THE UNIVERSITY OF CHICAGO

MANAGING DIVERSITY IN PERFORMANCE AND ENERGY CHARACTERISTICS ON EMBEDDED SYSTEMS

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

DEPARTMENT OF COMPUTER SCIENCE

BY
CONNOR KELLY IMES

CHICAGO, ILLINOIS
WINTER 2015

Dedicated to those who forgot their power cables.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Minimizing energy under timing constraints is a common task for embedded systems, which are often limited by available energy sources. Solving this problem even for a single system is hard, and current solutions are often designed for specific platforms and applications. Developing general techniques is difficult because of the variety in performance, power, and energy characteristics of different systems and applications. While heuristic solutions have often sufficed in the past, the increase in configurability and heterogeneity of modern systems means that no single heuristic will perform well on all platforms, or even for all applications on a single platform. A portable solution to the problem is needed.

This thesis first identifies two embedded systems that expose different performance/power and latency/energy tradeoff spaces and demonstrates that each achieves acceptable energy consumption under timing constraints by using different heuristics. Both systems use hardware designs that are common in today's cutting-edge devices, thus establishing the problem as a genuine concern for system and software designers. The first is a Sony Vaio tablet, a homogeneous multi-core system with a mobile Intel Haswell processor; the second is an ODROID development platform, a single-ISA heterogeneous multi-core system with an ARM big.LITTLE processor. We show that the Vaio uses a *race-to-sleep* heuristic to achieve acceptable energy efficiency while the ODROID uses a *never-idle* heuristic, and that choosing the wrong strategy can increase energy consumption as much as 14× the optimal. To address this problem, we present POET, a portable library and runtime that manages system resources to achieve near-optimal energy consumption while meeting soft real-time performance constraints. We show that POET meets performance targets with small error while consuming, on average, only 1.3% more energy on the Vaio compared to a dynamic optimal oracle, and 2.9% more on the ODROID. POET is designed to facilitate writing portable, energy-efficient applications by providing a simple and extensible API to users. It is released as open-source software under the BSD License.

CHAPTER 1

INTRODUCTION

Developers for embedded platforms must often write software that meets timing constraints in order to fulfill requirements for real-time systems or to provide quality-of-service guarantees. Since these systems are often limited by available energy (*e.g.,* a battery), energy consumption must also be kept to a minimum.

To assist designers in solving problems like this, systems are becoming more configurable. A common approach is to use dynamic voltage and frequency scaling (DVFS) [52, 64], which allows trading processor performance for power savings. Some systems also support sleep states or power-gating various components likes processor cores, caches, memory controllers, and DRAM [6, 37, 46]. Heterogeneous multi-core architectures, where different cores provide different performance and energy tradeoffs, are becoming more common in embedded devices.

The increasing variety of configurable components results in a diversity in performance/power and latency/energy tradeoff spaces. This exacerbates the already difficult problem of scheduling multiple resources. As a result, many designers implement heuristic solutions to ensure that their software meets timing constraints while approximating minimal energy. In particular, the *race-to-idle* heuristic is common because it is easy to implement and has achieved good results in practice [7, 44, 51, 56]. Other strategies are often designed or implemented for specific platforms and applications [53, 60, 63, 69]. However, recent studies indicate that a single heuristic is no longer likely to perform well on all systems [18, 32, 43]. A general solution to the problem that supports the growing diversity in systems is needed. We address the problem of minimizing energy under soft real-time constraints in a portable manner with POET: the Performance with Optimal Energy Toolkit. To our knowledge, POET is the first feedback control system that achieves near-optimal energy consumption under timing constraints in a portable manner.

1.1 The Need for a General Solution

Table 1.1: Properties for the two embedded systems.

Platform	Processor	Cores	Core Types	Speeds (GHz)	TurboBoost	HyperThreads	Num. Configs
SVT11226CXB	Intel Haswell	2	1	.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

To study the diversity in behavior of configurable components, we evaluate two modern embedded systems – a Sony Vaio tablet with a mobile Intel Haswell processor and an ODROID development board with an ARM big.LITTLE processor. Table 1.1 demonstrates the variety in configurability of these two platforms. For our analysis, we use eight standard benchmarks, none of which were originally designed to provide timing predictability.

It is common to use heuristic approaches to approximate minimal energy while meeting timing constraints, so we are motivated to examine the behavior of heuristics on our two systems. We find that the Vaio uses a *race-to-sleep* heuristic to achieve near-optimal energy efficiency while the ODROID prefers a *never-idle* heuristic instead. If the wrong approach is used, a remarkable amount of energy can be wasted. For example, if the Vaio were to use race-to-idle or never-idle, it consumes almost 50% more energy than optimal on average when meeting a latency target that requires only half of its performance capacity. The ODROID uses, on average, about 3.7× the optimal energy if it were to use race-to-sleep or race-to-idle. Other scenarios can do even worse – up to 14× over optimal on the Vaio and 9× over optimal on the ODROID!

The consequences of choosing the wrong heuristic are unacceptable. An additional burden is now placed on embedded systems developers – software must either be rewritten for different systems or a general solution to the problem must be found. We conclude that a general solution that supports the growing diversity in systems is the preferable choice. The difference in energy consumption offered by a truly optimal, system-specific solution over a near-optimal, portable solution often may not be worth the cost of redesigning and rewriting the software each time a new platform is introduced. Figure 1.1 proposes a high-level design for a portable solution – it receives a *timing constraint* and *resource specification* as input, and provides a *minimal-energy resource*

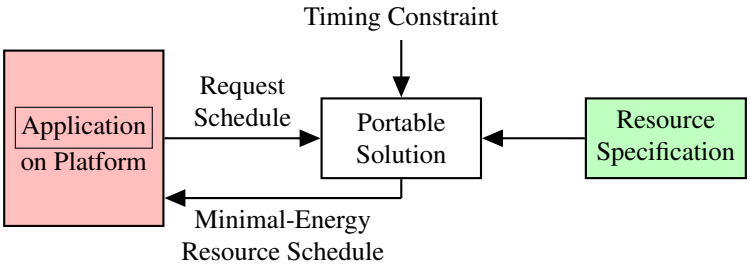

Figure 1.1: A portable design for minimizing energy under timing constraints.

schedule as its output.

1.2 Introducing POET

To meet the need for a general solution to the problem, we introduce POET. This C library automatically schedules system resources without the need for code changes when moving software to new platforms. Our evaluation shows that it meets timing constraints with low error (less than 2% on average), achieves near-optimal energy consumption (2.1% over optimal on average), and is portable between systems. POET uses feedback control to ensure that timing constraints are met, then solves a linear optimization problem to determine a resource allocation schedule for the application that minimizes system energy consumption. Users provide system configuration specifications and a performance target, and POET handles the rest. Integrating POET requires adding only a few extra lines of code (12 - 30) to an application. For capturing the performance and power metrics that POET requires, we update the Application Heartbeats API [31] to read power and energy data from various sources.

We analyze heuristic behavior for eight benchmarks on our two systems to motivate POET's necessity, then evaluate POET using those same applications to demonstrate its effectiveness. Despite the applications not being originally designed for timing predictability, we find that POET provides ease of use, predictable application performance, near-minimal energy consumption, and adaptability to dynamic input and system behavior.

1.3 Summary of Contributions

This thesis makes the following contributions:

1. Extends the Application Heartbeats library [31] with a new *hb-energy* interface to capture power and energy metrics.
2. Examines two real-world embedded systems and determines that they expose different performance/power and latency/energy tradeoff spaces.
3. Shows that different heuristics are required on each system to achieve acceptable energy consumption while meeting timing constraints, establishing the need for a general resource allocation solution to the problem.
4. Describes the design and implementation of POET, *Performance with Optimal Energy Toolkit*, which uses feedback control to provide soft real-time guarantees and linear optimization to minimize energy consumption.
5. Presents an empirical evaluation of POET on the two embedded systems, demonstrating its ability to meet timing constraints with low error and achieve near-optimal energy consumption in a portable fashion.
6. Provides open-source releases of Application Heartbeats 2.0, POET, system configurations, and benchmark patches[1].

The rest of this thesis is organized as follows. Chapter 2 presents background on the problem and related work. Chapter 3 outlines the mathematical formulations used to describe the problem. Chapter 4 describes the platforms we examine and the applications we use. We introduce Heartbeats 2.0 in Chapter 5, then characterize the two systems in terms of power, performance, latency, and energy. Chapter 6 presents the formulation and evaluation of heuristic approaches. The POET framework and implementation are described in Chapter 7, followed by an analysis in Chapter 8. Chapter 9 describes potential future work and Chapter 10 concludes this thesis.

1. Available at `http://poet.cs.uchicago.edu/`

CHAPTER 2

BACKGROUND AND RELATED WORK

Minimizing energy consumption under timing constraints is an important task for systems of all sizes and has been studied extensively both in theory and in practice. It is especially important for embedded platforms where efficient energy consumption is a top priority in system and software design. This chapter begins with a short description on power and energy, followed by a history of the evolution of system architectures and their configurability with respect to performance and power management. We then motivate this work by discussing the need for a general solution to the problem, wrapping up with a section on work that is similar to POET.

2.1 Power and Energy

We begin with a brief background power, energy, and the relationship between them. Elementary physics describes power, measured in Watts, as the rate at which an electric charge passes through an electric potential (voltage) difference. Energy, measured in Joules, is the potential to deliver electric charge; we conceptualize it as a finite resource that can be stored and consumed. Power is the rate of energy consumption.

Static power, the power used when a CMOS integrated circuit (IC) is not switching transistors, is computed as the product of leakage current I_{leak} and supply voltage V_{dd} [62]:

$$P_s = I_{leak} V_{dd} \tag{2.1}$$

Dynamic power, the power used when an IC is switching transistors, is proportional to the effective capacitance C_{eff} of the circuit, the applied voltage V_{dd}, and the clock frequency f [30]:

$$P_d \propto \frac{1}{2} C_{eff} V_{dd}^2 f \tag{2.2}$$

The total power draw in a system includes both static and dynamic power. Voltage and frequency are correlated [42], meaning their values in Eqn. 2.2 cannot be manipulated independently. A higher clock frequency requires higher voltage to maintain stable circuit operation, while conversely, a lower frequency can be sustained with a lower voltage. It then follows that reducing the voltage and frequency can offer significant power savings due to the V^2 relationship.

Energy is computed as the integral of power w.r.t. time:

$$E = \int P \cdot dt \tag{2.3}$$

If power is fixed, energy is trivially computed as the product of power and elapsed time:

$$E = P \cdot \Delta t \tag{2.4}$$

On embedded platforms, energy consumption is especially important because of limited energy resources, like a battery. However, a battery's capacity, and therefore its life, does not scale linearly with current draw [47]. An approximation of a battery's capacity was determined empirically by Peukert in the late 1800s:

$$C = \frac{k}{I^{\alpha}} \tag{2.5}$$

In this formulation, k is a constant determined by the chemical properties and design of the battery, and I is the current being drawn. In a perfect battery, $\alpha = 0$, making the battery capacity fixed. Higher power consumptions typically draw higher current because of the increase in voltage necessary to keep the circuit behavior stable. This also results in more heat being generated which causes an increase in electrical resistance in the circuit, meaning even more current is required to maintain stable operation. In general, lowering power consumption not only extends the battery life but also increases the total amount of energy that can be delivered before it is depleted.

2.2 Evolution of Architecture

Performance and power considerations impact the design and configurability of computer systems. Modern systems expose *configurable* components which allow software to manipulate the system state to exploit the tradeoffs between performance and power. Effective management of these resources can lead to significant energy savings, whereas improper use can waste energy reserves.

Processors draw a large portion of the total power in modern computers. For systems of all sizes, it is common to control processors and individual cores to trade performance and power using dynamic voltage and frequency scaling (DVFS) [52, 64]. It is critical to note that the energy consumption required to complete a fixed amount of work depends not only on the power being drawn, but on the work rate achieved at a particular clock frequency [57]. If a work rate is disproportionately low compared to the power saved by reducing voltage and frequency, energy may not be saved and could be wasted instead. In short, lowering voltage and clock frequency is not always more energy efficient. DVFS has been widely explored in the past and theoretically optimal scheduling algorithms are available for uniprocessor systems [4].

Many systems now support varying levels of processor *sleep states* [37] and power-gating components [6, 46] to further reduce power usage. With the end of Dennard scaling [21, 23], single processor performance has begun to stall despite the introduction of super-scalar, out-of-order processors with aggressive branch prediction and larger caches [30]. Multi-core and multi-processor systems are now the norm and are becoming more prevalent on embedded systems.

Until recently, the primary compute resources of multi-core and multi-processor systems have typically been *homogeneous*, where each processor and core is identical. Even so, research in scheduling for *heterogeneous* systems dates back to at least the mid-2000s [40, 41]. Heterogeneous computing is often discussed using different vocabulary, sometimes depending on specific features like the instruction set architecture (ISA) or the performance and power characteristics of the processors. Other times the phrasing is just a matter of preference. Elewi et al. provide an overview on keeping track of the terminology [22]. To summarize, systems with multi-cores that support differ-

ent ISAs are often called *heterogeneous multiprocessor (HMP)* platforms, while those that share an ISA are referred to as *asymmetric multiprocessor (AMP)* platforms. Alternative names for the latter include *uniform multiprocessors*, *uniform heterogeneous multiprocessors*, *single-ISA heterogeneous multi-core architectures*, and *shared-ISA asymmetric multi-core architectures*. We refer to the ARM big.LITTLE processor used in this work as a single-ISA heterogeneous multi-core.

Heterogeneous systems are now becoming widespread in the mobile device domain, promising better energy efficiency and longer battery life. This trend is expected to continue for systems of all sizes as energy becomes a key limiter in performance and therefore crucial in the design of future processors [12]. We already see more aggressive designs using heterogeneity to increase energy efficiency, such as GreenDroid [26], which generates customized cores from hot segments of code. Recent research has yielded power-aware scheduling approaches for heterogeneous systems that are both theoretical or simulated [19, 22] and evaluated experimentally [54, 59]. For example, Chen and John project core configurations and program resource demands to a unified multi-dimensional space and use the weighted Euclidean distance between the two to guide scheduling, reducing energy consumption and increasing throughput [19]. Muthukaruppan et al. build a framework using price theory (from economics) to build a distributed and scalable power management system for heterogeneous multi-cores under thermal design power constraints which supports DVFS, load balancing, and task migration [59]. Cao et al. evaluate software written in managed languages to determine how to schedule the software and associated Virtual Machine services (like the interpreter and garbage collector) to achieve good performance and energy consumption [16].

Empirical studies have shown that managing multiple resources simultaneously is more energy efficient than managing a single resource alone [7, 18, 20, 66, 67]. While we explore the effects of manipulating DVFS frequency, core count, and core type to save energy, there are often other configurable components in systems. Memory controllers, network bandwidth, disk allotment, LCD and backlight settings are all examples of other resources that can be manipulated to reduce power and energy consumption [10, 17, 49]. Maggio et al. manage core allocation and clockspeed [45]

while Bitirgen et al. coordinate frequency, cache, and memory bandwidth on a chip multiprocessor (CMP) [11]. AbouGhazaleh et al. coordinate processor speed and cache in PACSL [3].

As a result of these evolutions, managing timing and energy on embedded systems is becoming a more complex task. To best address problems relating to timing and energy, solutions should support both homogeneous and heterogeneous multi-core platforms, and account for multiple configurable components.

2.3 Motivation for a Portable Solution

Even for uniprocessor systems with DVFS and idle states, scheduling for optimal energy consumption under timing constraints is an intractable problem, though some algorithms have been shown to be within a bounded distance of optimal [5]. Studies performed in the early and mid-2000s found that the complexity of theoretically optimal scheduling algorithms provide little to no benefit on real systems [7, 44, 51, 56]. Instead, heuristic approaches are often used to meet performance or latency requirements while approximating minimal energy. These studies show that the *race-to-idle* heuristic meets timing constraints and often achieves close to optimal energy consumption. More recent studies, however, indicate that this trend is starting to change [18, 32, 43].

Because of the variability in systems, developers for embedded software often optimize approaches for specific applications or platforms to achieve good energy consumption. For example, both Thiagarajan et al. [60] and Zhu et al. [69] focus their efforts on mobile web browsers. Thiagarajan et al. analyze browser energy consumption and Zhu et al. investigate scheduling them on heterogeneous systems to minimize energy under performance constraints. Petrucci et al. [53] build an energy-aware scheduler for a specific heterogeneous architecture style (essentially a big.LITTLE). Wang et al. [63] develop a real-time instruction-level loop scheduling technique for reducing leakage energy specifically for VLIW architectures. Such solutions for minimizing energy are not portable, meaning they must be redesigned and rewritten when applying to new applications or platforms. In addition, new hardware may expose different configurable resources

and present different performance and power tradeoffs.

We are therefore motivated to examine state-of-the-art embedded platforms and study their requirements for optimal energy consumption under timing constraints. The heuristic evaluations presented in this thesis (Chapter 6) confirm the non-portability of heuristics on embedded platforms using hardware that is common today – a trend we expect to continue. As a result, resource scheduling approaches must be re-evaluated by either seeking new heuristics or creating new algorithms.

2.4 Related Work

We create a portable solution to the resource allocation problem that abstracts software from system-specific resources and configurations. Our solution, a library and runtime called POET, uses control theory to meet timing constraints and linear programming to minimize energy consumption. Control theory techniques provide a formal framework for reasoning about dynamic behavior in systems, and previous approaches have successfully used feedback control to manage timing constraints [8, 25, 31, 33, 55, 61, 68]. However, POET users do not need to be experts in control theory or optimization. To allow their applications meet timing constraints and minimize energy in a portable manner, they only need to make small changes to their code to integrate POET and let the runtime handle the rest. This remainder of this section describes related work that is similar to POET – specifically, solutions that behave as a middleware or runtime and those that use control theory.

Rajkumar et al. propose Q-RAM (QoS-based Resource Allocation Model) [55] for allocating multiple resources to concurrent applications to maximize system utility and meet real-time constraints. Q-RAM uses feedback from the operating system to measure resource consumption for applications, but the approach is not energy-aware nor is the runtime overhead quantified, making it difficult to determine its usefulness in practice. Sojka et al. [58] create a middleware framework for soft real-time constraints with a focus on abstraction, modularity, and portability between hard-

ware and software platforms. It supports applications that do not have a concept of jobs, which POET requires, but does not address energy concerns and requires a non-trivial amount of manual work to determine suitable system parameters for applications. Zhang et al. present ControlWare [68], another middleware layer for meeting QoS constraints with a focus on abstraction. ControlWare uses control theory like POET, but targets internet services executed on servers rather than software on embedded systems. It is also not energy-aware.

Snowdon designs and implements Koala [57], an OS-level approach for performance, power, and energy management that supports a wide range of platforms, from embedded systems to high-end servers. While Koala supports various policies and performs well on different classes of systems and applications, it wholly depends on models based on prior system characterization (*i.e.,* no feedback at runtime) and does not provide any formal guarantees. Hoffmann et al. introduce PTRADE [31], which is the most similar work to POET – it uses feedback control to meet performance goals, but minimizes power rather than energy consumption. Additionally, PTRADE uses heuristic solutions rather than a truly optimal scheduling algorithm.

CHAPTER 3

ENERGY OPTIMIZATION FORMULATION

This chapter generalizes the resource allocation problem for completing tasks under timing constraints while minimizing energy consumption. Prior work by Hoffmann expresses the problem of minimizing energy under timing constraints as a linear program [32], which we expand on with the following formulations. First, we assume the following about a given task:

1. Begins at time 0.
2. Has a deadline at time τ.
3. Has a base application speed b, which is achieved when using a minimum system resource allocation.
4. Has W work units in its workload.

We assume following about the system configurations:

1. $\exists C$ configurations s.t. $\forall c, c \in \{0, \ldots, C-1\}$
2. $\forall c, \exists s_c$ *speedup* over b.
3. $\forall c, \exists p_c$ *powerup*, a power consumption value normalized to that of configuration β, where $s_\beta = 1$ (*i.e.,* β is the configuration that results in the base application speed).
4. By convention, $c = 0$ is the system *sleep* state with $s_0 = 0$ and $p_0 = 0$.
5. By convention, $c = 1$ is the system *idle* state with $s_1 = 0$ and $p_1 = p_{idle}$, a constant property of the system.
6. By convention, $c = C - 1$ is the configuration that allocates all resources to a task.

For example, in a dual-core system with five DVFS settings, there are 12 configurations ($2\,cores \times 5\,speeds + sleep + idle$). Then for scheduling a task using the provided configurations, $\forall c$:

1. $r_c = s_c \cdot b$ is the true performance (work rate) achieved for a task running in configuration c.
2. $\exists \tau_c$ s.t. $0 \leq \tau_c \leq \tau$ where τ_c is the amount of time scheduled for configuration c.
3. $\tau_c \cdot r_c$ is the work completed in configuration c.
4. $\tau_c \cdot p_c$ is the energy consumed in configuration c.

Now we formulate energy minimization under a timing constraint with the following linear program:

$$\textit{minimize} \quad \sum_{c=0}^{C-1} \tau_c \cdot p_c \tag{3.1}$$

$$s.t. \quad \sum_{c=0}^{C-1} \tau_c \cdot r_c = W \tag{3.2}$$

$$\sum_{c=0}^{C-1} \tau_c = \tau \tag{3.3}$$

$$0 \leq \tau_c \leq \tau, \qquad \forall c \in \{0, \ldots, C-1\} \tag{3.4}$$

Furthermore, linear programming theory states that an optimal solution exists when $\forall i \in \{0, \ldots, C-1\}, \tau_i = 0$, except for, at most, two distinct values of i [13, 39].

Our formulation assumes that a single application is running on the system and therefore can take full advantage of a configuration's performance capacity and power consumption. The ability to meet timing constraints and minimize energy is restricted to the set of known configurations. In Chapter 6 we will describe how different heuristics map onto this formulation. Chapter 7 will describe how POET solves the linear program defined by Eqns. 3.1–3.4.

CHAPTER 4

PLATFORMS AND APPLICATIONS

4.1 Platforms

This section describes the two platforms we characterize, analyze heuristics for, and use for our evaluation of POET. Both systems run Ubuntu 14.04 LTS and use the `cpufrequtils` interface to control DVFS settings. Table 4.1 presents an overview of the two systems' configurability. Table 4.2 shows their basic power characteristics.

4.1.1 Sony VAIO SVT11226CXB

The Sony VAIO SVT11226CXB tablet PC is a homogeneous dual core system with a mobile Intel Haswell processor (22nm technology). It supports hyperthreading, TurboBoost, and eleven DVFS settings ranging from 600 MHz to 1.501 GHz, the last of which enables TurboBoost. The system uses Linux kernel 3.13.0.

The Vaio claims to support running at different DVFS frequencies on different virtual cores, but our experience leads us to conclude that this is not actually the case. Using mixed values results in non-deterministic performance behavior. As a result, we only allow configurations where all cores are set to the same frequency. As seen in Table 4.1, the best performance improvement (speedup) on the Vaio is gained by increasing the DVFS frequency.

We measure power consumption on the Vaio using the Haswell processor's Model-Specific Register (MSR), which provides energy consumption metrics.

4.1.2 Hardkernel ODROID-XU+E

The ODROID-XU+E [28] development platform is a heterogeneous system with an ARM big.LITTLE processor. Specifically, it has the Samsung Exynos5 Octa SoC (28nm technology) containing quad

Table 4.1: System configurations.

System	Configuration	Settings	Max Speedup
Vaio	clock speed	11	2.72
	cores	2	1.81
	hyperthreads	2	1.10
ODROID	big cores	4	6.10
	big core speeds	9	1.97
	LITTLE cores	4	3.94
	LITTLE core speeds	8	2.40

Table 4.2: System power characteristics.

System	Idle Power	Min Power	Max Power
Vaio	2.50 W	3.04 W	8.05 W
ODROID	0.12 W	0.17 W	8.14 W

core Cortex-A15 (big) and Cortex-A7 (LITTLE) clusters. The Cortex-A15 cores support nine DVFS settings, from 800 MHz to 1.6 GHz. The Cortex-A7 cores support eight DVFS settings, from 500 MHz to 1.2 GHz. All cores in a cluster must operate at the same frequency. This system uses Linux kernel 3.4.104. Unlike the Vaio, the ODROID gains the most speedup by increasing the number of cores being used for a task (Table 4.1).

The ODROID uses the In Kernel Switcher (IKS) for managing processor assignment through cluster migration. IKS maintains compatibility with existing schedulers by modifying existing DVFS systems, but at a cost of supporting execution on only one cluster at a time. On the ODROID, compatibility is maintained by using dummy DVFS frequencies for the LITTLE cores so that the list of available frequencies do not overlap between the big and LITTLE cores. The Exynos cpufreq driver presents the LITTLE core frequencies at half their actual values – 250 MHz to 600 MHz.

The XU+E model sports four integrated INA-231 power sensors [35] for the Cortex-A15 cluster, Cortex-A7 cluster, DRAM, and GPU. We poll these sensors to collect power metrics on the ODROID.

4.2 Applications and Inputs

For our analyses, we use eight benchmarks to represent a variety of embedded applications, none of which were originally designed to meet timing constraints. This section describes the applications and the inputs used.

4.2.1 Benchmarks

The first five applications we choose are from the PARSEC benchmark suite [9]. Specifically, we use `blackscholes`, `bodytrack`, `facesim`, `ferret`, and `x264`. Both bodytrack and x264 process video input and could reasonably be required to perform at least as well as the frame rate of a live feed (*e.g.,* from an on-board video camera). Ferret is a toolkit for content-based similarity search of non-text data and should be able to satisfy a latency requirement for returning results to a user. Facesim creates realistic animations of a human face from a model and time sequence of muscle movements, and can also be expected to maintain a real-time frame rate.

The next two applications, `dijkstra` and `sha`, are from the ParMiBench benchmark suite [36]. Dijkstra performs single-source shortest path calculations on a graph. Two parallel implementations are available, so we choose the multiple queue implementation with the expectation that it scales out better than the single queue implementation. SHA (Secure Hash Algorithm) is a one-way hash function used in cryptography. On embedded systems, it can be used for secure storage and transmission of data and must perform fast enough to ensure timely communication.

The last application is `STREAM` [48], a synthetic benchmark for measuring sustainable memory bandwidth. STREAM represents a variety of memory-bound applications.

These benchmarks perform tasks that can reasonably be expected to run on embedded systems. They exhibit a range of behavioral characteristics commonly found in embedded multithreaded applications and are therefore representative of a wide range of embedded system workloads.

Table 4.3: Benchmark input 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

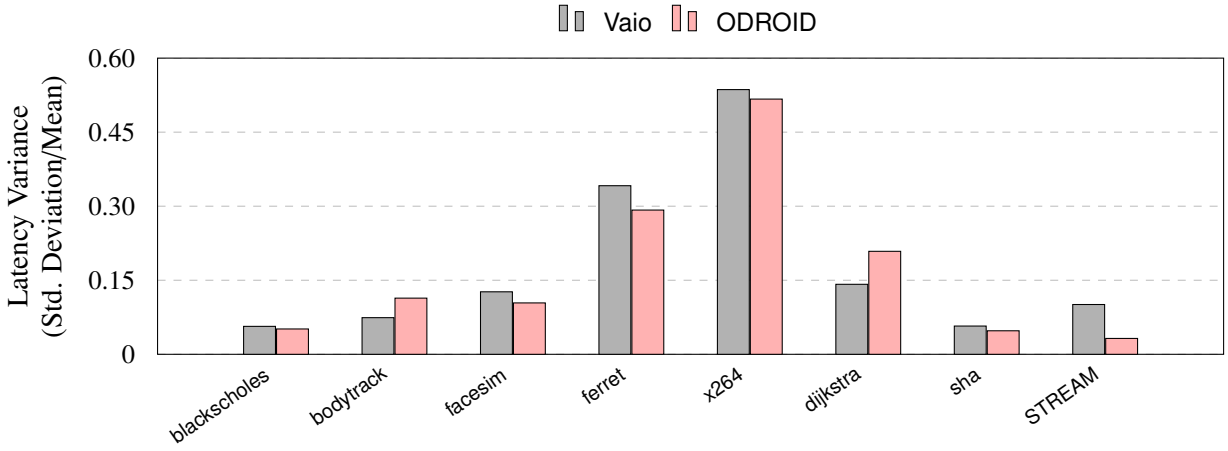

Figure 4.1: Application timing variability.

4.2.2 Inputs

All inputs used, presented in Table 4.3, are packaged with the original benchmarks, with the exception of the x264 input which is taken from a set of standard test sequences. Also, the blackscholes input is a subset of the original, using one million instead of ten million input lines. Figure 4.1 shows the timing variability of each benchmark using the aforementioned inputs, computed as the ratio of the standard deviation to the mean. These results indicate that the benchmarks have a range of performance behavior. Low variance implies natural predictability (*e.g.,* blackscholes and sha), whereas high variance implies more uncertainty (*e.g.,* ferret and x264). The results are similar for both platforms, indicating that the behaviors are a fundamental property of the applications, not the systems. We will discuss benchmark behavior further in Chapter 6.

CHAPTER 5

CHARACTERIZING SYSTEMS

5.1 Application Heartbeats 2.0

Collecting actionable performance and power data in a portable manner requires a general framework for capturing metrics and making them available both to software and to developers. The Application Heartbeats API [31], developed by Hoffmann et al., has been used previously to capture performance metrics for applications like the ones used in this thesis. We extend its functionality to additionally collect power and energy data.

The applications we consider have high-level loops, with a known amount of work completed in each loop iteration. A heartbeat can then be issued each time the loop is executed (or any X loop executions, where X is a positive integer). The Application Heartbeats library tracks performance by measuring the latency between the heartbeats. It then makes this data available for querying at runtime by other automatic systems, or to developers via a structured log file.

The original Application Heartbeats API is portable between systems as it does not require any interaction with the system or other hardware beyond reading the system clock, a capability that is available using standard software interfaces. Power and energy readings, on the other hand, require hardware support that is not inherently portable. In response, we develop a simple API called *hb-energy* to abstract capturing power and energy data from different sources. We assume that we have permission to access the required hardware resources and that the access is timely with minimal overhead (*i.e.,* capturing data does not unreasonably distort the results). No assumptions are made about what resource an implementation actually measures. For example, some implementations may collect energy data from a processor while others collect power readings for an entire system and convert the results to energy values. The Heartbeats library collects energy readings using the hb-energy interface, making its implementation portable. It then exposes power data for heartbeats to external sources, in addition to the performance data it already provided.

Table 5.1: API functions for hb-energy.

Function	Return Value	Parameters
hb_energy_init	int	void
hb_energy_read_total	double	int64_t *last_hb_time*, int64_t *curr_hb_time*
hb_energy_finish	int	void
hb_energy_get_source	char*	void
hb_energy_impl_alloc	hb_energy_impl*	void

The hb-energy API provides three core functions:

- **hb_energy_init**: Initialize hardware resources, global state variables, polling threads, etc.
- **hb_energy_read_total**: Return the energy consumed since the last reading.
- **hb_energy_finish**: Cleanup anything previously initialized.

Two auxiliary functions are also included:

- **hb_energy_get_source**: Returns a unique, human-readable name of the data source.
- **hb_energy_impl_alloc**: Allocates and returns the reference to a data structure containing function pointers for the implementation (effectively, a factory function).

Table 5.1 provides parameter and return value details for these functions.

The new Heartbeats library includes implementations of the hb-energy interface for both common and unique hardware. The first implementation gathers energy readings from an Intel Model-Specific Register (MSR) – it works with a variety of Intel processors and can be configured to read from multiple MSRs simultaneously. The second is specific to the ODROID-XU+E development board – it polls four embedded INA-231 power sensors for the Cortex-A15 cluster, Cortex-A7 cluster, DRAM, and GPU and converts power readings to energy. The third polls a WattsUp? PRO [2] external power monitor and converts power readings to energy. A dummy implementation is also included.

If needed, users can write their own implementations for different power/energy monitoring resources without the need to change any code in the Heartbeats library. They need only link with their custom implementation when compiling and deploying their application.

Integrating Heartbeats with an application is straightforward, and requires only a few addi-

```
// initialization
heartbeat_t* heart =
  heartbeat_acc_pow_init(window_size, buffer_depth, "heartbeat.log",
                         min_heartrate, max_heartrate, min_accuracy, max_accuracy,
                         1, hb_energy_impl_alloc(), min_power, max_power);
// execution of main loop
while(running) {
  heartbeat_acc(heart, count++, 1);
  doWork();
}
// cleanup
heartbeat_finish(heart);
```

Listing 5.1: Example of Heartbeat-enabled application code.

tional lines of code. Listing 5.1 presents a simple example, with variable declarations excluded for brevity. Heartbeats are typically made before or after each iteration of a main program loop. We consider each iteration to be the completion of a job, *e.g.,* encoding a single frame in the x264 benchmark. Up to *buffer_depth* heartbeat records are kept in memory at any time, which must be at least as large as *window_size*. The variables that begin with "min" and "max" declare a desired range for the metrics that are measured (performance, accuracy[1], and power), though it is outside the scope of the Heartbeats library to provide any enforcement of these declarations. The call to `hb_energy_impl_alloc` accesses the hb-energy interface to get the energy reader implementation at runtime in a portable fashion.

To summarize, Application Heartbeats 2.0 extends the original Application Heartbeats API to provide accuracy and power measurements in a portable manner. It is easy to integrate with existing software and extend when necessary.

5.2 Performance and Power

To *characterize* a system for an application, we configure the application to use Heartbeats 2.0 as described in Chapter 5.1. This requires only minimal code changes, and no other application behavior is affected. We then execute the application in all possible system configurations on each of our devices and collect the average performance and power consumption for each execution. While

1. Accuracy quantifies the quality of an application's result, but is not considered in this work.

the problem we address is minimizing energy (not power) under timing constraints, we must first understand the performance and power characteristics of the system configurations. This knowledge allows us to analyze the energy consumption of heuristics and to predict energy consumption at runtime when determining which system states to use in POET.

This section presents the performance and power characteristics for the Vaio and ODROID for various classes of applications. We first examine `x264`, which is representative of most of the applications we use. `STREAM` represents behavior for a class of memory-bound applications on both systems. Finally, `blackscholes` presents a performance and power tradeoff space similar to `x264`, but different enough on the ODROID to justify a separate analysis.

5.2.1 Vaio

The Vaio presents performance and power tradeoff spaces that would likely be familiar to many developers. Each point in Figure 5.1 represents the performance and power characteristics of a unique configuration for the `x264` application. Most of the other benchmarks produce similar behavior. The results spread across the range of attainable performance and power values in a linear manner. The four fastest (rightmost) clusters of values represent the behavior of all but the two slowest DVFS frequencies for one, two, three, and four cores. Although the system exposes eleven DVFS settings, all but the slowest two achieve the same performance and power, meaning the system is probably just running at the same frequency. As a result, most of the DVFS settings have no effect on performance and power. The two slowest DVFS settings behave differently from the higher ones and from each other, as is typically the case for DVFS. We saw this same behavior for all applications we tested, indicating that it really is a property of the system, not a particular application. Since the Vaio is a real device that can be purchased off-the-shelf, software needs to deal with this sort of behavior in practice. Any scheduling approach that estimates performance based on DVFS frequency would likely perform poorly on this system without first accounting for this unexpected behavior.

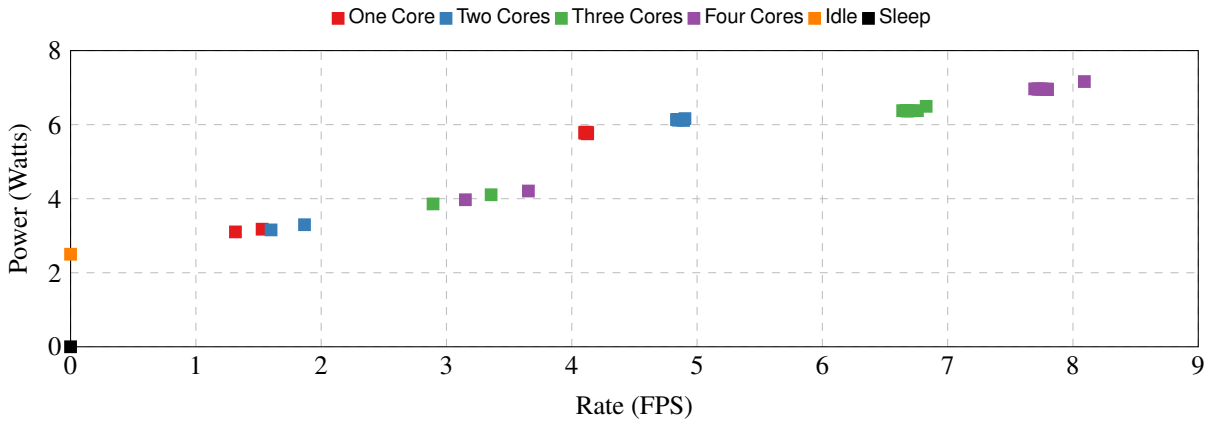

Figure 5.1: Performance and power tradeoffs for x264 on the Vaio.

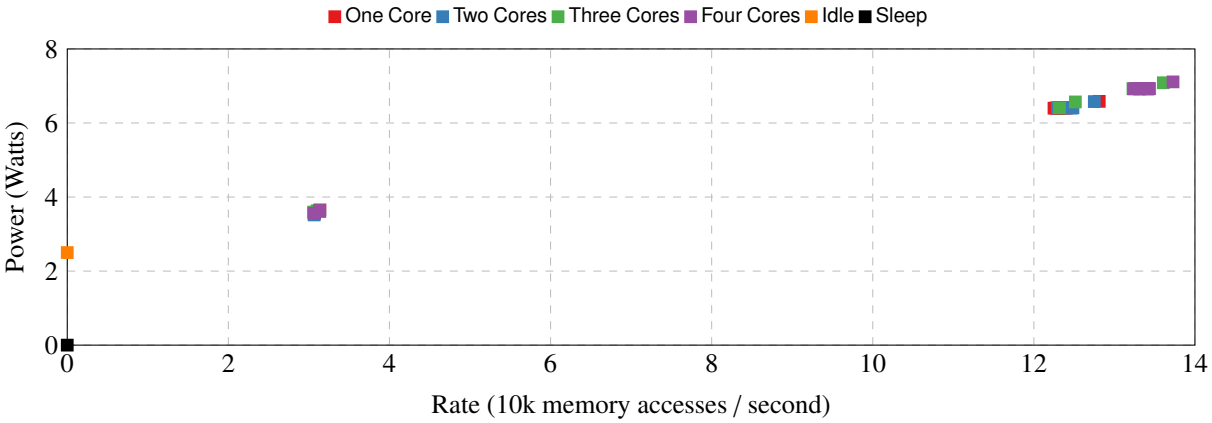

Figure 5.2: Performance and power tradeoffs for STREAM on the Vaio.

`STREAM` produces a tradeoff space with some notable differences, as seen in Figure 5.2. Perhaps the most obvious difference is the absence of configurations with performance and power values that lie between the extremes. Most points are clustered in the upper right (high performance, high power) and a few are clustered in the lower left (low performance, low power). Increasing the number of cores no longer results in any performance gain, which makes sense for memory-bound applications. We determine that there are only four Pareto-optimal states in this tradeoff space, typically using just a single core.

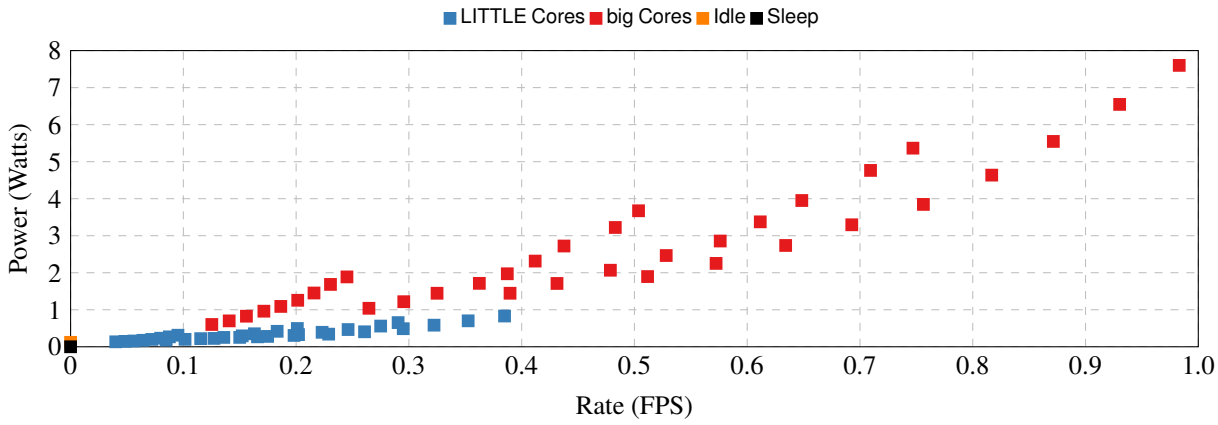

Figure 5.3: Performance and power tradeoffs for x264 on the ODROID.

5.2.2 ODROID

The ODROID presents performance and power spaces that are clearly distinct from those produced by the Vaio. The `x264` application again provides a similar tradeoff space to most other benchmarks, so we present this as an example in Figure 5.3. Here we plot the configurations by core type rather than the number of cores being used. For the big cores, it is clear which sections of the plot correspond to one, two, three, and four cores. Note that the x-axis is scaled differently than with the Vaio - the ODROID's actual performance is much lower. A single big core offers performance between 0.13 and 0.25 FPS and four cores offer performance between 0.51 and 0.98 FPS for this particular video. It is more difficult to discern for the LITTLE cores, but a single core achieves between 0.040 and 0.095 FPS while four cores achieve between 0.17 and 0.38 FPS. LITTLE cores use very little power, yet still overlap with the performance offered by many of the big core states, most notably when using just one or two cores from the big cluster. It is almost never desirable to use fewer than all four of the big cores – only a couple of states with three big cores are arguably useful as they provide performance between those offered by the LITTLE cores and those offered by all four big cores at a power level that is slightly lower than the best offered by all four big cores. The gap in performance and power between the Pareto-optimal LITTLE and big-core configurations is not very large.

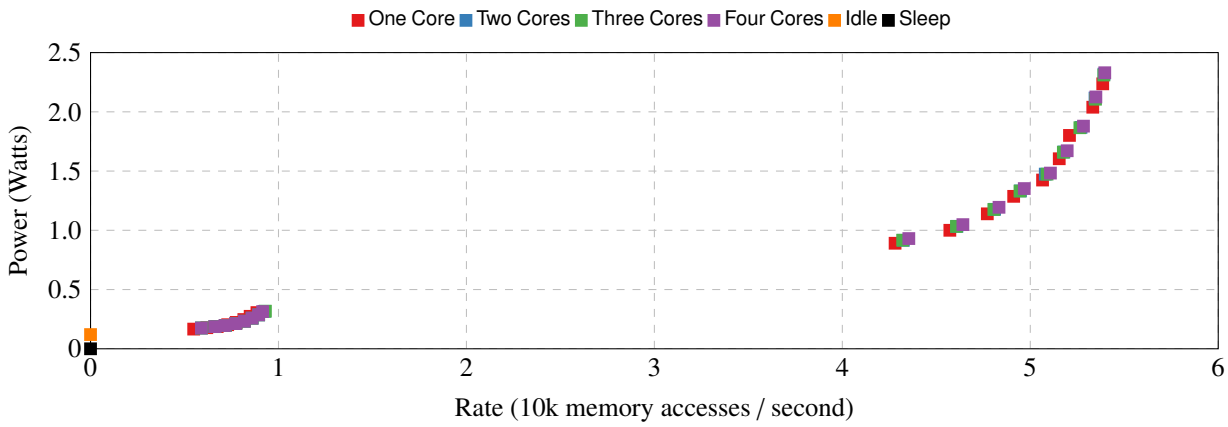

Figure 5.4: Performance and power tradeoffs for STREAM on the ODROID.

`STREAM` again produces a performance and power tradeoff space that is quite different from `x264`. In Figure 5.4, as with the Vaio, there is an absence of states between the extremes and a large gap in both performance and power between the clusters of states. The curves appear to have collapsed together such that varying the number of cores no longer affects the performance and power behavior, as also seen on the Vaio. Again, this is what we should expect for memory-bound applications. The close proximity of the points makes it difficult to determine the Pareto-optimal states, but using only states with a single core achieves good results in practice. While there is a correlation between power consumption and energy consumption, we will see later that although the configurations that use LITTLE cores are Pareto-optimal in the performance and power tradeoff space for `STREAM`, they are not optimal in the latency and energy tradeoff space.

Lastly, we examine the behavior of `blackscholes` on the ODROID in Figure 5.5. While it appears similar to `x264`, it is different enough to warrant attention. The performance range of the LITTLE-core configurations extends further into the range of the big-core configurations – four of the previously Pareto-optimal states that use four big cores are no longer optimal. This results in a larger power gap between the optimal states with LITTLE cores and those with big cores while only gaining a small amount of performance.

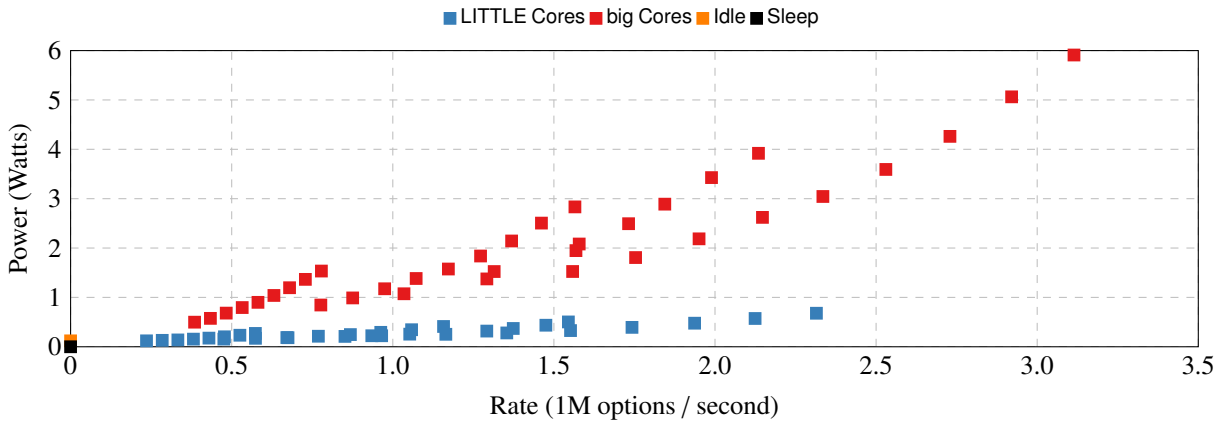

Figure 5.5: Performance and power tradeoffs for blackscholes on the ODROID.

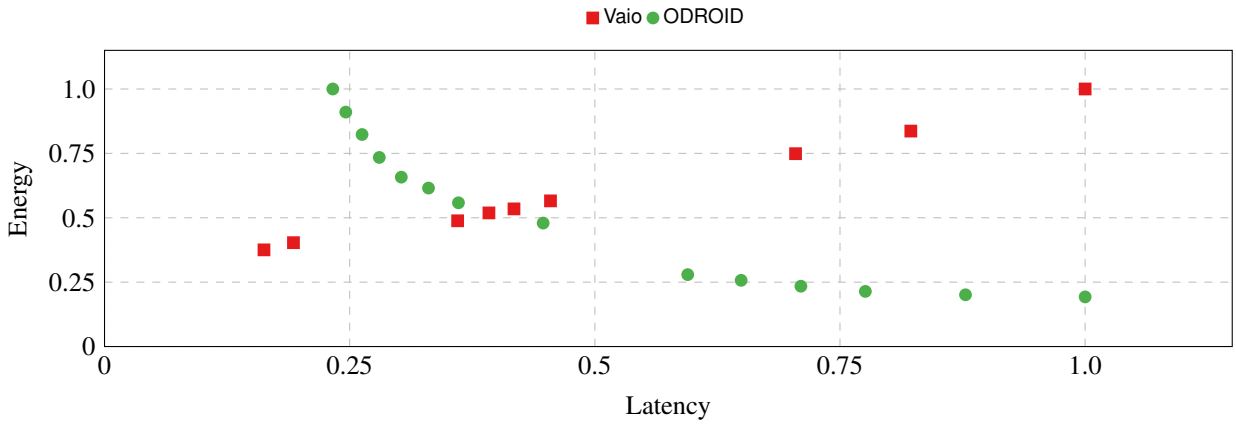

Figure 5.6: Latency and energy tradeoffs for x264.

5.3 Latency and Energy

While we measure application performance, the embedded systems community prefers to discuss scheduling in terms of meeting deadlines, formulated as meeting latency targets rather than performance goals. Also, the problem we address is minimizing energy, not power. With the performance and power characteristics of the Vaio and ODROID determined, we analyze the systems in terms of latency and energy. Figure 5.6 presents the latency and energy tradeoff spaces for both systems using the `x264` application. Only Pareto-optimal states from the performance and power tradeoff space are included – non-optimal states are discarded. Latency is normalized to the longest latency offered by an optimal state, *i.e.,* the lowest performing state. Energy is normalized to the

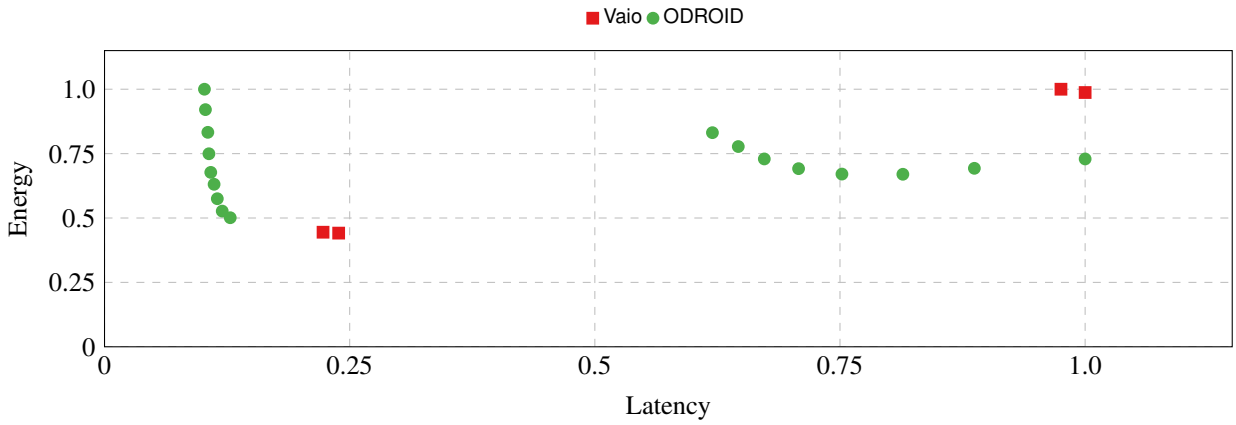

Figure 5.7: Latency and energy tradeoffs for STREAM using Pareto-optimal states from performance and power tradeoff space.

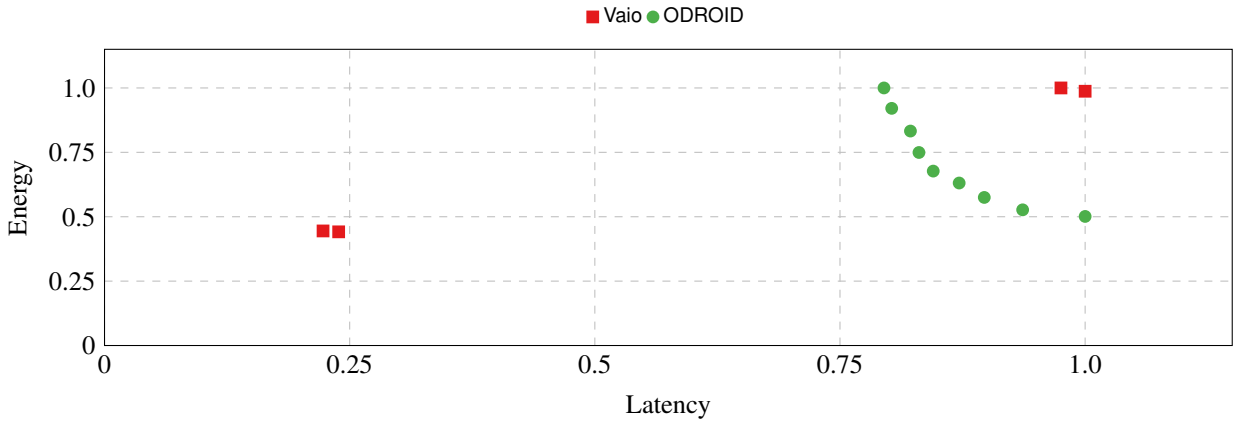

Figure 5.8: Latency and energy tradeoffs for STREAM with the ODROID's non-optimal LITTLE-core states removed.

highest-energy state.

The Vaio uses more energy as the latency increases meaning that faster configurations are more efficient. In fact, only the fastest state (lower left) is Pareto-optimal in this space – all other states both perform slower and are less energy efficient. Conversely, the states with higher latency use less energy on the ODROID – up to 4× less, in fact. As a result, if an application is willing to deal with higher latency (lower performance), it can gain significant energy savings.

`STREAM` presents a latency and energy tradeoff space for the ODROID in which an entire cluster turns out not to be optimal. Figure 5.7 shows the results of applying optimal performance and power states to the latency and energy tradeoff space. Despite offering power savings over the

ODROID's big cores, the LITTLE cores do not offer energy savings as they achieve similar energy consumption, but at a higher latency. This behavior is counter-intuitive – we should expect applications that are not compute-bound to achieve better energy efficiency on the low-power LITTLE cores. The failure to do so is likely a result of the differences in the caches between the clusters [1]. The LITTLE cluster (Cortex-A7) per-core L1 data caches support simple pattern-based automatic prefetching. The big cluster (Cortex-A15) per-core L1 data caches support out-of-order, speculative, non-blocking loads. Its L2 data cache is larger (2 MB compared to the LITTLE cluster's 512 KB) and has a more advanced prefetcher than the LITTLE cluster's L1 caches. At least in some cases, the microarchitecture of the LITTLE cores makes them unsuitable for memory-bound applications like `STREAM`.

If we remove the LITTLE cores from the tradeoff space and re-normalize, the result is Figure 5.8. Behavior similar to that seen with `x264` returns, where higher latency states offer better energy consumption. However, the available ranges for both latency and energy are less than with `x264` on the ODROID. While this provides less flexibility, the ODROID already consumes significantly less power running `STREAM` than it does running `x264` – see the raw power values presented in Figures 5.3 and 5.4. The Vaio, on the other hand, still offers latency and energy ranges similar to those it offers with `x264`.

CHAPTER 6

HEURISTICS EVALUATION

Practical difficulties in solving the optimization problem described in Chapter 3 often lead developers to use heuristic approaches. We consider three heuristics that map on to the structure of this problem as they meet the constraints, *i.e.,* they complete the required work by the given deadlines (Eqns. 3.2–3.4). They are not guaranteed to result in optimal energy consumption, but in many cases work well in practice.

This chapter examines the impact of the three heuristics on energy consumption for the Vaio and ODROID. First, we describe formulations of the *race-to-sleep*, *race-to-idle*, and *never-idle* heuristics. We then examine the energy consumpion for a single latency target on all applications tested. Finally, we analyze the impact that the latency target has on the energy consumpion of the heuristics.

6.1 Formulations

The first heuristic is *race-to-sleep*, which makes all resources available until a task completes, then puts system components to sleep. Our formulation of a perfect race-to-sleep assumes that no power is used by the measured components (*e.g.,* the processor) while in a sleep state. This maps onto our formulation by scheduling $c = C - 1$ until the job is finished, then scheduling the remaining time in $c = 0$. Formally:

$$\tau_{C-1} = \frac{W}{r_{C-1}} \tag{6.1}$$

$$\tau_0 = \tau - \tau_{C-1} \tag{6.2}$$

$$\tau_c = 0, \forall c \neq C - 1, 0 \tag{6.3}$$

Similarly, *race-to-idle*, also known as *race-to-complete* or *race-to-halt*, makes all resources

available until a task completes, then idles the system. Recall that an idle state has a fixed non-zero power consumption that is system-specific. Like *race-to-sleep*, *race-to-idle* maps onto our formulation by scheduling $c = C - 1$ until the job is finished, then scheduling the remaining time in $c = 1$ (instead of $c = 0$). Formally:

$$\tau_{C-1} = \frac{W}{r_{C-1}} \tag{6.4}$$

$$\tau_1 = \tau - \tau_{C-1} \tag{6.5}$$

$$\tau_c = 0, \forall c \neq C - 1, 1 \tag{6.6}$$

The last heuristic is *never-idle*, which attempts to keep the system busy and complete the task at the deadline. This may be achieved by using a subset of the available resources. *Never-idle* schedules time in two configurations, which we call *hi* and *lo*. The *hi* state is the lowest-power configuration that performs faster than is necessary to complete the task by the deadline. The *lo* state is the most energy-efficient configuration that is slower than necessary. Formally:

$$hi = \arg \min_{c \in \{0,\ldots,C-1\}} \left\{ p_c \mid r_c \geq \frac{W}{\tau} \right\} \tag{6.7}$$

$$lo = \arg \max_{c \in \{0,\ldots,C-1\}} \left\{ \frac{r_c}{p_c} \mid r_c < \frac{W}{\tau} \right\} \tag{6.8}$$

The resulting time spent in each state is then:

$$\tau_{hi} = \frac{W - r_{lo} \cdot \tau}{r_{hi} - r_{lo}} \tag{6.9}$$

$$\tau_{lo} = \frac{r_{hi} \cdot \tau - W}{r_{hi} - r_{lo}} \tag{6.10}$$

$$\tau_c = 0, \forall c \neq hi, lo \tag{6.11}$$

Race-to-idle is, for obvious reasons, easy to implement on any system – simply allocate all resources, then transition to the idle state when the required work is complete. It requires no insight

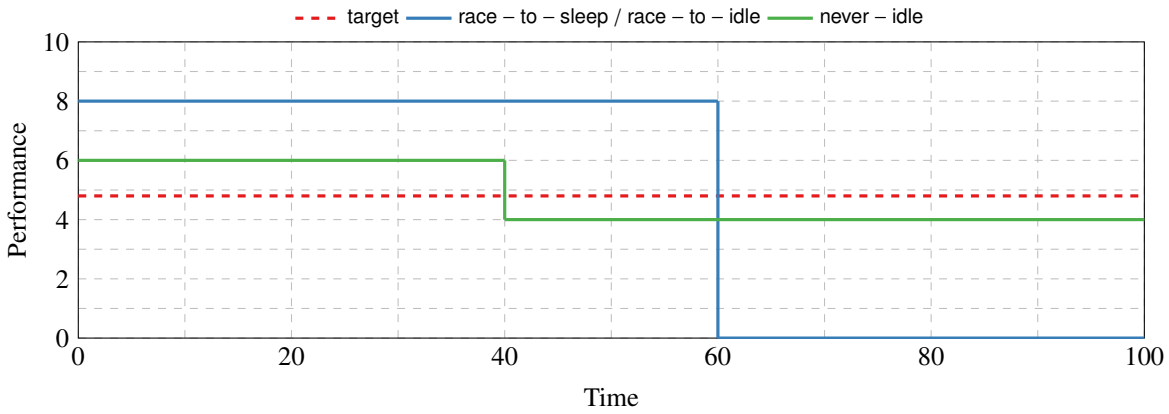

Figure 6.1: Example of race-to-sleep, race-to-idle, and never-idle heuristic behavior.

into the performance and power characteristics of the system being used and is therefore easily portable. *Race-to-sleep* is also easy to implement for the same reasons, though may require system-specific approaches to actually applying a sleep state. Additionally, transitioning to and from sleep states typically has more time and energy overhead than switching between DVFS states [43], which the other two heuristics use exclusively. For simplicity, these overheads are not accounted for in our formulations. *Never-idle* is somewhat more difficult to implement – more insight into the application's performance and the system's power characteristics for different configurations are required. The *hi* and *lo* states may vary between systems, so they cannot be chosen before the system is known and its properties understood. Figure 6.1 presents a time series example of each heuristic's performance behavior. The dashed line indicates the performance target.

6.2 Optimization Using Heuristics

Using the performance and power results from Chapter 5.2 and foreknowledge of input behavior, we derive an *oracle* that calculates the minimum amount of energy required to complete each benchmark while meeting a latency target. Naturally, this optimal energy result is subject to the accuracy of the configurations' performance and power values. In computing resource schedules, heuristic solutions are strictly limited by the accuracy of predetermined configuration performance

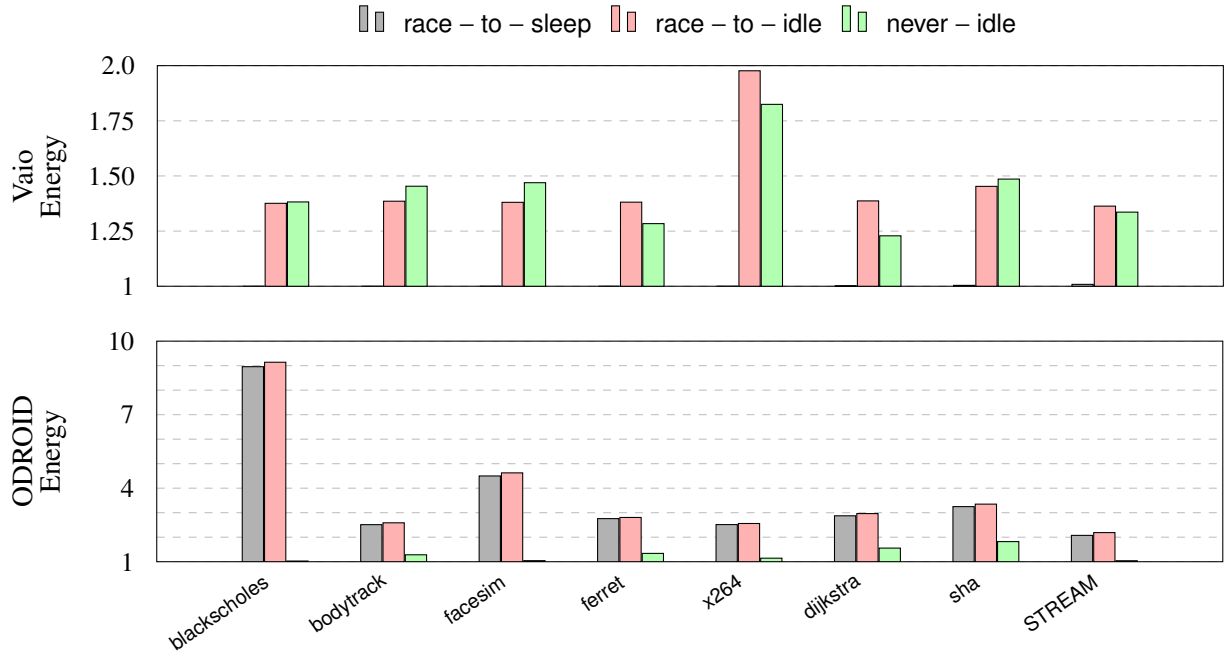

Figure 6.2: Energy consumption of heuristics on the Vaio and ODROID for a latency target of double their respective capacities.

values. Energy consumption is not accounted for explicitly when determining a schedule. We model the energy consumption of the *race-to-sleep*, *race-to-idle*, and *never-idle* heuristics for each benchmark and compare to the oracle on both platforms. The models assume perfect implementations with no latency or energy overhead for computation or changing system states. Energy is normalized to the oracle's for each benchmark and platform.

For each application i, a latency target requiring 100% of a system's performance capacity (configuration $C-1$) produces a minimum achievable latency m_i. We evaluate latency targets derived from performance goals that range from 5% to 95% of each system's performance capacity. Latency targets are inversely proportional to performance – for example, a 25% performance goal corresponds to a latency target that is $4 \times m_i$, and a 95% performance goal corresponds to a latency target that is slightly more than $1.05 \times m_i$.

We begin by examining the latency target that is double each system's minimum capability (*i.e.,* $2 \times m_i$, or 50% of their performance capacity) for each of the eight benchmarks. Figure 6.2 shows

that race-to-sleep is near-optimal on the Vaio for all benchmarks tested, averaging only 0.18% energy over optimal. Race-to-idle and never-idle average 46.28% and 43.30% energy consumption over optimal, respectively. This makes sense given that the Vaio is most energy efficient in its fastest state – it is desirable to complete as much work as possible in this state, then eliminate power consumption by sleeping. On the ODROID, never-idle is the better choice, averaging 28.19% energy consumption over optimal. On average, race-to-sleep and race-to-idle consume 3.68× and 3.78× the optimal energy, respectively. Since running at slower speeds is more efficient on the ODROID, it is preferable to use configurations that result in higher latency but are still able to meet the target, despite the fact that it could spend time sleeping and not consuming any power. Even when a sleep state is not considered, never-idle is still preferable despite the ODROID's extremely low idle power.

6.3 Sensitivity to Latency Target

Now we analyze the effect the latency target has on the energy consumption achieved by the heuristics. We use the `x264` application as an example again and consider a range from 5% to 95% of each system's maximum performance. Figure 6.3 presents the results of the heuristics on the Vaio and ODROID relative to their oracles' energy consumption.

On both systems, the higher latency (lower performance, *e.g.,* 5%) targets suffer the most excess energy consumption when an inefficient heuristic is used. The Vaio would consume 13.91× optimal energy if it used race-to-idle and 2.66× if it used never-idle. Similarly, the ODROID would consume 6.84× optimal using race-to-idle and 5.19× optimal using race-to-sleep. Interestingly, never-idle appears not to perform well either at this high latency target, still consuming slightly more than 2× the optimal energy. In fact, the optimal approach involves running at the slowest configuration on the performance/energy convex hull, then entering the sleep state. None of these three heuristics arrive at this conclusion, and all perform unsatisfactorily in this scenario. High latency targets offer the most opportunity to save (or waste) energy.

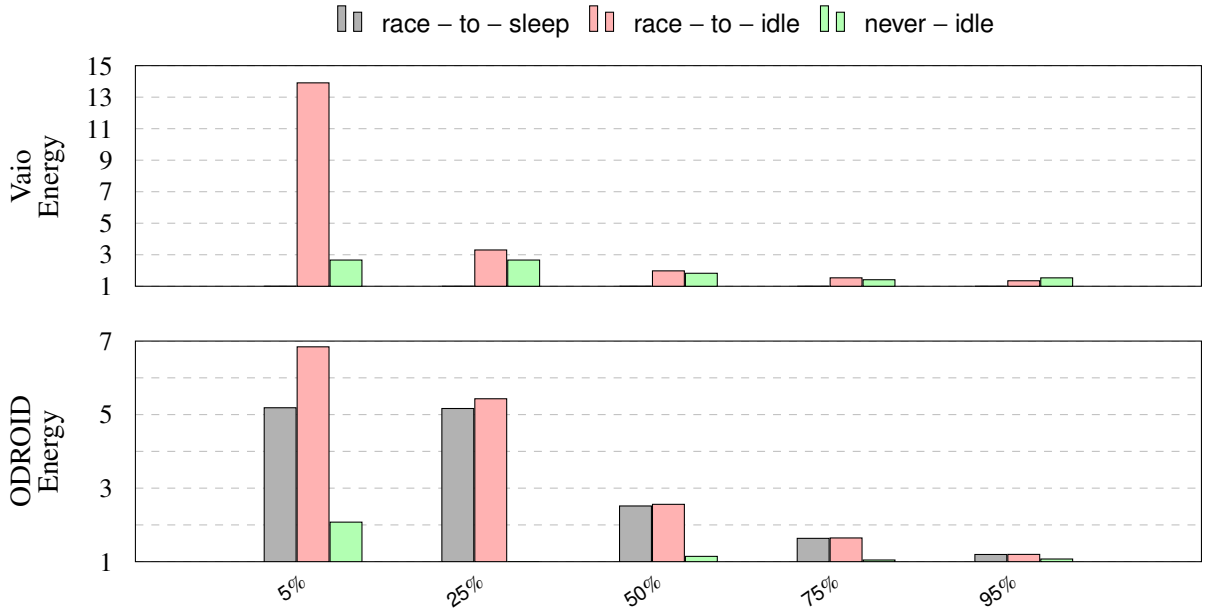

Figure 6.3: Results of heuristics for various latency targets using x264 on the Vaio and ODROID.

As the latency target decreases and approaches each system's minimum (maximum performance), the energy values converge. On the Vaio, the 25% target shows clear improvements in efficiency over 5%, and steadily improves at 50% and 75%. At 95%, the difference in energy consumption of the three heuristics is almost indistinguishable. There is less opportunity to save energy as each system's full resource capacity is required nearly the entire time to meet the deadlines. Similar behavior is seen on the ODROID (except between 5% and 25% as discussed). These results indicate that picking the correct resource allocation strategy is essential, especially for systems that are not fully utilized.

We briefly examine the results for `STREAM` in Figure 6.4 since it presented very different performance/power and latency/energy tradeoff spaces on both platforms (Chapter 5). The trends seen with `x264` hold in that race-to-sleep is still preferable on the Vaio while never-idle does best on the ODROID. However, the undesirable heuristics perform better than they did for `x264`, especially at higher latency (lower performance) targets. In a memory-bound application like `STREAM`, more time is spent transferring data to and from layers of the memory hierarchy and not actually completing work in the processor. This causes the application to run slower and consume more energy

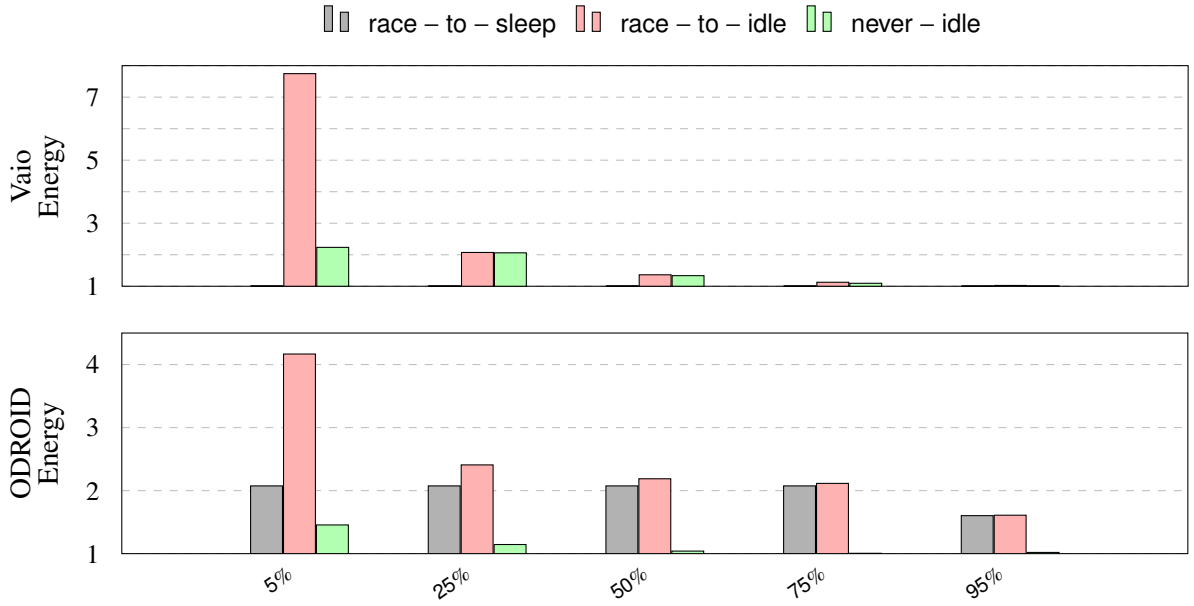

Figure 6.4: Results of heuristics for various latency targets using STREAM on the Vaio and ODROID.

than it would otherwise.

It is clear that the two platforms require different approaches to achieving acceptable energy consumption under timing constraints. Choosing the wrong approach for a system can result in wasted energy, especially when the system is underutilized. As explained in Chapter 2.3, this motivates the need for a better approach. Specifically, a portable solution is needed so that software and resource management solutions do not need to be changed when running on different systems.

CHAPTER 7

POET DESIGN AND IMPLEMENTATION

With the need for a general solution established, we now describe the POET framework and implementation. We explain what is required from users, describe the interface, and provide insight into the implementation of the runtime.

7.1 Resource Allocation Framework

An accurate and effective resource allocation framework for the problem we address must provide predictable timing in order for jobs to meet their deadlines, and it must minimize energy consumption. To keep the framework portable, we abstract performance and power from specific applications and systems, and use normalized values called **speedup** and **powerup**. We use control theory and compute a general control signal to handle the problem of providing predictable timing. The control signal is then used to solve the energy minimization problem using mathematical optimization. In computing a minimal energy schedule, POET is resilient to some error in speedup values, but is still subject to the accuracy of the predetermined configuration powerup values.

Figure 7.1 visualizes this approach, which we implement in POET. The application provides a *performance goal* (work rate) r_d which can be easily computed from a known workload W and desired latency (deadline) τ:

$$r_d = \frac{W}{\tau} \tag{7.1}$$

POET's runtime computes the *performance error* between the *performance goal* and the *performance feedback* and passes it to the **controller**. The controller then calculates a *generic control signal* that indicates the speed adjustment required to overcome the error. Given this control signal and a **resource specification** for the system, the **optimizer** produces a *resource schedule* that achieves the desired speedup while simultaneously minimizing energy consumption.

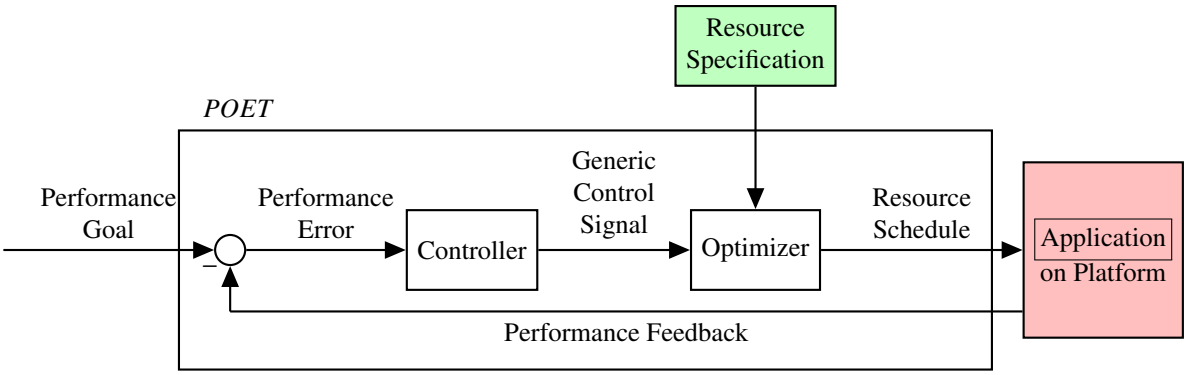

Figure 7.1: Overview of the POET runtime.

7.1.1 Controller

The controller computes the change in speedup required to cancel the error between the desired performance, r_d, and the measured performance, $r_m(t)$. POET models performance as:

$$r_m(t) = s(t-1) \cdot b(t-1) \tag{7.2}$$

The error $e(t)$ is then easily calculated as:

$$e(t) = r_d - r_m(t) \tag{7.3}$$

The controller performs its calculations at discrete time intervals to produce a new desired speedup, $s(t)$, implementing the *integral control law* [29]:

$$s(t) = s(t-1) + (1-\alpha) \cdot \frac{e(t)}{b(t)} \tag{7.4}$$

where α is a *pole* of the closed loop characteristic equation [24] such that:

$$0 \leq \alpha < 1 \tag{7.5}$$

The pole is configurable. Small values of α result in the controller being highly reactive to the input control signal, potentially producing a noisy signal itself. Large values, on the other hand, are less reactive to fluctuations in the input control signal, and may therefore be better suited to noisy systems.

As mentioned in Chapter 3, b is the base speed of the application on the system being used. The base speed may also change over time as a function of the application input, so POET continually estimates b as $b(t)$ at runtime to match current application behavior. This is accomplished using a Kalman filter [65]. 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 [65]:

$$\begin{cases} \hat{b}^-(t) &= \hat{b}(t-1) \\ e_b^-(t) &= e_b(t-1) + q_b(t) \\ k_b(t) &= \dfrac{e_b^-(t) \cdot s(t)}{[s(t)]^2 \cdot e_b^-(t)} \\ \hat{b}(t) &= \hat{b}^-(t) + k_b(t) \left[\dfrac{1}{d_m(t)} - s(t) \cdot \hat{b}^-(t) \right] \\ e_b(t) &= [1 - k_b(t) \cdot s(t-1)]\, e_b^-(t) \end{cases} \tag{7.6}$$

This formulates Kalman gain for the 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 [15].

A POET user does not need to know about Kalman filtering – $s(t)$ is computed by the controller, $d_m(t)$ and $q_b(t)$ are measured, and all other filter parameters are derived. The advantage of using the Kalman filter is that POET's formulation is independent of particular applications and systems. Computing the generic control signal $s(t)$ means that the controller does not reason about specific sets of resources, making it portable between systems.

Algorithm 1 Finding a Minimal-Energy Schedule.

Require: C ▷ system configurations, given by user
Require: $s(t)$ ▷ speedup, given by Eqn. 7.4
Require: ω ▷ discrete work units, given by application

```
under = {c | s_c ≤ s(t)}
over = {c | s_c > s(t)}
candidates = {⟨u, o⟩ | u ∈ under, o ∈ over}
energy = ∞
optimal = ⟨−1, −1⟩
schedule = ⟨0, 0⟩

for ⟨u, o⟩ ∈ candidates do                                   ▷ loop over all pairs
    ω_u = ω · ⌊ s_o · (s_u − s(t)) / (s(t) · (s_u − s_o)) ⌋   ▷ compute the work units to spend in each configuration in pair
    ω_o = ω − ω_u
    newEnergy = ω_u · p_u + ω_o · p_o                        ▷ compute energy of this pair
    if newEnergy < energy then                               ▷ compare energy to best found so far
        energy = newEnergy
        optimal = ⟨u, o⟩
        schedule = ⟨ω_u, ω_o⟩
    end if
end for

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

7.1.2 Optimizer

The optimizer translates the generic control signal generated by the controller (Eqn. 7.4) into a resource allocation strategy based on the resource specification it was provided (see Figure 7.1). The *schedule* is computed for the next ω work units.

The optimizer solves the linear optimization problem given by Eqns. 3.1–3.4. The approach is demonstrated in Algorithm 1. It requires as input: the system configurations given by the user, the speedup produced by the controller, and the number of work units given by the application (referred to as a *window* length in the implementation). It then partitions the configuration set into two subsets. The first subset contains all configurations with speedup less than or equal to the target speedup. The second subset contains the remaining configurations, with speedup greater than the target speedup. Algorithm 1 then loops over all pairs of configurations, with one configuration from each set, and determines how many work units to complete in each configuration to achieve

the target speedup. Note that it takes the floor of the value computed for the first configuration – since the value must be a whole number, this ensures that the actual speedup is not less than the target. This discrepancy contributes to the error computed in the next iteration (Eqn. 7.3), but is inversely proportional to the number of work units in a workload – the larger the workload, the smaller the potential error. For example, if a workload consists of 20 work units, this rounding down independently contributes strictly less than 5% error. The (normalized) energy that each pair would consume is then estimated, and if this value is lower than any previous estimate, the pair is remembered along with the computed schedule. When the algorithm completes, it returns the best configuration pair and their schedule. Since each set contains at most C elements, the upper bound of the algorithm complexity is $O(C^2)$.

7.1.3 Generality, Convergence, and Robustness

Both the controller and optimizer reason about speedup rather than true performance. Performance varies depending on the application, the input(s), and the system it is being executed on, but speedup is a general concept which can be applied to any application and system. Speedup, therefore, gives a more general metric for control. The controller's use of the Kalman filter to estimate base speed allows it to customize behavior for a specific application. The optimizer maintains platform independence by using the configurations provided as input to find an optimal solution without relying on heuristics, which we show may be system-specific or even application-dependent (Chapter 6).

Adaptive mechanisms that use control theory provide formal guarantees about the behavior of the system being controlled. It can be proven that the controller computes the correct speedup value to cancel the performance error and converges to the correct control signal once the Kalman filter has reached its correct estimate for the base application speed. This analysis is outside the scope of this thesis, but can be found in our publication of POET [34].

In analyzing POET's robustness to error, suppose that the speedup values provided for the

configurations in the resource specification are incorrect. Let s_c be the speedup with the largest error and Δ be its multiplicative error term. For example, $\Delta = 5$ means that s_c is off by a factor of five, and $\Delta = 0.5$ means that the actual speedup provided by configuration c is only half of that specified by s_c. The relationship between the base speed b and the speedup s_c means that an error in the speedup is equivalent to the same error in the base speed estimate. A formal analysis shows that the system is stable when

$$0 < \Delta < \frac{2}{1 - \alpha} \tag{7.7}$$

where α is the *pole*, subject to Eqn. 7.5 [34]. Effectively, the larger the pole value, the larger Δ can be. For example, if $\alpha = 0.1$, the largest Δ the system can handle is 2.2, whereas if $\alpha = 0.95$, the maximum Δ is 40. However, larger pole values cause the controller to respond more slowly to the input control signal, and as a result, the system will take longer to converge.

7.2 Prerequisites

Users must provide three pieces of information to POET:

1. A reference to the data structure provided by the Heartbeats library (Chapter 5.1).
2. The performance target for the application.
3. The system configurations and their associated performance and power characteristics.

The configurations may be restricted to those that are Pareto-optimal in the latency/energy tradeoff space, though it is not required to do so.

Fulfilling the first requirement is easy – just pass the Heartbeats data structure reference to POET during initialization. This allows POET to read performance and power metrics at runtime. The second requirement, providing a performance target, is managed through the Heartbeats API. This value is the real work rate as measured by Heartbeats, and can easily be derived from a latency goal. Minimum and maximum performance goals are specified as part of the Heartbeats initialization, so POET just takes the average of these two values as the desired performance target.

Finally, system configurations are provided to POET in two data structures. The first data structure is system-agnostic and contains a configuration identifier with the **speedup** and **powerup** values as described in Chapter 3 for each configuration. The second data structure can take any form that a developer considers appropriate for the system being used. It should contain the settings values of the resources to be manipulated in order to apply each configuration. POET includes a default format for this structure which contains a configuration identifier, the number of cores to use, and the DVFS setting to apply (core types, hyperthreading, and TurboBoost are managed through DVFS). In many instances, the second data structure (or at least the information required to create it) will be provided to developers. If not, the values can be derived by characterizing the system as described in Chapter 5.

7.3 Interface

The POET API provides three simple functions:

1. **poet_init**: Initializes and returns a *poet_state* data structure reference.
2. **poet_apply_control**: Executes the controller logic that implements Algorithm 1 and applies system configurations.
3. **poet_destroy**: Cleans up the *poet_state* data structure.

The initialization function, `poet_init`, takes as parameters: a reference to the Heartbeats data structure, references to the system configurations (two data structures), a reference to the function that applies system configurations, 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) is of 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 responsible for handling the *void* reference properly. 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 other two API functions,

`poet_apply_control` and `poet_destroy`, take the *poet_state* variable as their only parameter. The *poet_state* data structure contains all the runtime data required by the framework described in Chapter 3.

POET also provides auxiliary functions to load system configurations from files, apply system configurations, and discover the system's initial configuration. If the initial configuration cannot be determined, POET assumes that the system is in the state $C - 1$ (*i.e.,* the configuration that makes all resources available). If this turns out to be an incorrect assumption, the system will still successfully adapt, it just takes longer. The latter two of these auxiliary functions meet the *poet_apply_func* and *poet_curr_state_func* definitions, respectively, and can therefore be passed to `poet_init`. All of these auxiliary functions are platform-dependent and so are kept separate to maintain portability and allow users to write their own implementations, if required.

7.4 Runtime

During runtime, an application makes calls to the Heartbeats library at regular intervals, as shown in Listing 5.1. After each heartbeat, a POET-enabled application then makes a call to the function `poet_apply_control`, which contains POET's core logic. Listing 7.1 demonstrates the implementation of a POET-enabled application.

The window size that was specified in the initialization of the Heartbeats data structure indicates to POET how often it should act. At the completion of a window, POET executes its task. First it estimates the application's base speed with Eqn. 7.6, computes the performance error with Eqn. 7.3, and computes the speedup with Eqn. 7.4 in order to eliminate the error. Next, Algorithm 1 is run to determine the resource schedule that minimizes energy consumption and achieves the required speedup. Finally, POET applies the first of the two configurations returned by Algorithm 1 by executing the `poet_apply_func` function provided during initialization, enacting the newly computed schedule.

Recall that `poet_apply_func` is a platform-dependent operation. The implementation of this

```
// initialization
heartbeat_t* heart =
  heartbeat_acc_pow_init(window_size, 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(heart, nstates, control_states, cpu_states,
                              &apply_cpu_config, &get_current_cpu_state,
                              buffer_depth, "poet.log");
// execution of main loop
while(running) {
  heartbeat_acc(heart, count++, 1);
  poet_apply_control(state);
  doWork();
}
// cleanup
poet_destroy(state);
free(control_states);
free(cpu_states);
heartbeat_finish(heart);
```

Listing 7.1: Example of POET-enabled application code.

auxiliary function that is provided with POET invokes the `taskset` utility to force the application process and all of its threads onto the number of cores specified in the scheduled configuration. It then uses the `cpufrequtils` interface to adjust the DVFS settings of the cores (naturally, the *userspace* cpufreq governor must be in use for this change to take effect). Of course, developers can specify their own functions if needed or otherwise desired. For example, a system may have additional configurable resources that allow it to further manipulate performance and power, like additional memory controllers, caches, or network bandwidth.

The only remaining task after applying the first configuration is to apply the second one at the appropriate time during the next window period, as computed in the schedule by Algorithm 1. When the computed number of heartbeats to wait passes, POET executes the `poet_apply_func` function again to apply the second configuration, though no further computation is performed. When the heartbeat window completes, the control process repeats.

CHAPTER 8

POET EVALUATION

This chapter describes our experimental evaluation of POET. First, we prove its ability to meet soft real-time latency requirements imposed by an application. Next, we analyze POET's energy usage on our two systems and show that it also achieves near-optimal energy consumption. We then examine its reaction to input with multiple phases and evaluate its ability to adapt accordingly. Finally, we evaluate POET's ability to handle interference caused by other applications.

8.1 Meeting Timing Constraints

To test POET's ability to meet timing constraints, we execute each application with four latency targets, derived from performance goals ranging from 25% to 95% of each system's performance capacity. See Chapter 6.2 for a brief description of the relationship between latency and performance. The latency target derived from a 5% performance goal is not tested because it requires using an idle or sleep state, which POET does not support at this time.

Recall that an execution is composed of multiple jobs, as presented earlier in Table 4.3. We quantify POET's ability to meet timing constraints by comparing each job's actual latency to the target. We use a metric that is standard in control theory for evaluating the behavior of controllers – the Mean Absolute Percentage Error (MAPE) [24]. MAPE is an unforgiving metric since it penalizes every violation of a latency target. For an application composed of n jobs:

$$MAPE = 100\% \cdot \frac{1}{n} \sum_{i=1}^{n} \begin{cases} d_m(i) > d_r : & \left| \frac{d_m(i) - d_r}{d_r} \right| \\ d_m(i) \leq d_r : & 0 \end{cases} \tag{8.1}$$

where d_r is the specified latency requirement and $d_m(i)$ is the actual measured latency for the i-th job. In short, each missed deadline increases the MAPE value by an amount relative to the tardiness of the job completion.

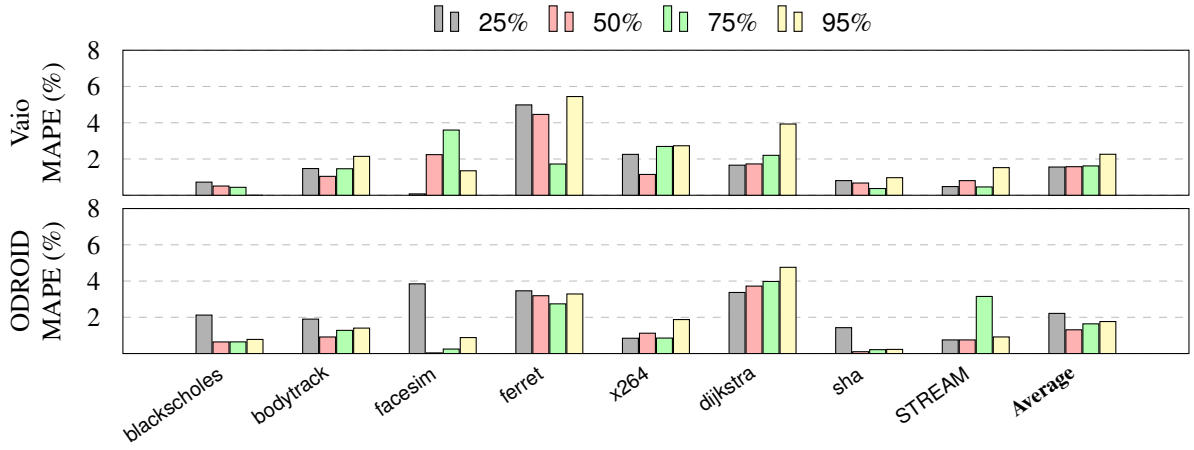

Figure 8.1: Latency error for each benchmark.

The MAPE values for the four latency targets tested (corresponding to 25%, 50%, 75%, and 95% of each system's performance capacity) are presented in Figure 8.1 for each benchmark on the Vaio and ODROID (lower is better, 0 is optimal). On both the Vaio and the ODROID, the average MAPE value for all benchmarks is below 2.5% for all latency targets. We presented the variability of each benchmark's behavior (as a function of their inputs) in Figure 4.1 and find that, in general, the benchmarks with larger latency variance result in higher MAPE values. This is to be expected since more volatile behavior is more difficult to control. Still, POET achieves low MAPE values even for applications that were not originally designed to support predictable timing.

8.2 Minimizing Energy

In Chapter 5 we collected and presented performance/power and latency/energy metrics for each application on both systems for all possible configurations, given the components we manipulate. We therefore have perfect knowledge of each application's behavior for the inputs used. We apply this knowledge to create an *oracle* that computes the minimum energy required to meet each latency target, *i.e.,* the energy consumed when the best configurations are chosen for each job with no computation or configuration switching overhead and perfect knowledge of the job's needs. Of course, POET has non-zero overhead and does not have foreknowledge of input behavior.

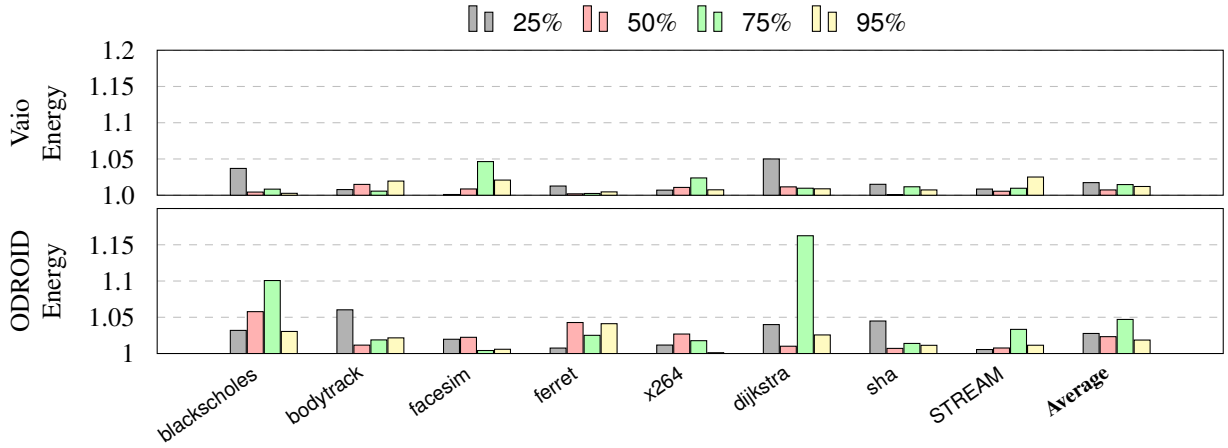

Figure 8.2: Normalized energy consumption for each benchmark.

For each execution that we calculated MAPE for, we also compute the energy consumption based on the power and timing results. We then compute the ratio of this energy consumption to the minimum achievable energy derived by the oracle. Figure 8.2 presents the energy consumption results for each benchmark on the Vaio and the ODROID, with energy normalized to the oracle's minimum (lower is better, 1 is optimal). Values greater than 1 show energy consumption above the optimal. Across all applications and latency targets, POET's energy consumption exceeds optimal by an average of 1.3% on the Vaio and 2.9% on the ODROID. This demonstrates that POET achieves near-optimal energy consumption in practice, even with the overhead of computing and applying system configurations.

The `dijkstra` application on the ODROID with a 75% target is the most difficult execution, exceeding optimal energy by about 16%. POET's runtime overhead is low, but not zero, and in this case forces a higher DVFS frequency than the optimal schedule requires. While this normally introduces a small additional energy cost, this area of the latency/energy tradeoff space results in a particularly high cost. Decreasing the latency by 2% (POET's approximate overhead) increases energy consumption by over 15%, which accounts for most of the additional energy consumed by POET compared to optimal.

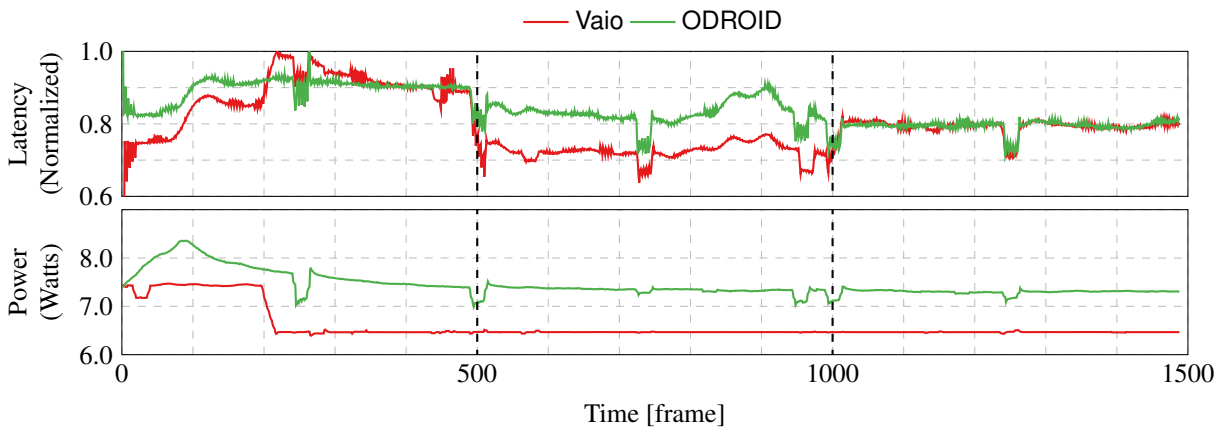

Figure 8.3: Uncontrolled processing x264 input with distinct phases.

8.3 Responding to Application Phases

We now examine POET's ability to cope with a workload that varies with time – specifically, we use a workload that exhibits three distinct phases. This test uses the `x264` benchmark with an input that is a combination of three different videos of varying encoding difficulty. As a result, we get an input with three distinct phases, each composed of 500 jobs (video frames). Figure 8.3 displays time series data for both latency and power on the Vaio and ODROID when they each run without POET in configuration $C-1$. Latency is normalized to the maximum measured for all iterations, *i.e.,* the empirically determined worst case. Raw power data is used here, making actual energy consumption for each system the integral of their respective curve. The bold dashed vertical lines indicate the separation of phases at frames 500 and 1000.

The three phases are clearly distinguishable by their changes in latency behavior. Interestingly, the two systems do not process each phase with the same relative difficulty. 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 phase is the easiest for the ODROID.

We now enable POET and set the latency target to the maximum measured latency in the uncontrolled execution, ensuring that the latency target is achievable for each job. Jobs (frames) that require less time or fewer resources to achieve the latency goal present opportunities to save

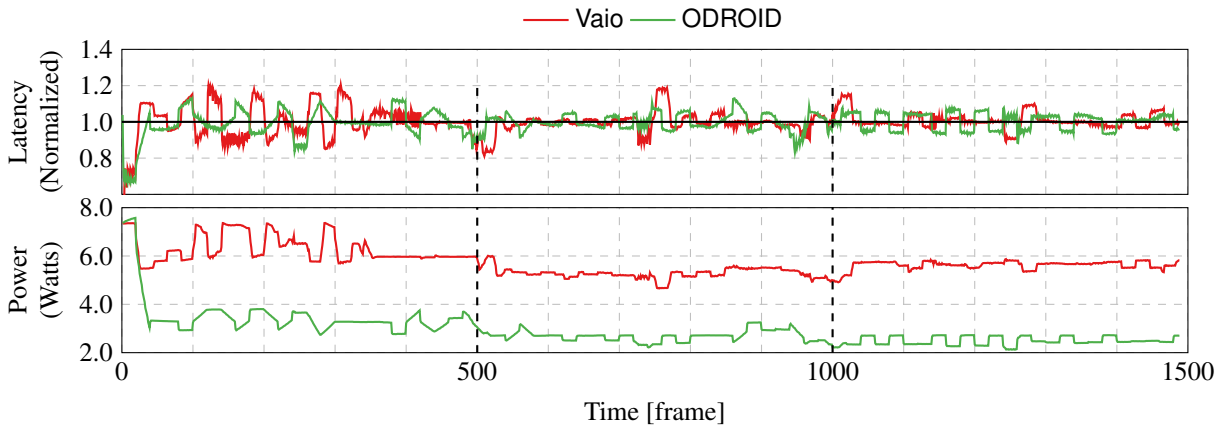

Figure 8.4: POET processing for x264 input with distinct phases.

energy. The results are presented in Figure 8.4, with latency values normalized to the target and actual power values used. Dips and spikes in both latency and power are visible at the beginning of each phase, showing the change in the input behavior and POET adapting to those changes. Other disturbances are a result of the controller responding to the unpredictable input (`x264` has high timing variability – see Chapter 4.2.2). POET is able to meet the latency target on both systems, achieving 2.2% MAPE on the Vaio and 2.0% MAPE on the ODROID.

8.4 Adapting to Other Applications

Now we evaluate POET's behavior when other applications cause disruptions to the system. It is assumed that this external load is not under direct control of POET and is not otherwise manipulating the system resource configurations. We launch the POET-enabled `bodytrack` application with a latency target, then approximately halfway through its execution, we launch another application. The second application consumes system resources, slowing down the POET-enabled bodytrack. In response, POET enables more system resources so that bodytrack continues to meet its latency target.

Figure 8.5 shows a time series of the resulting behavior, with the top portion displaying the results on the Vaio and the bottom portion displaying the results on the ODROID. Values are

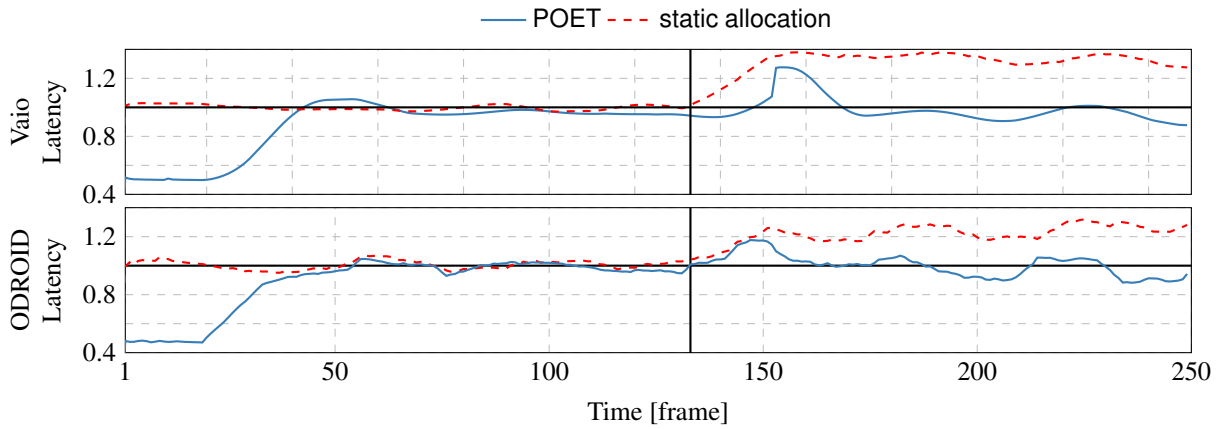

Figure 8.5: Normalized latency behavior of POET running with another application compared to a static allocation technique.

normalized to the latency target. The solid vertical lines indicate when the second application is launched. On both systems, the latency temporarily increases before POET adjusts the resource allocation to compensate for the disruption caused by the second application. The charts also include curves that show the latency achieved using a statically managed system which just selects the resource allocation at the beginning, thus highlighting the benefits of POET. With a statically managed system, the introduction of the second application dramatically increases the job latency. To quantify this effect, we compute the MAPE over the entire execution for both POET-enabled application and the static allocation technique. On the Vaio, POET's MAPE is 2.3%, which includes the period of adjustment after the second application is launched, while the static case has a MAPE of 16%. On the ODROID, POET achieves a 2.4% MAPE, while the static case suffers 12% MAPE.

8.5 Results and Limitations

We have demonstrated that POET achieves its design goal of meeting timing constraints while achieving near-optimal energy consumption. These results are achieved despite the facts that the applications tested were not originally designed to offer predictable timing and that the platforms used have completely different performance/power and latency/energy tradeoff spaces. Applica-

tions only require minimal modification to use POET and no code changes are needed to exploit the variation in resources that different platforms provide, even when those resources expose different performance, power, and energy characteristics. Only different resource specifications were needed, allowing us to use the same application and POET code on both systems. This testifies to POET's portability.

Naturally, there are some limitations to our approach. First, POET supports soft real-time constraints, not hard real-time guarantees. The controller is guaranteed to converge to the desired performance/latency target and is provably robust to errors, but the goal may be violated during the settling time (though more accurate speedup estimates produce better results). This is clearly demonstrated in Figure 8.5 when POET adapts to the presence of the second application. Additionally, applications or inputs that demonstrate high variability in behavior may still cause violations before the controller settles again, as seen in Figure 8.4 when controlling the high-variance `x264` application. This is further evidence of the tension between timeliness and efficiency [14] – the substantial energy savings on the ODROID comes at a cost of some latency errors.

POET is also sensitive to the resource specifications the user provides at initialization. While large errors can be tolerated, in practice it is better to classify applications by their behavior (*e.g.,* compute-bound, memory-bound) and use a different configuration set for each class. For example, most applications tested are similar to the `x264` benchmark, but `STREAM` and `blackscholes` exhibit behavior that is sufficiently different to warrant their own configuration sets.

The choice of the window size also impacts POET's behavior. Since there are a discrete number of jobs to complete in a window period, POET will experience timing error and energy overhead that is inversely proportional to the window size. We recommend a window size of at least 20, which allows an error range that is strictly less than a 5%. Smaller window periods allow POET-enabled applications to adapt more quickly, but are also more susceptible to this naturally occurring error and to other noise in the system.

At this time, POET's models do not account for the latency required to switch between con-

figurations. The best example in the work performed here is the migration between the big and LITTLE clusters on the ODROID's ARM processor. Application state needs to be transferred between clusters and hardware powered up or down. The loss in performance caused by switching is currently modeled as an inaccuracy in the speedup specified in the system configurations. This is likely to be a more serious concern when sleep states are used as their switching time is generally higher than between DVFS settings. Still, our results show that this simplification works well in practice, though it may be insufficient when dealing with resources that have extremely long transition latencies. In such a scenario, POET should be extended to explicitly account for state transition delays in the controller and optimizer.

Finally, POET assumes that only one running application (possibly consisting of multiple, communicating threads) needs to meet a deadline. Concurrently running multiple applications with timing constraints is not supported. Coordination between POET-enabled applications would be needed to prevent conflicts in scheduling shared resources and applying resource allocations.

CHAPTER 9

FUTURE WORK

As the configurability of systems continues to grow, it is already becoming infeasible to execute non-trivial applications and inputs in all possible system states in order to understand their behavior [57]. For example, running the `x264` application with a 500 frame 1080p video input on the ODROID in all 68 active configurations (*i.e.,* no sleep or idle states) takes around three days to complete. While the input for this example is almost certainly larger than necessary for learning sufficient information about a configuration's behavior, it is likely that the problem will persist and eventually become untenable. The configuration space grows exponentially with the introduction of new adjustable components and new resource settings, so a more intelligent approach is needed to understand behavior of system configurations. Mishra et al. propose LEO [50], a machine learning technique for estimating Pareto-optimal configurations, which has the potential to reduce the amount of time required to complete this task. Such approaches can be integrated with POET.

Even if all states can be evaluated experimentally in advance, the $O(C^2)$ complexity of Algorithm 1 could result in unacceptably high runtime overhead. In practice, the size of the set of configurations can be reduced by including only those configurations with Pareto-optimal performance/energy values. Sorting into Pareto-optimal takes $O(C \log_2(C))$ time and only needs to be done once. More creative approaches to identifying desirable configurations by speedup can also be implemented to further reduce the algorithm complexity. For example, configurations could be indexed in a sufficiently large pre-filled hash table, using the speedup as the key and reducing the complexity to constant time.

Other future work can address some of the current shortfalls in POET. Support for sleep states can be added to improve energy consumption on systems like the Vaio which benefit from a *race-to-sleep* heuristic. Sleeping is a system-specific operation, so developers could currently add this capability themselves without modifying POET. As mentioned in Chapter 8.5, handling sleep states could also be accompanied by introducing explicit modeling of state transition latencies.

A simple implementation could account for a single expected delay value, though it could be more complex if the transition latencies depend on the prior state. Support can also be added for coordinating multiple applications simultaneously – a priority-based scheme could ensure that high priority applications are allocated sufficient resources to meet their deadlines, and allocate for low priority applications using a best-effort approach.

POET can also be modified to provide more abstract control behavior. Currently it meets a target in one dimension (performance) while minimizing in another (energy consumption). It should be straightforward to introduce a runtime switch that allows the software to support either maximizing or minimizing in the second dimension. For example, a user may wish to meet a power target while maximizing performance. Embedded systems are sometimes over-provisioned and can draw power faster than they can dissipate the heat generated, which can cause the system to overheat and possibly cause irreparable damage to itself [25]. In such instances, meeting a power target is a reasonable goal, and short-lived violations may be tolerable. Alternatively, an application may wish to meet a performance target and maximize computation accuracy (for software that supports adjusting this kind of behavior), *e.g.,* a video encoder or decoder. Of course, new implementations of the *poet_apply_func* function definition need to be written to interpret the system-specific (or even application-specific) configurations and apply them (see Chapter 7).

POET could also be extended to allow applications to modify their performance (or power, accuracy, etc.) target during runtime. A simple implementation should require little or no changes beyond exposing this setting via a setter interface function. The POET controller would read the new target value, the last achieved value, and treat the difference as error which needs to be canceled. It would then compute the appropriate speedup control signal without ever being aware that the target was actually changed. More advanced approaches that are aware of changes to the target should be able to adapt faster.

We highlighted in Chapter 5 some unanticipated application behavior on the ODROID for `blackscholes` and `STREAM`. The former performed unexpectedly well on the LITTLE cores

while the latter did not achieve better energy efficiency using the LITTLE cluster than the big cluster. One question that immediately follows from the cache comparison between the clusters is whether the LITTLE cores would benefit from data prefetching. Similarly, what provided the most benefit to applications when running on the big cores – the out-of-order speculative execution, the data prefetching, or something else? Further evaluation of this big.LITTLE implementation could determine more precisely why these applications behaved as they did and potentially yield insights for improving heterogeneous architectures.

The work in this thesis evaluates two modern embedded platforms. However, the embedded systems field changes rapidly and there are already newer devices available with different capabilities that POET can be evaluated on. For example, some big.LITTLE systems now support Global Task Scheduling (GTS) [38] which allows any possible subset of cores to be allocated to a single task, with each cluster running at different frequencies. This greater flexibility comes at a cost of exponentially increasing the number of possible configurations. GTS offers opportunities for true simultaneous heterogeneous computing, meaning more advanced approaches may be required for efficient resource scheduling. This capability is not yet available in the mainline Linux kernel, but can be found in the Linaro Linux kernel [27].

Although POET's behavior has been analyzed on embedded platforms, it can be used on larger scale systems like laptops, desktops, and servers without modification (scale up). Distributed environments also require performance and power/energy management (scale out). A coordinated networked solution of POET could be designed and implemented for distributed systems. Furthermore, coordinating multiple distributed applications with shared resources can be explored.

CHAPTER 10

CONCLUSION

Embedded systems contain an increasing variety of configurable resources to assist developers in managing application and system behavior. The complexity in scheduling these resources leads many developers to use heuristic approaches to meet software timing constraints while approximating minimal energy consumption. This thesis examines two embedded multi-core systems and finds that they expose different performance/power and latency/energy tradeoff spaces. As a result, each requires a different heuristic approach to achieve acceptable results. Lack of portability like this leads designers to write their solutions for specific platforms or applications, meaning software must be rewritten for different systems. The need for a portable solution motivates us to develop POET – the Performance with Optimal Energy Toolkit. POET uses control theory and mathematical optimization to generalize the problem of energy-aware resource allocation, without relying on heuristics. We test POET on the two systems using a variety of benchmarks suitable for embedded platforms that were not originally designed to meet timing constraints. The results demonstrate that POET meets timing constraints and achieves near-optimal energy consumption, as determined by a dynamic optimal oracle. We release POET as open source software, along with benchmark patches and the configurations used in its analysis.

REFERENCES

[1] Cortex-a series processors. http://infocenter.arm.com/help/topic/com.arm.doc.set.cortexa/index.html.

[2] Watts Up? meters. http://www.wattsupmeters.com/.

[3] 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.

[4] Susanne Albers. Algorithms for dynamic speed scaling. In *STACS*, pages 1–11, 2011.

[5] Susanne Albers and Antonios Antoniadis. Race to idle: new algorithms for speed scaling with a sleep state. In *SODA*, 2012.

[6] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 248–259, Washington, DC, USA, 1999. IEEE Computer Society.

[7] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[8] L. Bertini, J.C.B. Leite, and D. Mosse. Statistical qos guarantee and energy-efficiency in web server clusters. In *ECRTS*, 2007.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *17th Conference on Parallel Architectures and Compilation Techniques*, 2008.

[10] W. Lloyd Bircher and Lizy K. John. Analysis of dynamic power management on multi-core processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 327–338, New York, NY, USA, 2008. ACM.

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

[12] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[13] S.P. Bradley, A.C. Hax, and T.L. Magnanti. *Applied mathematical programming*. 1977.

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

[15] Liyu Cao and Howard M. Schwartz. Analysis of the kalman filter based estimation algorithm: An orthogonal decomposition approach. *Automatica*, 40(1), 2004.

[16] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 225–236, Washington, DC, USA, 2012. IEEE Computer Society.

[17] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[18] Aaron Carroll and Gernot Heiser. Mobile multicores: Use them or waste them. In *HotPower*, 2013.

[19] Jian Chen and Lizy K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 927–930, New York, NY, USA, 2009. ACM.

[20] Hui Cheng and Steve Goddard. SYS-EDF: a system-wide energy-efficient scheduling algorithm for hard real-time systems. *IJES*, 4(2), 2009.

[21] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, Oct 1974.

[22] Abdullah Elewi, Mohamed Shalan, Medhat Awadalla, and Elsayed M. Saad. Energy-efficient task allocation techniques for asymmetric multiprocessor embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(2s):71:1–71:27, January 2014.

[23] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

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

[25] Yong Fu, Nicholas Kottenstette, Chenyang Lu, and Xenofon D. Koutsoukos. Feedback thermal control of real-time systems on multicore processors. In *EMSOFT*, 2012.

[26] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31(2):86–95, March 2011.

[27] George Gray. big.LITTLE software update. `http://www.linaro.org/blog/hardware-update/big-little-software-update/`.

[28] HardKernel. `http://www.hardkernel.com/main/products/prdt\_info.php?g\_code=G137463363079`.

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

[30] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[31] H. Hoffmann, M. Maggio, M.D. Santambrogio, A. Leva, and A. Agarwal. A generalized software framework for accurate and efficient management of performance goals. In *EMSOFT*, 2013.

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

[33] Henry Hoffmann. Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *ECRTS*, 2014.

[34] 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.

[35] Texas Instruments. `http://www.ti.com/product/ina231`.

[36] S.M.Z. Iqbal, Yuchen Liang, and H. Grahn. Parmibench - an open-source benchmark for embedded multiprocessor systems. *Computer Architecture Letters*, 9(2), 2010.

[37] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. *ACM Trans. Algorithms*, 3(4), November 2007.

[38] Brian Jeff. big.LITTLE technology moves towards fully heterogeneous global task scheduling. `http://www.arm.com/files/pdf/big_LITTLE_technology_moves_towards_fully_heterogeneous_Global_Task_Scheduling.pdf`.

[39] David H. K. Kim and Henry Hoffmann. Racing and pacing to idle: Minimizing energy under performance constraints. Technical Report TR-2014-10, University of Chicago, Dept. of Comp. Sci., 2014.

[40] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[41] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 64–, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[43] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *USENIX ATC*, 2011.

[44] 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.

[45] M. Maggio, H. Hoffmann, M.D. Santambrogio, A. Agarwal, and A. Leva. Power optimization in embedded systems via feedback control of resource allocation. *IEEE TCST*, 21(1), 2013.

[46] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Com-*

puter Architecture, ISCA '98, pages 132–141, Washington, DC, USA, 1998. IEEE Computer Society.

[47] T. Martin and D. Siewiorek. A power metric for mobile systems. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, ISLPED '96, pages 37–42, Piscataway, NJ, USA, 1996. IEEE Press.

[48] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 1995.

[49] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *38th Annual Symposium on Computer Architecture*, 2011.

[50] Nikita Mishra, Huazhe Zhang, John Lafferty, and Henry Hoffmann. A bayesian approach for minimizing energy under performance constraints. In *ASPLOS*, 2015.

[51] 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.

[52] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED*, 1998.

[53] Vinivius Petrucci, Orlando Loques, and Daniel Mosse. Lucky scheduling for energy-efficient heterogeneous multi-core systems. In *HotPower*, 2012.

[54] Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Power-performance modeling on asymmetric multi-cores. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 15:1–15:10, Piscataway, NJ, USA, 2013. IEEE Press.

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

[56] Saowanee Saewong and Ragunathan (Raj) Rajkumar. Practical voltage-scaling for fixed-priority rt-systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '03, pages 106–, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

[59] Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. Price theory based power management for heterogeneous multi-cores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 161–176, New York, NY, USA, 2014. ACM.

[60] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Singh. Who killed my battery?: Analyzing mobile browser energy consumption. In *WWW*, 2012.

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

[62] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. *ACM Comput. Surv.*, 37(3):195–237, September 2005.

[63] Meng Wang, Zili Shao, C.J. Xue, and E.H.-M. Sha. Real-time loop scheduling with leakage energy minimization for embedded vliw dsp processors. In *RTCSA*, 2007.

[64] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. *Mobile Computing*, 1996.

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

[66] Chuan-Yue Yang, Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. System-level energy-efficiency for real-time tasks. In *SOCRTD*, 2007.

[67] Heechul Yun, Po-Liang Wu, A. Arya, T. Abdelzaher, Cheolgi Kim, and Lui Sha. System-wide energy optimization for multiple dvs components and real-time tasks. In *ECRTS*, 2010.

[68] Ronghua Zhang, Chenyang Lu, T.F. Abdelzaher, and J.A. Stankovic. Controlware: a middleware architecture for feedback control of software performance. In *ICDCS*, 2002.

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