THE UNIVERSITY OF CHICAGO

BALANCING PERFORMANCE AND ENERGY IN COMPUTING SYSTEMS

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

DEPARTMENT OF COMPUTER SCIENCE

BY
CONNOR KELLY IMES

CHICAGO, ILLINOIS
JUNE 2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

There are so many people to thank, I cannot hope to list them all here.

My family... Your unwavering support, both before and during graduate school, made my endeavors possible. Through the ten years and living in five different states since college, time and distance have only heightened my appreciation for everything you have given me. Thank you.

Those who aided my journey getting here... Professor Michael Crowley at the University of Southern California, for a letter of recommendation five years after I graduated. My coworkers from Lockheed Martin, especially Gerald Nostrand for also writing a letter of recommendation.

My advisor, Professor Hank Hoffmann... Thanks for the exceptional mentoring and support, and for sending me *everywhere*, especially Portugal, Hong Kong, Greece, and France. I wish you could have joined in the global adventures, but thanks also for trusting me to "spread the gospel" and defend our work on my own. Now that you have tenure, my work here is done!

Others at the University of Chicago... Professors Andrew Chien and Haryadi Gunawi, for being on my Master's committee. Professor Shan Lu, for being on my doctoral dissertation committee. Nikita Mishra and David Kim, for collaborations and co-authorships. Huazhe (Harper) Zhang and Kevin Zhao, for helping with the grunt work. Ivana Marinčić, for being well-connected and helping me with my job hunt. The department technical and administrative staff, especially Bob Bartlett, Margaret Jaffey, Nita Yack, and Sandy Quarles—nothing would ever get done with you, you are all great!

Those who supported my research outside the university... Lars Bergstrom, for the summer position at Mozilla Research. Steve Hofmeyr, for research collaboration, hosting me at Lawrence Berkeley National Laboratory, and being on my dissertation committee. Our Proteus project collaborators at Rice, UT Austin, and MIT, especially Professors Krishna

Palem and Don Fussell.

Those providing support beyond research... Saeid Barati, for long chats, and for finding volleyball and softball groups where we can unwind, and sometimes even win. Rhys Povey (Physics), for organizing so many social events, barbecues, and our intramural volleyball team. Neville Eclov (Medical Physics), for having us on the Ionizers softball team and hosting classy parties in the off-season. Will Kong, Joe Wingerter, Kavon Farvardin, and Saeid Barati (again), for all the home brewing and other social activities. Roselyne Tchoua, for sharing in first-year struggles—it was nice having another adult to talk with! Gökalp Demirci, for casual drinks, and being the only one to show up for my first-year Thanksgiving potluck. Anne Farrell, for cookies, a bicycle, and impressively silent socialization. All the PSD Happy Hour regulars from the department, for banter over cheap pizza and beverages.

All members of Hank's group, past and present, especially those not yet mentioned but equally important to me—Zhixuan Zhou, Bernard Dickens, and Muhammad Husni Santriaji... Thanks for putting up with the sarcastic old guy in the room. I hope I managed to drop a little wisdom along the way—if you find it, I might need it back.

ABSTRACT

This dissertation addresses challenges in balancing performance and power/energy consumption in computing systems. Systems are often underutilized, meaning applications do not require all of a system's resources in order to achieve desired behavior, *e.g.,* an application performance goal or reasonable system energy consumption. Modern systems expose knobs for tuning resources, like processor frequency or core allocation, which have a quantifiable impact on application performance and system power consumption. The result is a tradeoff space that can be navigated by resource schedulers to achieve desired behavior, sacrificing one dimension in favor of another, *e.g.,* increased performance at the cost of increased power or energy consumption. The optimal knob settings required to achieve desired behavior depend on both the application and system, even changing during the course of execution as applications progress through different processing phases. The challenge in designing general and portable solutions to these problems arises from the diversity in both hardware and software systems.

We first address the problem of meeting application performance goals while minimizing energy consumption with two projects—POET and CoPPer. Both projects use control theory, which provides a formal framework for reasoning about dynamic systems, including convergence guarantees and robustness to model inaccuracies. In contrast, commonly used heuristic techniques cannot provide these guarantees, nor are they always portable. POET is a general solution that is portable between applications and systems—it is independent of different knob types and their allowable settings. POET produces resource schedules to exactly meet performance goals while achieving optimal energy consumption. CoPPer leverages recent power capping technology in place of software-managed Dynamic Voltage and Frequency Scaling (DVFS), which is being deprecated by hardware vendors. CoPPer overcomes challenges presented by the non-linear relationship between performance and power to meet performance goals while leaving the energy optimization to hardware, which responds more rapidly to changes in application resource requirements than software.

Finally, we address the problem of optimizing energy efficiency to minimize the execution cost of running applications with the CEES project. CEES uses machine learning classifiers, driven by low-level hardware performance counters, to predict the most energy-efficient knob settings at runtime based on current application resource utilization. By using performance counters, no application modifications are necessary. We evaluate this approach in the High Performance Computing (HPC) domain, more aggressively trading performance for energy savings than has historically been done, reducing the cost of scientific insight. Extrapolating from empirical single-node performance and power results, scaling the solution to hardware over-provisioned, power-constrained clusters could increase total cluster throughput by up to 24%.

The three projects presented in this dissertation dynamically adapt to changing application and system behavior at runtime, and are thus able to provide more optimal results than commonly used static resource scheduling techniques. Furthermore, the project designs are independent of particular applications and systems, making them portable to a wide range of computing platforms.

CHAPTER 1

INTRODUCTION

1.1 Thesis Statement

This dissertation explores two problems in balancing software performance and system energy consumption in computing systems: (1) meeting application performance goals while minimizing energy consumption, and (2) running applications as energy-efficiently as possible. We address the first problem using control theory to meet performance goals by tuning system knobs such as DVFS frequencies, core allocations, and hardware power caps. We treat the second as a classification problem, using machine learning classifiers, driven by low-level hardware counter metrics, to predict the most energy-efficient system knob settings at runtime.

1.2 Problem Description

Performance and energy consumption are conflicting goals in computing systems and must be appropriately balanced to satisfy user requirements. As power and energy become first-class concerns in systems ranging from low-power embedded platforms to large-scale High Performance Computing (HPC) environments, there is an increasing need for software to manage hardware resource allocations to achieve a desirable balance in performance and power/energy consumption. Fortunately, modern systems expose knobs to support trading performance and power.

For example, Dynamic Voltage and Frequency Scaling (DVFS) trades processor clock rate and power consumption for changes in throughput, and is ubiquitous in modern systems. Dynamic power P is proportional to the effective capacitance C of a processor, the applied voltage V, and the clock frequency f [49]:

$$P \propto \frac{1}{2}CV^2f \tag{1.1}$$

Higher clock frequencies typically result in higher performance, *e.g.,* for compute-bound applications. However, voltage and frequency are correlated, meaning their values in Equation 1.1 cannot be adjusted independently [79]. Higher clock frequencies require higher voltage to maintain stable circuit operation; conversely, lower frequencies can maintain operation at lower voltages. Reducing the voltage and frequency can offer significant power savings due to this non-linear relationship. Newer technologies allow specifying power caps for hardware resources and letting the components optimize their own behavior subject to the power constraint, *e.g.,* Intel's Running Average Power Limit (RAPL) tunes DVFS much faster than software [25]. Another common knob setting is the allocation of compute cores on multicore systems, allowing unallocated cores to enter low-power sleep states, or even shut down altogether to save power/energy.

If we consider each allowable combination of knob values to be a system *setting* or *configuration*, we can explore the impact of different knob combinations on performance and power consumption. Therefore, to understand the behavior of a particular application on a system, we can perform a characterization by testing the application in each system configuration and recording the behavior. The result is a *tradeoff space*, where each system configuration produces unique application performance and system power values. Often only a subset of the available configurations are actually desirable—it is straightforward to derive the Pareto-optimal configurations, or even those that lie on the (upper or lower) convex hull of the tradeoff space.

A significant amount of software is subject to performance requirements. Power or energy can then be optimized under the performance constraint to increase battery life or reduce power costs. Until recently, heuristic approaches were often sufficient to meet performance

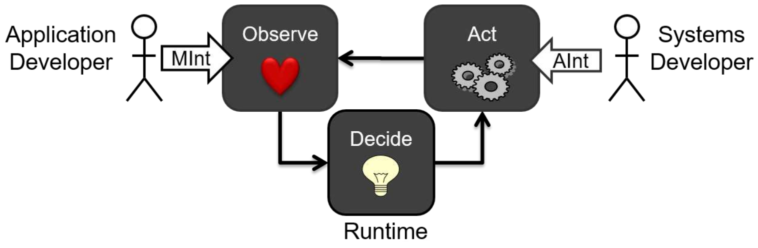


Figure 1.1: A self-aware computing (SEEC) runtime model – observe, decide, and act.

goals while achieving near-minimal energy consumption. However, heuristic approaches to balancing performance and power/energy rely on assumptions about the performance/power tradeoff space of an application and system that we have found are not portable between applications and modern systems. For example, the classic *race-to-idle* heuristic can achieve low energy consumption on one system, but high energy consumption on another compared with a *pace-to-idle* or *never-idle* approach [61]. While performance/power tradeoff spaces were mostly linear on older systems, advances in technology are making them more convex on modern systems. An important observation by Kim et al. is, "Unless the function of a given machine is a straight line originating from the idle state, (0, P_{idle}), an idling heuristic is never optimal for all instances" [73]. In short, the more convex the tradeoff space, the further from optimal any idling approach is.

A more optimal tradeoff in balancing performance and power is to maximize *energy efficiency*, *i.e.,* the amount of work completed per unit of energy consumed (analogous to maximizing the ratio of performance to power). Sometimes it is desirable simply to reduce the energy cost of a computation at the expense of slower execution. In the High Performance Computing (HPC) domain, maximizing energy efficiency can maximize the throughput of power-constrained, hardware over-provisioned clusters [104]. As with meeting performance goals, a similar problem with heuristics arises—the optimal approach varies between applications and systems.

The lack of both portability and optimality of heuristics demand that we develop more general approaches to addressing the problems in balancing performance and power/energy.

In his PhD dissertation, Hoffmann proposes a “self-aware” computing model (SEEC) that uses a closed-loop feedback design to *observe*, *decide*, and *act* [54]. Figure 1.1 demonstrates this concept. The SEEC model is more portable than many heuristics because it includes an observation step that measures behavior at runtime rather than relying strictly on assumptions made offline. We build on this high-level model and use feedback systems that measure application and system behavior during runtime to make informed changes to resource allocations as new information becomes available. Furthermore, our general feedback system designs do not depend on extremely accurate or complete models. Instead, they rely on runtime measurements to fine-tune their decisions.

1.3 Contributions

This dissertation addresses two common goals in managing software performance and system energy consumption. The first is the constrained optimization problem of meeting an application performance goal while minimizing energy consumption. Software performance constraints are common in applications that run on platforms ranging from embedded to server-class. The second addresses the problem of running software as energy-efficiently as possible, *i.e.,* maximizing the ratio of work completed to energy consumed. We address these two problems with three distinct projects:

- POET, the **P**erformance with **O**ptimal **E**nergy **T**oolkit: A portable, control-theoretic framework for meeting soft application performance goals while optimizing energy consumption, which we demonstrate by tuning DVFS settings and core allocations.
- CoPPer, **Co**ntrol **P**erformance with **P**ow**er**: A model-free, control-theoretic approach for meeting soft application performance constraints by tuning processor power caps and allowing the hardware to optimize energy consumption.
- CEES, **C**lassification of **E**nergy-**e**fficient **S**ettings: A machine learning classification framework for predicting energy-efficient system settings based on low-level hardware

performance counter metrics, which we demonstrate by tuning DVFS, socket allocations, and the use of HyperThreads.

Appendix A describes portable tools created to support these projects that we believe are useful to other researchers and developers. Appendix B provides open-source release information for POET, CoPPer, and supporting tools.

1.3.1 POET

POET addresses the problem of meeting soft real-time application performance goals while minimizing energy consumption. A significant amount of software is subject to performance constraints, from applications running on low-power embedded platforms to those in high-performance, high-power environments like on servers in datacenters. Optimizing energy consumption increases the usable runtime of battery-powered systems like tablets, smartphones, and smaller devices we now classify as Internet of Things; for always-on devices, it reduces the runtime energy costs.

Kim et al. prove that an optimal solution to this constrained optimization problem requires, at most, two system configurations from the convex hull of the tradeoff space [73]. POET uses control theory to meet the performance goal and linear optimization to select the optimal pair of system configurations that satisfy the performance constraint. While POET was originally designed for embedded systems [63], we later evaluated it on a server-class system [64].

POET's evaluation uses DVFS and core allocation as the system knobs to control performance and energy consumption. While significant prior work has used these knobs, their solutions tend to be restricted to particular applications and systems, limiting their portability. In contrast, POET's design is independent of any particular application or platform and their performance/power tradeoff spaces.

We find that POET achieves:

- **Ease of Use**: Integrating POET with applications only requires a few additional lines of code.
- **Predictable Performance**: POET meets a range of performance targets, minimizing the error between the goal and the achieved performance.
- **Energy Savings**: Using an offline oracle, we verify that POET achieves near-optimal energy consumption (*e.g.,* 1.3% over optimal on an Intel-powered tablet, 2.9% on an ARM big.LITTLE system), which includes POET's runtime overhead.
- **Adaptability**: POET adapts to phases in application and input behavior, achieving increased energy savings during periods of low computational demand. Additionally, POET adapts to noise in the system introduced by co-scheduled applications to ensure that performance goals are still met when feasible, and makes a best effort otherwise.

1.3.2 CoPPer

CoPPer also addresses the problem of optimizing energy consumption under a soft performance constraint, but does so by tuning hardware power caps. Recent trends suggest that software control of DVFS is being deprecated, making all prior software approaches that depend on DVFS obsolete. Linux kernel developers have acknowledged this trend [108]. In fact, the Linux kernel documentation notes, "the idea that frequency can be set to a single frequency is fictional for Intel Core processors. Even if the scaling driver selects a single P-State, the actual frequency the processor will run at is selected by the processor itself" [33].

However, hardware is not aware of application-level performance requirements, so a software component is still needed to ensure that performance constraints are respected. Fortunately, emerging interfaces let software set *power caps* on hardware, with hardware free to determine what DVFS frequencies should be used and when, so long as the average power over some time window is respected. For example, Intel's Running Average Power Limit

(RAPL) allows software to set power limits on hardware [25]. This poses a new challenge. Meeting performance constraints with DVFS is easy: simple linear models map changes in processor clock frequency to changes in application performance. Meeting performance requirements with power capping is harder: power and speedup have a non-linear relationship (Equation 1.1) and most applications exhibit diminishing performance returns with increasing clock frequency (and thus power), *i.e.,* are not entirely compute-bound.

CoPPer proposes to leverage hardware power capping to control performance, leaving the energy optimization to the hardware. Its evaluation uses Intel RAPL, which manages DVFS in hardware at finer-grained intervals than software can, while strictly respecting the imposed power limit. CoPPer makes the following contributions:

- Demonstrates the need for a DVFS alternative that allows software to manage performance/power tradeoffs.
- Proposes software-defined power capping as a replacement for software-managed DVFS.
- Presents CoPPer, a feedback controller that meets performance goals by manipulating hardware power caps, handles non-linearity in power cap/performance tradeoffs, and introduces adaptive *gain limits* to further reduce power when it does not increase performance.
- Evaluates CoPPer using Intel RAPL on a dual-socket, 32-core server. We find that CoPPer achieves performance guarantees similar to software DVFS control, but with better energy efficiency. Specifically, CoPPer improves energy efficiency by 6% on average with a 12% improvement for memory-bound applications. At the highest performance targets, CoPPer's gain limit saves even more energy: 8% on average and 18% for memory-bound applications.

1.3.3 CEES

In other scenarios, it is desirable to maximize energy efficiency, *i.e.,* complete a fixed-size computation using the least energy possible without a performance constraint. It is becoming well-established that the *race-to-idle* heuristic is not energy-efficient. It may then seem reasonable to simply run in the lowest-power system configuration, but this approach fails to account for elapsed time, so energy consumption may still exceed optimal even when consuming low power. Identifying the most energy-efficient system setting is challenging since the optimal setting varies depending on varying application resource demands.

We address this problem in the High Performance Computing (HPC) domain. It has historically been desirable in HPC to run software as fast as possible. The motivation behind this approach is either to: (1) get an answer to a scientific question as fast as possible, or (2) for the HPC cluster to complete as much work as possible, *i.e.,* maximize the cluster throughput. The goal of completing an application as fast as possible leaves little room for power/energy optimization, although some work has been done in this area (see Chapter 2). Until recently, maximizing application performance also solved the latter problem. Now we are moving toward the era of exascale systems with strict power budgets [74]. Hardware over-provisioning, which allows more systems to operate in a cluster than can actually be supported if each were consuming their maximum power budget, has been proposed as an approach to improve cluster throughput [104]. It works because systems do not usually require their maximum allowable power budget. In these new hardware over-provisioned, power-constrained clusters, maximizing energy efficiency instead of application performance will both decrease the cost of per-application scientific insight and maximize the throughput of the cluster.

Our solution is to treat this challenge as a classification problem, using samples from low-level hardware performance counters to drive machine learning algorithms that predict the most energy-efficient configuration to use. This approach does not require the software to provide its own instrumentation, nor does the classifier need to know anything about the application in advance. Using classification instead of estimation reduces the computational overhead of the software solution, at the cost of less insight into the resource predictor's decision. We tune DVFS, socket allocation, and the use of HyperThreads.

This work makes the following contributions:

- Proposes optimizing *energy efficiency* instead of *runtime* to decrease the cost of scientific computation.
- Establishes the problem complexity—of 15 different machine learning classification techniques evaluated, only some are suitable for optimizing energy efficiency.
- Demonstrates that sufficiently powerful classifiers can dramatically reduce energy consumption by accurately predicting energy-efficient system settings at runtime.
- Extrapolates from empirical results to posit that optimizing energy efficiency can improve the throughput of hardware over-provisioned, power-constrained systems by up to 24%.

CHAPTER 2
RELATED WORK

This chapter discusses work related to performance, power, and energy awareness. We first discuss work relating to POET and CoPPer—meeting performance constraints and optimizing energy consumption. We then discuss power and energy-aware work in the High Performance Computing domain, where trading performance for power/energy savings is less common. Finally, we describe research projects by other researchers that build off work in this dissertation.

2.1 Timing Constraints and Energy Awareness

Computing systems are often underutilized, leading to significant portions of time where application performance requirements can be met using less than the full system capacity [9, 95]. This trend has led to flourishing scheduling research that tailors resource usage to meet performance requirements, often while optimizing another dimension like power or energy consumption. Modern systems expose a variety of configurable resources, which energy-aware schedulers can tune to adjust performance and power behavior. This flexibility allows the system to adapt to different circumstances or different application needs, but also increases software complexity. The problem is exacerbated when approaches must achieve portability across a range of different systems, which expose different resources to software and exhibit diverse performance/power tradeoffs.

Software-based DVFS management is essential for many energy-aware scheduling algorithms [4, 75, 137]. At the processor level, DVFS has been used to meet performance requirements [88, 134] and implement power capping [81]. Allowing DVFS to be set separately on different cores provides further benefits [69, 113]. Similarly, managing DRAM DVFS increases energy efficiency [29, 31]. Recent survey papers devote entire sections to the various ways DVFS has been used in scheduling systems [98, 142]. In modern multi-core systems, allocating a subset of cores and sockets (with aggressive clock-gating to save power) is also common, and more recently comes with the additional challenge of scheduling for heterogeneous architectures [107].

One simple heuristic for minimizing energy is *race-to-idle*, which allocates all resources until a job completes and then idles the system until the next job arrives [9]. This heuristic is portable since it does not require knowledge about the system, but empirical studies show that it is not optimal [9, 22, 136, 138]. Kim et al. demonstrate that an optimal solution requires knowledge of how the different configurable resources in a system affect the specific application under control—information which *race-to-idle* does not use [73]. The same study shows that *race-to-idle* is dominated by a *pace-to-idle* heuristic, *i.e., pace-to-idle* is theoretically never worse than *race-to-idle* and can be much better.

It is not surprising that a number of different frameworks have arisen for intelligently controlling multiple resources to minimize energy. Many empirical studies have shown that it is more energy-efficient to coordinate multiple resources than to manage any one alone [9, 22, 136, 138]. Examples include systems that coordinate DVFS with core usage [23, 91, 114], coordination of processor and DRAM DVFS [21, 28, 38, 84], and DVFS with thread scheduling [113, 133]. Several other approaches coordinate processor-wide DVFS with adaptations to the memory system and processor pipeline [15, 35, 120]. For example, Dubach et al. coordinate several microarchitectural features [35], Petrucci et al. coordinate thread scheduling and the use of heterogeneous cores [107], while Maggio et al. coordinate core allocation and clock speed [91]. Bertini et al. coordinate tiers of a multi-tier webserver for e-commerce [13]. AbouGhazaleh et al. coordinate the speed of the processor and cache [1], while Yun et al. also coordinate the speeds of multiple on-chip components [138]. Liu et al. coordinate job scheduling and clock speed on clusters [87]. Bitirgen et al. coordinate clock speed, cache, and memory bandwidth in a multicore [15]. The METE system manages clock speed, memory bandwidth, and core usage [120]. Sinangil et al. co-design a processor architecture which exposes both monitoring and configurable resources with an operating system that dynamically manages those resources [122]. All of these approaches coordinate multiple resources, but do so using system-specific implementations, *e.g.,* METE's controller would have to be redesigned to work with Bitirgen et al.'s architecture.

Other frameworks have been proposed to meet real-time constraints by managing multiple resources. These approaches are typically implemented as middleware that take a specification of available resources and a performance goal, and then meet that goal. Rajkumar et al. propose a general framework (with system-specific implementation) for allocating resources to achieve real-time requirements, but this approach is not energy-aware [112]. Sojka et al. propose a portable middleware layer for allocating resources to meet soft real-time constraints, but this system does not minimize energy [123]. ControlWare is another middleware approach that uses control theory to meet quality-of-service constraints, but does not address energy concerns [140].

These approaches provide portable real-time guarantees, which is itself a hard problem, but they do not provide energy savings. LEO is a machine learning system that can meet performance constraints with minimal energy consumption [97]. LEO is very accurate and provides high energy savings, even with no prior knowledge of the application currently running. Its approach is extremely portable, but also incurs very high overhead. Interestingly, LEO and POET have complementary weaknesses—POET has low runtime overhead, but requires prior knowledge in the form of a configuration model while LEO has high overhead, but requires no prior knowledge. Follow-up work to LEO addressed this issue with CALOREE (discussed further in Chapter 2.3) [96]. PTRADE also uses control theory to manage general collections of resources [58], and is perhaps the most similar to POET. PTRADE minimizes power consumption, but not necessarily energy. In addition, PTRADE uses heuristic optimizations, which are not portable, while POET uses a true minimal-energy scheduling algorithm.

We also note two approaches that are complementary to POET. Zhao et al. manage processor speed to meet both reliability and timing constraints with minimal energy [141]. This problem is complicated by the effect DVFS scaling has on hardware reliability. It is possible that POET's general approach to resources other than DVFS might allow additional energy savings if incorporated in Zhao et al.'s work. He et al. propose adaptive energy management in the power circuitry itself to meet timing constraints while adapting the delivered energy to increase battery efficiency [47]. It is possible that combining POET's runtime-level resource management with this supply-level approach would further increase efficiency.

In the time since POET and CoPPer were developed, more recent works have focused on making it easier to create and use self-adaptive systems. Filieri et al. discuss a general approach to the controller design process [41] while Maggio et al. describe automatic methods for generating multiple-input multiple-output (MIMO) controllers [92]. Recent works leverage MIMO controllers to efficiently use system resources [109, 111] and could even be combined with approaches like CoPPer. Furthermore, software-level knobs (instead of system-level) are becoming more common. While the concept is not new, it still seems to be in its infancy—Shevtsov et al. perform a systematic literature review on software self-adaptation in order to outline current challenges and begin to generalize approaches [121]. One such general approach is SmartConf, which automatically adjusts software knobs to meet performance-sensitive application configurations and provide other optimizations [130].

2.2 High Performance Computing

Power and energy in HPC systems is a growing concern, though prior work in the area has often not allowed trading performance for power or energy savings. For example, Adagio uses DVFS to save energy with less than 1% increase in runtime [116]. Other work depends on accurate prediction of a code's critical path to reduce power where it will not slow down an application [70, 93]. Patki et al. propose to better utilize available power with hardware over-provisioning to increase total system throughput [104]. Sarood et al. have shown similar results: hardware over-provisioning increases performance given a power cap [118]. Hardware over-provisioning acknowledges that compute resources are no longer the primary factor limiting cluster size—power is—allowing us to more aggressively trade performance and power/energy consumption.

Other works demonstrate that low-level hardware performance counters can drive solutions for modeling and improving power/energy consumption [85, 119, 128, 135]. Using Dynamic Concurrency Throttling and DVFS to reduce energy consumption without performance loss, Curtis-Maury et al. use hardware events to create an energy-aware logistic regression model for predicting performance and power [24].

As the number of system settings increases and their interaction becomes more complicated, several approaches have turned to machine learning to manage them. Paragon [26] and Quasar [27] guarantee quality-of-service constraints in heterogeneous data centers using a scheduler based on the learning system that won the Netflix prize [10]. LEO (mentioned previously) uses a Hierarchical Bayesian Model to minimize energy for different system utilization requirements [97]. These approaches all estimate the performance and power of every possible system configuration, then search those estimates to find the best configuration that meets their operating constraints. The need to estimate every configuration's behavior is expensive, requiring half a second [97] to several seconds [26] of overhead. In contrast, our proposed classification approach simply returns the best system settings without predicting their actual energy efficiency, which requires orders of magnitude less overhead. Ferroni et al. use classification based on hardware events to select the best power model for an application from a predetermined set, and then assign resources to that application in a multi-tenant virtualized infrastructure [39]. This approach and our proposed approach are complementary, in that Ferroni et al. assign nodes as resources, while our proposed approach can fine-tune resource usage within a node.

Our approach for maximizing energy efficiency is most closely related to other node-level approaches for managing performance and energy. PUPiL maximizes node performance given a power cap by adjusting system settings to the particular needs of an application [139]. Chasapis et al. maximize performance for power-capped NUMA nodes by recognizing the effect that manufacturing variability can have on individual core performance [20]. Both of these approaches maximize performance for a given power constraint, but neither is capable of minimizing energy, which requires changing both power and performance. ParallelismDial is a node-level approach for managing application-level parallelism to increase energy efficiency [124]. ParallelismDial has similar goals to our proposed approach, but they are complementary—it works at the application level, while our approach operates at the system level. In future work, it would likely be beneficial to combine the two to further reduce overhead and improve energy savings.

2.3 Inspired Projects

Research in this dissertation, and POET in particular, has been used as the foundation for projects by other researchers.

Farrell and Hoffmann build on POET to create MEANTIME, which provides hard real-time guarantees instead of soft performance guarantees [36]. They use the concept of approximate computing to tune application accuracy when their runtime determines that a performance constraint may not be met. The approach usually only requires small changes to application accuracy, *e.g.,* peak signal-to-noise ratio and the bitrate of a video encoder, while still tuning system-level knobs, like in POET, to save energy.

Aforementioned work by Mishra et al. combines POET with machine learning in a project called CALOREE, which reduces the amount of offline work needed to generate the controller's resource specification [96]. CALOREE uses transfer learning, taking information from both previously seen applications and the current application, to model how interacting resources affect the current application's speedup. It then makes online updates to the controller's resource specification and pole value using more accurate models.

POET and follow-on work called Bard were also the starting point in an ongoing project called Proteus. Proteus is developing the FAST programming language which allows programmers to provide *intent specifications* that instruct the runtime about desired constraints and optimizations. FAST satisfies these intents by tuning both application and system knobs, *e.g.,* to manage application performance, application accuracy, and system power or energy consumption.

CHAPTER 3
POET: THE PERFORMANCE WITH OPTIMAL ENERGY TOOLKIT

This chapter describes POET (the **P**erformance with **O**ptimal **E**nergy **T**oolkit). Chapter 2 discussed existing work in managing performance and power/energy awareness. Like POET, a number of these approaches use feedback control to manage timing constraints [13, 43, 55, 58, 82, 84, 90, 91, 129, 140]. POET is most related to prior approaches that abstract resource management into a middleware or runtime [58, 112, 123, 140]. POET uses control techniques, which provide a formal framework for reasoning about the dynamic behavior of systems. POET is unique in its design for portability, its energy awareness, and the incorporation of a true minimal-energy resource allocation algorithm.

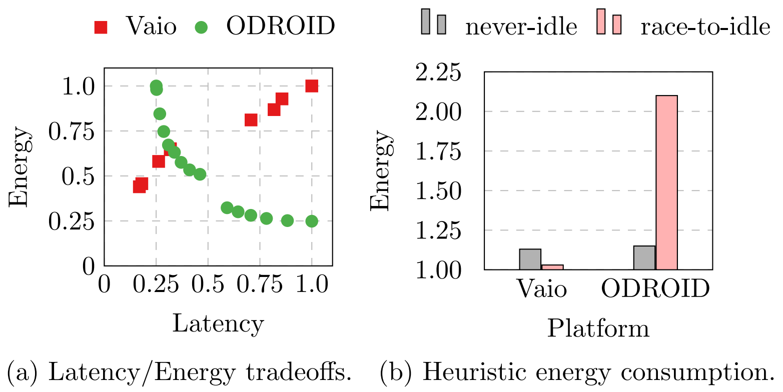
3.1 Motivation

To motivate the need for POET, we summarize prior work [61]. We evaluate timing and energy tradeoffs on two embedded platforms: a Sony Vaio SVT11226CXB tablet system with an Intel Haswell processor and an ODROID-XU+E ARM big.LITTLE development board. The two platforms have: (1) different configurable resources for performance/power management, and (2) timing/energy tradeoffs with different topologies. Resource allocation strategies that save energy on one are wasteful on the other.

Table 3.1 presents each system's configurable resources. The Vaio allows configuring the number of active cores, the use of HyperThreads, processor clock speed, and the use of TurboBoost. The ODROID supports configuring the number of active cores, whether the application uses the "big" (Cortex-A15 high-performance, high-power) or "LITTLE" (Cortex-A7 low-performance, low-power) cores, and the independent clock speeds of the big and LITTLE clusters.

Table 3.1: Two embedded platforms with different configurable components.

Platform	Processor	Cores	Core Types	Speeds (GHz)	TurboBoost	HyperThreads	Num. Configs
SVT11226CXB	Intel Haswell	2	1	0.6–1.5	yes	yes	46
ODROID-XU+E	ARM big.LITTLE	8	2 (A15, A7)	0.8–1.6, 0.5–1.2	no	no	70



(a) Latency/Energy tradeoffs. (b) Heuristic energy consumption.

Figure 3.1: Timing and energy behavior for encoding video on the Vaio and ODROID.

Our example features a video encoder, composed of jobs, where each job encodes a frame. We instrument the encoder to report the latency and platform energy consumption for each job. Figure 3.1a shows the tradeoffs between job latency and system energy consumption for each platform, where each point represents the average behavior of a different configuration. The x-axis shows latency, normalized to 1—the empirically determined worst case. The y-axis shows energy, normalized to 1—the highest measured energy. The tradeoffs are obviously very different for the Vaio and the ODROID. For the Vaio, energy increases as frame latency increases; *i.e.,* a slower job wastes energy. For the ODROID, energy decreases as frame latency increases; *i.e.,* slower encodings save energy.

The different shapes of these tradeoff spaces lead to different optimal resource allocation strategies. Empirical studies show that the *race-to-idle* heuristic, which makes all resources available and then idles after completing a job, is near-optimal on systems like the Vaio [9, 52, 53, 61, 99]. On systems like the ODROID, recent approaches save energy by keeping the system constantly busy, like the *never-idle* heuristic [19, 53, 61, 80, 86].

To demonstrate the importance of choosing the right strategy, we analyze the two heuristics on both platforms and compare their energy consumption to optimal (found by measuring every possible resource configuration). We set a latency target equal to twice the minimum latency and measure the energy consumption of encoding 500 video frames using each heuristic. Figure 3.1b shows the results, normalized to optimal. Both heuristics meet the latency target, but their energy consumptions vary tremendously. On the Vaio, *race-to-idle* is near-optimal, but *never-idle* consumes 13% more energy. Conversely, *never-idle* is near-optimal for the ODROID, but *race-to-idle* consumes 2× more energy.

These results demonstrate that resource allocation strategy greatly affects energy consumption, and more importantly, that heuristic solutions are not portable across systems. These two points motivate the need for an approach like POET, which provides near-optimal resource allocation while remaining platform-independent.

3.2 General and Portable Resource Allocation

The goal of POET's resource allocation framework is twofold. First, it must provide predictable timing so application jobs meet their deadlines. Second, it should minimize energy consumption given the timing requirement. While these two subproblems are intrinsically connected, they can be decoupled to provide a general solution. The complexity arises from the need to keep resource allocation general with respect to the running application and the hardware platform. We tackle the problem of providing predictable timing using control theory by computing a *generic control signal*. Using the computed control signal, we then solve the energy minimization problem using mathematical optimization.

Figure 3.2 illustrates our approach. The **application** informs the runtime of its job *performance goal*. Measuring each job start and completion time (*performance feedback*), POET's runtime computes the *performance error* and passes it to the **controller**. The controller uses the error to calculate a *generic control signal*, indicating how much the application speed should be altered. This signal is used by the **optimizer**, together with the

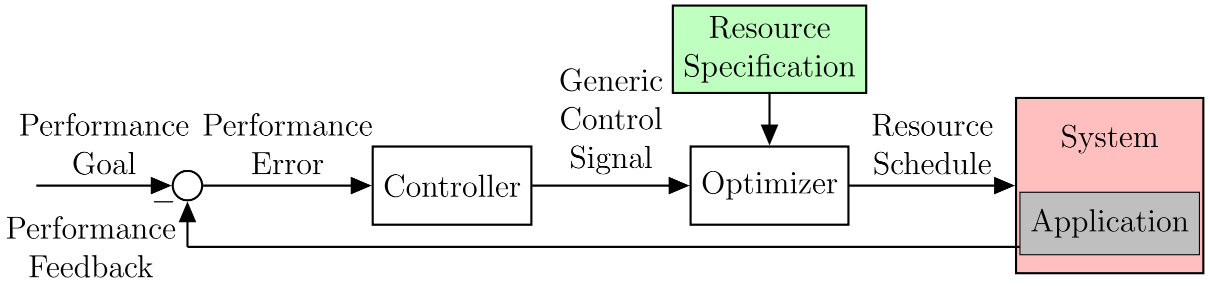


Figure 3.2: Overview of the POET runtime.

resource specification, to produce a *resource schedule* that achieves the desired performance goal while minimizing energy consumption. Both the controller and the optimizer are designed independently of any particular application and system. The only assumption made is that applications are composed of repeated jobs with a soft real-time performance goal.

3.2.1 Controller

The application provides a *performance goal* (work rate) R_{ref}, which is easily computed from a workload size (number of jobs) ω and desired latency goal (deadline) τ for those ω jobs:

$$R_{ref} = \frac{\omega}{\tau} \tag{3.1}$$

The controller cancels the error between the desired performance, R_{ref}, and the measured performance, $r_m(t)$, which it models as:

$$r_m(t) = s(t-1) \cdot b_r(t-1) \tag{3.2}$$

The error $e_r(t)$ is then easily computed as:

$$e_r(t) = R_{ref} - r_m(t) \tag{3.3}$$

The controller performs its calculations at discrete work (job) intervals to produce a new desired speedup, $s(t)$, implementing the *integral control law* [48]:

$$s(t) = s(t-1) + (1-\alpha) \cdot \frac{e_r(t)}{b_r(t)} \tag{3.4}$$

where α is a *pole* of the closed loop characteristic equation [40] such that α lies within the unit circle:

$$0 \leq \alpha < 1 \tag{3.5}$$

The pole is configurable. Small α values make the controller highly reactive, while large values make it slow to respond to external changes. However, a large α ensures robustness with respect to transient fluctuations and may be beneficial for very noisy systems. A small α will cause the controller to react quickly, potentially producing a noisy control signal.

The parameter $b_r(t)$ represents the application's base speed, which directly influences the controller. Different applications will have different base speeds. Applications may also experience *phases*, where base speed changes over time. To accommodate these situations, POET continually estimates base speed using a Kalman filter [131], which adapts $b_r(t)$ of Equation 3.4 to the current application behavior. Assuming minimal measurement variance (*i.e.,* even if an application is noisy, the signaling framework does not add additional noise) and denoting the application timing variance as $q_b(t)$, the Kalman filter formulation is standard:

$$\begin{cases} \hat{b}^-(t) &= \hat{b}(t-1) \\ e_b^-(t) &= e_b(t-1) + q_b(t) \\ k_b(t) &= \frac{e_b^-(t) \cdot s(t)}{[s(t)]^2 \cdot e_b^-(t)} \\ \hat{b}(t) &= \hat{b}^-(t) + k_b(t) \left[r_m(t) - s(t) \cdot \hat{b}^-(t) \right] \\ e_b(t) &= [1 - k_b(t) \cdot s(t-1)] \, e_b^-(t) \end{cases} \tag{3.6}$$

This formulates Kalman gain for job latency as $k_b(t)$, the *a priori* and *a posteriori* estimates of the base speed as $\hat{b}^{-}(t)$ and $\hat{b}(t)$, and the *a priori* and *a posteriori* estimates of the error variance as $e_b^{-}(t)$ and $e_b(t)$. The Kalman filter produces a statistically optimal estimate of the system's parameters and is provably exponentially convergent [18].

Unlike prior work, the POET controller does not reason about a particular set of resources, but computes a generic control signal $s(t)$. The advantage of using the Kalman filter is that POET's formulation is independent of particular applications and systems. POET provides formal guarantees about its steady-state convergence and robustness without requiring users to understand control theory or Kalman filtering—$s(t)$ is computed by the controller, $r_m(t)$ and $q_b(t)$ are measured, and all other parameters are derived.

3.2.2 Optimizer

The optimizer turns the generic control signal computed by the controller into a system-specific resource allocation strategy, translating the speedup $s(t)$ computed with Equation 3.4 into a *schedule* for the available resources. The *schedule* is computed for the next ω work units (jobs).

As shown in Figure 3.2, the optimizer takes, as input, a resource specification containing the set of available system configurations. There are C possible configurations in the system and by convention, we number the configurations from 0 to $C-1$. We use $c = 0$ to indicate the configuration where the least amount of resources is given to the application. In contrast, configuration $C-1$ maximizes the resource availability. Each configuration c is associated with performance and power values, speedup s_c and powerup p_c respectively, which are normalized to $c = 0$.

Given this information, POET solves the following optimization problem:

$$minimize \quad \sum_{c=0}^{C-1} \tau_c \cdot p_c \tag{3.7}$$

$$s.t. \quad \sum_{c=0}^{C-1} \tau_c \cdot s_c \cdot b_r(t) = \omega \tag{3.8}$$

$$\sum_{c=0}^{C-1} \tau_c = \tau \tag{3.9}$$

$$0 \leq \tau_c \leq \tau, \qquad \forall c \in \{0, \ldots, C-1\} \tag{3.10}$$

Equation 3.7 minimizes the total energy consumption. Equation 3.8 constrains all jobs to complete in the next control period. Equation 3.9 ensures that the time is fully scheduled and Equation 3.10 imposes that a non-negative time is assigned to each configuration. Solving linear optimization problems is, in general, hard, but this particular problem has a structure that makes it practical to solve. Feasible solutions are confined to a polytope in the positive quadrant defined by the two constraints Equations 3.8 and 3.9. Thus, linear programming theory states an optimal solution exists for this problem when all the τ_c are equal to zero except for (at most) two configurations [16].

Algorithm 1 takes the set of configurations C, the controller's speedup $s(t)$, and the number of work units ω in a control period. It then divides the configurations in two distinct subsets. The first subset contains all configurations with a speedup less than or equal to the target. The second contains the remaining configurations, *i.e.,* those with speedups greater than required. Subsequently, Algorithm 1 loops over all feasible pairs of configurations, with one from each subset, to determine how much time should be spent in each configuration given speedup constraint. If the energy of the pair is lower than any previous energy, the algorithm stores the current best pair, its energy, and its schedule. When the algorithm terminates, its output is the pair of chosen configurations and their assigned times. The algorithm tests all possible pairs from the two subsets, each of which contains at most C

Algorithm 1 Finding a Minimal-Energy Schedule.

```
Require: $C$                                      ▷ system configurations, given by user
Require: $\omega$                                 ▷ discrete work units, given by application
Require: $s(t)$                                   ▷ speedup, given by Equation 3.4
  $U = \{c \mid s_c \leq s(t)\}$
  $O = \{c \mid s_c > s(t)\}$
  $candidates = U \times O = \{\langle u, o\rangle \mid u \in U, o \in O\}$
  $energy = \infty$
  $optimal = \langle -1, -1\rangle$
  $schedule = \langle -1, -1\rangle$

  for $\langle u, o\rangle \in candidates$ do     ▷ loop over all pairs
      $\omega_u = \omega \cdot \frac{s_u \cdot (s_o - s(t))}{s(t) \cdot (s_o - s_u)}$     ▷ compute the work units to spend in each configuration in pair
      $\omega_o = \omega - \omega_u$
      $newEnergy = \frac{\omega_u}{s_u} \cdot p_u + \frac{\omega_o}{s_o} \cdot p_o$     ▷ compute energy of this pair
      if $newEnergy < energy$ then                ▷ compare energy to best found so far
          $energy = newEnergy$
          $optimal = \langle u, o\rangle$
          $schedule = \langle \omega_u, \omega_o\rangle$
      end if
  end for

  return $optimal$                                ▷ pair of configurations with minimal energy
  return $schedule$                               ▷ work units to spend in each configuration
```

elements, so an upper bound to the algorithm complexity is $O(C^2)$. We know that there is an optimal solution to the linear program with at most two non-zero τ_c (as the dual problem has two dimensions [16]). Therefore, Algorithm 1 will find a minimal-energy schedule.

3.2.3 Generality and Robustness

The controller and the optimizer both reason about speedup instead of absolute performance or latency. The absolute performance of the application, measured by the average latency of its jobs, will vary as a function of the application itself and the platform it executes on. However, speedup is a general concept and can be applied to any application and system, providing a more general metric for control. Moreover, the controller customizes the behavior of a specific application using the estimate of its base speed produced by the Kalman filter The optimizer operates in a platform-independent manner, using the available configurations provided as input to find the optimal solution, without relying on a particular heuristic that may be system-specific or application-dependent. Finally, the customizable pole α in Equation 3.4 allows for flexibility and robustness to inaccuracies and noise.

The ability to control robustness to inaccuracies and model errors is a major advantage of feedback control systems [40]. In particular, POET is stable and converges to the desired latency without oscillations provided that $0 \leq \alpha < 1$. Formal analysis of this behavior can be obtained by applying standard control techniques—see the original POET publication for further details [63].

In addition to provable convergence, the control formulation allows us to analyze POET's robustness to user error. In particular, suppose Δ is a multiplicative error term, indicating the largest error in the speedup values provided in the system configurations. That is, if the provided speedup is s_p, the real value is $s_p \cdot \Delta$. POET cancels the error despite inaccurate information if and only if $0 < \Delta < \frac{2}{1-\alpha}$. The value of α therefore determines how robust POET is to errors in speedup specifications. For example, when $\alpha = 0.1$, s_p can be off by a factor of 2 and the system is still guaranteed to converge. Users who can provide good system models will therefore use a small α, while less confident users can select a larger α. All the experiments in our evaluation use $\alpha = 0$ to test our implementation in the least forgiving setting. A detailed analysis of POET's robustness is presented in the original POET publication [63].

3.3 Implementation

We now describe how the POET framework is implemented as a C library. We specify the information that must be provided by POET's users, describe the library interface, and then discuss the implementation of the runtime engine.

```
#id        speedup        powerup
0          1              1
1          1.20           1.09
2          1.40           1.16
3          1.60           1.30
4          2.12           1.35
5          2.53           1.50
6          2.88           1.64
7          3.18           1.69
```

Listing 3.1: System-agnostic.

```
#id        frequency        cores
0          250000           0
1          300000           0
2          350000           0
3          400000           0
4          250000           1
5          300000           1
6          350000           1
7          250000           2
```

Listing 3.2: System-specific.

Figure 3.3: Snippets of POET configuration files.

3.3.1 External Inputs

POET requires three pieces of information from users. First, it needs the **resource specification**—system configurations and their associated speedup and powerup (timing and power) characteristics. Second, it needs *performance feedback*—timing and power measurements during runtime. Third, it requires the application to specify its *performance goal*.

The first input is the specification of available system configurations. POET separates the system configurations into two data structures. The first is system-agnostic and contains a configuration identifier along with **speedup** and **powerup** values. The second is system-specific and can take any form a developer considers appropriate to define a system configuration. POET includes a default format for this second structure which contains a configuration identifier, the DVFS frequency to apply, and the number of cores to use. Figure 3.3 presents snippets of actual configuration files representing both data structures. Speedup and powerup values are derived from an application characterization, *i.e.,* running a representative application in all configurations and measuring the behavior.

To measure both performance and power, we modify the Heartbeats API [58] to record power data along with timing statistics (see also Appendices A.1 and A.2). We then modify applications to issue heartbeats at appropriate points during processing, typically after the completion of every job. POET can then use this captured timing and power data at runtime.

The third necessary input is a performance target. Performance goals can change during runtime, and POET automatically adapts.

3.3.2 Software Interface

Note that this interface described here has changed slightly from the original POET publication, primarily to decouple POET and the Heartbeats API [63]. Users typically interact with only three POET functions. `poet_init` initializes POET and returns a `poet_state` data structure reference, which stores the control state required to implement the framework described in Chapter 3.2. `poet_apply_control` executes the controller, runs Algorithm 1, and configures the platform. `poet_destroy` cleans up the `poet_state` data structure.

The `poet_init` function takes as parameters: the performance goal, the window period over which the controller operates, the system's configurations, and a reference to the function that applies the given system configurations. It also receives an optional reference to the function that determines the system's current state and a log file name. The first system configuration data structure (system-agnostic) is of type `poet_control_state_t`, and the second (system-specific) has type `void`. The two functions passed by reference are the only ones that need to know the format of the second data structure and are therefore passed the `void` type reference given to `poet_init` as parameters. The first of these two functions must have a signature that matches the `poet_apply_func` definition and the second must match the `poet_curr_state_func` definition.

The `poet_apply_control` function takes as parameters: the `poet_state` reference, a unique identifier (for logging purposes), and the measured performance and power values (*e.g.,* as recorded by the Heartbeats API). The `poet_destroy` function just takes the `poet_state` reference as a parameter. An additional function, `poet_set_performance_goal`, allows changing the performance goal at runtime.

A separate header exposes auxiliary functions to load system configurations from files, discover the initial system configuration, and apply system configurations. These are for the default file formats and system knobs described in the previous section. The latter two of these functions meet the `poet_curr_state_func` and `poet_apply_func` definitions, respectively, and can be passed to `poet_init`. These auxiliary functions are platform-dependent and thus kept separate to maintain portability, allowing users to easily substitute their own versions. They are, however, generic enough that most Linux users do not need to write their own unless they want to tune different knobs.

3.3.3 Runtime

After an application measures its performance and power, *e.g.,* with a heartbeat, it makes a call to `poet_apply_control`, which contains POET's core logic. When a window period completes, the POET runtime calculates the estimated base speed with Equation 3.6, then computes the performance error with Equation 3.3. It subsequently computes the control signal from Equation 3.4 to determine the speedup necessary to eliminate the error. Finally it determines the energy-minimal resource schedule that achieves the necessary speedup using Algorithm 1 and applies the first configuration in the schedule by executing the provided `poet_apply_func` function.

The last step, a call to the `poet_apply_func` specified by the user, is a platform-dependent operation. For example, the function included with POET for Linux platforms invokes the `taskset` utility to force the process and all of its threads onto the desired number of cores and uses the `cpufrequtils` interface to adjust the cores' DVFS frequencies. Developers can specify their own function—for example, a system may require a different approach to change the number of active cores, or a system may have additional configurable resources that a user wants to tune, like memory and network bandwidth.

The only remaining task is to apply the second configuration in the schedule at the appropriate time during the next window period. When the computed subset of jobs in the window period are complete (ω_u in Algorithm 1), the `poet_apply_func` function executes again, but no further computation is performed. When window period completes, the control process repeats.

The code snippet in Listing 3.3 is an example of application code, highlighting the POET function calls. The complete modification of an existing application requires only a handful of function calls plus associated variable declarations. The min and max heartrate, accuracy, and power values in the Heartbeats initialization are not used, so they can safely be set to any value. When initializing POET, the user should specify the system's configurations, encoded in `control_states` and `cpu_states`.

```c
// initialization
heartbeat_t* hb =
  heartbeat_acc_pow_init(window_size, log_buffer_depth, "heartbeat.log",
                         min_heartrate, max_heartrate, min_accuracy, max_accuracy,
                         1, hb_energy_impl_alloc(), min_power, max_power);
get_control_states(NULL, &control_states, &nstates);
get_cpu_states(NULL, &cpu_states, &nstates);
poet_state* state = poet_init(perf_goal, nstates, control_states, cpu_states,
                              &apply_cpu_config, &get_current_cpu_state,
                              window_size, log_buffer_depth, "poet.log");
// execution of main loop
while(running) {
  heartbeat_acc(hb, count++, 1);
  poet_apply_control(state, count, hb_get_windowed_rate(hb), hb_get_windowed_power(hb));
  doWork();
}
// cleanup
poet_destroy(state);
free(control_states);
free(cpu_states);
heartbeat_finish(hb);
```

Listing 3.3: Example of POET application code.

3.4 Experimental Design

This section details the applications used to evaluate POET and the different evaluation platforms. The evaluation is broken down into two categories: an embedded systems evaluation and a server-class system evaluation.

3.4.1 Applications

To represent a wide variety of embedded applications, we use eight different benchmarks, none of which were originally written to provide predictable timing. We choose applications that do not enforce any timing guarantees to challenge POET's approach as much as possible.

The first five applications are included in the PARSEC benchmark suite [14]. Specifically, we use `blackscholes`, `bodytrack`, `facesim`, `ferret`, and `x264`. `Bodytrack` and `x264` process video input and could be required to match the frame rate of a live feed (*e.g.,* from an on-board camera). `Ferret` is a toolkit for content-based similarity search of non-text data and should satisfy a latency requirement on how fast results are returned to users. `Facesim` creates realistic animations of a human face from a model and time sequence of muscle movements, and must maintain a real-time frame rate. The sixth and seventh applications are `dijkstra` and `sha`, from the ParMiBench benchmark suite [68]. `Dijkstra` computes single-source shortest paths in a graph (we use the parallelized multiple queue implementation). `SHA` (Secure Hash Algorithm) is used for secure storage and transmission of data and must maintain response time to ensure timely communication. The last application is `STREAM` [94], a synthetic benchmark for measuring sustainable memory bandwidth, representing a variety of memory-bound applications.

All benchmarks are modified as discussed in Chapter 3.3.3, adding Heartbeats and POET calls. All application inputs used are packaged with the original benchmarks, with the exception of the `x264` input which comes from a set of standard test sequences.

3.4.2 Evaluation Platforms

The first part of the evaluation uses two modern embedded devices with different hardware—a Sony VAIO SVT11226CXB tablet PC with a mobile Intel Haswell processor, and a Hardkernel ODROID-XU+E ARM big.LITTLE development platform. We selected these two platforms because prior work has shown that they expose different timing and energy trade-

Table 3.2: System power characteristics.

System	Idle Power	Min Power	Max Power
Vaio	2.50 W	3.04 W	8.05 W
ODROID-XU+E	0.12 W	0.17 W	8.14 W
Server	17.90 W	37.80 W	199.26 W

offs [61]. Both platforms run Ubuntu Linux 14.04 LTS. The Vaio uses kernel 3.13.0, while the ODROID runs kernel 3.4.104.

The second part of the evaluation uses a dual-socket server system, where each socket contains 8 cores. With HyperThreading, the system exposes 32 virtual cores. There are 16 DVFS settings available, including TurboBoost. The server system also runs Ubuntu 14.04 LTS with Linux kernel 3.13.0.

A **configuration** is a unique combination of allowable values for the system resources. In all cases, `cpufrequtils` controls processor clock speeds and `taskset` controls core allocation. While the Vaio claims to support different frequency settings on different virtual cores, our experience leads us to conclude that this is not the case. Thus, we allow only configurations where all cores are set to the same frequency. The ODROID's version of the Exynos5 Octa does not support executing on the big and LITTLE clusters simultaneously, and all cores in a cluster must operate at the same frequency. We use a mainline Linux kernel with the default In Kernel Switcher for managing cluster migration. On the server system, we set the DVFS frequency on all cores, like on the Vaio.

Capturing power/energy metrics naturally requires hardware resources that expose power or energy data to software. To capture power measurements on the Vaio and the server system, we use the Model-Specific Registers (MSRs) provided by Intel [115]. On the ODROID, we poll INA-231 power sensors to capture power data for the A15 and A7 clusters as well as for the DRAM and for the GPU [66]. Basic power figures for the three platforms are shown in Table 3.2. The modified version of Heartbeats includes energy readers for some common hardware (*e.g.,* the MSR) and exposes a simple interface for extending to new hardware.

Table 3.3: Embedded systems evaluation inputs and configuration details.

Application	**Input**	**Jobs**	**Window Size**
blackscholes	1 million options	400 batches	20
bodytrack	sequenceB	261 frames	20
facesim	Storytelling	100 frames	20
ferret	corel:lsh	2,000 queries	20
x264	ducks_take_off	500 frames	20
dijkstra	input_small	1,000 paths	20
sha	in_file(1-16)	1,000 hashes	50
STREAM	self-generated	1,000 updates	50

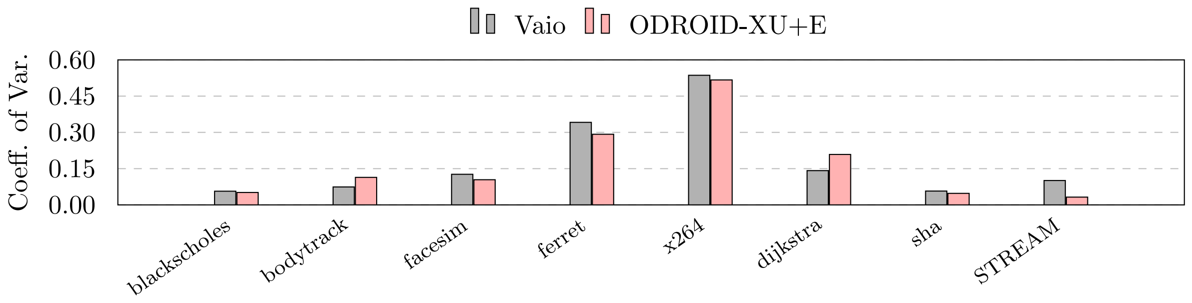


Figure 3.4: Embedded systems application latency variability.

Collecting power data on new platforms with different power or energy monitors is easy and does not require any modifications to POET.

Table 3.3 shows the inputs used for each of the applications on the embedded platforms. We quantify this inherent unpredictability by measuring the latency of each job and computing the coefficient of variation (ratio of standard deviation to the mean) over all jobs in an application. Figure 3.4 demonstrates this unpredictability for each application when running without POET. The figure shows that our applications have a range of natural behavior from low variance (implying natural predictability, *e.g.,* `blackscholes`) to high variance (meaning that the application naturally has widely distributed latencies, *e.g.,* `x264`). The variability in the applications is largely the same across platforms, indicating that it is a fundamental property of the applications and not the devices.

Table 3.4 lists the inputs used for each of the applications on the server system. The server-class system is significantly more powerful than the embedded systems. The overhead

Table 3.4: Server-class system evaluation inputs and configuration details.

Application	**Input**	**Jobs**	**Window Size**
blackscholes	10 million options	400 batches	50
bodytrack	sequenceB	261 frames	50
facesim	Storytelling	100 frames	20
ferret	corel:lsh	2,000 queries	50
x264	rush_hour	1,500 frames	100
dijkstra	input_large	1,000 paths	50
sha	in_file(1-16)	1,000 hashes	50
STREAM	self-generated	1,000 updates	50

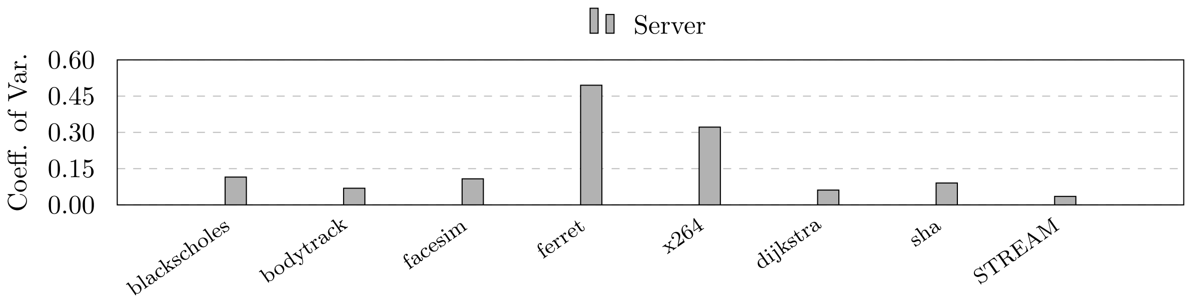


Figure 3.5: Server system application latency variability.

of changing resource allocations is also higher due to the larger core count. As a result, we increased both the size or length of some inputs and the window period size. Again, we quantify the variability in the applications and present the results in Figure 3.5. As expected, they are similar to those from the embedded systems.

3.5 Embedded Systems Evaluation

This section describes POET's evaluation on embedded systems—the Vaio and the ODROID-XU+E (henceforth referred to simply as the ODROID)—and is divided into five parts. First, we demonstrate POET's ability to meet performance goals, formulated as job latency requirements. Next, we quantify the energy consumption of the resulting system, then compare the energy of POET's general approach to one that controls latency and energy tradeoffs using just DVFS. We then evaluate POET's ability to adapt to input with multiple phases, and finally, its ability to run subject to the interference of other applications.

3.5.1 Meeting Latency Targets

To test POET's ability to meet latency targets, we first *characterize* each application i by executing in all possible configurations on both systems to determine the minimum average latency m_i and derive an *oracle* to be used for our analysis. This oracle determines an optimal resource schedule for each target without missing any deadlines, and has no computation or configuration switching overhead. For each application, we impose four latency targets. The targets cover a wide range of achievable goals, from 25% to 95% of each system's performance capacity, *i.e.,* a 25% goal means that the target is set to $4 \times m_i$. Applications are launched in the maximum-resource configuration (configuration $C-1$ as described in Chapter 3.2.2). POET observes application behavior during the first window period, then begins adapting the system to meet the latency goals while minimizing energy consumption.

We quantify POET's ability to meet the latency goals by measuring each job's latency and comparing it to the goal. We then compute the Mean Absolute Percentage Error (MAPE), a standard metric in control theory to evaluate the behavior of controllers [40]. For an application composed of n jobs:

$$\text{MAPE} = 100\% \cdot \frac{1}{n} \sum_{i=1}^{n} \begin{cases} d_m(i) > D_{ref} : & \frac{d_m(i) - D_{ref}}{D_{ref}} \\ d_m(i) \leq D_{ref} : & 0 \end{cases} \tag{3.11}$$

where D_{ref} is the specified job latency requirement (deadline) and $d_m(i)$ is the measured latency for the i-th job. In other words, for each missed deadline we add a term that depends on the relative tardiness between the target and measured latency.

Figure 3.6 presents the MAPE values for each application for the four latency targets on both the Vaio and the ODROID. In general, the larger the variance in the application behavior, the larger the error. This is not surprising since more volatile applications are

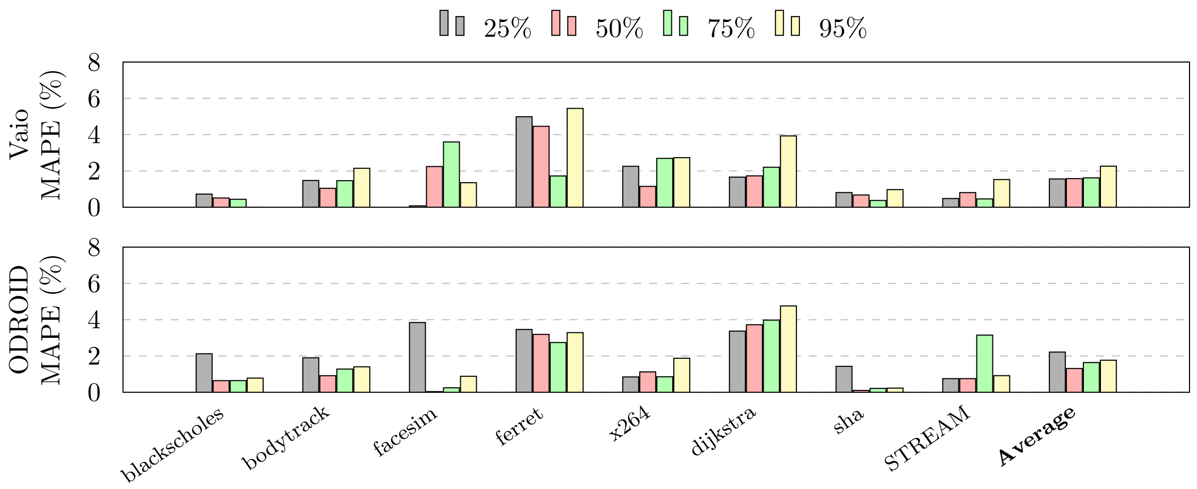


Figure 3.6: Embedded systems latency error for different latency targets (lower is better, 0 is optimal).

harder to control. However, the results indicate that MAPE values are generally low. On the Vaio, the average MAPE for all applications is well below 2.5% for all targets, typically closer to 1.5%. The ODROID presents similar results. The MAPE metric is unforgiving since it penalizes every violation of the latency target, yet POET achieves low MAPE even for applications that were not inherently designed to support predictable timing.

3.5.2 Energy Minimization

This section evaluates POET's energy minimization strategy. As discussed, we have measured latency and energy consumption for all applications in all configurations. Therefore, the oracle has perfect knowledge of each application's behavior. We quantify POET's energy consumption by comparing its achieved energy to the oracle derived from the application characterization. Although the oracle has zero overhead, meeting all latency targets while simultaneously achieving optimal energy is not actually possible in practice as it would require knowledge of the future and no overhead.

For each application, we run POET for each latency target and record the achieved energy consumption. We then compute the ratio of the energy consumption with POET

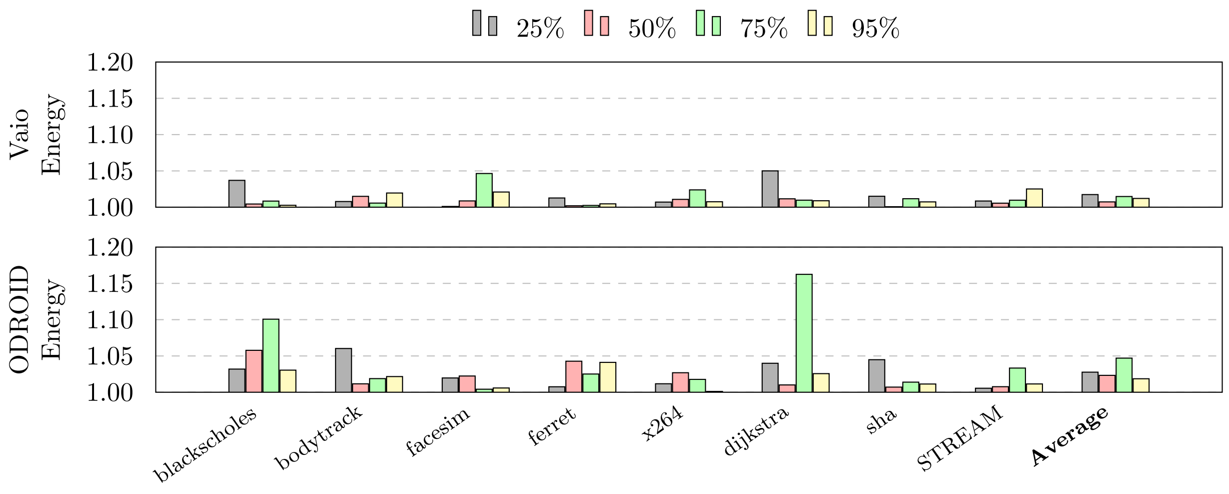


Figure 3.7: Embedded systems energy consumption for different latency targets (lower is better, 1 is optimal).

to the effective minimal energy. Unity represents minimal energy and values greater than 1 show energy consumption above the optimal. Figure 3.7 presents the normalized energy data for each application on both the Vaio and the ODROID. The data includes the overhead of POET's runtime, which consumes additional energy executing the control and optimization tasks. On average across all applications and targets, POET's energy consumption exceeds optimal by 1.3% on the Vaio and by 2.9% on the ODROID. These results demonstrate that POET achieves near-optimal energy consumption in practice.

The most troublesome test is `dijkstra` on the ODROID with a latency target of 75%, which exceeds optimal by about 16%. The true optimal schedule just barely achieves this goal by varying the DVFS setting between 1.2 and 1.1 GHz. Any overhead larger than 2% requires a clock speed of 1.3 GHz. Unfortunately, this is in the area of steeply diminishing returns for the ODROID. Compensating for this overhead almost entirely accounts for the energy difference between POET and optimal. POET's runtime overhead is small, but non-zero, so POET uses the higher clock speed. POET's overhead is due in part to its generality; *i.e.,* its ability to handle multiple actuators that may affect energy and latency. The next section highlights the benefits of this generality.

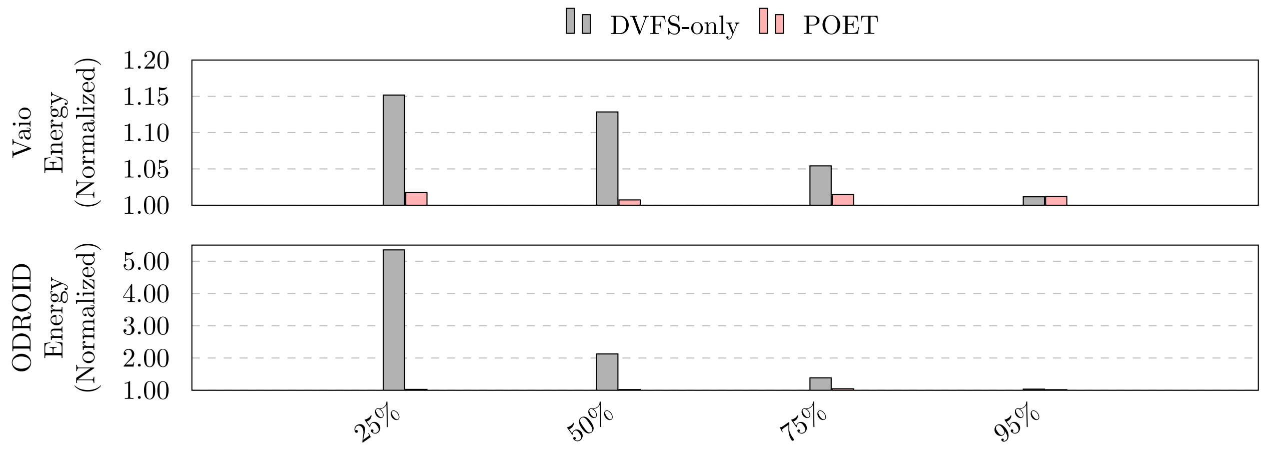


Figure 3.8: Comparison of average energy consumption with DVFS-only versus POET (lower is better, 1 is optimal).

3.5.3 Comparison with DVFS

Several energy management approaches have been proposed that optimally tune DVFS settings to meet timing constraints while reducing energy consumption [4]. In this section, we compare POET's energy consumption to an approach which only uses DVFS. To do so, we develop system configuration files that only specify changes in DVFS settings and deploy POET on both embedded platforms with these configurations. We compare this *DVFS-only* approach to POET's more general approach which coordinates multiple resource types and uses different resources on different platforms.

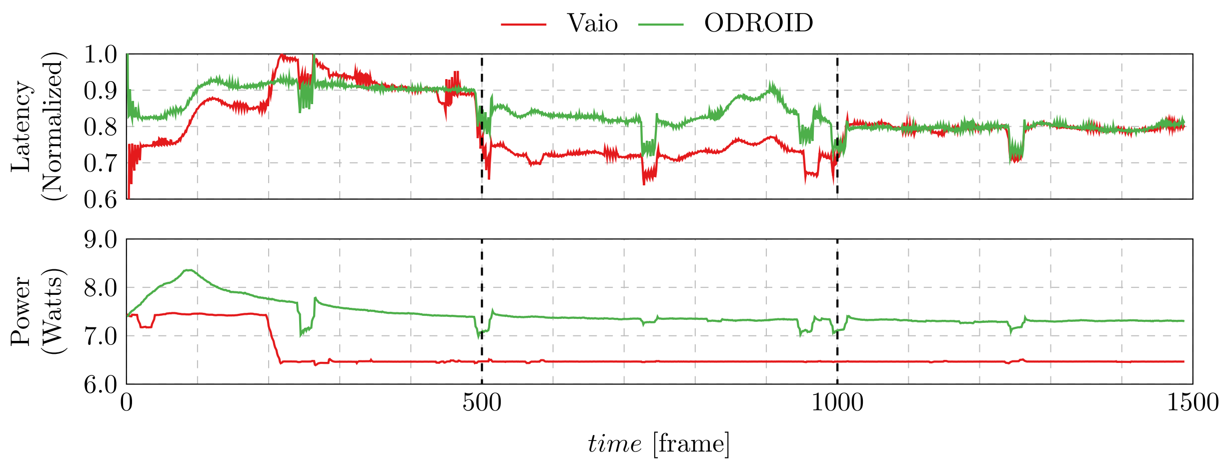
Figure 3.8 summarizes the data comparing a DVFS-only approach to POET. The charts show latency targets on the x-axes and energy consumption normalized to optimal on the y-axes (for POET, this is the same data shown in Figure 3.7). For each latency target, the figure shows the average energy over optimal (across all benchmarks) for both DVFS-only and POET on both platforms. At the higher latency (lower performance, *e.g.,* 25%) targets, POET saves substantial energy. The energy savings are especially high on the ODROID as POET is able to take advantage of cluster migration and the low-power LITTLE cores, whereas a DVFS-only approach cannot exploit this feature. This data clearly demonstrates that systems provisioned for a rarely-seen, worst-case latency can greatly benefit from POET's generalized approach. This is also further confirmation of prior studies showing that DVFS by itself is not optimal [57, 95].

3.5.4 Responding to Application Phases

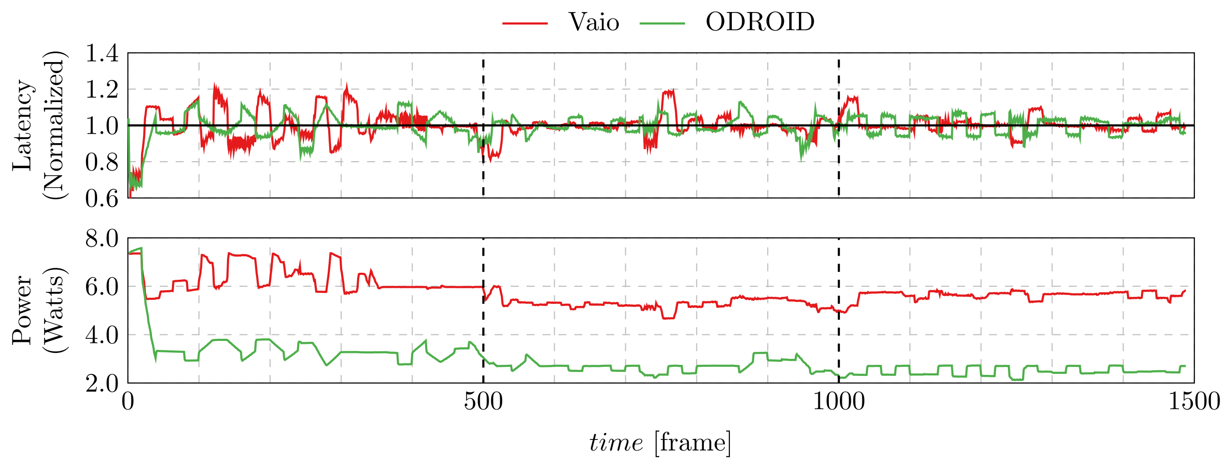
In this test, we examine POET's ability to cope with input whose workload varies with time. We execute the `x264` application using an input that is a combination of three videos of varying difficulty. The input thus has three distinct phases, each composed of 500 jobs (frames). Figure 3.9a shows time series data for both job latency and power on the Vaio and the ODROID when they run without POET in their highest-performing configurations. Latency is normalized to the maximum latency measured for any job, *i.e.,* the empirically determined worst case. We use this worst case result to derive the latency target for the POET tests. Frames that take less time are easier to encode and require fewer system resources to meet a latency goal compared to the frame that takes the most time, presenting opportunities to save energy. We present the raw data for power; energy is the integral of those curves.

The phases are clearly distinguishable by the change in latency at frames 500 and 1000. The two embedded systems do not process each phase with the same relative latency. The first phase is the most difficult (highest latency) for both systems, but the second phase is the easiest (lowest latency) for the Vaio, while the third one is the easiest for the ODROID.

We enable POET using the maximum measured latency identified in the first experiment as the target. Figure 3.9b demonstrates that POET is able to meet the latency target on both systems. Dips and spikes are visible at the beginning of each phase, showing the change in input behavior and POET adapting to the change. Despite these variations, latency goals are respected: MAPE is 2.2% on the Vaio and 2.0% on the ODROID. At the same time, energy is near-minimal over the course of execution: energy is 1.7% greater than optimal on



(a) Uncontrolled behavior.



(b) Meeting a performance target with POET.

Figure 3.9: Processing `x264` input with distinct phases on embedded systems.

the Vaio and 3.6% over optimal on the ODROID.

3.5.5 Adapting to Other Applications

This test evaluates POET's behavior when other applications are present in the system, using an external load that is not under POET's direct control. We launch a POET-enabled application (`bodytrack`) with a target latency, then halfway through its execution, we launch another application. This second application consumes resources, slowing down the POET-enabled application. POET then assigns more resources to its own application so that it

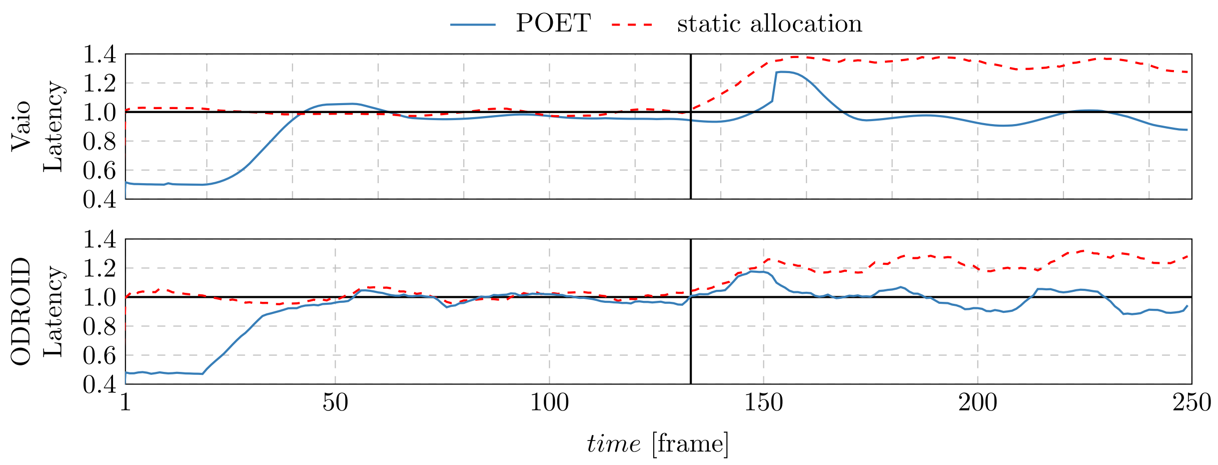


Figure 3.10: POET adapting to a background application on embedded systems.

continues to meet its latency target.

Figure 3.10 demonstrates POET's behavior on the two embedded platforms. The thick vertical lines indicate when the second application is launched. On both systems, we see the latency temporarily increase before POET adjusts the resource allocation. The charts also show a static resource scheduling approach that selects the application's system resources at the beginning of the execution. In the static case, the introduction of the new application dramatically increases the job latency, but it is not able to adapt like POET. On the Vaio, POET's MAPE is 2.3% over the entire execution (including the period of adjustment to the new load), while the static case has a MAPE of 16%. On the ODROID, POET achieves 2.4% MAPE, while the static case is 12%.

3.6 Server-class System Evaluation

This section describes POET's evaluation on a server-class system, which is divided into five parts. First, we demonstrate POET's ability to meet performance goals, again formulated as job latency requirements, then compare the energy consumption results to optimal. Next, we evaluate POET's ability to adapt to input with multiple phases, its ability to run subject to interference from another application, then finally discuss some differences with the

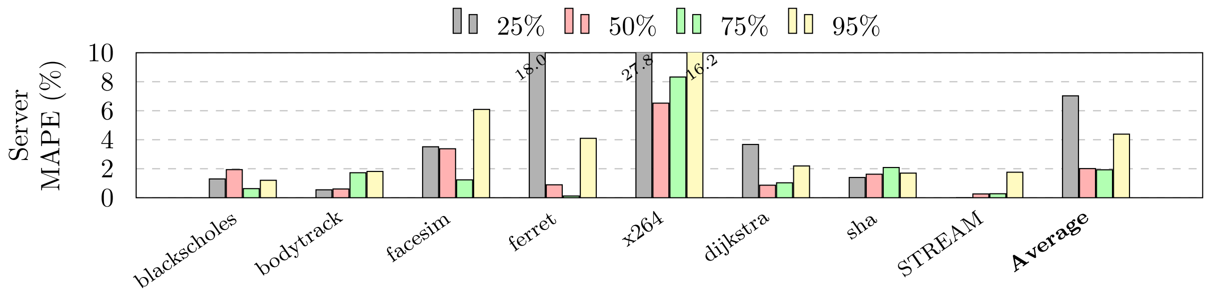


Figure 3.11: Server system latency error for different latency targets (lower is better, 0 is optimal).

embedded systems evaluation and quantify some overhead values.

3.6.1 Meeting Latency Targets

We demonstrate that POET is able to meet latency targets for each application on the multicore server-class system. As with the embedded systems, we *characterize* each application by executing in all possible configurations without POET, derive an oracle, and set latency targets for each application that range from 25% to 95% of their respective performance capacities. The MAPE metric for quantifying POET's ability to meet the latency goals was described earlier in Equation 3.11 (Chapter 3.5.1).

Figure 3.11 presents the each application's MAPE values for the four latency targets on the server-class system. As before, the relationship between application variability and MAPE is clear: higher variance typically results in higher MAPE since more volatile applications are more difficult to control. Still, the error values are generally low—on par with the behavior observed on embedded systems. There are a few outliers, particularly with `ferret` and `x264`, which are both high-variability applications. `Ferret`'s threads are asynchronous, so work continues to be performed while system changes are being applied which introduces unpredictability into timing measurements. `X264` is continuously creating and destroying threads, sometimes causing errors when assigning threads to cores with `taskset`. These issues are more apparent on the server-class system because of its significantly higher

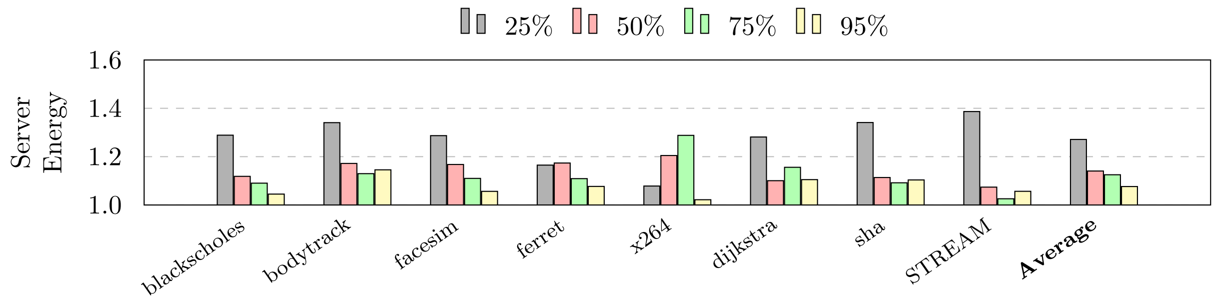


Figure 3.12: Server system energy consumption for different latency targets (lower is better, 1 is optimal).

parallelism and performance compared to the embedded systems. Further increasing the size of the window period or adjusting the pole value in POET's controller helps offset these kinds of issues. Still, POET achieves low MAPE for most executions on the multicore server system, despite the applications not being originally designed to provide predictable timing.

3.6.2 Energy Minimization

We now use the oracle, derived from each application's characterization, to evaluate POET's ability to minimize energy consumption when meeting the four latency targets on the server-class system.

Figure 3.12 presents the ratio of energy consumption to optimal for each latency target, (again) where unity is optimal and values greater than 1 show energy consumed over optimal. As before, executions include the POET runtime overhead and the overhead imposed by changing system configurations. On the resource-constrained embedded systems, the overhead of changing resource allocations was low, but the multicore server system requires additional time and energy to apply changes. The overhead of computing the resource schedule is dwarfed by the cost of applying DVFS settings to 32 cores and sometimes switching application core assignments between one and two processor sockets (more on this in Chapter 3.6.5). Despite this challenge, POET still achieves near-optimal energy consumption.

The 25% target is clearly the most inefficient, and in fact is not actually achievable for

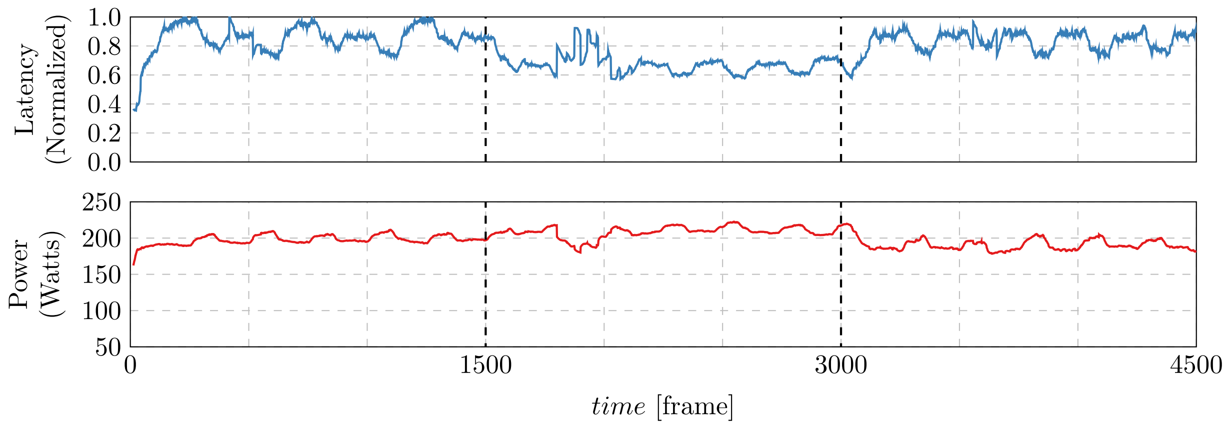
`STREAM` without idling, which POET does not support. `Ferret` and `x264` appear to be efficient at the 25% target, but this is just a side effect of their high MAPE. These observations are consistent with studies on server-class systems that demonstrate how inefficient these machines are when running at low utilizations [9, 139].

3.6.3 Responding to Application Phases

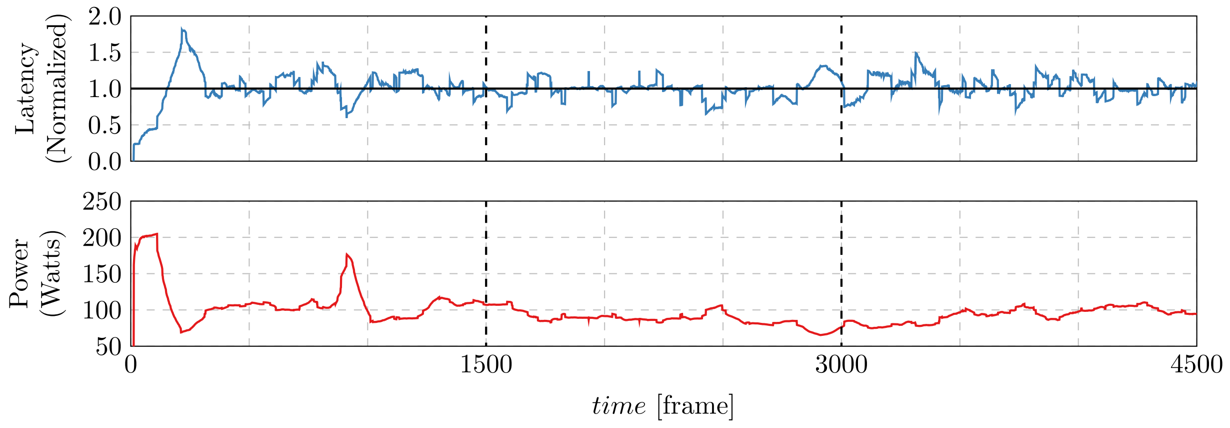
We again examine POET with an application input that exhibits changes in its behavior over time. The analysis is the same as in the embedded systems evaluation, except that we increase each video phase length so that each phase is 1,500 jobs (frames), for a total of 4,500 jobs.

Figure 3.13a shows the time series data for latency and power consumption when running the application without POET in the highest-resource configuration ($C - 1$). We again normalize latency to the maximum recorded value. The phases are clearly distinguishable by the change in latency at frames 1,500 and 3,000. In the embedded systems evaluation, we noted that the two systems did not process each phase with the same relative latency. The first phase was the most difficult (highest latency) for both systems, but the second phase was the easiest (lowest latency) on one while the third was the easiest on the other. Now on our server system we find that the first and third phases are just about the same level of difficulty and the second phase is easiest.

Figure 3.13b demonstrates enabling POET with a target that is about half of the system's maximum performance (twice the minimum latency). We launch the application in the highest resource configuration. During the first 100 frames (the first window period), POET observes the application behavior, hence the low latency and higher power consumption. The first resource adjustment overshoots the latency target, reducing power consumption below where it will stabilize. Latency and power settle around frame 300, or the end of the second adjustment period. Later fluctuations are a result of variability in the input video



(a) Uncontrolled behavior.



(b) Meeting a performance target with POET.

Figure 3.13: Processing `x264` input with distinct phases on a server system.

(per Figure 3.5, `x264` inputs exhibit high variance). There is a discernible drop in power after frame 1,500, indicating the start of the second phase where fewer resources are required to meet the latency target. Power then increases after frame 3,000 when the processing again becomes more difficult. Despite these variations, latency goals are respected: MAPE is 5.6% and energy is 20.2% greater than optimal, which are similar to the `x264` results shown previously.

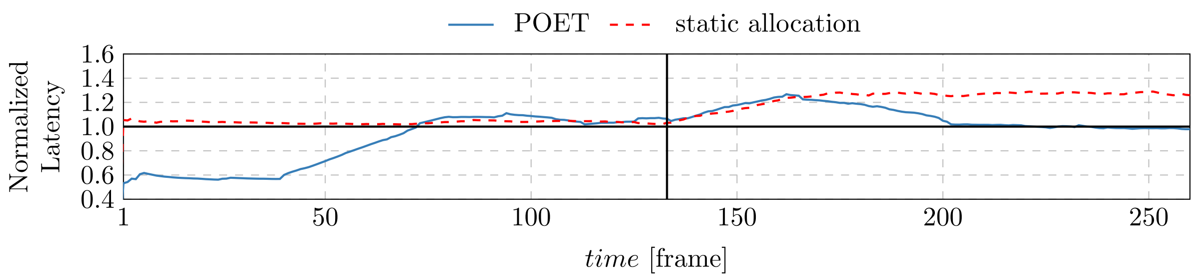


Figure 3.14: POET adapting to a background application on a server system.

3.6.4 Adapting to Other Applications

Again, we demonstrate POET adapt to changes in system resource behavior at runtime using the `bodytrack` application and a performance target of about 50% capacity. Halfway through the execution, we launch an application in the background that does not use POET, but consumes system resources. POET adapts by allocating more system resources, *i.e.,* increasing the DVFS speeds and/or allocating more cores to `bodytrack` so that it continues to meet the original soft latency goal.

Figure 3.14 presents a time series for this scenario, including the POET-controlled execution and another that uses a static resource allocation strategy that fixes the resource assignments at the start of the execution. The y-axis is normalized to the latency target, and the vertical line indicates when the second application is launched. For this test, we reduce the window size from 50 to 40 frames which allows for more window periods during the execution but increases volatility. As with the previous experiments, we launch the POET-controlled `bodytrack` application in the highest-resource setting, configuration $C-1$. During the first window period, POET observes application behavior, then makes its first resource allocation decision at frame 40. By frame 80, the end of the first period of adjustment, the average window job latency is near the target. After the second application is launched, there is a temporary increase in latency. POET detects this change and allocates additional resources so that the latency goal continues to be met. This adaptation results in 5.0% MAPE for the entire execution. In contrast, the static allocation strategy fails to

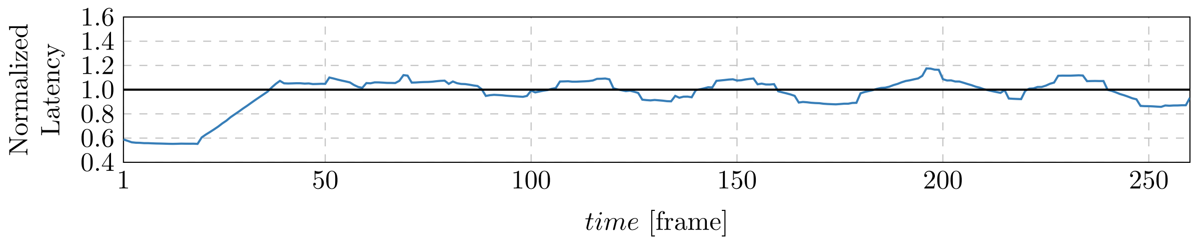


Figure 3.15: POET with insufficient window size.

meet job deadlines after the second application begins, resulting in 23.9% MAPE.

POET adjusts resource allocations to adapt to changes in application behavior. Assuming there are sufficient resources still available, a POET-controlled application will continue to meet its soft deadlines, despite interference within the system.

3.6.5 Configurations and Overhead

An important difference between the embedded systems analysis and the server-class system analysis is the choice of the window period sizes for applications. For example, the `bodytrack` executions used a window size of 20 on the embedded systems, but we used a size of 50, and later 40, for evaluations on the multicore server-class system for the same application. Faster application performance and larger number of resources on the server-class system increase the relative overhead of changing system resource allocations at runtime, making window size changes necessary. Figure 3.15 demonstrates the results of using a window size of 20, which is too small, for a latency target of about 50% capacity. Although MAPE is still low at 2.95%, the controller oscillates, failing to converge.

We measure the overhead of three resource allocation tasks: (1) the POET controller and optimizer, which we call *POET Core*, (2) application core assignment with `taskset`, which we call *Affinity*, and (3) changing frequency settings with *DVFS* for the 32 virtual cores. The latter two are executed by the platform-specific function defined by `poet_apply_func` (Chapter 3.3.2). Compared to a perfect implementation that requires no computation or resource allocation overhead and always meets the latency goal, each POET Core execution adds 0.12 ms average latency overhead, each Affinity change averages 62.24 ms, and each DVFS change averages 65.08 ms. The POET Core overhead is negligible, but the others add 2.36% and 2.47% timing overhead to the example in Figure 3.15, totaling almost 5%. That cost is like adding a whole additional frame to the window period. Increasing the window size to 50 reduces the Affinity and DVFS overhead to less than 1% each for this `bodytrack` performance target. Faster applications require longer window periods to reduce the performance impact caused by the fixed overhead of changing resource allocations.

The POET design models overhead as error and lets the control dynamics naturally correct any overhead. This approach works best on small-scale systems like the embedded systems we evaluated, but clearly has drawbacks on the larger system evaluated here. As explained above, we can overcome this drawback by using larger windows to amortize overhead. We could also extend POET to explicitly account for overhead and the cost of switching configurations. Such an approach would force POET to be conservative about switching configurations and likely reduce energy savings. A third approach would be to build hardware and operating system support for rapid configuration changes. We believe supporting this kind of adaptability is key for future multicore systems, as faster configuration changes increase the potential for energy savings.

3.7 Discussion of Results and Limitations

Our results show that POET achieves the goals of providing predictable timing and near-minimal energy across multiple platforms. These results are obtained despite the facts that: (1) the tested applications were not originally designed to offer predictable timing, and (2) the test platforms vary greatly in size and capability, and have different latency/energy tradeoffs. The applications require only minimal modifications to run with POET, but no other changes are needed to exploit the different resources and tradeoffs that different platforms offer.

These results also demonstrate some limitations of the approach. POET supports only soft real-time constraints. The controller is guaranteed to converge to the desired performance/latency goal and is provably robust to errors, but goals may be violated during the settling time, as seen in Figures 3.10 and 3.14 when POET adapts to the presence of new applications in the system. In addition, highly variable applications can still cause temporary violations before the control action settles again, as seen in Figures 3.9b and 3.13b when controlling the high-variance `x264` application. This is further evidence that there is a tension between timeliness and energy reduction [17], *e.g.,* the tremendous energy savings on the ODROID come at a cost of some latency errors compared to the *race-to-idle* heuristic.

POET is also sensitive to the resource specifications provided by the user. While the controller can tolerate large errors, in practice it is best to classify applications by their behavior, *e.g.,* compute or memory-bound, and use different configurations for each class of application. POET's models also do not currently account for the time required to switch between configurations. Instead, this overhead is modeled as an inaccuracy in the specified speedup. Our results show that this simplification works well in practice, but it may not be sufficient with different resources that have extremely long configuration transition latencies. In that case, the POET controller and optimizer should be extended to account explicitly for the overhead of switching configurations.

Finally, POET currently assumes that only one of the running applications should meet a performance/latency deadline. POET's Kalman filter guarantees that even when other applications are present in the system, the controller will compute the correct speedup to be applied, as demonstrated in Figures 3.10 and 3.14. However, future work could extend POET with a priority scheme allowing multiple POET-enabled applications to work concurrently. In that scheme, high-priority applications would be allocated the needed resources and lower-priority applications would run in best-effort mode.

3.8 Bard: An Extension of POET

In this section, we briefly describe Bard, and extension to POET that adds the ability to meet soft power constraints and maximize performance. For a complete description and evaluation, see the original Bard publication [62].

3.8.1 Motivation

An application on an embedded device (like a smartphone) may need to meet timing constraints when executing in the foreground to deliver a sufficiently high level of performance to satisfy users. The same application may have tasks that execute when running in the background that are not time-sensitive and would benefit the user by keeping power consumption low so as not to drain the battery. Other systems that normally provide timing guarantees must continue operating with reduced capacity during hardware failures (*e.g.,* a broken cooling fan) or during periods of extremely low energy reserves (*e.g.,* at night or during cloudy weather for a system that harvests energy from a solar panel).

A naive approach to solving this problem is to integrate two different solutions into the application: one that meets timing constraints, and another for power. If the libraries are not properly managed, they may compete for control of system resources. This can result in failing to meet either goal, causing poor performance and high power consumption. Furthermore, it adds additional complexity to the program and increases the overhead in development and validation testing. Furthermore, many existing solutions are specific to particular applications or systems, *i.e.,* they are not portable.

POET's portability makes it a perfect starting point for solving this problem. Bard extends POET by adding the ability to track both performance and power behavior, allowing the user to decide which constraint/optimization scheme to use at any time. In summary, Bard: (1) meets either timing or power constraints, (2) optimizes the other, and (3) remains portable across systems.

3.8.2 Further Generalizing POET's Design

Bard's formulations are very similar to POET. The key to rapidly switching between performance and power constraints is to continually estimate both base values. The Bard controller generates its *generic control signal*, $x(t)$, which is either *speedup* or *powerup*:

$$x(t) = x(t-1) + (1-\alpha) \cdot \frac{e_x(t)}{b_x(t)} \tag{3.12}$$

Here $b_x(t)$ represents either base speed or base power, depending on the constraint, both of which are continually estimated by a Kalman filter [131].

The constrained optimization problem for meeting a performance goal and minimizing energy consumption remains the same as in POET (Equations 3.7–3.10). To meet a power goal P_{ref} and maximize performance, the problem becomes:

$$\begin{aligned}
\mathit{maximize} \quad & \sum_{c=0}^{C-1} \tau_c \cdot s_c && (3.13)\\
s.t. \quad & \sum_{c=0}^{C-1} \tau_c \cdot p_c \cdot b_p(t) \le P_{ref} && (3.14)\\
& \sum_{c=0}^{C-1} \tau_c = 1 && (3.15)\\
& \tau_c \ge 0, \qquad \forall c \in \{0, \dots, C-1\} && (3.16)
\end{aligned}$$

where τ_c is the proportion of time spent in configuration c.

Algorithm 2 further generalizes the POET optimizer behavior (Algorithm 1 in Chapter 3.2.2) to compute either a minimal-energy or maximal-performance schedule. It takes as input the configuration set C, workload size ω, the *constraint* type, and the generic control signal $x(t)$. The algorithm similarly loops over all feasible configuration pairs and determines the optimal schedule that achieves the speedup or powerup given by the controller.

We add the `constraint` type enumeration (`PERFORMANCE` or `POWER`) to the POET API ini-

Algorithm 2 Finding an Optimal Configuration Schedule.

```
Require: $C$                                   ▷ system configurations, given by user
Require: $\omega$                              ▷ discrete work units, given by application
Require: $constraint$                          ▷ PERFORMANCE or POWER, given by user or application
Require: $x(t)$                                ▷ speedup/powerup, depending on constraint, given by Equation 3.12
  $U = \{c \mid x_c \leq x(t)\}$
  $O = \{c \mid x_c > x(t)\}$
  $candidates = U \times O = \{\langle u, o\rangle \mid u \in U, o \in O\}$
  $cost = \infty$
  $optimal = \langle -1, -1\rangle$
  $schedule = \langle -1, -1\rangle$

  for $\langle u, o\rangle \in candidates$ do                              ▷ loop over all pairs
      $\omega_u = \omega \cdot \frac{x_u \cdot (x_o - x(t))}{x(t) \cdot (x_o - x_u)}$     ▷ compute the work units to spend in each configuration in pair
      $\omega_o = \omega - \omega_u$
      if $constraint = POWER$ then                                      ▷ compute cost of this pair
          $newCost = \frac{\omega_u}{s_u} + \frac{\omega_o}{s_o}$                      ▷ normalized latency
      else
          $newCost = \frac{\omega_u}{s_u} \cdot p_u + \frac{\omega_o}{s_o} \cdot p_o$          ▷ normalized energy
      end if
      if $newCost < cost$ then                                          ▷ compare cost to best found so far
          $cost = newCost$
          $optimal = \langle u, o\rangle$
          $schedule = \langle \omega_u, \omega_o\rangle$
      end if
  end for

  return $optimal$                              ▷ pair of configurations with minimal cost
  return $schedule$                             ▷ work units to spend in each configuration
```

tialization function. If `PERFORMANCE` is specified, Bard meets a performance target and minimizes energy. If `POWER` is specified, Bard meets the power target and maximizes performance (minimizes latency). We then replace the setter function `poet_set_performance_goal` with `poet_set_constraint_type` to support changing the constraint type and goal at runtime.

3.8.3 Evaluation

We evaluate Bard on the same Vaio tablet as POET, and use a newer ARM big.LITTLE device, an ODROID-XU3, which supports running both the big and LITTLE cluster simultaneously. Although Bard is independent of the system-specific configurations, we need a slightly different format than in POET to use in our evaluation. The new system-specific

```
#id         speedup          powerup
0           1                1
1           1.55             1.06
2           2.11             1.11
3           2.16             1.12
4           2.66             1.17
5           3.36             1.22
6           4.51             1.31
7           5.11             1.37
```

Listing 3.4: System-agnostic.

```
#id cores frequencies
0   0x01  200000,-,-,-,200000,-,-,-
1   0x01  300000,-,-,-,200000,-,-,-
2   0x01  400000,-,-,-,200000,-,-,-
3   0x03  200000,-,-,-,200000,-,-,-
4   0x01  500000,-,-,-,200000,-,-,-
5   0x07  200000,-,-,-,200000,-,-,-
6   0x0F  200000,-,-,-,200000,-,-,-
7   0x07  300000,-,-,-,200000,-,-,-
```

Listing 3.5: System-specific.

Figure 3.16: Snippets of Bard configuration files.

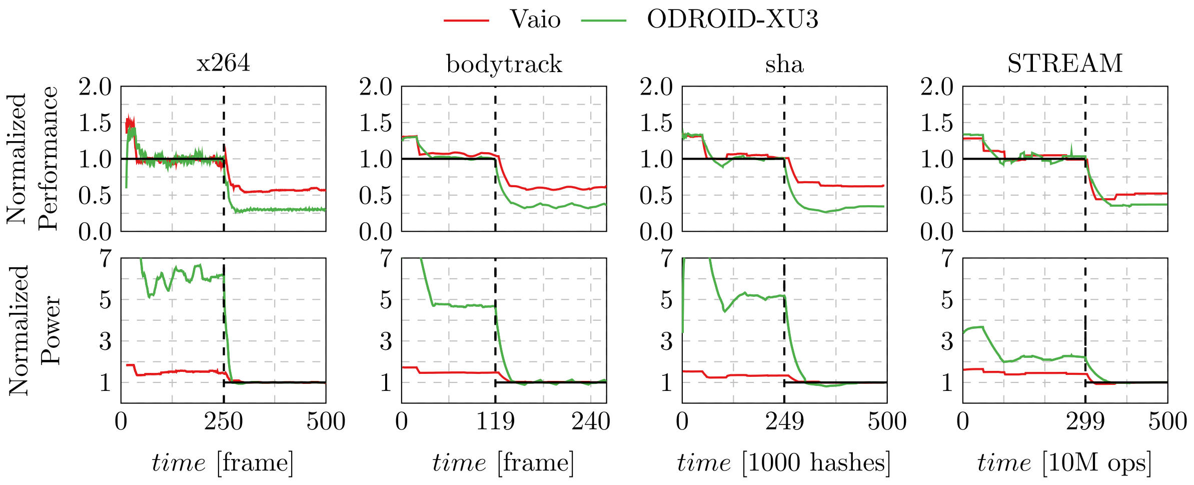


Figure 3.17: Changing constraint from timing to power.

configuration in Figure 3.16 contains an identifier, a core mask (instead of core count), and a comma-delimited list of DVFS frequencies to apply (instead of a single frequency). A dash indicates that a DVFS frequency does not need to be applied to a core, either because the frequency does not matter or it is already managed through another affected core. For example, configuration with $id = 7$ assigns cpu0, cpu1, and cpu2 (core mask 0x07) and sets the DVFS frequency on cpu0 to 300 MHz and cpu4 to 200 MHz. On this particular system, cores 0–3 are in one DVFS domain and cores 4–7 are in another. Therefore, applying a DVFS setting of 300 MHz on cpu0 sets the same frequency on cores 1–3, and applying a frequency of 200 MHz on cpu4 sets the same frequency on cores 5–7. Note that configuration with $id = 7$ does not use cores 4–7, but the DVFS frequency is set anyway (to its lowest value to save energy).

In Figure 3.17, we observe the behavior of launching four applications with a performance goal, then switching to a power goal. The top portion has performance data, normalized to the target from the first half of the execution. The bottom portion of the charts shows power data, normalized to the target set in the second half of the execution. The dashed vertical lines indicate when the switch from performance to power constraints is made. The solid black horizontal lines represent the goals.

The results demonstrate several important features. Bard first keeps the performance at or above the target, then keeps power at or below the new target. MAPE quantifies performance deficit in the first half and power cap violations in the second half. Similarly, our oracle computes the minimal-energy schedule, then the maximal-performance schedule to determine the total efficiency of the execution. On the Vaio, Bard achieves 1.58% MAPE and 92% efficiency on average. On the ODROID, it achieves 1.97% MAPE and 91% efficiency. As Figure 3.17 shows, the time taken to switch between the timing and power constraints is quite small—just one period. The small fluctuations seen in the first portion of some executions (like `x264`) are not caused by Bard, but rather by the variability of the application inputs which makes them difficult to control. These results demonstrate that Bard achieves its design goal of providing a single, portable framework for meeting either timing or power constraints and dynamically switching between the two.

CHAPTER 4
COPPER: CONTROL PERFORMANCE WITH POWER

This chapter describes CoPPer (**Co**ntrol **P**erformance with **P**ow**er**), a software system that uses adaptive control theory to meet application performance goals by manipulating hardware power caps [65]. As software management of DVFS is becoming deprecated, we propose CoPPer as a replacement. CoPPer has three key features. First, it works on applications without prior knowledge of their specific performance/power tradeoffs; *i.e.,* it does not require a system or application-specific power cap/performance model based on precharacterization, making it suitable for general purpose computing workloads composed of repeated jobs. Second, it uses a Kalman filter to adapt control to *non-linearities* in the power cap/performance relationship. Third, it introduces adaptive *gain limits* to prevent power from being over-allocated when applications cannot achieve additional speedup. That is, if a workload's performance does not improve with expanded power limits, CoPPer will not allocate additional power, whereas standard control-theoretic approaches take no additional power-saving action. Thus, CoPPer saves energy in many cases compared to existing DVFS-based approaches while maintaining its formal guarantees.

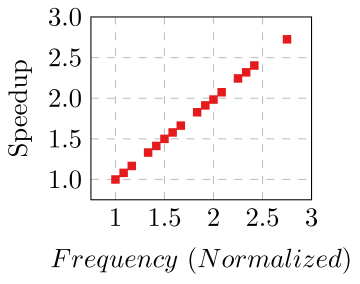
4.1 Motivation

We discuss the current processor landscape as justification for why DVFS might soon no longer be controllable by software. We then illustrate how the simple linear models that work well for meeting performance requirements by tuning DVFS can produce sub-optimal behavior when directly applied to power capping.

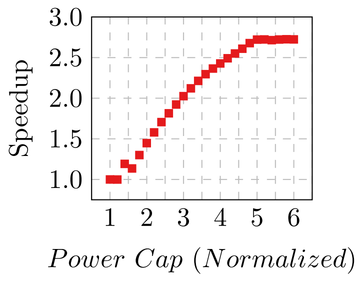
4.1.1 The Future of Software DVFS

There are strong indications that DVFS will not be directly controllable by software in future processors. Since SandyBridge, Intel processors take software DVFS settings as suggestions, and hardware has been free to dynamically alter the actual clock speed and voltage independently from the software-specified setting [33, 108]. With the Skylake architecture, Intel has been actively campaigning to move DVFS management wholly to hardware and instead have software specify power. The hardware is then free to rapidly change DVFS settings to achieve better performance while still respecting those power limits [110]. For example, if software sets power limits requiring any 50 ms time window to average 100 W, hardware is free to use turbo mode to speed up the processing of any bursty work within that 50 ms, as long as it compensates by running in a low-power state for some of that time.

Of course, even as DVFS shifts to hardware, it will still be software's responsibility to provide its own notion of either "best" or "good enough" performance, subject to hardware-imposed constraints like thermal design power and energy consumption costs. The capability to specify power caps and simultaneously provide some optimization is already provided by interfaces like Intel's Running Average Power Limit (RAPL) [25]. Recent work shows that a combination of RAPL and software resource management can achieve even better performance while guaranteeing power consumption [139]. What is still needed, however, is the software component that guarantees performance without using DVFS. We address this need with CoPPer, which provides soft performance guarantees by manipulating hardware power caps, thus allowing the hardware to perform fine-grained optimizations. Because CoPPer operates at the software level, it can still benefit from additional hardware-level optimizations to further increase energy efficiency, like on-chip voltage regulators [7] or adaptive management of power circuitry [47].



(a) DVFS.



(b) Power cap.

Figure 4.1: DVFS and power cap performance impacts for `vips`.

4.1.2 The Challenges of Actuating Power

DVFS is being replaced with hardware power capping, but meeting performance targets with power caps instead of DVFS settings introduces new challenges. Figure 4.1 demonstrates how the compute-bound `vips` application's performance is affected by DVFS frequencies (Figure 4.1a) compared to processor power caps (Figure 4.1b) on our evaluation system. Three challenges are immediately apparent from the figures. First, DVFS produces a linear response in performance, but power capping is *non-linear*. Second, power capping has *diminishing returns*: as power increases, the change in performance becomes smaller and eventually stops increasing altogether. Third, *the range* of DVFS settings is much smaller than power settings: the ratio of the maximum to minimum DVFS setting is 2.75, but power capping has a ratio of over 6 (as can be seen from the x-axes).

The linear relationship between DVFS and performance makes it easy to apply textbook control-theoretic techniques to build a performance management system based on DVFS, and many examples exist in the literature [40, 46, 71, 81, 109, 117, 140]. With DVFS, control models assume that—for compute-bound applications—a 2× change in frequency produces a 2× change in performance. Applying the same techniques to build a performance management system based on power capping is more complicated. The major issue is that controllers based on time-invariant linear models will have varying error dependent on the current power cap. The simple solution is to build a linear model that never overestimates

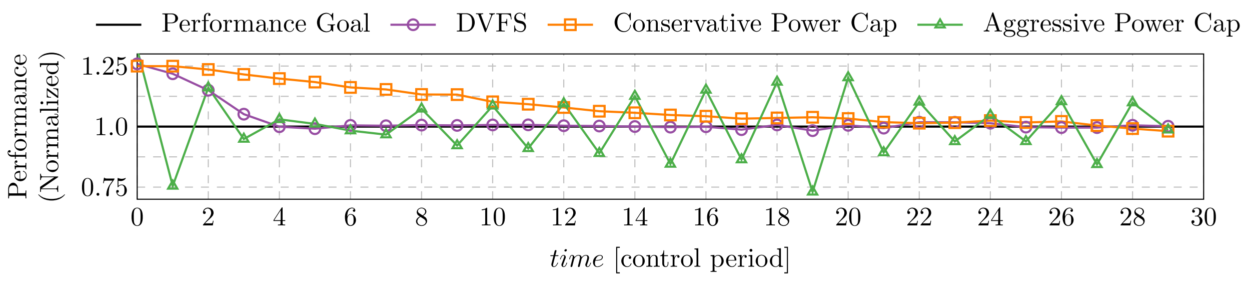


Figure 4.2: DVFS and power capping with linear models.

the relationship between power and speedup [40]. The downsides to this approach are: (1) a developer must know the maximum error for any application the system might run, and (2) the overestimate slows the control reaction.

Figure 4.2 shows the difference between a controller based on a linear DVFS model extracted from Figure 4.1a and two (one conservative and one aggressive) based on fitting time-invariant linear models to the power capping data from Figure 4.1b. All approaches start at the maximum DVFS or power setting and must bring performance down to the required level while minimizing energy. Figure 4.2 shows the DVFS controller quickly reaches the desired performance, but the conservative power capping controller is much slower to react. The conservative approach never violates the performance requirement, but its slow reaction wastes energy. The aggressive approach overreacts, oscillating around the performance target instead of settling on it. These results demonstrate how sensitive power capping approaches can be to their input models. The next section describes an adaptive control design for meeting performance goals with power capping that overcomes the difficulties highlighted by this example without requiring a user-specified model.

4.2 A General Power Capping Design

CoPPer's goal is to provide soft performance guarantees, with the competing goal of keeping power as low as possible. To achieve the best energy consumption, a power capping

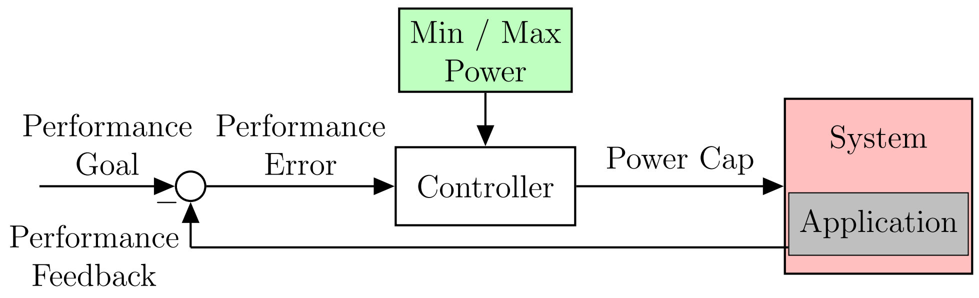


Figure 4.3: CoPPer's feedback control design.

framework for meeting performance targets must not allocate more power than is actually needed by an application. CoPPer uses an adaptive control-theoretic approach to meet soft real-time performance constraints and employs a *gain limit* to proactively reduce power consumption when it determines that power is over-allocated. For maximum portability, CoPPer is independent of any particular system, application, and power capping implementation.

4.2.1 Adaptive Controller Formulation

Figure 4.3 presents CoPPer's feedback control design. CoPPer requires three pieces of information at runtime: (1) the soft *performance goal*, (2) *performance feedback*, and (3) the **minimum and maximum power** that the **system** allows. A user provides CoPPer with the performance goal, R_{ref}. At runtime, the application measures its own performance, $r_m(t)$, which it provides to CoPPer. The minimum and maximum power caps, U_{min} and U_{max}, are system properties.

The **controller** first computes the *performance error*, $e_r(t)$, between the desired and measured performance:

$$e_r(t) = R_{ref} - r_m(t) \tag{4.1}$$

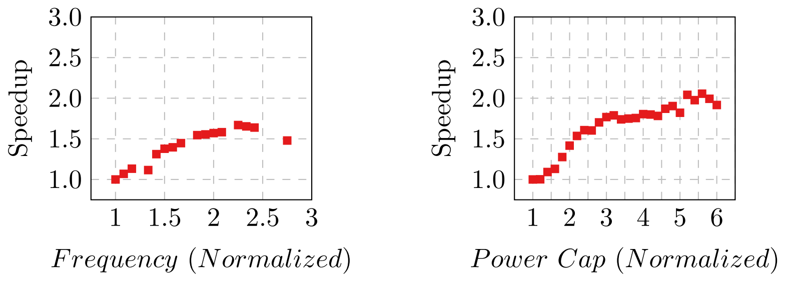
It then computes a *speedup* value as:

$$s(t) = max\left(1, gain(t) \cdot min\left(s(t-1) + \frac{e_r(t)}{b_r(t)}, \frac{U_{max}}{U_{min}}\right)\right) \tag{4.2}$$

where $s(t-1)$ is the speedup signal generated in the previous iteration, $b_r(t)$ is the base speed estimate produced by a Kalman filter [131], and $gain(t)$ (where $0 < gain(t) \leq 1$) is a time-varying value that scales the control response. The gain is described in more detail shortly (Chapter 4.2.2). Other feedback controllers use similar formulations, but without the gain [62, 63]. Clamping between 1 and $\frac{U_{max}}{U_{min}}$ prevents slow controller response if R_{ref} was previously unachievable (in control terminology, this is an *anti-windup* mechanism). Finally, the new *power cap* to be applied is computed as:

$$u(t) = U_{min} \cdot s(t) \tag{4.3}$$

In Chapter 4.1.2, Figure 4.1b shows that, unlike with DVFS frequencies, a scalable compute-bound application's speedup is a non-linear function of the power cap. Figure 4.2 then illustrates how formulating a controller based on a linear model can cause the controller to converge very slowly, or to not converge at all. CoPPer overcomes this limitation by treating the application's base speed, $b_r(t)$, as a time-varying value and estimating it with a Kalman filter. In practice, this approach is analogous to estimating a non-linear curve with a series of tangent lines, each with slope $b_r(t)$. Thus, CoPPer's use of the Kalman filter allows it to overcome the problematic non-linear relationship between performance and power caps.



(a) DVFS. (b) Power cap.

Figure 4.4: DVFS and power cap performance impacts for `HOP`.

4.2.2 The Gain Limit

Applications exhibit different performance behaviors when controlling DVFS frequencies or power caps. In many cases, allocating more power to an application does not actually increase its performance. With `vips` in Figure 4.1, performance scales (linearly) as higher DVFS frequencies are applied, but eventually a predefined maximum allowable frequency is reached. Performance increases (non-linearly) as higher power caps are applied, until a little beyond the system's thermal design power (TDP). Unfortunately, TDP is not a reliable indicator of the maximum power cap that can be applied efficiently. Figure 4.4 demonstrates the power cap/performance behavior of the `HOP` application, where performance levels off well before the system TDP, and begins exhibiting performance unpredictability before beginning to achieve some small increases in average performance again once the power cap is *greater than* than the system TDP. These behaviors make it difficult for controllers to efficiently meet performance targets. CoPPer uses a *gain limit* to avoid over-allocating power when it is not useful, *e.g.,* for unachievable performance targets in `vips` or moderate targets in `HOP`.

The gain limit is used in computing the $gain(t)$ term in Equation 4.2. This term is initially 1, and will remain so until the controller settles. Based on the performance history, gain may be reduced to lower the speedup when CoPPer detects that the extra speedup is not beneficial (and thus wasting power). In general, the convex properties of performance/power tradeoff spaces ensure that reducing the speedup never increases power consumption, and in most cases reduces it.

Intuitively, if the performance error value computed in Equation 4.1 is low, then the system has converged to the performance target and the speedup signal should remain where it is. However, if error values are high but the *difference* in error values between iterations is low, the controller has settled, but the performance target is not achievable. It may then be beneficial to reduce the speedup, and thus the power. Speedup is reduced by setting gain to:

$$gain(t) = 1 - \alpha_c \cdot e_{ns}(t) \cdot \Delta e_{ns}(t) \tag{4.4}$$

where α_c ($0 \leq \alpha_c < 1$) is the *gain limit*, a constant that controls how low the gain can go, and:

$$e_n(t) = \frac{|e_r(t)|}{R_{ref}} \tag{4.5}$$

$$\Delta e_n(t) = |e_n(t-1) - e_n(t)| \tag{4.6}$$

$$e_{ns}(t) = 1 - \frac{1}{e_n(t)+1} \tag{4.7}$$

$$\Delta e_{ns}(t) = \frac{1}{\Delta e_n(t)+1} \tag{4.8}$$

Since R_{ref} is the performance target, $e_n(t)$ is the absolute normalized performance error. $\Delta e_n(t)$ in Equation 4.6 is the absolute change in $e_n(t)$ since the previous iteration. Equations 4.7 and 4.8 compute values $e_{ns}(t)$ and $\Delta e_{ns}(t)$, which determine how much impact $e_n(t)$ and $\Delta e_n(t)$ have on reducing the speedup. Both $e_{ns}(t)$ and $\Delta e_{ns}(t)$ lay in the unit circle. As normalized performance error $e_n(t)$ approaches 0, $e_{ns}(t)$ also approaches 0, which reduces the impact of the gain limit in Equation 4.4. Conversely, if the error is high, $e_{ns}(t)$ approaches 1 and the gain limit will have a greater impact on the speedup. As the change in normalized performance error $\Delta e_n(t)$ approaches 0, $\Delta e_{ns}(t)$ approaches 1, thus increasing the gain limit's impact in Equation 4.4. Conversely, if the change in error is high, $\Delta e_{ns}(t)$ approaches 0 and the gain limit will have less impact on the speedup. Therefore, Equation 4.4 reduces the speedup signal by a factor of at most α_c, with the greatest change in speedup occurring when the absolute performance error $e_n(t)$ is high and the absolute change in error $\Delta e_n(t)$ is low. Setting $\alpha_c = 0$ disables the gain limit entirely, corresponding to $gain(t) = 1$ in Equation 4.2.

Note that in Equation 4.2, the upper clamping is performed prior to applying the gain. High performance errors force a speedup value too high for the gain to overcome if the speedup is not clamped first. In cases of high performance error, the gain limit's effectiveness is also constrained by the accuracy of U_{max} and U_{min}. If speedup is not clamped at a reasonable upper value, the gain limit must be quite high to overcome the inaccuracy.

Recall that CoPPer holds $gain(t)$ at 1 until the controller converges. Until then, the controller is an adaptive deadbeat control system that retains the corresponding control-theoretic guarantees [40]. Specifically, it can converge to the desired performance in as little as one control iteration. At that point, CoPPer computes Equations 4.4–4.8, which may change $gain(t)$. In control-theoretic terms, a non-zero gain is equivalent to adding a zero to the characteristic equation of the closed loop system. Given the definition of $gain(t)$, it is equivalent to a zero greater than 1, which moves the controller in the opposite direction from the feedback signal. Normally, this would be undesirable behavior, but this is exactly the behavior we want when we are no longer seeing performance improvements by increasing the power cap.

4.2.3 Using CoPPer

Application-level feedback provides high-level metrics that are conducive to goal-oriented software and has been shown to provide a more reliable measure of application progress than low-level metrics like performance counters or memory bandwidth [58]. Many applications that are subject to performance constraints already measure performance and integrate with runtime DVFS controllers to meet performance targets. A performance target is any positive real value that makes sense for the application, and can conceivably be configured from any number of sources, *e.g.,* a command line parameter, a configuration file, or dynamically via a software interface. At desired time or work intervals called *window periods*, the application measures its performance and calls the controller. Developers for this class of applications already perform these tasks, so all that remains is to replace function calls to an existing DVFS controller with those for CoPPer.

CoPPer is designed to be independent of any particular system, application, and power capping implementation. It is initialized with a performance target, the minimum and maximum allowed power values, and the starting power cap.[1] After each window period, the `copper_adapt` function is called with an identifier and the current application performance. This function returns the new power cap, which is then applied to the system. For example:

```
// initialize CoPPer
copper cop;
copper_init(&cop, perf_goal, pwr_min, pwr_max, pwr_start);
// application main loop
for (i = 1; i <= NUM_LOOPS; i++) {
  do_application_work();
  if (i % window_size == 0) {
    // end of window period
    perf = get_window_performance();
    powercap = copper_adapt(&cop, i, perf);
    apply_powercap(powercap);
  }
}
```

Listing 4.1: Using CoPPer to compute and apply power caps.

The underlined functions simply replace the existing DVFS-related ones.

1. An accurate starting power cap is optional, but knowing the initial configuration helps the controller to settle as quickly as possible.

4.3 Experimental Design

This section details the platform and applications used to evaluate CoPPer. We then quantify application performance variability, which directly impacts an application's ability to be controlled. Finally, we describe the control approaches CoPPer is evaluated against.

4.3.1 Testing Platform

We evaluate CoPPer using a server-class system with dual Xeon E5-2690 processors running Ubuntu 14.04 LTS with Linux kernel 3.13.0. Each processor socket has its own memory controller and supports 8 physical cores and 8 HyperThreads, for a total of 32 virtual cores in the system. The processors support 15 DVFS settings, 1.2–2.9 GHz, plus TurboBoost up to 3.3 GHz. We bind applications to all 32 virtual cores and interleave with both memory controllers using `numactl`. To record runtime power behavior, we read energy from each socket's Model-Specific Register using the EnergyMon API and sum the two values [59, 115] (see also Appendix A.2). Energy measurements are only used to evaluate CoPPer, they are *not* required in practice.

RAPL supports a variety of zones, otherwise known as power planes, for controlling the power limits on different hardware components. The *Core* power plane controls the power cap for all cores in a socket while the *Uncore* power plane, typically only available on client hardware, controls the power cap for on-board graphics hardware. The *DRAM* power plane, only available on server-class hardware,[2] controls the power for main memory. The *Package* power plane encompasses the *Core* the *Uncore* power planes, the last-level cache, and memory controllers. Intel Skylake processors support the *PSys* (or *Platform*) power plane for managing the entire system-on-chip [37]. The *Package* and *PSys* zones both support *long_term* and *short_term* power constraints; other zones only support a single

2. We have also seen the DRAM zone on client hardware, but the RAPL documentation does not back up this observation.

constraint. See the RAPL documentation for a more complete description.

For our experiments, we enable TurboBoost and set power caps for the RAPL *short_term* constraint at the *Package* level. We keep the system's default time window of 7812.5 μs. On our evaluation system, RAPL specifies a minimum of 51 W per socket, although we found that 25 W per socket is a more reasonable lower bound. RAPL specifies a maximum of 215 W per socket, and although this is far more than a socket ever actually seems to use, it is still an acceptable maximum value since CoPPer will not allocate more power than is necessary. Therefore, we specify a 50 W lower bound and a 430 W upper bound to be split evenly between the sockets, *e.g.,* a system power cap of 200 W sets a 100 W limit on each socket. A more complex power partitioning scheme could conceivably be applied to systems with heterogeneous architectures or otherwise unbalanced behavior between power capped components, but that is beyond the scope of this work.

To apply RAPL power caps, we provide an easy-to-use tool called `RAPLCap` (see Appendix A.3), but we stress again that power capping implementations are independent of CoPPer. For example, the `apply_powercap` function used in Listing 4.1 might be:

```c
raplcap rc;

void apply_powercap(double powercap) {
  uint32_t n = raplcap_get_num_sockets(&rc);
  raplcap_limit rl = {
    // time window = 0 keeps current time window
    .seconds = 0.0,
    // share powercap evenly across sockets
    .watts = powercap / (double) n
  };
  for (uint32_t i = 0; i < n; i++) {
    raplcap_set_limits(i, &rc, RAPLCAP_ZONE_PACKAGE, NULL, &rl);
  }
}
```

Listing 4.2: Applying a power cap with RAPLCap.

The RAPL interface sets a limit on average power consumption over a time window, with

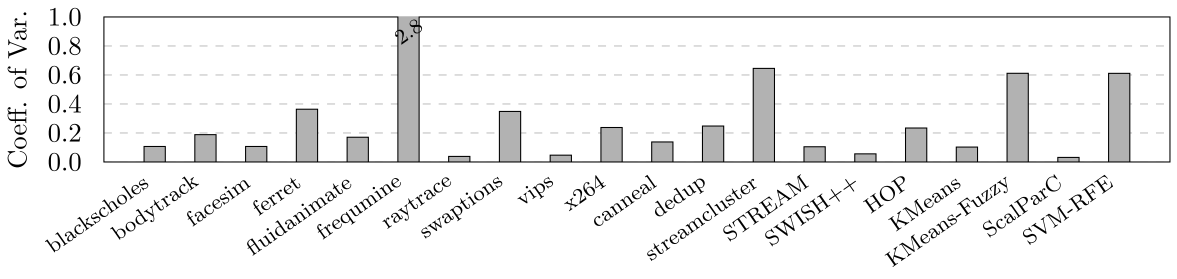


Figure 4.5: Application job performance variability.

hardware controlling DVFS and power allocation within that window.

4.3.2 Applications and Inputs

Our experiments use applications from the PARSEC benchmark suite [14], MineBench benchmark suite [100], `STREAM` [94], and `SWISH++` [89]. We instrument the applications with a modified Heartbeats interface to measure performance, as real applications would [56]. PARSEC provides a wide variety of parallel applications that exhibit different ranges of performance and power behavior. MineBench provides a representative set of data mining applications, some of which support parallel execution. `STREAM` is a synthetic benchmark that stresses main memory and represents memory-bound applications. `SWISH++` is a file indexing and search engine. All inputs are delivered with or generated directly from the benchmark sources, with the exception of `dedup` which uses a publicly available disc image, and `raytrace` and `x264` which are from standard test sequences.

Table 4.1 presents our application configurations. Applications contain top-level loops, where each loop iteration completes a *job*. For very high performance applications, we batch a fixed number of iterations into a single job. As is common in feedback control systems, CoPPer executes at fixed job intervals called *window periods*. For example, CoPPer will compute a new power cap every 50 video frames in `x264`. We select window periods that are sufficiently long to prevent changing power caps too frequently, but small enough to allow CoPPer to adapt behavior in reasonably responsive times intervals.

Table 4.1: Application inputs and configuration details.

Application	Input	Jobs	Window Size
blackscholes	10 million options	400	20
bodytrack	sequenceB	261	20
canneal	2500000.nets	384	50
dedup	FC-6-x86_64-disc1.iso	421	50
facesim	Storytelling	100	20
ferret	corel:lsh	2,000	50
fluidanimate	in_500K.fluid out.fluid	160	20
freqmine	webdocs_250k.dat	140	40
raytrace	thai_statue.obj	200	20
streamcluster	2000000 (points)	200	20
swaptions	self-generated	1,000	50
x264	rush_hour	500	50
vips	orion_18000x18000.v	795	50
STREAM	self-generated	1,000	50
SWISH++	swish++-large-126M.index	2,900	50
HOP	particles_0_64	400	50
KMeans	edge	200	20
KMeans-Fuzzy	edge	200	20
ScalParC	F26-A32-D250K.tab	200	20
SVM-RFE	outData.txt	400	50

Applications exhibit variability in their performance behavior, with some behaving more predictably than others. Figure 4.5 demonstrates the behavior of the applications used in this evaluation when running in an uncontrolled setting (default system power caps). Naturally, better predictability typically results in lower error in meeting performance targets, as will be shown in Chapter 4.4.1.

If a window period is too short, the overhead of changing system settings combined with signal noise resulting from application variability prevents performance controllers from converging. Both actuation overhead and performance predictability can be improved by increasing the size of window periods. However, longer window periods mean that it takes controllers more time to converge on a target and reduces their responsiveness to changes in application or system behavior. In practice, choosing a window period is dependent on the both the application and system properties, as well as the deployment context.

We set a lower bound of 20 jobs per window period for the benefit of the sophisticated DVFS-only controller we evaluate against (discussed shortly). It divides the discrete number of jobs in a window period between two DVFS frequencies, so that the average window performance precisely meets the target. Thus a minimum of 20 jobs ensures less than 5% performance error in its scheduling. CoPPer suffers no such scheduling limitation, but we make the accommodation for the DVFS controller anyway in an effort to provide the most challenging comparison possible. In our evaluation, we are also sometimes limited by the size of the test inputs, *e.g.,* for `facesim`. As a result, an execution might contain only a few window periods during which the controllers can possibly be converged since they must have an initial observation window period followed by an initial period of adjustment.

4.3.3 Execution and Analysis

Prior to performing the evaluation, we first *characterize* the behavior of all applications by running them without any control at each of the evaluation system's DVFS frequencies and measuring their performance and power behavior. From these characterizations, which are only required for our analysis and not in practice, we derive an *oracle* with perfect foreknowledge of job behavior and no computation overhead. For each performance target used in the evaluation, the oracle produces a DVFS schedule that never misses a performance goal. Until a job is complete, it runs at the highest-performance frequency for the application (which is not always the highest frequency or the TurboBoost setting). Once a job completes, the oracle then sets the most energy-efficient frequency and aggressively places the processor cores in a low-power sleep state, with no delay or transition overhead. The oracle is thus an ideal *performance* DVFS governor and a good baseline for comparison—modern Linux systems provide a real performance governor, which is not as efficient as our oracle. With the exception of unachievable performance targets, we compare the energy consumption of all the executions in the evaluation against this oracle to determine their relative energy efficiency.

The different analyses compare CoPPer with various *gain limits* against a simple linear DVFS controller (based on modification of an existing controller for meeting power requirements [81]) and a sophisticated DVFS controller that meets soft performance constraints and schedules for optimal energy consumption (POET) [63]. The simple linear DVFS controller estimates the ratio of control change (a primitive application-specific base speed estimate) in the first iteration, whereas a textbook controller requires this value at initialization and is rarely as good as our runtime estimate. It then uses an $O(log(n))$ algorithm to map speedup values to the lowest of n DVFS frequencies that meets the performance target, which is also an improvement over textbook approaches in that limiting the controller to discrete DVFS settings prevents oscillations.

The sophisticated DVFS controller requires a system model that maps DVFS frequencies to speedup and powerup values. It uses this model to divide window periods between two DVFS settings to meet a performance target precisely, where the schedule is computed using an $O(n^2)$ algorithm to find the best energy consumption subject to the performance constraint. This approach results in low error and often higher energy efficiency than the simple approach, as Chapter 4.4.1 will show. We also use a much more efficient DVFS actuation function than the sophisticated DVFS controller comes with, reducing its actuation overhead by two orders of magnitude.

We provide the DVFS controllers with linear models (*e.g.,* a $2\times$ change in frequency results in a $2\times$ change in performance and power), which works quite well on our evaluation system. It should be noted, however, that poor models can cause slow, oscillating, non-convergent, or otherwise unpredictable behavior in model-driven controllers. In contrast to the DVFS controllers, CoPPer does not require a model, only the minimum and maximum power values, and therefore can run in constant $O(1)$ time.

4.4 Evaluation

This section evaluates CoPPer. We first show that CoPPer achieves similar error and higher energy efficiency than both a simple and a sophisticated DVFS controller. Next, we show that CoPPer improves energy efficiency for memory-bound applications and that its gain limit avoids over-allocating power when performance targets are not achievable. We then show the advantages of using an adaptive controller by demonstrating its behavior for an application with a phased input and in response to interference caused by multiple concurrent applications. Finally, we quantify CoPPer's runtime overhead.

4.4.1 Efficiently Meeting Performance Goals

We begin by quantifying CoPPer's ability to achieve high energy efficiency while meeting soft performance goals. We use *gain limits* of 0.0 (disabled) and 0.5. For this analysis, we consider the steady-state behavior of the controllers. Therefore, each controller is initialized with the same $s(t)$ value for $t = 0$ (see Equation 4.2 in Chapter 4.2.1).

For each application, we define and evaluate three different performance goals which specify how much to favor performance over energy consumption: high, medium, and low. We define the high performance goal to mean that the application must maintain at least 90% of top performance. The medium and low goals correspond to maintaining 70% and 50% of top performance respectively. While it is likely that most users would want high performance, we include the others to demonstrate CoPPer's ability to meet a range of different goals. We note that actual performance values provided to CoPPer are application-specific, as described in Chapter 4.2.3 (*i.e.,* not a percentage).

We quantify the ability to meet performance goals with low energy with two metrics:

- *Energy efficiency* is the ratio of the ideal *performance* governor's energy consumption (as computed by the oracle) to the actual energy consumption achieved.
- *Mean Absolute Percentage Error* (MAPE) quantifies the error between the desired per-

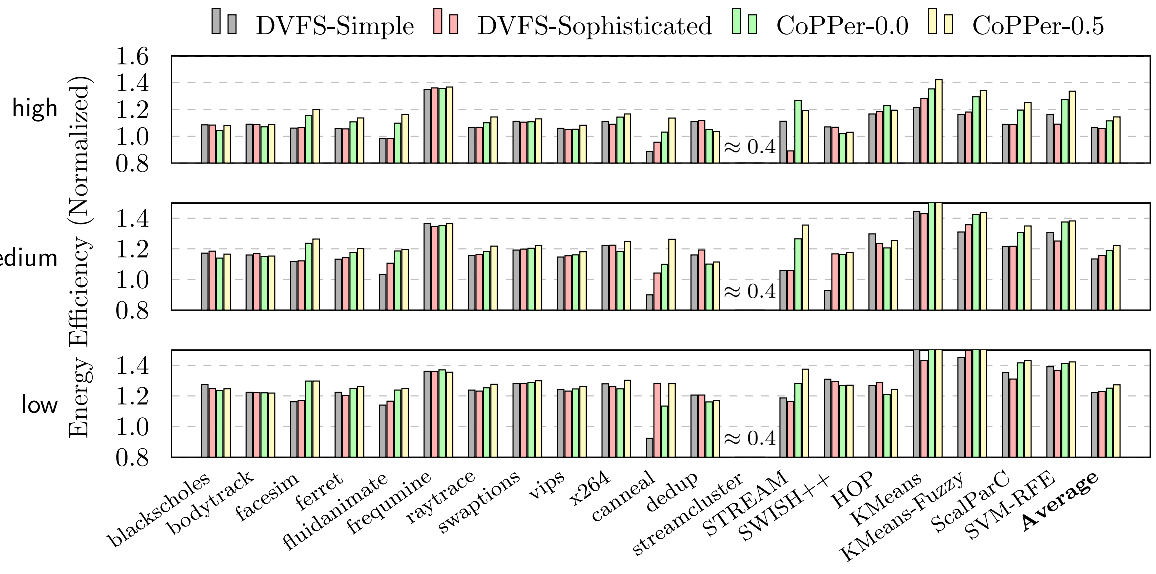


Figure 4.6: Application energy efficiency for DVFS controllers and CoPPer, with and without a gain limit, for **high**, **medium**, and **low** performance targets (higher is better). Results are normalized to an ideal *performance* DVFS governor.

formance and the achieved performance; it is a standard metric for evaluating control systems [40].

MAPE computes the performance error for an application with n jobs and a performance goal of R_{ref} as:

$$\text{MAPE} = 100\% \cdot \frac{1}{n} \sum_{i=1}^{n} \begin{cases} r_m(i) < R_{ref} : & \frac{R_{ref} - r_m(i)}{R_{ref}} \\ r_m(i) \geq R_{ref} : & 0 \end{cases} \tag{4.9}$$

where $r_m(i)$ is the achieved performance for the i-th job. Each failure to achieve the performance target increases MAPE by an amount relative to how badly the target was missed.

Figures 4.6 and 4.7 present the energy efficiency and MAPE values for all applications and targets. Despite the challenges described in Chapter 4.1.2 (*e.g.,* non-linearity and larger range in the power cap/performance relationship), CoPPer achieves higher energy efficiency and similar MAPE compared to both the simple and sophisticated DVFS controllers for most applications and performance targets. Compared to the sophisticated DVFS controller,

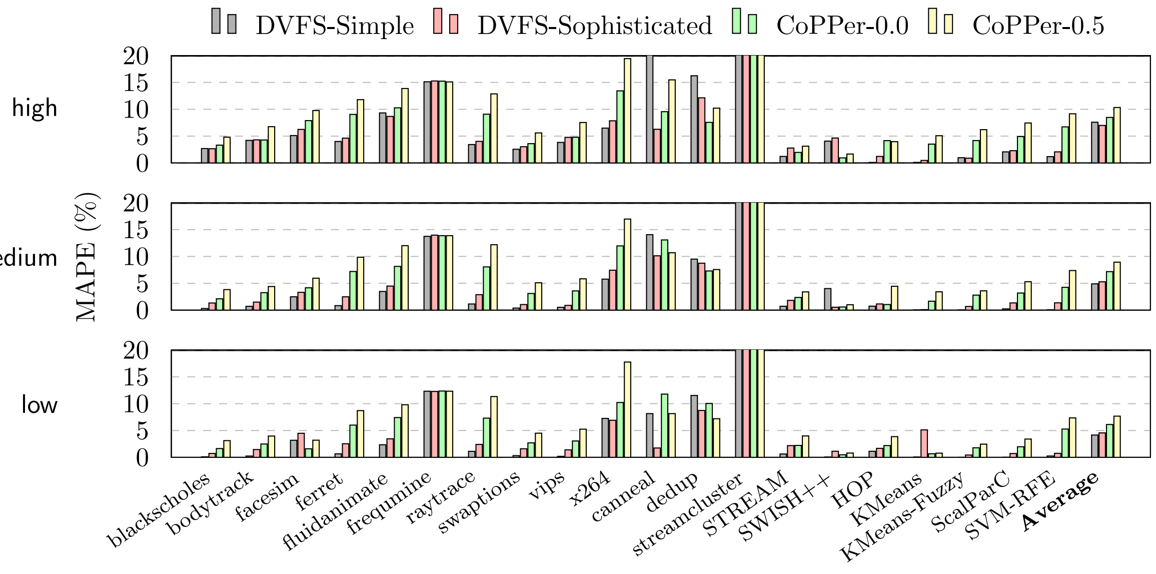


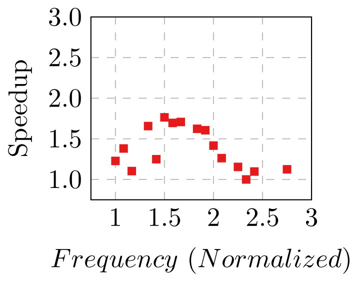
Figure 4.7: Application performance error for DVFS controllers and CoPPer, with and without a gain limit, for for high, medium, and low performance targets (lower is better).

Table 4.2: Energy efficiency of CoPPer with gain limits of 0.0 and 0.5 compared to the sophisticated DVFS controller (higher is better).

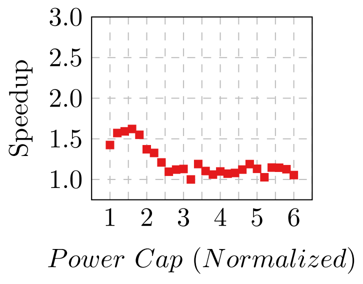
	Energy Efficiency vs DVFS	
Performance	**CoPPer-0.0**	**CoPPer-0.5**
high	1.05	1.08
medium	1.03	1.06
low	1.02	1.04
Average	**1.03**	**1.06**

CoPPer is on average 3% more energy-efficient with no gain limit and a 6% more efficient with gain limit 0.5.

Table 4.2 shows the average energy efficiency gains of CoPPer compared to the sophisticated DVFS controller for different performance goals. Note that CoPPer's energy efficiency gains increase as the performance goal increases. For high goals, the DVFS approach must use TurboBoost, which is inefficient. CoPPer, however, can set a power cap that allows the performance goal to be met and leave the decision to Turbo or not to hardware, which has more information about whether that choice is appropriate—exactly the motivation to move DVFS control to hardware and allow software to cap power instead.



(a) DVFS.



(b) Power cap.

Figure 4.8: DVFS and power cap performance impacts for `streamcluster`.

`Freqmine` and `streamcluster` are outliers for both energy efficiency and MAPE. `Freqmine` is composed of repeated jobs, but its behavior is not predictable with feedback—the application uses a recursive algorithm that causes job performance to continually slow as it progresses. This behavior is quantified by its high job variability as shown in Figure 4.5 in Chapter 4.3.2. `Streamcluster` exhibits a performance/power tradeoff space that does not scale well beyond a fairly low DVFS setting or power cap, as demonstrated in Figure 4.8. In fact, its performance degrades dramatically as resource allocation increases. Even CoPPer's gain limit cannot adapt since it detects a change in performance when trying to reduce the power cap. The ideal *performance* DVFS governor (the oracle) knows not to allocate higher frequencies since it has access to the application-specific characterizations, but our runtime controllers do not have this information. If we were to specify `streamcluster`-specific power cap ranges for CoPPer, the results would be similar to the other applications; the DVFS controllers, however, would need entirely new models.

4.4.2 Controlling Memory-bound Applications

Our benchmark set contains 6 memory-bound applications: `KMeans`, `KMeans-Fuzzy`, `ScalParC`, `STREAM`, `streamcluster`, and `SVM-RFE`. CoPPer achieves noticeably higher energy efficiency for these applications than with DVFS. Table 4.3 summarizes the average ratio of energy efficiencies across all performance targets for these, comparing CoPPer with and without a gain limit to the sophisticated DVFS controller.

Table 4.3: Energy efficiency of CoPPer with gain limits of 0.0 and 0.5 compared to the sophisticated DVFS controller for memory-bound applications (higher is better).

	Energy Efficiency vs DVFS	
Performance	**CoPPer-0.0**	**CoPPer-0.5**
high	1.15	1.18
medium	1.09	1.11
low	1.06	1.08
Average	**1.10**	**1.12**

We see that even without a gain limit, CoPPer already improves on the sophisticated DVFS controller's energy efficiency by 10% on average. With a gain limit of 0.5, the improvement rises to 12%. These are significant energy savings. CoPPer performs especially well compared to DVFS with these memory-bound applications for the higher performance targets. Again, even without a gain limit, CoPPer improves energy efficiency by 15% for the high performance target. DVFS can benefit from TurboBoost at high performance targets for many applications, but the higher DVFS frequencies also result in unnecessarily high energy consumption for memory-bound applications. By setting power caps instead of forcing DVFS frequencies, CoPPer achieves better energy savings by allowing the processor to scale frequencies more quickly between computational and memory-intensive periods. The gain limit provides significant energy savings for memory-bound applications with only a small loss in performance. In general, we advocate the use of 0.5 gain limit in practice since it produces almost no difference in MAPE, but can provide significant energy savings for memory-bound applications.

4.4.3 Reducing Power for Unachievable Goals

Sometimes performance targets simply are not achievable. This could be due to a user requesting too much from an application given the available processing capability, or the application may just want to run as fast as possible. When a performance target is unachievable, a naive resource controller will continue to increase resource allocations like DVFS frequen-

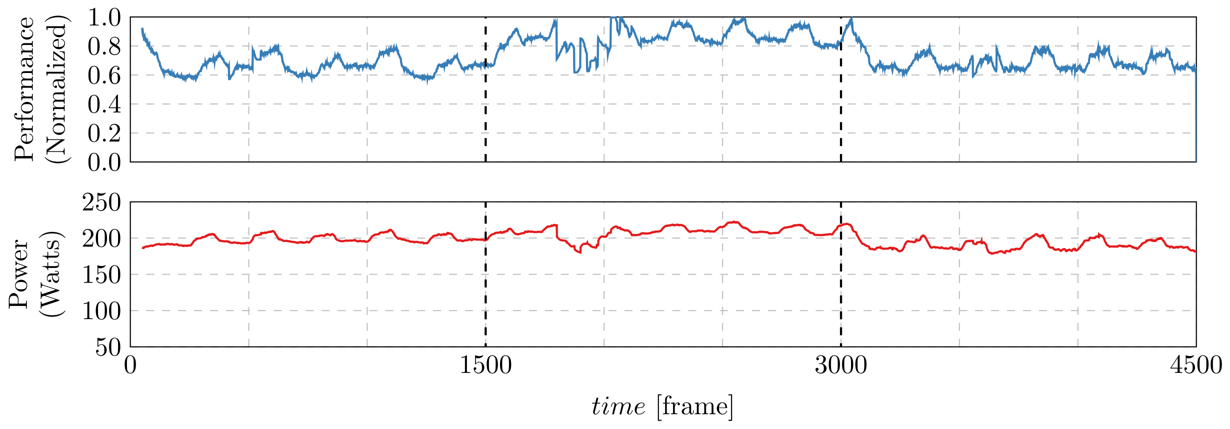
Table 4.4: Average energy efficiency for unachievable performance targets, normalized to the sophisticated DVFS controller.

Controller	Energy Efficiency
DVFS-Sophisticated	1.00
CoPPer-0.0	1.00
CoPPer-0.2	1.01
CoPPer-0.5	1.10
CoPPer-0.6	1.16
CoPPer-0.8	1.29
CoPPer-0.99	1.46

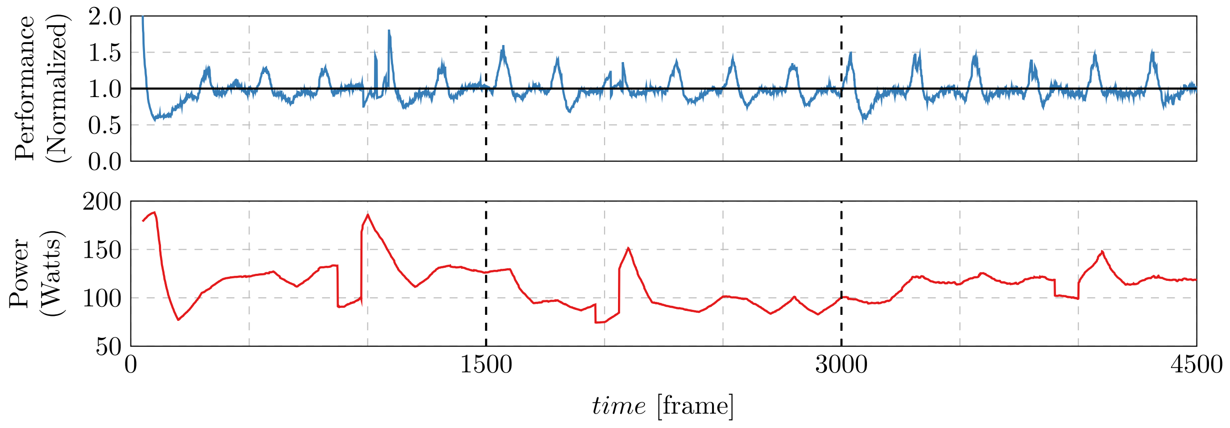
cies or power caps in an attempt to improve performance, needlessly wasting energy. In this part of the evaluation, we demonstrate that CoPPer's *gain limit* helps avoid this pitfall. In Chapter 4.2.2, we explained that the gain limit's effectiveness is constrained by the accuracy of the minimum and maximum power values. For this experiment, we use a more reasonable (and safer) maximum power limit, the evaluation system's TDP of 270 W.

For each application, we set an unrealistically high performance target—1000× greater than what the system can actually achieve. We then execute both the sophisticated DVFS controller and CoPPer with a range of gain limit values. As the performance target is not actually achievable, MAPE is meaningless. Instead, we normalize energy efficiency to the sophisticated DVFS controller. Table 4.4 presents the results for select gain limits.

The DVFS controller runs in the TurboBoost setting for the entirety of each execution. We also verified that the simple DVFS controller and the evaluation system's real Linux performance governor achieved nearly identical results as the sophisticated controller. As should be expected, CoPPer without a gain limit behaves similarly. With gain limits enabled, CoPPer achieves increasingly better energy efficiency for small increases in application runtime. A 0.5 gain limit demonstrates a significant improvement in energy efficiency—a 10% increase over the sophisticated DVFS controller. A gain limit of 0.99 increases energy efficiency by 46% over the DVFS controller, though suffers a 20% loss in performance as it pulls power consumption back too far.



(a) Uncontrolled behavior.



(b) Meeting a performance target with CoPPer.

Figure 4.9: Processing `x264` input with distinct phases.

These results clearly demonstrate the gain limit's advantages. For achievable performance targets, it has minimal impact on controller behavior. For unachievable targets, it can greatly improve energy efficiency over the sophisticated DVFS controller.

4.4.4 Adapting to Runtime Changes

This experiment demonstrates CoPPer's ability to respond to changes in application behavior at runtime. We run `x264` with a video input that exhibits three distinct levels of encoding difficulty. Figure 4.9a demonstrates the uncontrolled behavior of the input, with performance normalized to the maximum achieved. Dashed vertical lines denote where phase changes occur. The first phase has the lowest average performance and is therefore the most difficult to encode, followed closely by the third phase. The second phase has the highest performance, meaning it is the easiest part of the video to encode. Frames that are easier to encode offer an opportunity to save energy when meeting a performance target, as fewer resources are needed to satisfy the constraint. In the uncontrolled execution, power is consistently high as no changes to resource allocations are being made.

Figure 4.9b shows the time series for CoPPer with a gain limit of 0.0 for the medium performance target and a window size of 50 frames. Performance is normalized to the target. Now the performance remains mostly fixed (per the constraint), whereas the power consumption fluctuates as the power cap changes. Power values are now inversely proportional to the performance behavior seen in Figure 4.9a since CoPPer is able to reduce power consumption to save energy during phases of easier encoding. Of course, the actual power consumption recorded for any given frame does not necessarily match the power cap—it may be lower.

The fluctuations around the performance target in each phase are a result of input variability. The uncontrolled execution in Figure 4.9a testifies to the variability's presence in the input. We see similar behavior in other applications to varying degrees, but this visual clarifies where performance error (MAPE) comes from, and why some applications are difficult to control. Still, CoPPer meets the performance target with high energy efficiency and low error. For this execution, energy efficiency is 1.25 compared to the ideal *performance* governor (the oracle) and MAPE is 6.48%.

4.4.5 Multiple Applications

This section evaluates CoPPer's resilience to interference from another application. We begin the experiment by launching each application with a performance target. Roughly halfway through each execution, we launch a second application which was randomly selected from

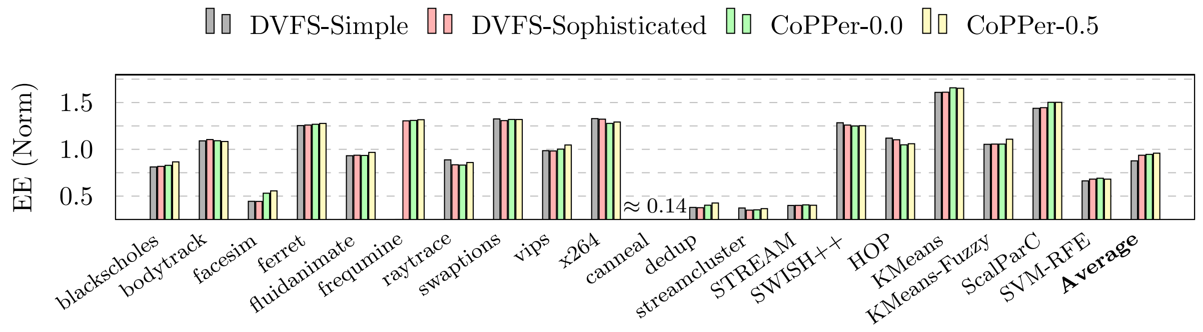


Figure 4.10: Application energy efficiency for DVFS controllers and CoPPer, with and without a gain limit, under interference by a second application (higher is better). Results are normalized to an ideal *performance* DVFS governor.

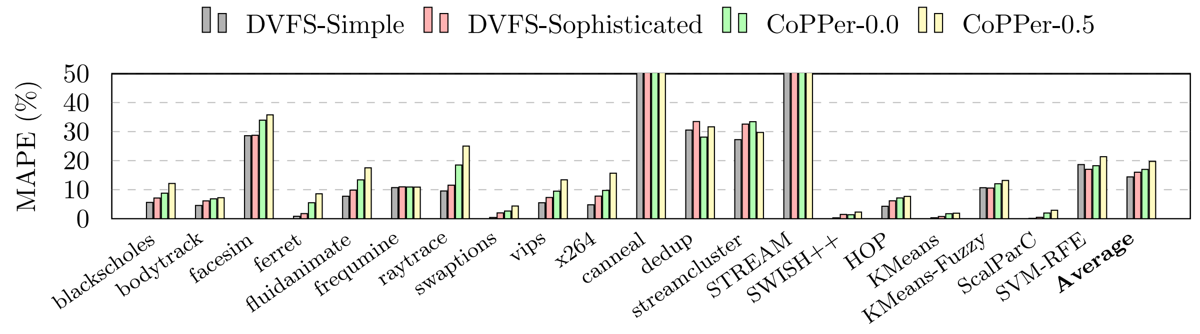


Figure 4.11: Application performance error for DVFS controllers and CoPPer, with and without a gain limit, under interference by a second application (lower is better).

the PARSEC benchmark suite. The second application does not perform any DVFS or power control, but introduces interference into the system by consuming resources.

Figures 4.10 and 4.11 present the energy efficiency and MAPE results for each application. As should be expected, MAPE is higher than in previous experiments given that there is significant disturbance to system resources, which also makes the application more difficult to control even when the controller recognizes the disturbance and adapts to it. Some applications (*e.g.,* `facesim`, `canneal`, `dedup`, `streamcluster`, `STREAM`, and `SVM-RFE`) were not able to achieve the performance target after the second application is started, simply because there were not sufficient resources remaining in the system. Instead, the controller makes a best effort. These applications drag down the average energy efficiency, which otherwise remains high like in the single application analysis (*e.g.,* `KMeans and ScalParC`). Still, the average across all applications is nearly as good as the ideal *performance* governor would achieve in the absence of interference.

4.4.6 Overhead Analysis

This section quantifies the runtime overhead of changing power caps and DVFS frequencies.

The granularity for configuring DVFS settings varies between systems. In some cases cores are individually configurable, sometimes they are grouped into multiple voltage domains per socket, and other times simply managed at a socket level. As the number of cores increases, the overhead of managing their DVFS settings in software can become prohibitively high. Naively, we have to set 32 DVFS frequencies, or one for each logical core. We reduce this to 16 by limiting ourselves to the physical cores, which requires additional knowledge of the processor topology and virtual core number to physical core mapping. In practice, having to discover this mapping is an additional burden on developers looking to reduce their overhead when using DVFS. The RAPL interface exposes power capping with socket granularity, which allows the hardware to more efficiently manage voltage and frequency settings at smaller component and time scales than software can. As a result, less work needs to be performed by software. For example, our evaluation system requires us to set only two power caps—one per socket.

We compare the overhead of power capping and setting DVFS frequencies on our evaluation system. Each test is run one million times, with the average of 5 runs presented. Power capping alternates between applying the lowest and highest power caps (50 and 430 W, split between the two processor sockets) in each iteration, and achieves an average iteration time of **15.6 μs**. DVFS alternates between setting the minimum and maximum frequencies (1.2 GHz and TurboBoost) on all 16 physical cores, achieving an average iteration time of **118.5 μs**. As noted in Chapter 4.3.3, our experiments use a better actuation function than the sophisticated DVFS controller comes with, so this average overhead is much improved over the double-digit millisecond overhead the controller previously achieved [64]. Yet, the overhead imposed by power capping is still another order of magnitude lower than the improved DVFS overhead. Limiting software to coarse-grained power management at a socket level provides clear runtime benefits over more fine-grained DVFS management.

These values do not even include the overhead of actually computing the correct settings to apply for each window period. The simple DVFS controller runs in $O(log(n))$ time, where n is the number of available DVFS frequencies, and actuates once per window period. The sophisticated DVFS controller we compared CoPPer with computes a true optimal DVFS schedule which requires $O(n^2)$ time, and usually requires actuating two different DVFS settings per window period. This optimal algorithm is what allows the DVFS controller to achieve such good energy efficiency results, and remains practical due to the small number of DVFS settings. CoPPer runs in constant $O(1)$ time and requires only one actuation per window period.

Our evaluation demonstrates that CoPPer's approach to meeting software performance goals achieves better energy efficiency and similar error to both a simple and a state-of-the-art optimal DVFS controller. Furthermore, CoPPer is much lower overhead in both computing resource schedules and applying changes to system settings.

CHAPTER 5

CEES: CLASSIFICATION OF ENERGY-EFFICIENT SETTINGS

This chapter describes CEES (**C**lassification of **E**nergy-**e**fficient **S**ettings), focusing on the High Performance Computing (HPC) domain. Energy and power consumption will be first-class constraints in future exascale systems [125]. For example, these systems are predicted to have strict operating budgets of approximately 20 MW [74]. Additionally, exascale operating systems are required to actively "decrease the cost per scientific insight" [12]. Minimizing application energy, even if runtime is increased, can: (1) greatly reduce the cost (in Joules) of scientific insight, and (2) increase application throughput in hardware over-provisioned, power-constrained systems—such as future exascale computers [104, 118]—by allowing more applications to execute on a cluster simultaneously.

We propose to dynamically monitor application and system behavior at runtime and use machine learning classification to predict the most energy-efficient resource settings as the application executes. The proposed approach has two distinct benefits. First, it can be applied to new applications without modifying either the classifier or application, as it does not require application-level instrumentation or hooks. Second, classification's low overhead makes it suitable for running in-situ on compute nodes.

5.1 Motivation

We first describe how maximizing energy efficiency is distinct from maximizing performance or minimizing power consumption. As such, finding the most energy-efficient system setting requires new techniques. We then discuss the challenges of learning a model for mapping application/system behavior to energy-efficient system settings.

5.1.1 Energy Efficiency is a Unique Challenge

Energy efficiency is the ratio of work completed per unit of energy consumed. For example, HPC clusters can formulate energy efficiency as the ratio of applications completed per Joule of energy used. If an application execution represents some unit of scientific insight, then this metric expresses the cost of scientific insights in terms of operational costs (energy consumption). Thus, maximizing energy efficiency directly reduces the cost of scientific insight by minimizing application energy consumption.

Some recent energy-aware approaches in HPC reduce energy consumption while maintaining application performance [70, 93, 116]. As these approaches do not trade application speed, they reduce energy strictly by reducing power. While it is tempting to use the same approaches to minimize energy by simply removing the constraint on application runtime, failing to account for the impact of lower-power resource settings on execution time can result in worse energy consumption. Also, while raising processor clock speed typically reduces application runtime, the requisite increase in power results in increased energy consumption (recall Equation 1.1 in Chapter 1.2). In short, maximizing energy efficiency *does not* mean using as little power as possible *or* running as fast as possible—it requires finding the optimal tradeoff between execution time and power consumption.

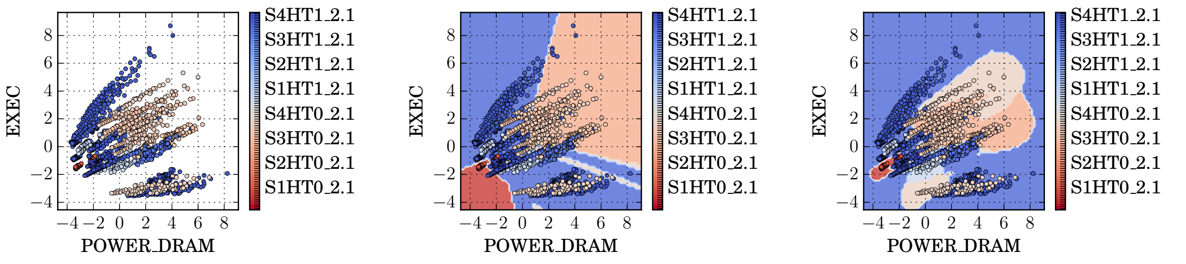
Typical resource management approaches in HPC clusters primarily attempt to optimize application completion time, only worrying about power consumption to the extent that the total cluster power constraint is not violated, and ignoring energy consumption altogether. Although there are various techniques for assigning jobs to nodes, minimizing their runtime is usually achieved by running the individual compute nodes as fast as possible, *i.e.,* the *race-to-idle* heuristic. Using *race-to-idle* makes resource scheduling on a node easy, but it is never as energy-efficient as a more intelligent approach that understands how system settings affect performance and power tradeoffs on a per-application basis [73]. *Racing* to finish a job as fast as possible minimizes execution time, but the increase in power dwarfs the time savings, resulting in high energy consumption.

In addition to reducing costs, maximizing energy efficiency will also improve overall system throughput in power-constrained environments. In hardware over-provisioned clusters, the hardware can draw more total power than the infrastructure can physically deliver to the system if nodes use *race-to-idle* [118]. Effectively, power is now the primary factor limiting cluster size and throughput, not available compute resources. More aggressively trading performance and power means that optimizing energy efficiency increases total cluster throughput by: (1) allowing more applications to run in parallel due to lower per-application power consumption, while (2) still considering application runtime due to its impact on energy consumption. Therefore, new resource management approaches are required that focus solely on minimizing energy to reduce the cost of science (in Joules), even if application runtime is increased.

5.1.2 Learning Energy Efficiency

Identifying energy-efficient combinations of resource settings is challenging as there is no universal best setting for a system—it depends on the application and its configuration, even varying for different inputs. Even in a parallel system with homogeneous nodes and perfectly uniform application behavior, optimal settings can vary dramatically due to manufacturing variation [2]. Furthermore, many applications progress through different *phases* during execution. For example, an application may transition between compute and memory-intensive processing, causing the most energy-efficient setting to change during runtime. It is therefore not optimal to choose a single setting statically when launching the application, necessitating a dynamic approach instead.

We explore learning approaches that adjust system settings to minimize energy. Specifically, we tune socket allocation, the use of HyperThreads, and processor DVFS. The learning component picks settings for the combination of these resources such that, at any point dur-



(a) Training data only.

(b) SVM classifier with a linear kernel, recall=0.456.

(c) SVM classifier with a RBF kernel, recall=0.710.

Figure 5.1: Training data and learned decision boundaries for two SVM classifiers using two primary features.

ing the application execution, the system is operating in its most energy-efficient state.

Our goal is for the learner to find the most energy-efficient setting without requiring any application-level changes. Therefore, we use existing hardware performance counters as *features*. The learner then determines a function that maps some subset of these features into the most energy-efficient system setting. However, system settings in modern compute nodes have very complicated interactions, so it is essential to use learning mechanisms that can produce accurate mappings despite this complexity.

We briefly illustrate this complexity using just two features—the performance counters `POWER_DRAM` (a measure of memory usage) and `EXEC` (a measure of CPU usage). Figure 5.1a visualizes the behavior of our training data (21 common HPC benchmarks) with respect to normalized performance counter values. Each data point is the average recorded `POWER_DRAM` and `EXEC` behavior for a training application running in a single resource configuration on our evaluation system. There are 88 unique resource configurations, or possible *labels*, accounting for different combinations of the socket count `S`, whether HyperThreads `HT` are used, and the DVFS frequency (*e.g.,* 2.1 GHz). All 88 data points belonging to a particular application are assigned *the same label*—the resource configuration with the best average energy efficiency for that application; *i.e.,* all 88 data points for an application have the same color. With this labeling, a good learner will recognize suboptimal behavior and produce the most energy-efficient settings to use instead. The problem is clearly complex—no intuitive pattern emerges that obviously maps CPU and memory usage into the most energy-efficient system settings.

To successfully map these features into accurate predictions, the learner must be able to handle this complexity, which not all learning mechanisms can. Consider Figure 5.1b, which illustrates the accuracy of a Support Vector Machine (SVM) classifier using a linear kernel. The shaded regions indicate the label (system settings) that the SVM classifier predicts for a range of feature values; the training data is overlaid for comparison. This linear SVM's *recall*—the fraction of system settings that are accurately predicted when simply replaying the training data—is only 45.6%, a clear indication that this classifier is not effective. In contrast, Figure 5.1c demonstrates a SVM with a radial basis function (RBF) kernel, which achieves 71.0% recall—better, though perhaps still with room for further improvement. Note that recall is simpler than cross-validation, since the classifier already saw the same input during supervised training. If a classifier cannot perform accurate recall, it will likely perform even worse in deployment, when it is exposed to data it has not yet seen.

5.2 Classifying System Settings

We propose to predict energy-efficient settings at runtime, without the overhead of estimating the behavior of all possible system settings before producing a result. As with many prior works in managing power/energy in HPC systems, we use hardware performance counters to measure application and system behavior.

Figure 5.2 demonstrates our proposed approach. While an **application** runs on the **compute node**, hardware performance counters are polled in the background at regular intervals. For our experiments, we use the PCM tool to collect performance counter data [103]. The *PCM sample* data is scaled, then processed using Principal Component Analysis (PCA) to identify which fields correlate well with energy efficiency. We also use **feature**

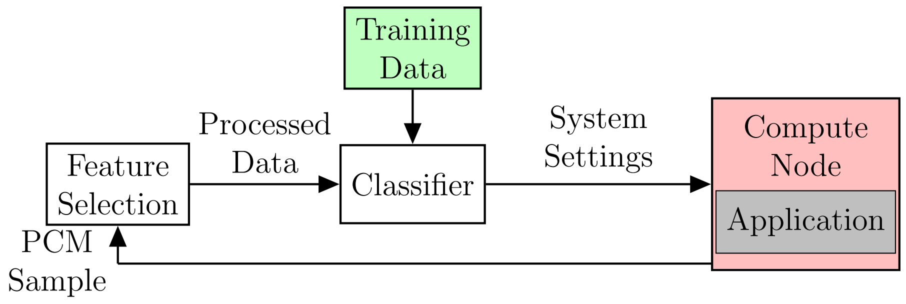


Figure 5.2: Design for using machine learning classifiers to predict energy-efficient system settings based on performance counter behavior.

selection to limit the number of hardware counters used by the classifier to reduce runtime overhead (evaluated in Chapter 5.4.2). Using this *processed data*, the **classifier** predicts the most energy-efficient *system settings* to use, which are then actuated on the system. The process then repeats at the next time interval.

5.2.1 Training Data

A classifier must be trained before it can be used. To collect training data, we characterize the behavior of benchmark applications on the target platform by running them in all possible settings and collect hardware performance counter results. In other words, if there are N different allowable settings, each application is executed N times, or once in each setting. For M training applications, there are $N \times M$ feature vectors in the training set.

Characterization can be time-consuming, but only needs to be done once for a platform and can be completed in a reasonable period of time by keeping application execution times short. Choosing applications that are representative of those that will be used on the system improves the likelihood that the classifier can accurately predict settings during runtime. Additionally, training applications should be chosen that exercise the system hardware components in different patterns to cover a wide range of possible use cases.

Application energy efficiency (EE) is defined as the amount of work completed per unit of energy (J) used. In general, and in our evaluation, a *complete application execution* is the measure of completed work. Our classifiers, like many prior works that use performance counters, need a measure to quantify application progress *during* runtime. Low-level hardware performance counters do not have a metric for quantifying true application progress (work completed), but prior works have successfully used instructions retired by the system (INST). Some prior works have even attempted to measure only those instructions considered useful in measuring application progress, *e.g.,* ignoring spinlocks or parallelization/synchronization instructions [26]. Using INST is an imperfect solution for optimizing total application energy efficiency, but suffices, as the evaluation will demonstrate. The classifier then uses the following formulation *as a proxy* to quantify energy efficiency for training:

$$EE = \frac{INST}{J} \tag{5.1}$$

For each of M applications, the N feature vectors are labeled using that application's most energy-efficient setting. Thus the classifier learns both efficient and inefficient behavior so that it produces an efficient prediction when it observes similar runtime behavior:

$$\mathbf{FeatVec}_{mn} \mapsto \operatorname*{argmax}_{i \in N} EE_i \quad \forall m \in M,\ \forall n \in N \tag{5.2}$$

The total instruction count is not fixed for an application execution—the count typically increases with the application execution time, *e.g.,* due to background processes like PCM or kernel tasks. As such, it is important to note that labeling the most energy-efficient configuration using Equation 5.2 is different than using the minimum-energy execution from the characterization. There are two reasons for using instruction count in computing energy efficiency. First, energy efficiency can be quantified at any point during an execution, making it a useful metric for runtime behavior analysis. Second and more importantly, Equation 5.1 is a function of events happening *at the time.* Total energy requires knowing all events that *will* happen during the application execution, introducing the possibility that the classifiers

Table 5.1: Overview of system-level performance counters.

Performance Counter	**Description**
EXEC	Instructions per nominal CPU cycle
IPC	Instructions per cycle
FREQ	Frequency relative to nominal CPU frequency
AFREQ	FREQ, excluding the time when the CPU is sleeping
L3MISS	L3 cache line misses
L2MISS	L2 cache line misses
L3HIT	L3 Cache hit ratio
L2HIT	L2 Cache hit ratio
L3MPI	L3 Cache misses per instruction
L2MPI	L2 Cache misses per instruction
READ	Memory read traffic
WRITE	Memory write traffic
INST	Number of instructions retired
Proc Energy	Energy consumed by the processor
DRAM Energy	Energy consumed by the DRAM

might be learning something about application input rather than the way hardware events correspond to energy consumption. Accounting for instructions helps to avoid this pitfall.

5.2.2 Performance Counters

Performance counter metrics are available at different levels of granularity—system, socket, and core. For simplicity, we limit ourselves to system-wide data. Table 5.1 lists the performance counters that we process for our experiments [132].

Performance counters are also translated into rates (as needed), which is necessary in order to vary the sampling interval without scaling values (evaluated in Chapter 5.4.2). Because we use PCM to collect performance counter metrics, we read more hardware counters than we process (Table 5.1). In practice, a fielded solution would reduce overhead by only reading and processing counters that are used. Most prior works aggressively limit the hardware counters they access, both to reduce sampling overhead and to reduce computation in their models [5, 24, 85].

5.3 Experimental Design

This section describes our experimental setup, including the evaluation system, applications used for training and evaluation, and the classification algorithms tested. We perform our evaluation on a quad-socket, 80-physical core system with 512 GB DRAM running Ubuntu Linux 14.04 LTS. With HyperThreads, there are 160 compute threads available, *i.e.,* 20 physical and 20 virtual on each socket. We use Linux kernel 4.4.0 with the intel_pstate driver disabled so that we can use the userspace DVFS governor.

5.3.1 Training Applications

Training applications are representative of HPC workloads and are selected from the NAS Parallel Benchmarks [8], Lawrence Livermore Lab's Co-design benchmarks (`AMG` [50], `Kripke` [76], `LULESH` [72], `Quicksilver` [77]), and Argonne's CESAR Proxy-apps (`XSBench` [127], `RSBench` [126]). Other applications include `CoMD` [78], Berkeley's `HPGMG-FV` [3], a partial differential equation solver (`jacobi`), and `STREAM` [94]. Additionally, we include a characterization of system idling behavior. In total, there are 21 unique applications/characterizations used for training. Each application is configured to run with 160 threads to match the number of compute cores on the evaluation system and to use NUMA memory interleaving.

Each performance counter is used as a *feature* for classification. Performance counter values are converted to rates, normalized, then PCA is applied. Figure 5.3 quantifies the percentage of variance contributed by the top 10 performance counters in the feature space for our system and training applications, accounting for 99% of the total variance.

5.3.2 Evaluation Applications

We evaluate classifier performance on four complex bioinformatic HPC applications: `HipMer` [44], `IDBA` [106], `Megahit` [83] and `metaSPAdes` [102]. These four are the leading applications

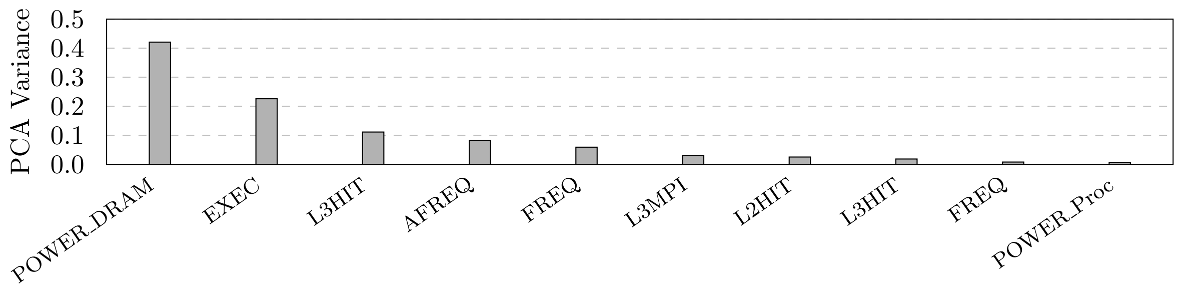


Figure 5.3: PCA explained variance ratio for the top 10 performance counters, accounting for 99% of variance.

that perform *de novo* genome assembly, which is one of the most computationally challenging bioinformatics problems. There are HPC systems, such as NERSC's Genepool [101], that are predominantly used to concurrently execute many single-node applications, such as genome assemblers and comparative analysis [34]. The datasets can be very large (for example, metagenomes can have raw sequence datasets on the order of terabytes), and the algorithms are hard to scale efficiently. For example, of the four applications, only `HipMer` scales efficiently to distributed memory systems. Consequently, *de novo* assemblers are typically run on very large shared-memory systems, with at least 0.5 TB of memory. Thus our experimental platform is typical of the sort of hardware that would be used for these kinds of applications. A single execution assembling a large genome could take days on our large evaluation system, making it prohibitive to exhaustively characterize the full range of allowable settings, motivating the need for a learning-based solution.

These four applications can also be considered representative of a wider range of HPC applications. They implement complex pipelines, with multiple different stages that require different resource adaptations to run energy-efficiently—some are compute-intensive, some I/O-intensive, and some communication-intensive. Exactly which stages are used and how much they contribute to the performance depends to a large degree on the program configuration and the input datasets. Although they are solving the same problem, they are implemented in very different ways, with different programming languages, different algorithms and different data flows. For example, HipMer can have up to 20 different stages, whereas Megahit may have only a few. Overall, these applications provide a broad coverage of a range of different bioinformatics approaches (frequency counting, graph traversal, alignment, sorting, etc.). The applications are thus prime candidates for our approach of using classification to predict the appropriate settings at runtime.

Like with training applications, we configure each evaluation application to run with 160 threads, except for metaSPAdes which uses 80 threads. We fix the application inputs and configurations for our evaluation, although they support a variety of configurations and inputs which affect performance and energy consumption behavior. Solely to use as a baseline in our evaluation, we perform a time-consuming DVFS-only characterization by running them in each DVFS setting with all 160 virtual cores allocated (for metaSPAdes, all 80 physical cores are used as the baseline instead). From these results we derive a DVFS *Oracle* which knows, for each application, the most energy-efficient static DVFS frequency—*each application has a different most energy-efficient static setting.* Note also that this data is *not* used in classifier training.

5.3.3 Classification Algorithms

This section identifies and briefly describes the classifiers we use. For data processing and classifier implementations, we use the Machine Learning toolkit scikit-learn, version 0.19.1 [105]. We do not attempt to optimize algorithm performance or prediction accuracy by tuning any implementation knobs. We demonstrate the feasibility of using classification without the need for fine-tuning.

With the labeled training data in Figure 5.1a, it is clear that the a useful classification algorithm must handle a complex space. We validate this hypothesis by performing an offline analysis of 15 algorithms using our training data. One of the metrics we looked at was training data recall, which we present in Figure 5.4. The algorithms are: AdaBoost

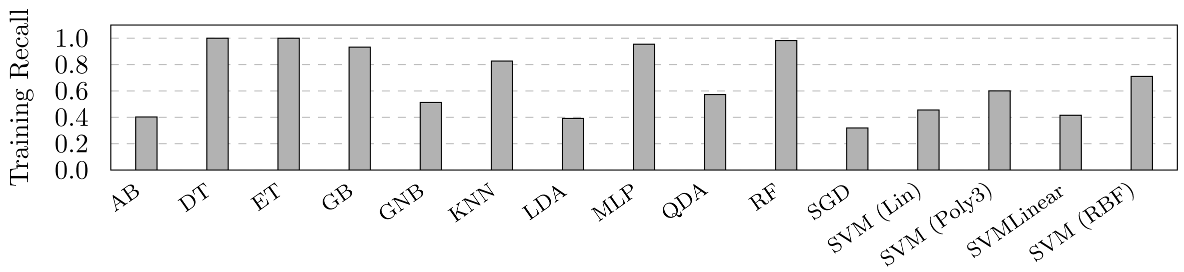


Figure 5.4: Training data recall for 15 classification algorithms (higher is better, 1 is optimal).

(AB), Decision Tree (DT), Extra Trees (ET), Gradient Boosting (GB), Gaussian Naive Bayes (GNB), K-Nearest Neighbors (KNN), Linear Discriminant Analysis (LDA), Multi-layer Perceptron (MLP), Quadratic Discriminant Analysis (QDA), Random Forest (RF), Stochastic Gradient Descent (SGD), Support Vector Machine with a linear kernel (SVM (Lin)), SVM with a degree=3 polynomial kernel (SVM (Poly3)), SVM with a different linear kernel (SVMLinear), and SVM with a radial basis function kernel (SVM (RBF)). Some approaches have poor recall and thus are not likely to perform well in practice.

When a mis-prediction occurs, the impact on energy consumption varies—some are sub-optimal but still reduce energy consumption over the naive *race-to-idle* heuristic, while others actually make it worse. We tested SVM (Lin) (Figure 5.1b) online—in most cases it reduced energy, but consumed 22% *more energy* than *race-to-idle* with the `Megahit` application, demonstrating the importance of selecting a good classifier. We choose five promising algorithms that support a range of different classification techniques to use in the evaluation:

1. ET: an extremely randomized decision tree, similar to Random Forest [45].
2. GB: fits multiple regression trees on the negative gradient of the deviance loss function [42, 105].
3. KNN: a simple majority vote of the nearest neighbors from the training data ($k = 5$, by default).
4. MLP: a neural network optimizing the log-loss function using *lbfgs*, a *tanh* activation function, and four layers [51].

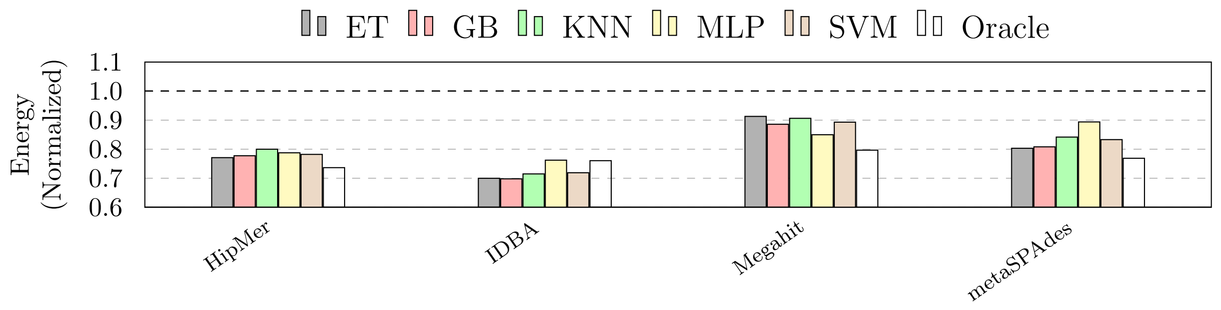


Figure 5.5: Average application energy consumption using four performance counters at 5 second prediction intervals (lower is better).

5. SVM: a maximum margin classifier using a radial basis function (RBF) kernel.

The only classifier with non-default configurations is MLP, in order to add additional layers to better represent deep learners, and to specify the activation and solver functions. Our evaluation compares the energy consumption of real application executions when using these five classifiers in different configurations.

5.4 Evaluation

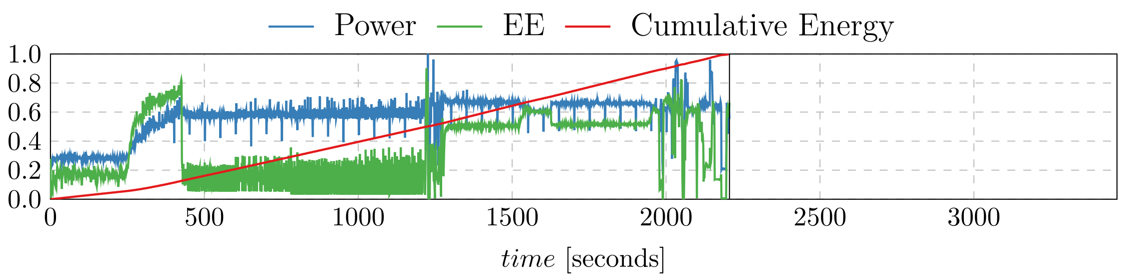
We now evaluate the effectiveness of using machine learning classifiers to predict energy-efficient system settings during application runtime. First, we compare against the naive *race-to-idle* heuristic, *i.e.,* all sockets allocated with HyperThreads and DVFS set to TurboBoost, and against a DVFS *Oracle*. We then quantify how varying sampling/prediction intervals and the number and types of features impacts classifier effectiveness. We discuss how application dynamics affect the classifiers and evaluate the overhead of different parts of the classifier runtime. Finally, we explore using separate classifiers for taskset and DVFS, then discuss limitations.

5.4.1 Reducing Energy Consumption

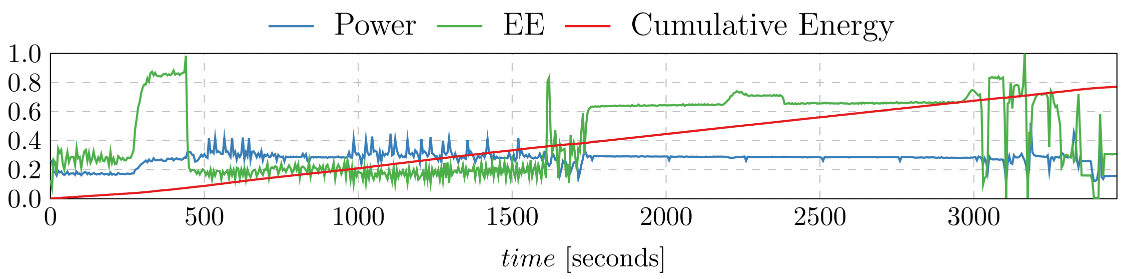
We first demonstrate how effective runtime classification is at reducing energy consumption. Figure 5.5 demonstrates results for each application, normalized to the *race-to-idle* setting (dashed line) for each. On average across all four applications, energy consumption is reduced by 19.3%—there is a 31.3% increase in runtime and 38.5% reduction in average power consumption. Extrapolating from these empirical results, a hardware overprovisioned, power-constrained cluster could increase throughput by 24%; the *Oracle* could achieve up to 30% increase in throughput. The most extreme results are from the `IDBA` and `Megahit` applications. For `IDBA`, each classifier outperforms the *Oracle*—the average energy savings is 28.1%, and as much as 30.2% when using the GB classifier. Conversely, for `Megahit`, the classifiers have the highest energy consumption over the *Oracle*, though still always better than *race-to-idle*. `Megahit` saves 11% energy on average—8.7% in the worst case with ET, and 15% at best with MLP. The behavior for these applications is discussed further in Chapter 5.4.3.

Computing the *Oracle* requires expensive offline characterization to determine the best static setting for each application and its input/configuration, making it impractical to determine for most applications and difficult to beat. It also does not have any overhead except for running PCM in the background. The *Oracle* can only be beat when an application does not require its full taskset (socket and HyperThreads) allocation to run efficiently. HPC applications are designed to parallelize well, meaning this is not the typical use case, but opportunities do occur, *e.g.,* during prolonged memory- or I/O-intensive phases.

To demonstrate how energy savings are achieved, Figure 5.6 shows the runtime, power, energy efficiency, and cumulative energy consumption for executions of the `HipMer` application (Y values are normalized across both time series for each metric). Figure 5.6a runs `HipMer` in the *race-to-idle* heuristic, and although the runtime is short, the high power also results in poor energy consumption. In contrast, Figure 5.6b demonstrates running with the



(a) `HipMer` in naive static *race-to-idle* heuristic.



(b) `HipMer` with ET classifier at 5 second intervals.

Figure 5.6: Execution time, power, energy efficiency, and energy consumption behavior for the `HipMer` application (a) without and (b) with classification.

ET classifier—the execution is 29% longer, but the 40% average power savings results in nearly 20% less energy consumption in total, despite the increase in runtime. Each figure shows clear phases in the execution, indicated by relatively long-term changes in both power and energy efficiency (per Equation 5.1, a function of instruction rate and power). Classifiers adapt to changes in feature values like instruction rate and power by changing their predictions to run the application more efficiently, thus decreasing total energy consumption.

5.4.2 Classifier Interval and Feature Selection

There are a variety classifier settings to configure at runtime. This section evaluates the classifiers' ability to reduce energy consumption by varying prediction interval and the number and types of features used.

The sampling and prediction interval dictates how quickly a classifier can respond to changes in application behavior, but also incurs overhead and affects a classifier's suscep-

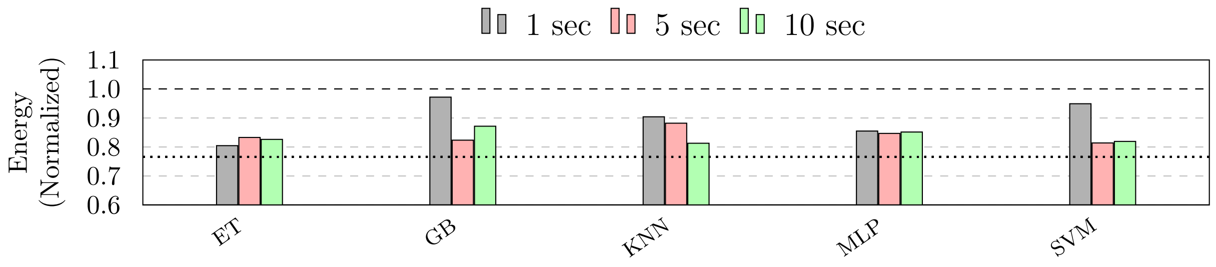
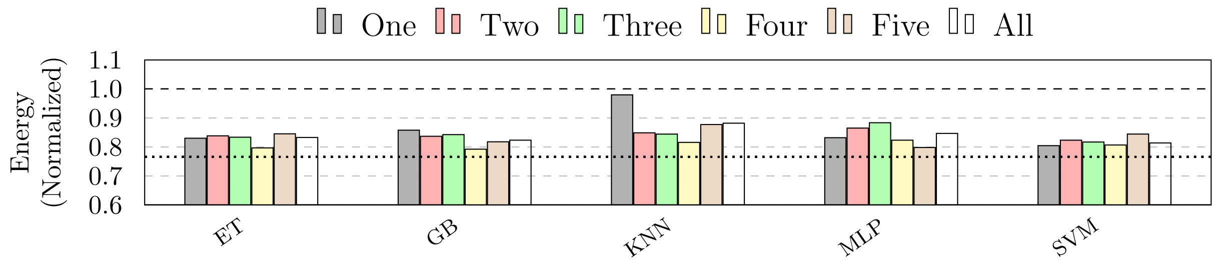


Figure 5.7: Average energy consumption for different sampling/prediction intervals (lower is better).

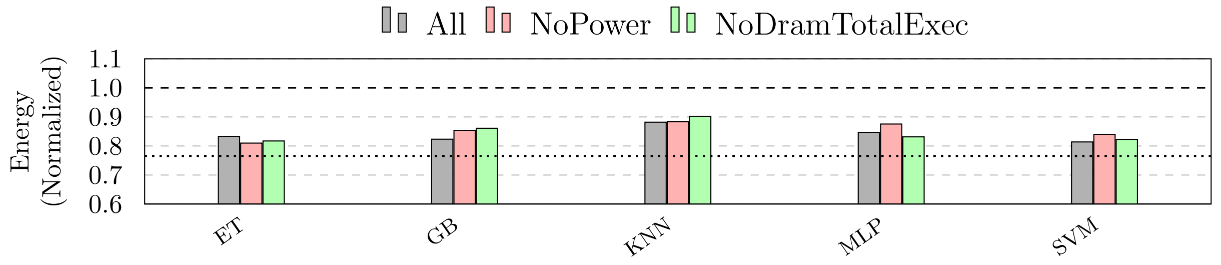
tibility to noise. Figure 5.7 shows the normalized energy consumption, averaged across applications, for each classifier at 1, 5, and 10 second intervals. The thin dotted line at 0.75 is the average normalized energy consumption for the *Oracle*. The 1 second interval is clearly the worst, only outperforming the others for the ET classifier. It is closer to the *race-to-idle* heuristic than the *Oracle* for GB and SVM, but still better than *race-to-idle*. Both the 5 and 10 second intervals perform well—although the 10 second interval does better for one particular classifier (KNN), the 5 second interval is similar on average and is more responsive. We elect to use the 5 second interval for the remainder of our experiments, as it provides a good tradeoff of energy efficiency and responsiveness to changing application behavior.

In Figure 5.3 (Chapter 5.3.1) we quantified the variation contribution of different features (performance counters). Now we evaluate how varying the features impacts classifier behavior since the number and types of features contribute to runtime complexity and prediction accuracy. We run with 5 second sampling/prediction intervals and present the results in Figure 5.8.

Figure 5.8a varies the number of features used. `POWER_DRAM` is the only feature used for *One*, `POWER_DRAM` and `EXEC` are used for *Two*, etc. For *All*, all 16 features are used. Using a single feature works surprisingly well for most classifiers, but performs poorly with KNN, which too often predicts settings that use only a single socket. It is a little surprising that a single feature can be so effective otherwise, but `POWER_DRAM` does account for 42% of the



(a) Average energy consumption based on number of features.



(b) Average energy consumption based on types of features.

Figure 5.8: Average energy consumption, varying the number and types of available features (lower is better).

explained variance, and is correlated with other performance counters like cache hit and miss rates. Using two or three features also works quite well, but four features is the best for most classifiers, accounting for 84% of the variance. The *Four* configuration was previously broken down by application in Chapter 5.4.1.

Figure 5.8b selectively removes available features. *NoPower* drops the `POWER_DRAM`, `POWER_Proc`, and `POWER_Total` features. This is a particularly important scenario, as not all systems have power/energy counters available at runtime (training data would have to be collected with additional power/energy instrumentation). When all three power-related features are ignored, `L3MISS_Rate` is used as the first feature, and `EXEC` remains the second. (If `POWER_DRAM` is excluded exclusively, `POWER_Total` becomes the primary feature instead.) Offline analysis also shows that `READ_Rate` can be used as the primary feature, depending on the exact training data. *NoDramTotalExec* drops `POWER_DRAM`, `POWER_Total`, and `EXEC`, making `AFREQ` and `WRITE_Rate` the primary and secondary features. In general, *All* outperforms the other two, but *NoPower* does slightly better for the ET classifier, as does

NoDramTotalExec with MLP. None of the configurations perform poorly, demonstrating that the classification approach is robust to different numbers and types of features.

5.4.3 Application Dynamics

The results thus far raise the question: why does the *Oracle* often perform better than the classifiers? First, we recall that the *Oracle* requires each application to be characterized across all DVFS settings with their current input and configuration, so is not practical to discover for most applications in practice. Additionally, the *Oracle* does not incur any overhead except sampling PCM (runtime overheads are quantified next in Chapter 5.4.4), and can only be beat when applications do not require all sockets and HyperThreads. In fact, for 3 of the 4 evaluation applications, classification usually achieves energy consumption close to that of the *Oracle*, and sometimes better.

Energy consumption penalties are incurred primarily by two factors: (1) overhead from changing settings, and (2) running in inefficient settings. While there is no particular threshold for determining when changes to performance counter values result in a new prediction, we quantify the difficulty in controlling applications by examining performance counters' coefficients of variation during the course of an application execution. Since applications move through phases, a high coefficient of variation for an execution does not cause problems on its own. The difficulty arises when performance counter values fluctuate rapidly enough to cause the classifiers to predict new system settings, constantly incurring actuation overhead and spending time in suboptimal settings.

In Figure 5.9, we examine `POWER_DRAM` and `EXEC` performance counter behaviors without any runtime actuation, with all sockets/cores allocated and using the maximum DVFS frequency without TurboBoost (to avoid fluctuations incurred when temporarily running in higher DVFS frequencies). For the first 10 minutes of each execution, we compute coefficient of variation over 5 second windows to demonstrate the variability that our classifiers see.

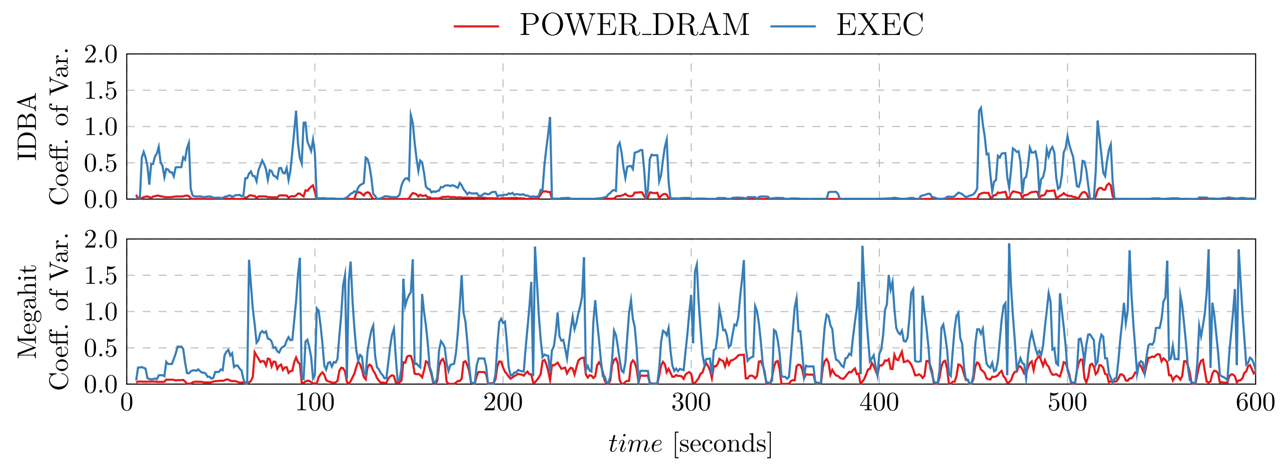


Figure 5.9: Runtime coefficient of variation over a 5 second sliding window (lower values indicate more stable application behavior).

Table 5.2: Frequency of settings changes for ET classifier in the *All* configuration.

Application	**Prediction**	**Taskset**	**DVFS**
HipMer	32.7%	25.6%	23.1%
IDBA	39.9%	37.4%	18.8%
Megahit	46.6%	37.9%	39.8%
metaSPAdes	41.9%	32.9%	28.6%

We present results for `IDBA`, which in most cases outperforms the *Oracle*, and for `Megahit`, which exhibits rapid fluctuations and consistently performs worse than the *Oracle* (but still better than *race-to-idle*). While `IDBA` does exhibit fluctuations, they are typically short-lived—most of the time the performance counter values are relatively stable. In contrast, `Megahit` is constantly noisy, resulting in frequent changes to system settings.

For example, Table 5.2 quantifies how frequently predictions and system settings change when using the ET classifier in the *All* configuration. `Megahit` has the highest frequency of changing predictions, but more importantly (and difficult to quantify), is how extreme the fluctuations' impacts are. With `IDBA`, the classifier prefers to switch between one socket and four sockets, always with HyperThreads enabled; DVFS is typically around 1.8 GHz, other times running in TurboBoost. With `Megahit`, the classifier chooses four sockets with

Table 5.3: Classification runtime overheads.

Stage	**Average Overhead (ms)**
Init – Scaling/PCA	4.50
Init – ET	29.01
Init – GB	3456.34
Init – KNN	3.50
Init – MLP	1278.30
Init – SVM	68.92
PCM Sampling	14.70
Scaling/PCA	0.08
Predict – ET	0.75
Predict – GB	0.36
Predict – KNN	0.39
Predict – MLP	0.14
Predict – SVM	0.13
Actuation	≈ 100

HyperThreads, four sockets without HyperThreads, and one socket with HyperThreads; DVFS is consistently around 1.7 or 1.8 GHz, periodically running in TurboBoost when all cores also being used. `Megahit` also poses an additional technical challenge—it appears to constantly destroy and spawn threads, which sometimes makes managing its taskset difficult.

In short, the dynamic behavior that applications exhibit can make predicting energy-efficient settings at runtime a challenge. Handling variability is not something the classifiers learn during training. However, any resource management system that is dependent on runtime feedback must find a good balance between responsiveness and susceptibility to noisy data—a challenge not limited to using classification.

5.4.4 Overhead

For classification to be practical, the runtime overhead must be reasonable. This section quantifies the overhead of different components/stages of the classification pipeline.

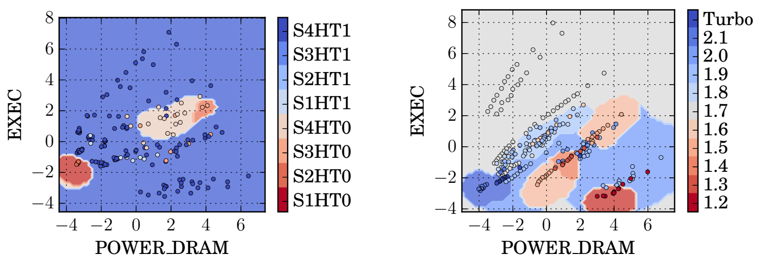
Table 5.3 breaks down the runtime overhead on our evaluation system. First, the data transformation (Scaling/PCA) and classifier must be initialized. Initialization incurs the highest overhead, but only needs to be performed once and is trivial compared to total application runtime. All classifiers we evaluated initialize within a few seconds, and some are much faster, initializing within a few dozen milliseconds.

To estimate the overhead incurred by reading the performance counters (PCM Sampling), we configure PCM to poll counters 1000 times at an interval that is faster than achievable (so it does not wait between reads), and time the execution. As a result, the average value for PCM also accounts for its initialization and teardown time.

Transforming the PCM data (Scaling/PCA) prior to running the classifier is extremely fast, averaging about 80 μs. Prediction overheads vary by classifier, but the average overheads for each are strictly less than 1 ms. The actuation overhead, while high compared to sampling and prediction, is a task that any runtime resource allocator that manages DVFS and taskset has to incur. Although dependent on application behavior, in half to three-fourths of cases we find that no actuation overhead is incurred since the classifier's prediction does not change at each sampling interval (Table 5.2). When there is a change, the overhead is on the order of 100 ms, which is consistent with observations on other server-class systems [64].

It is also import to note that PCM, the classifier, and the actuator run in parallel with the application under control, so the application does not stop making progress while the resource management components work.

As mentioned previously, we have not attempted to optimize any of these overhead results. We expect that it is readily possible to reduce overhead, particularly in reading performance counters (PCM samples more performance counters than we actually use) and integrating with other tools for managing taskset [124]. Additionally, if initialization time is a concern, classifier state could be stored and reloaded between executions, or the classifier could just run continuously on the system.



(a) Taskset only, recall=0.869. (b) DVFS only, recall=0.645.

Figure 5.10: Training data and learned decision boundaries for SVM when classifying for taskset and DVFS separately.

5.4.5 Using Separate Classifiers

Our analysis thus far has used a single classifier to predict a label that is a combination of both taskset (socket allocation, including HyperThreads) and the DVFS frequency (including TurboBoost). With 8 taskset options and 11 DVFS settings, there are 88 total possible labels for the classifier to choose from. After labeling the training data, only a subset of these 88 will actually be used, but the size of that subset could increase with additional training data.

An alternative approach is for one classifier to predict the taskset, and a separate classifier to predict the DVFS frequency. Figure 5.10 visualizes the training and recall for SVM (like Figure 5.1c in Chapter 5.1.2) using separate classifiers. Note that there are fewer data points, as DVFS is fixed at TurboBoost when training for taskset, and taskset is fixed at all sockets and HyperThreads when training for DVFS. Taskset's recall is quite good, while DVFS is not as good as before. There is, of course, additional overhead in running two classifiers instead of one, but Chapter 5.4.4 demonstrated that data processing overhead is not significant, and there is still only a single instance of PCM and the actuators running.

Figure 5.11 quantifies the behavior when we use two separate classifiers simultaneously. In each case, the same type of classifier is used for both taskset and DVFS prediction. In three of five cases, *Separate* actually outperforms the *Single* approach. However, it is also possible for *Separate* to perform rather poorly, as seen with the MLP classifier. The

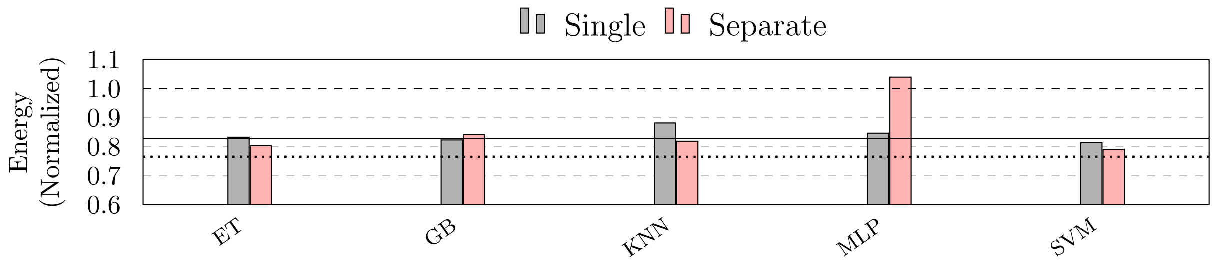


Figure 5.11: Average energy consumption when using a single or separate classifiers for system knobs (lower is better).

average MLP behavior is representative of the applications tested, *i.e.,* not caused by an outlier—in three of the four applications evaluated, MLP classifying taskset and DVFS separately performed worse than the *race-to-idle* heuristic. Of course, taskset and DVFS prediction may benefit from using different classification algorithms. We tested taskset-only and DVFS-only classifiers in isolation and empirically determined that, for our system and applications, the ET classifier works best for taskset prediction and the SVM classifier works best for DVFS prediction. The solid horizontal line indicates the energy consumption for this ET/SVM classifier mix. While this avoided the poor results seen with MLP, it actually performed slightly worse than three of the four remaining *Single* classifier approaches. Using a single, unified classifier that learns both taskset and DVFS together ultimately produces more reliable predictions.

5.4.6 Discussion of Results and Limitations

Our evaluation demonstrates that machine learning classification driven by low-level features is an effective approach for improving energy efficiency. In fact, a variety of different classifiers are useful for predicting energy-efficient system settings at runtime, even without classifier tuning. No single classifier appears to have a clear advantage over others, though future work on fine-tuning the training data and classifiers, combined with evaluations on other platforms, may eventually produce a near-optimal implementation.

We also did not attempt to optimize the runtime's overhead—reading performance counters, data transformations, or system actuation. Although low, there is still room for improvement, which could support faster sampling and prediction intervals and further improve energy consumption.

We currently only manage socket/core allocation at system-level. Because applications are launched with a fixed number of threads, those threads are forced to share compute cores when we bind applications to fewer sockets or disable HyperThreads during runtime. This over-subscription is usually not as efficient as matching the thread count to the available cores, *e.g.,* running 80 threads on our 80 physical cores is more efficient that binding 160 threads to those 80 cores. Integration with application-level parallelism can further improve energy efficiency and reduce resource contention when binding applications to smaller tasksets [124].

On large systems such as our evaluation platform, the processor and DRAM consume the majority of system power, which we are able to measure [25, 103]. There are other components like hard disks and network interfaces that are not currently accounted for. Similarly, extending the approach to support heterogeneous architectures like those using GPUs would also be beneficial. While it is not common practice to instrument these other components for power/energy monitoring, such feedback could help resource management solutions achieve better total energy efficiency.

CHAPTER 6
CONCLUSION

This dissertation addresses balancing application performance and system energy consumption in computing systems. The first two projects, POET and CoPPer, are adaptive control-theoretic approaches that solve the constrained optimization problem of meeting soft performance goals while minimizing energy consumption. The third project uses machine learning classification techniques, driven by hardware performance counters, to predict energy-efficient system settings at runtime to strictly optimize the performance/power tradeoff, thus minimizing energy consumption. Unlike static resource allocation techniques, our solutions dynamically adjust resource allocations based on runtime feedback to adapt to changing application and system behavior.

Our early work demonstrates that heuristic approaches for meeting timing constraints and reducing energy consumption are not portable between applications and systems, *e.g.,* the classic *race-to-idle* heuristic can be completely the wrong approach to use [61]. Therefore, we propose POET, a portable approach to resource management that functions independent of platform timing/energy tradeoffs or particular system knobs. POET uses control theory to meet soft performance (or latency) goals and mathematical optimization to minimize energy consumption. We demonstrate POET's ability to meet soft performance constraints and achieve near-optimal energy consumption on two very different embedded systems—a Vaio tablet with an Intel Haswell processor, and an ODROID-XU+E with a heterogeneous ARM big.LITTLE SoC. We then demonstrate POET's effectiveness on a dual-socket server-class system with an order of magnitude more knob settings and significantly more compute power than the embedded platforms. In addition, we briefly discuss Bard, a follow-on project which modifies POET to allow switching between performance or power goals while optimizing energy or performance respectively.

Software management of DVFS is becoming deprecated, necessitating a replacement for the abundance of controllers that previously relied on DVFS to meet performance constraints. We respond to this demand with CoPPer, which also uses adaptive control-theoretic techniques to meet soft performance goals while optimizing energy. CoPPer actuates hardware power caps instead of DVFS, but leaves the optimization to the hardware, which can adapt to changing resource demands more quickly than software. CoPPer addresses new challenges that arise from power capping, without requiring an explicit behavior model. We evaluate CoPPer on a dual-socket system using Intel's Running Average Power Limit (RAPL), demonstrate that it meets soft performance constraints, and outperforms a state-of-the-art DVFS controller (a modified version of POET) in achieving low energy consumption. Additionally, CoPPer introduces a novel *gain limit* to prevent over-allocating power when it is not beneficial, *e.g.,* for memory-bound applications or unachievable performance goals.

Finally, we shift focus to the High Performance Computing (HPC) domain and propose optimizing energy efficiency to minimize the runtime cost of computation. Beyond the obvious cost savings achieved from reducing energy, we extrapolate from our single-system results to show that optimizing energy efficiency in hardware over-provisioned, power-constrained clusters (as we expect future exascale HPC systems to be) increases the total throughput of the cluster to complete more science for the same runtime costs. To avoid modifying applications, we map low-level hardware performance counters (features) to energy-efficient system knob settings. We propose using machine learning classification techniques to predict energy-efficient settings in-situ on compute nodes, since they are low-overhead compared to existing estimation approaches used in HPC resource management. The complexity of the feature space mappings makes simple classification approaches insufficient for solving this resource allocation problem. We evaluate five of the most promising classifiers and demonstrate that they dramatically reduce the energy consumption of production HPC genome assembly applications on a large shared-memory compute node with low overhead and robustness to the availability of different performance counters.

6.1 Future Work

While this dissertation is on performance and power/energy consumption, there are other tradeoffs available in computing systems. For example, approximate computing is an ongoing area of research. A resource allocation scheme could dynamically choose from a set of functions that offer different performance and computation precision, or a configurable hardware component can increase performance or reduce energy consumption at the expense of less accurate results. Computation accuracy could be a goal or optimization in any of the three major projects presented in this dissertation and would be useful on systems ranging from embedded platforms to HPC clusters. It is a particularly challenging area of research because different applications have different concepts of what it means to be accurate. Accuracy can also be difficult to quantify, especially when such values are dependent on human perception, *e.g.,* for various types of multimedia. Furthermore, the range of acceptable accuracies is context-dependent, *e.g.,* by application and/or deployment scenario.

Our control-theoretic projects support meeting a single constraint and optimizing in another dimension. However, software must often manage multiple constraints, *e.g.,* meet a performance goal but also respect a power cap. Combining multiple controllers is a logical next step, but they must be coordinated so as not to conflict with each other. In simple cases, controllers tune knobs that do not affect each other, in which case little or no direct coordination is needed, *e.g.,* one controller tunes application-level knobs while another tunes system knobs. Otherwise the controllers must either communicate directly or be configured such that they can still adapt correctly, *e.g.,* by using different pole values and/or window periods. In either case, they should not be tuning the same knobs, otherwise their internal state would be inaccurate—they would fail to meet their constraints and any optimization they attempt to provide could suffer as well.

Our machine learning classification framework currently optimizes energy efficiency on individual compute nodes for single-node jobs. To actually maximize the throughput of hardware over-provisioned, power-constrained clusters, the solution must be adapted to scale out to multiple nodes. Naively deploying the current solution on all nodes in a cluster will not achieve this, and could result in suboptimal energy efficiency and violations of the global power constraint. First, a component to schedule jobs while respecting the cluster power constraint is needed, likely in coordination with accurate predictions of how much power individual nodes require while running their assigned jobs energy-efficiently (keeping in mind that the power consumption fluctuates as applications move through phases). Additionally, tighter coordination between nodes is needed to support jobs that scale across multiple nodes, so that one node does not slow down the rest, resulting in reduced total energy efficiency. A scaled-out approach will need to account for other components like network switches and consider the location of different compute nodes—individual racks have their own power constraints, and the distance between communicating nodes impacts application performance.

System resources other than CPU and DRAM should be coordinated since they also impact performance/power tradeoffs—GPUs and other accelerators are particularly interesting. Data storage (*e.g.,* hard disks, SSDs, future non-volatile memories) and network interfaces are also critical, particularly in low-power environments like embedded systems and high-throughput and communication-intensive environments like HPC.

Unfortunately, power/energy sensors on modern computing systems are relatively rare, with the exception of Intel processors. Model-based approaches are too inaccurate and cannot provide runtime guarantees, and custom/ad-hoc instrumentation requires special equipment and does not scale well in production. To better enable energy-aware applications, more hardware needs to include power/energy instrumentation—not just on processors and SoCs, but on other hardware components as well, *e.g.,* DRAM, disks, and networking components. Such capabilities would more easily enable both research and development of practical solutions for coordinating a wide variety of system resources in an energy-aware manner. Similarly, datacenters and HPC clusters can benefit from instrumenting external components like network switches and network-attached storage devices.

Finally, the availability of runtime feedback from which to make informed decisions is a fundamental requirement of self-aware systems. Applications and users want to specify contextual goals, *e.g.,* frames processed in a video encoder, not low-level metrics like instructions retired. Hardware performance counters are the de facto standard today, but do not provide high-level metrics like a true measure of application progress. Few such mechanisms are currently available, and typically require modifying applications to provide appropriate instrumentation, *e.g.,* using Heartbeats. Language constructs or operating system interfaces could provide more portable approaches for applications to report metrics. Some measurements could potentially be inferred from the software design, or at least only require a developer to specify a few attributes in their code without linking to additional libraries. A self-aware system can then rely on these portable metrics rather than requiring the developer to provide instrumentation.

APPENDIX A

TOOLS FOR SELF-AWARE SYSTEMS RESEARCH

Tool building is often under-appreciated by the computer science research community, yet is crucial to enabling practical systems research. The projects in this dissertation led to the development of a variety of portable tools that we make available to the community at large. Links to published code are available in Appendix B.

In Chapter 1.2, we described the *observe*, *decide*, and *act* framework that Hoffmann proposed in his "self-aware" computing model [54]. While the core research in this dissertation focuses on the *decide* phase, the tools described in this chapter are relevant to the *observe* and *act* phases.

A.1 Runtime Instrumentation and Profiling

The control-based projects presented in this dissertation (POET, Bard, CoPPer) are designed for applications that have high-level loops for which we enforce predictable timing behavior. For applications to capture performance and power/energy data, they need a mechanism to capture and record metrics. Application developers often create their own instrumentation and profiling tools, which is mostly unnecessary and means that data is stored in a variety different formats, making it troublesome to process. Hoffmann et al. proposed `Heartbeats` to specify and record performance goals for self-aware applications [56]. We extended this work in POET and Bard and released `Heartbeats 2.0`, which includes a primitive interface for portably capturing energy metrics.

We later developed a simplified interface, aptly called `Simple Heartbeats`, which allows more flexibility in recording application behavior. For example, it can record performance and power behavior for discontinuous loops or separate application code phases and is more portable across operating systems. Java and Rust language bindings and abstractions are also available for Simple Heartbeats, some of which are integrated with Servo, Mozilla's parallel web browsing engine [11]. These code bases are also used in other projects not described in this dissertation.

We also created `Heartbeats Simple Classic` which wraps Simple Heartbeats and EnergyMon (described in the next section) while behaving more like Heartbeats 2.0. This interface benefits from the simplicity of the original Heartbeats design while maintaining portability, and is used in CoPPer's evaluation.

A.2 EnergyMon: A Portable Interface for Runtime Energy Monitoring

There is a growing need for software to access power and energy data in-situ, like for the projects in this dissertation. Designing and evaluating POET and Bard led us to conclude that a portable software interface for accessing energy metrics at runtime is necessary. The primitive interface in Heartbeats 2.0 is not as portable as needed and does not allow capturing power/energy data in isolation. In response, we designed `EnergyMon`.

Accessing power/energy data requires writing system-specific code which is challenging for researchers and engineers, and results in applications that are not portable across platforms. The diversity in sensor properties and current ad-hoc approaches for using them presents a significant challenge. The problem is more complex than simply locating and reading the correct files, hardware registers, or memory addresses. A portable interface must account for other factors:

- **Data Type:** Whether sensors expose *power* or *energy* data.
- **Access:** Privileges/constraints, *e.g., exclusive access*, particularly for external devices.
- **Units of measurement:** The power or energy units that sensors expose measurements in, and also their *level of precision.*
- **Overflow:** Maximum values for sensor counters.

- **Refresh Interval:** How frequently sensors update.
- **Interface:** System abstractions, protocols, and data formats.

Sensors which only report power data *must be polled* at their refresh interval so as not to lose data after the sensor refreshes. Power/energy readings may need to be *extracted from other data* or *transformed* before they are meaningful [6, 30, 115]. Developers may even have to implement different *protocols* and *data formats* [6, 30]. Furthermore, even a single system equipped with power or energy monitoring can offer multiple approaches to obtain it [25, 66, 115]. Correctly and accurately collecting power and energy data requires understanding and properly managing sensor properties and behavior.

The first important design decision is which *data type* to expose in a common interface. We choose energy, which unlike power, is not explicitly a function of time and can be recorded cumulatively, simplifying the interface. With an energy provider that records total energy consumed since some time $\tau = 0$, a power-aware application can simply record energy E at the beginning and end of a time interval and compute the average power as $\Delta E / \Delta \tau$. In this way, the interface easily supports applications interested in energy or power.

Next, a common interface can expose energy metrics in a standard and sufficiently precise *unit of measurement* at a large enough data length to avoid *overflow*. Therefore, applications do not need to track various unit types and do conversions. A deceptively simple but important insight is not our choice of units or data length, but rather that counter overflow is one of the diverse sensor properties that applications should not have to worry about. We choose microjoules as 64-bit unsigned integers which are precise enough for the use cases we anticipate and avoid the need for floating point data types. At 100 W of power, it would take about 5,850 years to overflow, and in our experience, modern sensors are too imprecise and models too inaccurate to justify smaller units.

The `EnergyMon` C interface defines an `energymon` struct that contains seven function pointers and a `state` variable:

- `finit`: Initialize the energy monitor.
- `fread`: Get the cumulative energy in microjoules.
- `ffinish`: Destroy the energy monitor.
- `fsource`: Get the name for the energy monitoring source.
- `finterval`: Get the refresh interval in microseconds.
- `fprecision`: Get the best possible precision in microjoules.
- `fexclusive`: Get if exclusive access is required.
- `state`: Pointer to a struct that the implementation uses to maintain internal state, *e.g.,* file descriptors or thread data.

This interface exposes some of the diverse properties described previously, like *refresh interval* and sensor *precision*, while hiding other properties that are not important to most software, like the *underlying interfaces*, *device protocols*, and *data formats*.

For any time τ, where $\tau = 0$ is the energy monitor's initialization, the following invariants hold for energy readings $E(\tau)$:

$$\forall \tau \geq 0 \qquad \tau_i \leq \tau_j \implies 0 \leq E(\tau_i) \leq E(\tau_j) \tag{A.1}$$

$$\tau = 0 \not\Longrightarrow E(\tau) = 0 \tag{A.2}$$

Equation A.1 states that for any time at or after initialization, energy is always non-negative and monotonically increasing. Equation A.2 states that the initial value does not have to be zero (could be larger). The following example of the EnergyMon C interface demonstrates computing the average power of a worker function (error checking excluded):

```
energymon em;
uint64_t start_uj, end_uj, start_usec, end_usec;

// get the energymon instance and initialize
energymon_get_default(&em);
em.finit(&em);

```

```c
// get start time/energy
start_usec = gettime_us(); // e.g. wrap clock_gettime
start_uj = em.fread(&em);

// application-specific processing
do_work();

// get end time/energy
end_usec = gettime_us();
end_uj = em.fread(&em);

printf("Average power of do_work() in Watts: %f\n",
       (end_uj-start_uj) / (double) (end_usec-start_usec));

// destroy the instance
em.ffinish(&em);
```

Listing A.1: Reading energy values and computing power with EnergyMon.

Other functions provide additional information about the implementation that software can use to make runtime decisions. For example, applications can use the name provided by `fsource` for debugging or logging. `finterval` can be used to determine a lower bound on sleep time for polling the energy monitor at regular intervals. `fprecision` might inform a self-adaptive system how much noise to expect in power/energy data. `fexclusive` could be used in deciding to share energy data with other software components. The project also includes command-line utilities for integration with shell scripts, although we do not describe them further here.

EnergyMon was integrated with Servo, Mozilla's parallel web browsing engine [11]. A complete design and evaluation is available in the original publication [59] and an extended technical report [60]. EnergyMon is also used in CoPPer's evaluation, as well other research projects not described in this dissertation, *e.g.,* the ongoing development of the FAST language described in Chapter 1.3.3.

A.3 Power Capping

Evaluating Intel's Running Average Power Limit (RAPL) [25] ultimately led to the design and evaluation of the CoPPer project. Again, we found a distinct lack of tools to aid us—most material we found used low-level approaches for interfacing with RAPL that were hardware-dependent and required detailed understanding of RAPL as defined in Intel's software developer's manuals [67].

In recent years, the Linux kernel exposed some RAPL functionality via the `powercap` file system interface [32]. Perhaps the biggest advantage of the powercap interface is that a user does not have to configure the system's model-specific registers, which have very particular platform-dependent encodings. However, there were no software utilities available to assist in discovery of these RAPL capabilities, or in proper system actuation. Therefore, we developed a `Linux powercap bindings` library and collection of applications to aid software developers with these tasks. These tools are now available in repositories for Debian-based Linux distributions, including Ubuntu beginning with version 18.04 LTS "Bionic Beaver." The library usage is straightforward, for example (error checking excluded):

```
uint32_t i, npackages;
powercap_rapl_pkg* pkgs;

// get number of processor sockets
npackages = powercap_rapl_get_num_packages();

// initialize
pkgs = malloc(npackages * sizeof(powercap_rapl_pkg));
for (i = 0; i < npackages; i++) {
  // last parameter is a "read-only" flag (set to false)
  powercap_rapl_init(i, &pkgs[i], 0);
}

// set caps of 100 Watts for short_term, 50 Watts for long_term constraints on each package
for (i = 0; i < npackages; i++) {
  powercap_rapl_set_power_limit_uw(&pkgs[i], POWERCAP_RAPL_ZONE_PACKAGE,
    POWERCAP_RAPL_CONSTRAINT_SHORT, 100 * 1000000);
  powercap_rapl_set_power_limit_uw(&pkgs[i], POWERCAP_RAPL_ZONE_PACKAGE,
    POWERCAP_RAPL_CONSTRAINT_LONG, 50 * 1000000);
}

// now cleanup
for (i = 0; i < npackages; i++) {
  powercap_rapl_destroy(&pkgs[i]); {
}
free(pkgs);
```

Listing A.2: Setting RAPL power caps with the powercap library.

Command-line utilities, not described further here, also make integration with scripts easy.

While the powercap interface is sufficient for many use cases, it has some limitations. First, the configuration is not as fine-grained as it could be due to the kernel developers' choice of data units (microjoules, microwatts, and microseconds). Second, it is only useful on systems running the Linux kernel. We address these limitations with `RAPLCap`, a more general interface for accessing RAPL that provides multiple implementations. The two primary implementations are for Linux—the first wraps the Linux powercap bindings library; the second accesses the system model-specific registers which allows more fine-grained configuration, but must be updated as new hardware becomes available. The RAPLCap interface is also straightforward to use, for example (error checking excluded):

```
raplcap rc;
raplcap_limit rl_short, rl_long;
uint32_t i, nsockets;

// get the number of RAPL packages/sockets
nsockets = raplcap_get_num_sockets(NULL);

// initialize
raplcap_init(&rc);

// set caps of 100 Watts for short_term, 50 Watts for long_term constraints on each socket
// a time window of 0 leaves the time window unchanged
rl_short.watts = 100.0;
rl_short.seconds = 0.0;
rl_long.watts = 50.0;
rl_long.seconds = 0.0;
for (i = 0; i < nsockets; i++) {
  raplcap_set_limits(&rc, i, RAPLCAP_ZONE_PACKAGE, &rl_long, &rl_short);
}

// cleanup
raplcap_destroy(&rc);
```

Listing A.3: Setting RAPL power caps with RAPLCap.

Using a common interface allows software to link against different library implementations, depending on the target hardware platform or operating system, without requiring changes to application code. Work is ongoing to implement RAPLCap on other operating systems like Apple OSX and Microsoft Windows, which expose different methods for configuring RAPL. RAPLCap also includes command-line utilities, which are not described further here.

APPENDIX B
PUBLISHED CODE

This appendix provides links for much of the source code developed as part of this dissertation. Note that locations may change with time, and APIs may change from their presentation in this dissertation as demands evolve. Projects generally contain usage instructions and citation information. In general, bug reports and code patches (*e.g.,* pull requests) are welcome.

Table B.1 provides URLs for the core research code in this dissertation.

Table B.1: Project code URLs.

Project	**Language**	**URL**
POET/Bard Resources	–	http://poet.cs.uchicago.edu
POET	C	https://github.com/libpoet/poet
Bard	C	https://github.com/libpoet/bard
CoPPer	C	https://github.com/powercap/copper
CoPPer Evaluation Helper & Resources	C	https://github.com/powercap/copper-eval

Table B.2 provides URLs for code developed to assist the research in this dissertation that may be useful for wider audiences.

Table B.2: Tool code URLs.

Tool	**Language**	**URL**
Heartbeats 2.0	C	https://github.com/libheartbeats/heartbeats
Simple Heartbeats	C	https://github.com/libheartbeats/heartbeats-simple
Simple Heartbeats Classic	C	https://github.com/libheartbeats/heartbeats-simple-classic
Simple Heartbeats Rust Bindings	Rust	https://github.com/libheartbeats/heartbeats-simple-sys
Simple Heartbeats Rust Abstractions	Rust	https://github.com/libheartbeats/heartbeats-simple-rust
Simple Heartbeats Java Bindings/Abstractions	Java	https://github.com/libheartbeats/heartbeats-simple-jni
EnergyMon	C	https://github.com/energymon/energymon
EnergyMon Rust Bindings	Rust	https://github.com/energymon/energymon-sys
EnergyMon Rust Abstractions	Rust	https://github.com/energymon/energymon-rust
EnergyMonitor Rust Trait	Rust	https://github.com/energymon/energy-monitor-rs
EnergyMon Java Bindings/Abstractions	Java	https://github.com/energymon/energymon-jni
Linux powercap bindings	C	https://github.com/powercap/powercap
RAPLCap	C	https://github.com/powercap/raplcap

BIBLIOGRAPHY

[1] Nevine AbouGhazaleh, Alexandre Ferreira, Cosmin Rusu, Ruibin Xu, Frank Liberato, Bruce Childers, Daniel Mosse, and Rami Melhem. "Integrated CPU and L2 Cache Voltage Scaling Using Machine Learning". In: *LCTES*. 2007.

[2] Bilge Acun, Phil Miller, and Laxmikant V. Kale. "Variation Among Processors Under Turbo Boost in HPC Systems". In: *ICS*. 2016.

[3] Mark F. Adams, Jed Brown, John Shalf, Brian van Straalen, Erich Strohmaier, and Sam Williams. *HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems*. Tech. rep. LBNL-6630E. LBNL, 2014.

[4] Susanne Albers. "Algorithms for Dynamic Speed Scaling". In: *STACS*. 2011.

[5] Claudia Alvarado, Dan Tamir, and Apan Qasem. "Realizing Energy-efficient Thread Affinity Configurations with Supervised Learning". In: *IGSC*. 2015.

[6] AmeriDroid. *ODROID Smart Power*. 2016. URL: `http://ameridroid.com/products/smart-power`.

[7] Yuxin Bai, Victor W. Lee, and Engin Ipek. "Voltage Regulator Efficiency Aware Power Management". In: *ASPLOS*. 2017.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. "The NAS Parallel Benchmarks–Summary and Preliminary Results". In: *SC*. 1991.

[9] Luiz André Barroso and Urs Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (2007).

[10] Robert M. Bell, Yehuda Koren, and Chris Volinsky. *The Bellkor 2008 Solution to the Netflix Prize*. 2008.

[11] Lars Bergstrom, Brian Anderson, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. "Engineering the Servo Web Browser Engine using Rust". In: *ICSE*. 2016.

[12] Michael Berry, Thomas E. Potok, Prasanna Balaprakash, Henry Hoffmann, Raju Vatsavai, Prabhat, and Robinson Pino. *Machine Learning and Understanding for Intelligent Extreme Scale Scientific Computing and Discovery*. DOE ASCR Workshop Report. 2015.

[13] Luciano. Bertini, J.C.B. Leite, and Daniel Mosse. "Statistical QoS Guarantee and Energy-Efficiency in Web Server Clusters". In: *ECRTS*. 2007.

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *PACT*. 2008.

[15] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. “Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach”. In: *MICRO*. 2008.

[16] Stephen P. Bradley, Arnoldo C. Hax, and Thomas L. Magnanti. *Applied Mathematical Programming*. 1977.

[17] Giorgio C Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer, 2006.

[18] Liyu Cao and Howard M. Schwartz. “Analysis of the Kalman Filter Based Estimation Algorithm: An Orthogonal Decomposition Approach”. In: *Automatica* 40.1 (2004).

[19] Aaron Carroll and Gernot Heiser. “Mobile Multicores: Use Them or Waste Them”. In: *HotPower*. 2013.

[20] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Martin Schulz, Eduard Ayguadé, Jesus Labarta, and Mateo Valero. “Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes”. In: *ICS*. 2016.

[21] Jian Chen and Lizy Kurian John. “Predictive Coordination of Multiple On-Chip Resources for Chip Multiprocessors”. In: *ICS*. 2011.

[22] Hui Cheng and Steve Goddard. “SYS-EDF: a system-wide energy-efficient scheduling algorithm for hard real-time systems”. In: *IJES* 4.2 (2009).

[23] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. “Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps”. In: *MICRO*. 2011.

[24] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. “Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores”. In: *PACT*. 2008.

[25] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. “RAPL: Memory Power Estimation and Capping”. In: *ISLPED*. 2010.

[26] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware Scheduling for Heterogeneous Datacenters”. In: *ASPLOS*. 2013.

[27] Christina Delimitrou and Christos Kozyrakis. “Quasar: Resource-efficient and QoS-aware Cluster Management”. In: *ASPLOS*. 2014.

[28] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F Wenisch, and Ricardo Bianchini. "CoScale: Coordinating CPU and Memory System DVFS in Server Systems". In: *MICRO*. 2012.

[29] Qingyuan Deng, David Meisner, Luiz E. Ramos, Thomas F. Wenisch, and Ricardo Bianchini. "MemScale: active low-power modes for main memory". In: *ASPLOS*. 2011.

[30] Electronic Education Devices. *WattsUp .Net Meter*. 2016. URL: `http://www.wattsupmeters.com/`.

[31] Bruno Diniz, Dorgival Guedes, Wagner Meira Jr., and Ricardo Bianchini. "Limiting the Power Consumption of Main Memory". In: *ISCA*. 2007.

[32] Linux Kernel Documentation. *Power Capping Framework*. 2015. URL: `https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt`.

[33] Linux Kernel Documentation. *Intel P-State driver*. 2016. URL: `https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt`.

[34] Sudip S Dosanjh, Shane Canon, Jack Deslippe, Kjiersten Fagnan, Richard A Gerber, Lisa Gerhardt, Jason Hick, Douglas Jacobsen, David Skinner, and Nicholas J Wright. "Extreme Data Science at the National Energy Research Scientific Computing (NERSC) Center." In: *PARCO*. 2013.

[35] 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.

[36] Anne Farrell and Henry Hoffmann. "MEANTIME: Achieving Both Minimal Energy and Timeliness with Approximate Computing". In: *USENIX ATC*. 2016.

[37] Eyal Fayneh, Marcelo Yuffe, Ernest Knoll, Michael Zelikson, Muhammad Abozaed, Yair Talker, Ziv Shmuely, and Saher A. Rahme. "4.1 14nm 6th-Generation Core Processor SoC with Low Power Consumption and Improved Performance". In: *ISSCC*. 2016.

[38] Wes Felter, Karthick Rajamani, Tom Keller, and Cosmin Rusu. "A Performance-Conserving Approach for Reducing Peak Power Consumption in Server Systems". In: *ICS*. 2005.

[39] Matteo Ferroni, Andrea Corna, Andrea Damiani, Rolando Brondolin, Juan A. Colmenares, Steven Hofmeyr, John D. Kubiatowicz, and Marco D. Santambrogio. "Power Consumption Models for Multi-Tenant Server Infrastructures". In: *TACO* 14.4 (Nov. 2017).

[40] Antonio Filieri, Henry Hoffmann, and Martina Maggio. “Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees”. In: *ICSE*. 2014.

[41] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D’ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro V. Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. “Control Strategies for Self-Adaptive Software Systems”. In: *ACM Trans. Auton. Adapt. Syst.* 11.4 (Feb. 2017).

[42] Jerome H. Friedman. “Greedy Function Approximation: A Gradient Boosting Machine.” In: *Ann. Statist.* 29.5 (Oct. 2001).

[43] Yong Fu, Nicholas Kottenstette, Chenyang Lu, and Xenofon D. Koutsoukos. “Feedback Thermal Control of Real-time Systems on Multicore Processors”. In: *EMSOFT*. 2012.

[44] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. “HipMer: An Extreme-Scale De Novo Genome Assembler”. In: *SC*. 2015.

[45] Pierre Geurts, Damien Ernst, and Louis Wehenkel. “Extremely Randomized Trees”. In: *Machine Learning* 63.1 (2006).

[46] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. “SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit”. In: *2nd USENIX Windows NT Symposium*. 1998.

[47] Liang He, Lipeng Gu, Linghe Kong, Yu Gu, Cong Liu, and Tian He. “Exploring Adaptive Reconfiguration to Optimize Energy Efficiency in Large-Scale Battery Systems”. In: *RTSS*. 2013.

[48] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. 2004.

[49] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 3rd ed. 2003.

[50] Van Emden Henson and Ulrike Meier Yang. “BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner”. In: *Appl. Numer. Math.* 41.1 (Apr. 2002).

[51] Geoffrey E. Hinton. “Connectionist Learning Procedures”. In: *Artificial Intelligence* 40.1 (1989).

[52] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 1st. 2009.

[53] Henry Hoffmann. “Racing and Pacing to Idle: An Evaluation of Heuristics for Energy-aware Resource Allocation”. In: *HotPower*. 2013.

[54] Henry Hoffmann. “SEEC: A Framework for Self-Aware Management of Goals and Constraints in Computing Systems”. PhD thesis. Massachusetts Institute of Technology, 2013.

[55] Henry Hoffmann. “CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems”. In: *ECRTS*. 2014.

[56] 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.

[57] 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.

[58] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. “A Generalized Software Framework for Accurate and Efficient Management of Performance Goals”. In: *EMSOFT*. 2013.

[59] Connor Imes, Lars Bergstrom, and Henry Hoffmann. “A Portable Interface for Runtime Energy Monitoring”. In: *FSE*. 2016.

[60] Connor Imes, Lars Bergstrom, and Henry Hoffmann. *A Portable Interface for Runtime Energy Monitoring: Extended Analysis*. Tech. rep. TR-2016-08. University of Chicago, Department of Computer Science, 2016.

[61] Connor Imes and Henry Hoffmann. “Minimizing Energy Under Performance Constraints on Embedded Platforms: Resource Allocation Heuristics for Homogeneous and Single-ISA Heterogeneous Multi-Cores”. In: *EWiLi*. 2014.

[62] Connor Imes and Henry Hoffmann. “Bard: A Unified Framework for Managing Soft Timing and Power Constraints”. In: *SAMOS*. 2016.

[63] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. “POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints”. In: *RTAS*. 2015.

[64] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. "Portable Multicore Resource Management for Applications with Performance Constraints". In: *MCSoC*. 2016.

[65] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. *Handing DVFS to Hardware: Using Power Capping to Control Software Performance*. Tech. rep. TR-2018-03. University of Chicago, Department of Computer Science, 2018.

[66] Texas Instruments. *INA231 28-V, Bi-Directional, Zero-Drift, Low-/High-Side, I2C Out Current/Power Monitor w/ Alert in WCSP*. 2015. URL: `http://www.ti.com/product/ina231`.

[67] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Sept. 2016.

[68] Syed M. Z. Iqbal, Yuchen Liang, and Hakan Grahn. "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems". In: *Computer Architecture Letters* 9.2 (2010).

[69] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget". In: *MICRO*. 2006.

[70] Nandini Kappiah, Vincent W. Freeh, and David K. Lowenthal. "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs". In: *SC*. 2005.

[71] Christos Karamanolis, Magnus Karlsson, and Xiaoyun Zhu. "Designing Controllable Computer Systems". In: *HotOS*. 2005.

[72] Ian Karlin, Jeff Keasler, and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. 2013.

[73] 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.

[74] Peter Kogge, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, and Robert Lucas. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. 2008.

[75] Pratyush Kumar and Lothar Thiele. "p-YDS algorithm: An Optimal Extension of YDS Algorithm to Minimize Expected Energy For Real-Time Jobs". In: *EMSOFT*. 2014.

[76] Adam J. Kunen, Teresa S. Bailey, and Peter N. Brown. “KRIPKE - A Massively Parallel Transport Mini-App”. In: *American Nuclear Society M&C*. 2015.

[77] Lawrence Livermore National Laboratory. *Co-design at Lawrence Livermore National Lab – Quicksilver*. 2017. URL: https://codesign.llnl.gov/quicksilver.php.

[78] Los Alamos National Laboratory. *CoMD*. 2016. URL: https://github.com/ECP-copa/CoMD.

[79] Etienne Le Sueur and Gernot Heiser. “Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns”. In: *HotPower*. 2010.

[80] Etienne Le Sueur and Gernot Heiser. “Slow Down or Sleep, That is the Question”. In: *USENIX ATC*. 2011.

[81] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. “Power Capping: A Prelude to Power Shifting”. In: *Cluster Computing* 11.2 (June 2008).

[82] Baochun Li and Klara Nahrstedt. “A Control-Based Middleware Framework for Quality-of-Service Adaptations”. In: *IEEE Journal on Selected Areas in Communications* 17.9 (1999).

[83] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. “MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph”. In: *Bioinformatics* 31.10 (2015).

[84] Xiaodong Li, Ritu Gupta, Sarita V. Adve, and Yuanyuan Zhou. “Cross-component Energy Management: Joint Adaptation of Processor and Memory”. In: *TACO* 4.3 (2007).

[85] Simone Libutti, Giuseppe Massari, Patrick Bellasi, and William Fornaciari. “Exploiting Performance Counters for Energy Efficient Co-Scheduling of Mixed Workloads on Multi-Core Platforms”. In: *PARMA-DITAM*. 2014.

[86] Jian (Denny) Lin, Wei Song, and Albert M.K. Cheng. “Real-energy: A New Framework and a Case Study to Evaluate Power-aware Real-time Scheduling Algorithms”. In: *ISLPED*. 2010.

[87] Cong Liu, Xiao Qin, and Shuang Li. “PASS: Power-Aware Scheduling of Mixed Applications with Deadline Constraints on Clusters”. In: *ICCCN*. 2008.

[88] Zhijian Lu, Jason Hein, Marty Humphrey, Mircea Stan, John Lach, and Kevin Skadron. “Control-theoretic Dynamic Frequency and Voltage Scaling for Multimedia Workloads”. In: *CASES*. 2002.

[89] Paul J. Lucas. *SWISH++*. 2014. URL: http://swishplusplus.sourceforge.net/.

[90] Martina Maggio, Enrico Bini, Georgios Chasparis, and Karl-Erik Årzén. "A Game-Theoretic Resource Manager for RT Applications". In: *ECRTS*. 2013.

[91] Martina Maggio, Henry. Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. "Power Optimization in Embedded Systems via Feedback Control of Resource Allocation". In: *IEEE TCST* 21.1 (2013).

[92] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. "Automated Control of Multiple Software Goals Using Multiple Actuators". In: *ESEC/FSE*. 2017.

[93] Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. "A Run-Time System for Power-Constrained HPC Applications". In: *ISC*. 2015.

[94] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE TCCA Newsletter* (1995).

[95] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. "Power Management of Online Data-intensive Services". In: *ISCA*. 2011.

[96] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. "CALOREE: Learning Control for Predictable Latency and Low Energy". In: *ASPLOS*. 2018.

[97] 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.

[98] Sparsh Mittal. "A Survey of Techniques For Improving Energy Efficiency in Embedded Computing Systems". In: *IJCAET* 6.4 (2014).

[99] 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.

[100] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Jayaprakash Pisharath, Gokhan Memik, and Alok Choudhary. "MineBench: A Benchmark Suite for Data Mining Workloads". In: *IISWC*. 2006.

[101] NERSC. *GENEPOOL*. 2017 (accessed Jan, 2018). URL: `http://www.nersc.gov/users/computational-systems/genepool/`.

[102] Sergey Nurk, Dmitry Meleshko, Anton Korobeynikov, and Pavel Pevzner. "metaSPAdes: a new versatile de novo metagenomics assembler". In: *ArXiv e-prints* (Apr. 2016). arXiv: `1604.03071 [q-bio.GN]`.

[103] opcm. *Processor Counter Monitor (PCM).* 2016. URL: `https://github.com/opcm/pcm`.

[104] Tapasya Patki, David K. Lowenthal, Anjana Sasidharan, Matthias Maiterth, Barry L. Rountree, Martin Schulz, and Bronis R. de Supinski. "Practical Resource Management in Power-Constrained, High Performance Computing". In: *HPDC*. 2015.

[105] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (Nov. 2011).

[106] Yu Peng, Henry C. M. Leung, Siu-Ming Yiu, and Francis Y. L. Chin. "IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth". In: *Bioinformatics* 28.11 (2012).

[107] Vinivius Petrucci, Orlando Loques, and Daniel Mosse. "Lucky Scheduling for Energy-Efficient Heterogeneous Multi-Core Systems". In: *HotPower*. 2012.

[108] Nicolas Pitre. *Teaching the scheduler about power management.* 2014. URL: `http://lwn.net/Articles/602479/`.

[109] 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.

[110] Meghana R. *An overview of the 6th generation Intel$^{®}$ Core^{TM} processor (code-named Skylake).* Mar. 2016. URL: `https://software.intel.com/en-us/articles/an-overview-of-the-6th-generation-intel-core-processor-code-named-skylake`.

[111] Amir M. Rahmani, Bryan Donyanavard, Tiago Müch, Kasra Moazzemi, Axel Jantsch, Onur Mutlu, and Nikil Dutt. "SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management". In: *ASPLOS*. 2018.

[112] Ragunathan Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. "A Resource Allocation Model for QoS Management". In: *RTSS*. 1997.

[113] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. "Thread Motion: Fine-Grained Power Management for Multi-Core Systems". In: *ISCA*. 2009.

[114] Sherief Reda, Ryan Cochran, and Ayse Kivilcim Coskun. “Adaptive Power Capping for Servers with Multithreaded Workloads”. In: *IEEE Micro* 32.5 (2012).

[115] Efi Rotem, Alon Naveh, Doron Rajwan amd Avinash Ananthakrishnan, and Eli Weissmann. “Power management architecture of the 2nd generation Intel® Core^{TM} microarchitecture, formerly codenamed Sandy Bridge”. In: *Hot Chips*. 2011.

[116] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. “Adagio: Making DVS Practical for Complex HPC Applications”. In: *ICS*. 2009.

[117] Muhammad Husni Santriaji and Henry Hoffmann. “GRAPE: Minimizing Energy for GPU Applications with Performance Requirements”. In: *MICRO*. 2016.

[118] Osman Sarood, Akhil Langer, Laxmikant Kale, Barry Rountree, and Bronis de Supinski. “Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems”. In: *CLUSTER*. 2013.

[119] Hiroshi Sasaki, Yoshimichi Ikeda, Masaaki Kondo, and Hiroshi Nakamura. “An Intra-Task DVFS Technique based on Statistical Analysis of Hardware Events”. In: *CF*. 2007.

[120] 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 PER* 39.1 (2011).

[121] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. “Control-Theoretical Software Adaptation: A Systematic Literature Review”. In: *IEEE Transactions on Software Engineering* PP.99 (2017).

[122] Yildiz Sinangil, Sabrina M Neuman, Mahmut E Sinangil, Nathan Ickes, George Bezerra, Eric Lau, Jason E Miller, Henry C Hoffmann, Srini Devadas, and Anantha P Chandraksan. “A Self-Aware Processor SoC using Energy Monitors Integrated into Power Converters for Self-Adaptation”. In: *Symposium on VLSI*. 2014.

[123] 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”. In: *Journal of Systems Architecture - Embedded Systems Design* 57.4 (2011).

[124] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. “Holistic Run-time Parallelism Management for Time and Energy Efficiency”. In: *ICS*. 2013.

[125] ExaOSR Team. *Key Challenges for Exascale OS/R*. URL: `https://collab.mcs.anl.gov/display/exaosr/Challenges`.

[126] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. “Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations”. In: *EASC*. 2014.

[127] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. “XSBench – The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis”. In: *PHYSOR*. 2014.

[128] Ghislain Landry Tsafack Chetsa, Laurent Lefèvre, Jean-Marc Pierson, Patricia Stolf, and Georges Da Costa. “Exploiting Performance Counters to Predict and Improve Energy Performance of HPC Systems”. In: *Future Generation Computer Systems* (Aug. 2013).

[129] 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”. In: *IJES* 4.2 (2009).

[130] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. “Understanding and Auto-Adjusting Performance-Sensitive Configurations”. In: *ASPLOS*. 2018.

[131] Greg Welch and Gary Bishop. *An Introduction to the Kalman Filter*. Tech. rep. TR 95-041. UNC Chapel Hill, Department of Computer Science.

[132] Thomas Willhalm. *Intel PCM Column Names Decoder Ring*. 2014. URL: `https://software.intel.com/en-us/blogs/2014/07/18/intel-pcm-column-names-decoder-ring`.

[133] 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.

[134] 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.

[135] Xingfu Wu, Valerie Taylor, Jeanine Cook, and Philip J. Mucci. “Using Performance-Power Modeling to Improve Energy Efficiency of HPC Applications”. In: *Computer* 49.10 (Oct. 2016).

[136] Chuan-Yue Yang, Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. “System-Level Energy-Efficiency for Real-Time Tasks”. In: *SOCRTD*. 2007.

[137] Frances Yao, Alan Demers, and Scott Shenker. “A Scheduling Model for Reduced CPU Energy”. In: *FOCS*. 1995.

[138] Heechul Yun, Po-Liang Wu, Anshu Arya, Tarek Abdelzaher, Cheolgi Kim, and Lui Sha. “System-Wide Energy Optimization for Multiple DVS Components and Real-Time Tasks”. In: *ECRTS*. 2010.

[139] Huazhe Zhang and Henry Hoffmann. “Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques”. In: *ASPLOS*. 2016.

[140] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. “ControlWare: A Middleware Architecture for Feedback Control of Software Performance”. In: *ICDCS*. 2002.

[141] Baoxian Zhao, Hakan Aydin, and Dakai Zhu. “Energy Management Under General Task-Level Reliability Constraints”. In: *RTAS*. 2012.

[142] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. “Survey of Energy-Cognizant Scheduling Techniques”. In: *TPDS* 24.7 (2013).