3 (line 9).

5.3 Evaluation

In this section we evaluate CALOREE. We first describe our experimental setup, then evaluate the CALOREE-based schedulers, and finally present statistical analysis on CALOREE prediction accuracy and overhead.

Algorithm 4 Multi-node Iterative Scheduling Algorithm

Input: Tolerance: TOL $= 10^{-6}$
Performance prediction matrix from ESP: $\tilde{\mathbf{A}}$
Jobs arriving in stream: $J_1, J_2, \ldots, J_T$
Number of processors: q.
Exploration-exploitation parameter: η ($\eta \geq 1$)

1: Initialize work matrix: $\mathbf{W} = \mathbf{0}_{m \times q}$
2: **for all** i = 1:T **do**
3: Obtain job $J_i = (a_i, v_i)$. where a_i is the application index and v_i denotes the work for that application.
4: **for all** j = 1:q **do**
5: Expected work, $\mathbf{w}_{tmp} = \mathbf{W}(:, j)$; updated with new job's work $w_{tmp}[i] = w[a_i, j] + v_i$
6: Expected schedule time, $s[j] = \min_{\mathbf{y} \in \mathscr{C}} \|\mathbf{y}\|_1$,
where $\mathscr{C} = \{\mathbf{y} : \tilde{\mathbf{A}}\mathbf{y} = \mathbf{w}_{tmp}, \mathbf{y} \geq 0\}$.
7: **end for**
8: Greedily choose processor $P : P = \arg\min_{i \in [q]} s_i$ with least expected scheduling time.
9: Run Algorithm 3 for processor P with $\mathbf{w}_{tmp}$ amount of Total work, $\tilde{\mathbf{A}}$ as the performance prediction matrix and $\eta = 5$.
10: **end for**
11: **return** Total scheduling time for all processors: **s**.

5.3.1 Experimental Setup

Experimental System and Benchmarks

Our test platform is composed of four dual-socket Ubuntu 14.04 system with SuperMICRO X9DRL-iF motherboards and two Intel Xeon E5-2690 processors. These processors have eight cores, hyper-threading (eight additional virtual cores), and TurboBoost. Each node has 64 GB of RAM.

We use 15 benchmarks from different suites including PARSEC (`blackscholes`, `fluidanimate`, `swaptions`, `x264`) Bienia et al. [2008], Minebench (`Kmeans`, `HOP`, `svmrfe`) Narayanan et al. [2006], Rodinia (`cfd`, `particlefilter`, `vips`, `btree`, `backprop`, `bfs`) Che et al. [2009] and SEEC (`Dijkstra`, `jacobi`) Hoffmann et al. [2011b]. These benchmarks test a range of important 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). We construct multiprocessor workloads by randomly selecting benchmarks from this list. When we need more than 15 benchmarks, we allow duplicates.

We measure performance of all applications as wall-clock execution time. Interference is the slowdown one application experiences when co-scheduled with one or more other applications. We evaluate the schedulers in terms of time to complete scheduled jobs. We evaluate the accuracy of interference predictors in terms of the difference between the predicted and actual slowdown.

Points of Comparison

We compare CALOREE with:

- *Activity Vectors* – Activity vectors maximize resource usage variance Merkel et al. [2010]; *i.e.*, they co-schedule applications with widely differing resource needs. Extensions to activity vectors have made similar ideas suitable for dynamic consolidation in virtualized data centers Tembey et al. [2014] and for scheduling dynamically arriving applications Dey et al. [2013]. We compare against the original activity vector approach for the single-node case and against the extension to dynamically arriving applications in the multi-node case. This approach results in compute-intensive and memory intensive applications being scheduled together. Unlike CALOREE none of these approaches produce quantified slowdown predictions, but instead make decisions to co-schedule or not. We find that biasing this approach to be more sensitive to different resources produces different results. We consider variants biased toward:
  - *Memory (MEM)*: favors co-scheduling applications with different bandwidth needs.
  - *Instructions per cycle (IPC)*: favors co-scheduling applications with differing compute needs.
  - *L3 Request (L3R)*: favors co-scheduling applications with differing L3 cache needs.
- *Random (RND)*: co-schedules applications randomly.
- *Oracle*: represents the best schedule given perfect knowledge of application interference;

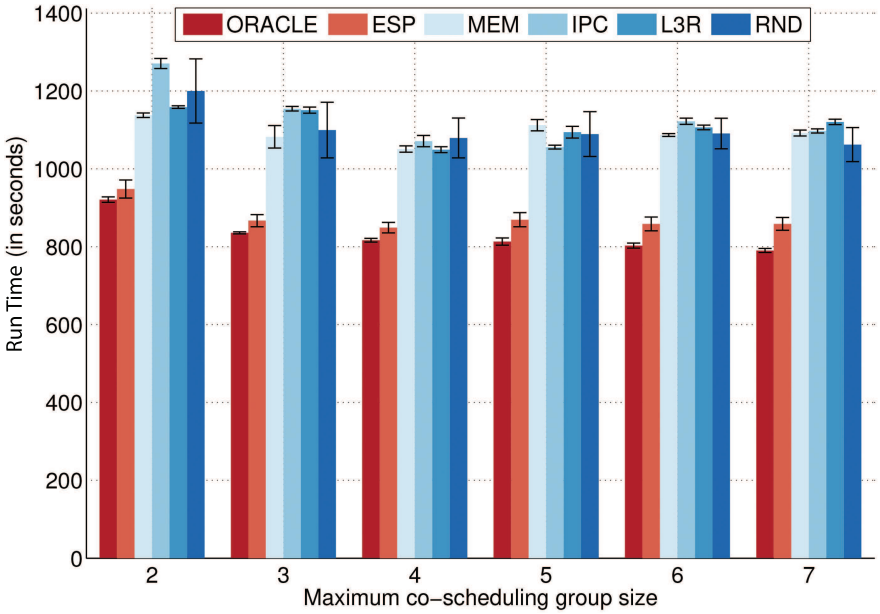


Figure 5.1: Single processor scheduling performance. On an average over different k, CALOREE is 25% better than MEM, 29% better than IPC, 27% better than L3R, 26% better than RND, and only 5% worse than ORACLE.

i.e., it is equivalent to knowing the true performance $\mathbf{A}$ in Equation (5.9). Not implementable in practice, we construct the oracle through a brute force search for all application mixes.

5.3.2 *Single Node Scheduling Results*

To test single-node batch scheduling, we launch all 15 benchmarks listed in Section 5.3.1. The schedulers order application execution to minimize total job completion time. We vary k, the maximum number of applications that can be co-scheduled from two to six. We find that it is never beneficial to schedule more than six applications simultaneously and no scheduler (other than random) ever attempted to do so even when more were allowed. We run 15 separate experiments for each k, where each experiment differs by the split between training and testing data for CALOREE.

Figure 5.1 shows the results. The x-axis is k, the y-axis is the time to complete the work, and there is a bar showing the mean scheduling time for each of our points of comparison with an error

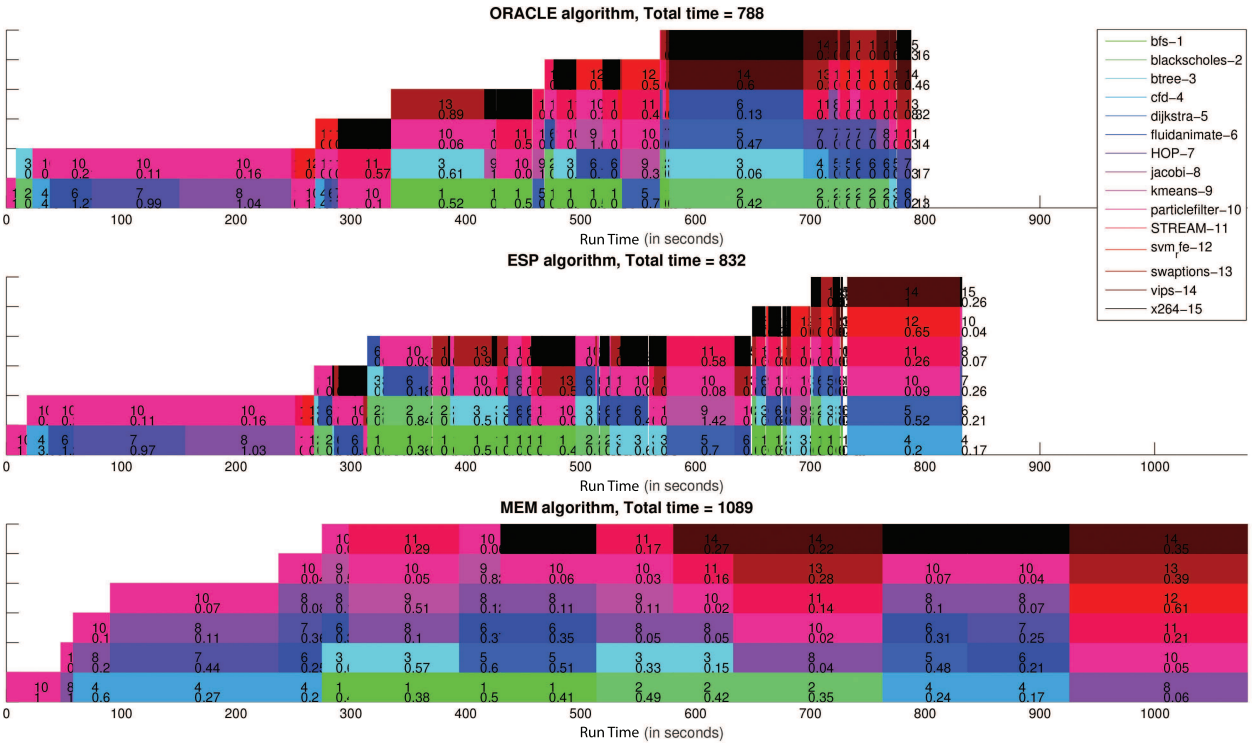


Figure 5.2: Comparison of schedules obtained from different algorithms with up-to 6 co-scheduled applications ($k = 6$) (more compact is better). Each block represents an application running with other applications. The number in the top of the block represents the application index. The number at the bottom represents the slowdown the application experiences. The schedule built using CALOREE algorithm squeezes the applications together better than the MEM baseline.

bar showing one standard deviation. Overall, CALOREE performs much better than the baseline algorithms and only slightly worse than the ORACLE. On average over different k, CALOREE is 25% faster than MEM, 29% faster than IPC, 27% faster than L3R and 26% faster than RND. These results include prediction and scheduling overhead, yet CALOREE is only 5% worse than the ORACLE, which has no overhead and knows the future. We conclude that CALOREE produces highly accurate predictions and yet is efficient enough for practical use.

To provide intuition on the different schedulers, Figure 5.2 illustrates the schedules produced by: ORACLE (top), CALOREE (middle), and MEM (bottom) when they are allowed to co-schedule up to six applications. For each chart, the horizontal axis represents time in seconds. Each colored block represents an application running with other applications. The top number in

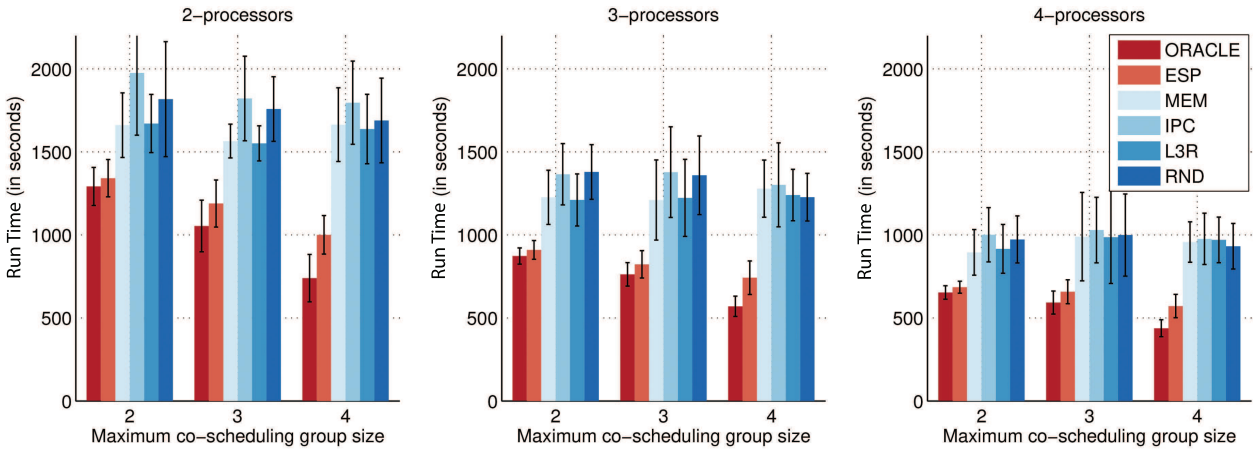


Figure 5.3: Comparison of multi-node scheduling times for stream of 40 applications (lower is better). On an average over different processes and tuples, CALOREE is 47% better than MEM, 61% better than IPC, 47% better than L3R and 54% better than RND.

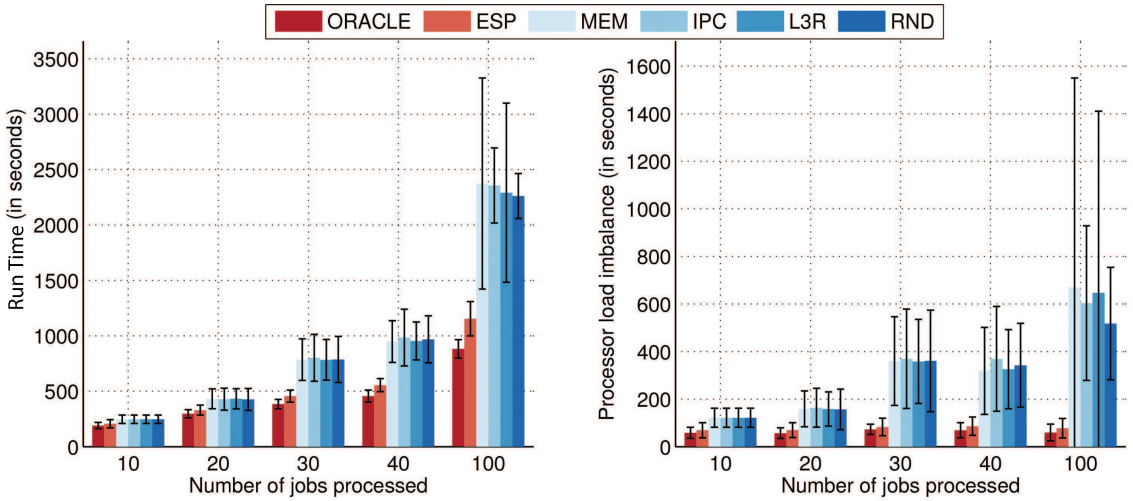


Figure 5.4: Multi-node scheduling performance with varying number of incoming jobs, allowing up to 4 co-scheduled applications per node (lower is better). The left figure shows scheduling time (in seconds) – on average CALOREE is 60% better than MEM, 61% better than IPC, 58% better than L3R and 58% better than RND. The right figure shows the load imbalance (the difference between the highest and lowest scheduling time across nodes). As we increase the number of jobs the load imbalance increases faster for the baseline algorithms and seems relatively constant for CALOREE and the ORACLE.

the block represents the application index (see the legend for mapping from index to name), and the bottom number represents the actual slowdown the application experienced at that time (this actual slowdown is determined after the fact). A better schedule is more compact and completes further

to the left. Clearly, CALOREE's schedule is more compact and closer to the ORACLE than MEM. ORACLE runs the maximum number of applications together less than half of the time. MEM, in contrast, runs the maximum allowed number of applications most of the time. Overall in this particular instance, the scheduling time for the ORACLE is 788 seconds, 832 for CALOREE, and 1089 for MEM.

5.3.3 Multi-node Scheduling Results

In this section we discuss multi-node scheduling performance. We use two, three, and four copies of our base test platform (described above). We assume that applications arrive randomly from our set of 15 benchmarks (so some benchmarks may have multiple instances live in the system). The challenge is to assign an application to a processor as it arrives such that the total impact on the system is minimized; *i.e.*, put the application on the node that minimizes interference.

The oracle computes the best possible schedule, the heuristics use activity vectors to select the best node and the CALOREE-based approach uses Algorithm 4. Figure 5.3 summarizes the results for 2, 3 and 4 processors and for $k = 2,3$ or 4. As we increase the number of processors, the scheduling time improves for all approaches. CALOREE performs almost as well as the ORACLE, performing 5% to 13% worse on an average. In fact, CALOREE is always within 1 standard deviation of ORACLE. CALOREE performs significantly better than the activity vector algorithms. On average over different processes and tuples, CALOREE is 47% better than MEM, 61% better than IPC, 47% better than L3R, and 54% better than RND. Additionally, the standard deviation in performance for CALOREE is much lower compared to the baseline algorithm, leading to much better performance predictability. This predictability is further evidence of our claim that CALOREE allows the schedulers to avoid bad predictions.

5.3.4 *Multi-node Sensitivity to Job Size*

We also study the scheduling performance as a function of the total number of applications scheduled. To be clear, jobs are still scheduled as they arrive (the multi-node scheduler does not reorder applications), we simply have more of them. Specifically, we send up to 100 jobs to a system with 4 nodes and we are allowed to co-schedule up to 4 applications together. For each experiment, we perform 15 trials with different random job arrivals. All results report the mean with error bars indicating one standard deviation.

Figure 5.4 shows how scheduling performance changes as a function of the total number of applications scheduled. The figure consists of two charts: the one on the left showing scheduling time as a function of applications scheduled and the one on the right showing load imbalance in terms of the largest difference in execution time between one processor and another. The number of applications scheduled is shown on the x-axis and the time on the y-axis. As the number of applications increases, CALOREE's relative performance improves compared to the activity vector approaches. The key to this result is CALOREE's accurate quantification of interference. The ability to directly reason about slowdown allows the CALOREE-based approach to rank scheduling decisions and always choose the one with the least impact on the performance of both the running applications and the application that just arrived.

CALOREE's foresight becomes more crucial as the number of jobs increase because bad decisions can create severe load imbalance on the parallel machine, as shown on the right side of Figure 5.4. As we increase the number of jobs the processor load imbalance increases vastly for the activity vector approaches and seems relatively constant for CALOREE as well as ORACLE. On an average CALOREE is 60% better than MEM, 61% better than IPC, 58% better than L3R and 58% better than RND. Again, the standard deviation (shown by the error bars) for CALOREE is much lower than the baselines—leading to much more predictable performance for latency sensitive applications.

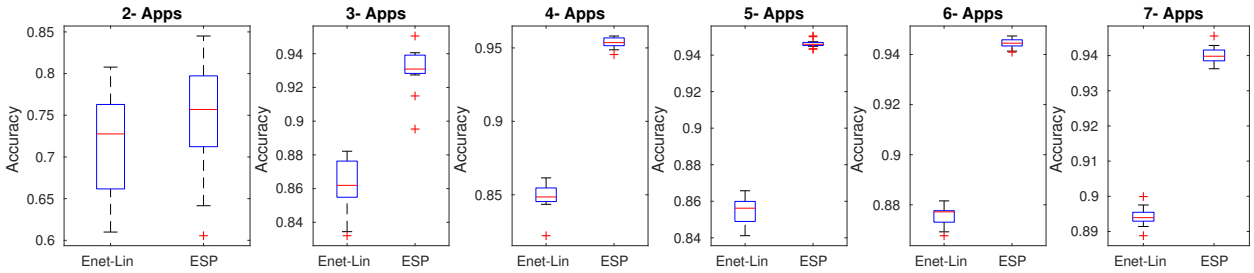


Figure 5.5: Prediction accuracy comparison between Enet-lin and CALOREE.

Table 5.2: Overhead

	k	2	3	4	5	6	7
CALOREE	Model training (in seconds)	0.42	0.59	0.68	0.69	0.48	0.51
	Prediction time (in seconds)	0.00056	0.003	0.0076	0.0198	0.04	0.06
	Training sample size (in %)	70%	40 %	10 %	4%	1 %	1%
Scheduling Algorithm 3	Linear program (in seconds/job)	0.008	0.014	0.023	0.06	0.117	0.22

5.3.5 CALOREE Prediction Accuracy

We compare CALOREE with the *elastic-net* regularization method on the linear model (*Enet-lin*) described in Section 5.1.3. To evaluate quantitatively, we measure the *accuracy* of the predicted performance $\hat{\mathbf{z}}$ with respect to the true data $\mathbf{z}$, by computing the *coefficient of determination* (R^2 value):

$$\text{accuracy}(\hat{\mathbf{z}}, \mathbf{z}) = \left(1 - \frac{\|\hat{\mathbf{w}} - \mathbf{w}\|_2^2}{\|\mathbf{w} - \bar{\mathbf{w}}\|_2^2}\right)_+, \tag{5.11}$$

where $\mathbf{w} = \min(\mathbf{z}, 1)$ and $\hat{\mathbf{w}} = \min(\hat{\mathbf{z}}, 1)$. This metric captures how well the predicted results correlate with the measured results. Unity represents perfect correlation.

Figure 5.5 shows a box-plot for out-of-sample predictive accuracy of CALOREE and Enet-lin when we train both the models with 70% data and test the prediction on the remaining data. For $k = 2$, CALOREE performs only slightly better than the Enet-lin with on average 76% accuracy whereas the baseline is 73% accurate. For $k > 2$, the prediction accuracy for the CALOREE varies from 93% to 96% with very small variance. On the other hand, Enet-lin is only between 85% to 88% accurate.

5.3.6 Overhead

CALOREE has an offline sampling phase followed by an estimation phase. Once we have collected the samples from the batch of co-scheduled applications, we run CALOREE to obtain the predictions for the rest of the combinations of the applications. We have summarized the overhead results in the Table 5.2. We require less than 0.7 seconds to build the performance model for a batch of k-tuples running together. Once the model is built, the prediction time for CALOREE is very small and would range from 0.5 milliseconds to 60 milliseconds. For such small training data, the prediction accuracy for $k = 2$ is around 80% and for $k > 2$ it is around 86%.

The scheduling overhead, again has two components: first obtaining the application's batch run profile which is done using CALOREE, and then solving the linear program to obtain the schedule. The first part (in the top of Table 5.2) is done offline. The vast bulk of online work is done by Equation (5.10), which solves a very sparse optimization problem. The total overhead per scheduled job for Algorithm 3 – the online portion of the approach – ranges from 0.008 seconds for $k = 2$ to 0.22 seconds for $k = 7$. Almost all of this work could be parallelized on a multicore (or accelerated with SIMD instructions) but that is beyond the scope of this work. We note that it is quite practical to consider co-scheduling up to 4 jobs. The overhead of scheduling 7 jobs may become prohibitive, but it is never useful on our test system. This is not an insignificant amount of time but this approach is suitable for long running applications and the benchmarks that we have used in this work are all long running benchmarks with at least 10 seconds of individual runtime.

5.4 Summary

This work explores machine learning methods for predicting application interference in computing systems. Specifically, we explore several state-of-the-art regularization techniques for high-dimensional problems—when many more features are available than samples—and we conclude that existing linear techniques are not accurate enough, while existing higher-order techniques are accurate but slow. Inspired by these observations we present ESP, a combination of linear feature

selection with higher-order model building that achieves the practicality of linear models with the accuracy of higher-order models. We have demonstrated that ESP's quantitative predictions produce significantly better schedules than existing heuristics for both single and multi-node systems, with up to 1.8× improvement in application completion time and significantly lower variance. Additionally, ESP achieves much higher prediction accuracies than prior approaches—over 93% when considering three or more applications. We have made the source code available so that others may improve on or compare with ESP. This work makes the following contributions: 1) ESP, a regularization method for predicting application interference, 2) Demonstration of ESP in both single and multi-node schedulers, 3) Comparison to existing heuristic techniques on real systems, 4) Comparison of ESP's predictive accuracy to other regularization methods, 5) Open-source code release[1]. ESP helps scheduling at multiple levels: it determines which applications should run together on a single node, it determines which node a new application should be scheduled on, and it avoids disastrous decisions that heuristic schedulers cannot. Support for data analytics has become an important research topic in computing systems, and this work explores how data analytics can be used to improve computing systems.

We therefore combine linear and non-linear approaches to produce accurate and practical predictions. ESP's key insight is to split regularization modeling into two parts: *feature selection* and *model building*. ESP uses linear techniques to perform feature selection, but uses quadratic techniques for model building. The result is a highly accurate predictor that is still practical and can be integrated into real application schedulers.

ESP assumes there is a known (possibly very large) set of applications that may be run on the system and some offline measurements have been taken of these individual applications. Specifically, ESP measures low-level hardware features like cache misses and instructions retired during a training phase. At runtime, applications from this set may be launched in any arbitrary combination. The goal is to efficiently predict the interference (*i.e.*, slowdown) of co-scheduled applica-

1. The code is available at https://github.com/ESPAlg/ESP

tions.

CHAPTER 6

FUTURE WORK AND CONCLUSION

6.1 Conclusion

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 Hoffmann et al. [2012], Gentzsch et al. [2005], Dini et al. [2004], Salehie and Tahvildari [2009], Kephart [2005], Laddaga [1999].

While much recent work has built systems to support learning and big data, in this work we use learning and big data to build better systems. Specifically by proposing CALOREE, a combination of machine learning and control for managing resources to meet performance requirements with minimal energy. CALOREE's unique contribution is showing how machine learning and control theory can be combined at runtime to provide more reliable performance and lower energy than either in isolation. Furthermore, this combination is not just practical, it provides formal guarantees

that the system will converge to the desired performance.

This work explores machine learning methods for predicting application interference in computing systems. Specifically, we explore several state-of-the-art regularization techniques for high-dimensional problems—when many more features are available than samples—and we conclude that existing linear techniques are not accurate enough, while existing higher-order techniques are accurate but slow. Inspired by these observations we present ESP, a combination of linear feature selection with higher-order model building that achieves the practicality of linear models with the accuracy of higher-order models. We have demonstrated that ESP's quantitative predictions produce significantly better schedules than existing heuristics for both single and multi-node systems, with up to 1.8× improvement in application completion time and significantly lower variance. Additionally, ESP achieves much higher prediction accuracies than prior approaches—over 93% when considering three or more applications. We have made the source code available so that others may improve on or compare with ESP.

6.2 Future Work

We have proposed methods to control single application under dynamic resource requirement and a method to estimate application interference. An interesting direction for future work is how to control resource allocation for multiple applications on a multi-node system. It is a very difficult problem because of computational blowup with multiple applications across multiple resources and multiple nodes. Another interesting direction is finding more low-level features for better estimation of application interference.

BIBLIOGRAPHY

L.A Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12): 33–37, Dec 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.443.

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

Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 83. IEEE Computer Society Press, 2012.

Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A Kim. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 51. IEEE Computer Society Press, 2012.

Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485974. URL `http://doi.acm.org/10.1145/2485922.2485974`.

Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, pages 153–166. ACM, 2010.

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

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

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

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

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. ISSN 0163-5999.

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

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.

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

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

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.

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

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. ISSN 1556-4665. doi: 10.1145/2382570.2382572. URL `http://doi.acm.org/10.1145/2382570.2382572`.

Yuhao Zhu and Vijay Janapa Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*, 2013.

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.

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

Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *ASPLOS*, 2015.

Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. ISBN 047126637X.

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

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*, 2015a.

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

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.

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.

Henry Hoffmann. Jouleguard: energy guarantees for approximate applications. In *SOSP*, 2015.

Lennart Ljung. *System Identification: Theory for the User*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999. ISBN 0-13-656695-2.

Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated multi-objective control for self-adaptive software design. In *FSE*, 2015.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Trans. on Knowl. and Data Eng.*, 22(10):1345–1359, October 2010. ISSN 1041-4347. doi: 10.1109/TKDE.2009.191. URL `http://dx.doi.org/10.1109/TKDE.2009.191`.

R. M. Bell, Y. Koren, and C. Volinsky. The bellkor 2008 solution to the netflix prize. Technical report, ATandT Labs, 2008.

W.S. Levine. *The control handbook*. CRC Press, 2005.

A Hoerl and R Kennard. Ridge regression, in encyclopedia of statistical sciences, vol. 8, 1988.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. Resense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Trans. Archit. Code Optim.*, 10(4), December 2013.

Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *SOCC*, 2014.

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.

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.

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)*.

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.

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*, 2015b.

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

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.

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.

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

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

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.

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.

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.

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

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

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

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.

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.

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.

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

Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2027-6. doi: 10.1145/2517351.2517354. URL `http://doi.acm.org/10.1145/2517351.2517354`.

Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. Swift: A feedback control and dynamic reconfiguration toolkit. In *2nd USENIX Windows NT Symposium*, 1998.

Raghavendra Pothukuchi, Amin Ansari, Petros Voulgaris, and Josep Torrellas. Using multiple input, multiple output formal control to maximize resource efficiency in architectures. In *ISCA*, 2016.

Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In *Job Scheduling Strategies for Parallel Processing*, pages 103–127. Springer, 2002.

Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Computer Architec-*

ture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on, pages 158–169. IEEE, 2015.

Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. Mantis: Automatic performance prediction for smartphone applications. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, pages 297–308. USENIX Association, 2013.

Chi Xu, Xi Chen, Robert Dick, and Zhuoqing Morley Mao. Cache contention and application performance prediction for multi-core systems. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 76–86. IEEE, 2010.

Xi Chen, Chi Xu, Robert P Dick, and Zhuoqing Morley Mao. Performance and power modeling in a multi-programmed multi-core environment. In *Proceedings of the 47th Design Automation Conference*, pages 813–818. ACM, 2010.

Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *MICRO*, 2015.

Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, Galen M. Shipman, and Chita R. Das. D-factor: A quantitative model of application slow-down in multi-resource shared systems. *SIGMETRICS Perform. Eval. Rev.*, 40(1), June 2012.

Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: Accelerating resource allocation in virtualized environments. *SIGARCH Comput. Archit. News*, 40(1), March 2012.

Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS*, 2014.

Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. Stay-away, protecting sensitive applications from performance interference. In *Middleware*, 2014.

Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

Jianqing Fan and Jinchi Lv. Sure independence screening for ultrahigh dimensional feature space. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(5):849–911, 2008.

Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

Jacob Bien, Jonathan Taylor, and Robert Tibshirani. A lasso for hierarchical interactions. *Annals of statistics*, 41(3):1111, 2013.

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

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

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

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.

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

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

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.

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.

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

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

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.

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

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. ISBN 978-1-4503-2458-8. doi: 10.1145/2525526.2525850. URL `http://doi.acm.org/10.1145/2525526.2525850`.

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.

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.

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. doi: 10.1109/DEXA.2005.173.

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

Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009. ISSN 1556-4665. doi: http://doi.acm.org/10.1145/1516533.1516538.

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

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

Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*, 2016.

Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. Mete: meeting end-to-end qos in multicores through system-wide resource management. In *SIGMETRICS*, 2011.

Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical Report TR 95-041, UNC Chapel Hill, Department of Computer Science.

David H. K. Kim, Connor Imes, and Henry Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *CPSNA*, 2015.

Michel Tokic. Adaptive ε-greedy exploration in reinforcement learning based on value differences. In *KI*. 2010.

John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, pages 19–25, December 1995.

Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *ICS*, 2002.

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.

Thomas Willhalm. Intel performance counter monitor-a better way to measure cpu utilization. *last modified August*, 16, 2012.

Henry Hoffmann, Martina Maggio, Marco D Santambrogio, Alberto Leva, and Anant Agarwal. Seec: a general and extensible framework for self-aware computing. 2011b.